

**CSC547/ECE547 Semester Lab Project  
SPRING 2020**

**Final Project Report**

# **Cloud Bursting to AWS from VCL**

**Team 2**

**Lead: Prashanth Mallyampatti**

**Vaibhav Singh**

**Shantanu Chandorkar**

**Christian Perez Ortiz**

**Rahul Aettapu**

## Table of Contents:

<b>Abstract:</b>	<b>4</b>
<b>Introduction:</b>	<b>5</b>
Problem Statement:	5
Motivation:	5
<b>Requirements:</b>	<b>6</b>
Apache VCL:	6
AWS:	6
Functional Requirements:	6
FR-1. Provisioning of AWS instances:	6
FR-2. Store the AWS instances data:	6
FR-3. Dashboard to display AWS Instance details:	7
FR-4. Deletion of AWS instances:	7
Non-Functional Requirements:	7
NFR-1. Availability:	7
NFR-2. Scalability:	7
NFR-3. Privacy:	7
NFR-4. Data-Retention:	7
NFR-5. Transparent:	7
NFR-6. Maintainability:	7
NFR-7. Idempotent:	8
NFR-8. Cost Effective	8
NFR-9. Security	8
<b>System Environment:</b>	<b>9</b>
<b>Architectural Design:</b>	<b>10</b>
VCL System:	11
VCL Service:	11
VCL Operations:	11
VCL Database:	12
VCL Webserver:	12
VCL Dashboard:	13
AWS System:	13
AWS Management Console:	13
AWS Operations:	14

Cloud Bursting System:	14
Scripts:	14
AWS Database:	15
AWS Dashboard:	16
Logs:	16
<b>Implementation:</b>	<b>17</b>
System setup:	17
Threshold Check:	17
AWS Instance Creation:	18
Parse User & Request Data:	19
Create & Store Key Pair:	19
Create Security Group:	20
Create Instance:	20
Fetch and Store Results:	21
Reservation Flow	22
AWS database:	23
AWS dashboard:	24
Modes, Pages & Action Functions (PHP web application):	24
Database queries & user session:	24
awsdetailsFunc Action Function & pre-rendered html:	24
awsdeleteform Action Function & Forms:	25
Private Key links:	25
Instance removal:	26
<b>Verification and Validation:</b>	<b>27</b>
<b>Schedule and Personnel:</b>	<b>35</b>
<b>Results:</b>	<b>38</b>
<b>Future Work:</b>	<b>39</b>
<b>References:</b>	<b>40</b>
<b>Appendices:</b>	<b>41</b>

**Abstract:**

In this class project, we implemented a solution to enhance the current VCL architecture to handle resource scarcity due to excessive demand on the system. We architected, developed and integrated a new cloudbusting provisioning framework that enables the system to detect excessive demand from a user and burst any extra demand to AWS EC2 public cloud seamlessly. We implemented the feature to be transparent to the user and that bursting only happens in the case of resource scarcity or load spikes. We also ensured that multiple users are able to reserve VMs at the same time on AWS. We provide the user the functionality to delete provisioned EC2 instances and remove all data that has been entered by the user. We also introduce a basic IAM system in which an admin user can seamlessly stop a malicious user's AWS instances.

## Introduction:

- **Problem Statement:**

The task that was provided in this project was to improve the existing VCL architecture by adding a new feature to burst onto the public cloud whenever required. This feature had to be transparent so that the users can use the same web portal for reserving VMs on the public cloud. Cloud bursting had to be done only when the resources in the VCL were finished.

Cloud Bursting is to dynamically deploy the applications that normally run on a private cloud into a public cloud to meet the customer requirements and handle peak demands when private cloud resources are insufficient.

- **Motivation:**

The motivation of this project is to ensure that the system can handle the increasing capacity of customer reservation requests in a cost effective manner. Some of the ways to handle this were to add additional capital equipment to the existing VCL cloud or consider the option of bursting to a public cloud. With the uncertainty in the demand from users, the additional capital equipment would be either surplus or insufficient. Hence the option of additional physical infrastructure was not good enough. Bursting to public cloud on demand is the best option economically and technically.

## Requirements:

- **Apache VCL:**

NCSU provides a remote access service that allows you to reserve a computer with a desired set of applications for yourself, and remotely access it over the Internet. This service is powered by the Apache Software Foundation's Virtual Computing Lab (VCL) software[1][2]. It provides various applications such as Matlab, Maple, SAS, Solidworks, Linux, Solaris and numerous Windows environments and many others to all NC State students and faculty.

- **AWS:**

Amazon Web Services (AWS)[3] is a subsidiary of Amazon that provides on-demand cloud computing platforms and APIs to individuals, companies, and governments, on a metered pay-as-you-go basis. In aggregate, these cloud computing web services provide a set of primitive abstract technical infrastructure and distributed computing building blocks and tools. One of these services is Amazon Elastic Compute Cloud[4], which allows users to have at their disposal a virtual cluster of computers, available all the time, through the Internet.

- **Functional Requirements:**

1. FR-1. Provisioning of AWS instances:

Bursting instances to Amazon Web Services - EC2 has to take place for a user in NCSU Virtual Computing Lab - VCL on predetermined instance threshold limits. This requires hybrid cloud architecture to be built, with AWS as the public cloud and VCL as the private cloud provider. To do this we need to integrate the AWS console to the VCL system such that API calls would help with the communication between the two cloud environments.

2. FR-2. Store the AWS instances data:

Once the connection is established, the data to be passed and fetched to/from AWS needs to be stored. This would be an essential step in further tasks such as reservation modification, deletion and displaying it on the user dashboard.

3. FR-3. Dashboard to display AWS Instance details:

A custom AWS dashboard has to be built for each user displaying essential details of his/her reservation and to be interactive enough for accessing his/her private keys for the reservations made and delete reservations upon request.

4. FR-4. Deletion of AWS instances:

Users should be able to delete reservations and cleanup operations have to be performed. All of the above requirements form the functional requirements of this project.

- Non-Functional Requirements:

Some of the non-functional requirements that we intend to achieve are as follows.

1. *NFR-1. Availability:*

Resources for reservations must be always available for users.

2. *NFR-2. Scalability:*

Multiple users must be able to reserve multiple reservations at any given point of time.

3. *NFR-3. Privacy:*

AWS reservations made for a user must not be accessible or visible for other users.

4. *NFR-4. Data-Retention:*

User reservations and their details must be persistent on multiple user's sessions. Security - Secure login and unique private keys have to be provided to the user for reservations made on AWS. Transparent

5. *NFR-5. Transparent:*

Users should be unaware of the underlying architecture and be able to make AWS reservations through the existing VCL system.

6. *NFR-6. Maintainability:*

Integrating AWS cloud to VCL should be as modular as possible, so that future developments can be made easily.

7. *NFR-7. Idempotent:*

Develop a code base which when multiple times called and executed should not change the existing state of the system.

8. *NFR-8. Cost Effective:*

The integration should always prefer spinning up instances on cheaper private clouds than on AWS.

9. *NFR-9. Security:*

The AWS/VCL admin should be able to restrict/ terminate VMs created by a malicious user.



## System Environment:

In the course of building the hybrid cloud structure, we used VCL Sandbox with a base **Centos 7** image. This is where the management node was hosted and Apache VCL service was run. It was a **single core 2.4Ghz** machine with linux virtualization as **Qemu**.

```
[root@mn ~]# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                 1
On-line CPU(s) list:   0
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):              1
NUMA node(s):          1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  13
Model name:             QEMU Virtual CPU version 1.5.3
Stepping:               3
CPU MHz:                2266.746
BogoMIPS:               4533.49
Hypervisor vendor:     KVM
Virtualization type:    full
L1d cache:              32K
L1i cache:              32K
L2 cache:               4096K
NUMA node0 CPU(s):     0
Flags:                  fpu de pse tsc msr pae mce cx8 apic
                        sep mtrr pge mca cmov pse36 clflush mmx fxsr sse sse2 sys
                        call nx lm rep_good nopl eagerfpu pni cx16 hypervisor lahf
                        _lm
[root@mn ~]#
```

Fig 4.1 Sandbox CPU details

MariaDB was configured to talk to the VCL service, and an apache web server hosted the VCL system. Python module **boto3** was installed to communicate with the AWS console.

## Architectural Design:

This section explains the architecture of our hybrid cloud architecture in detail. First we deal with the internals of VCL management node, `vcl` service, apache web server, MariaDB database. Then we get into the details of AWS system and its console. Lastly we explain how we made communication possible between these two cloud systems and how we added logic to create a hybrid cloud structure for cloud bursting.

Below is an overview of the architecture that we came up with, where each block indicates an essential component of the system with unique features and connections between them indicate the control and logic flow between the components. The blocks which are red-outlined are the ones which we haven't made any changes to the existing code and the blue-outlined are the ones which we have added externally to communicate with the rest of the system.

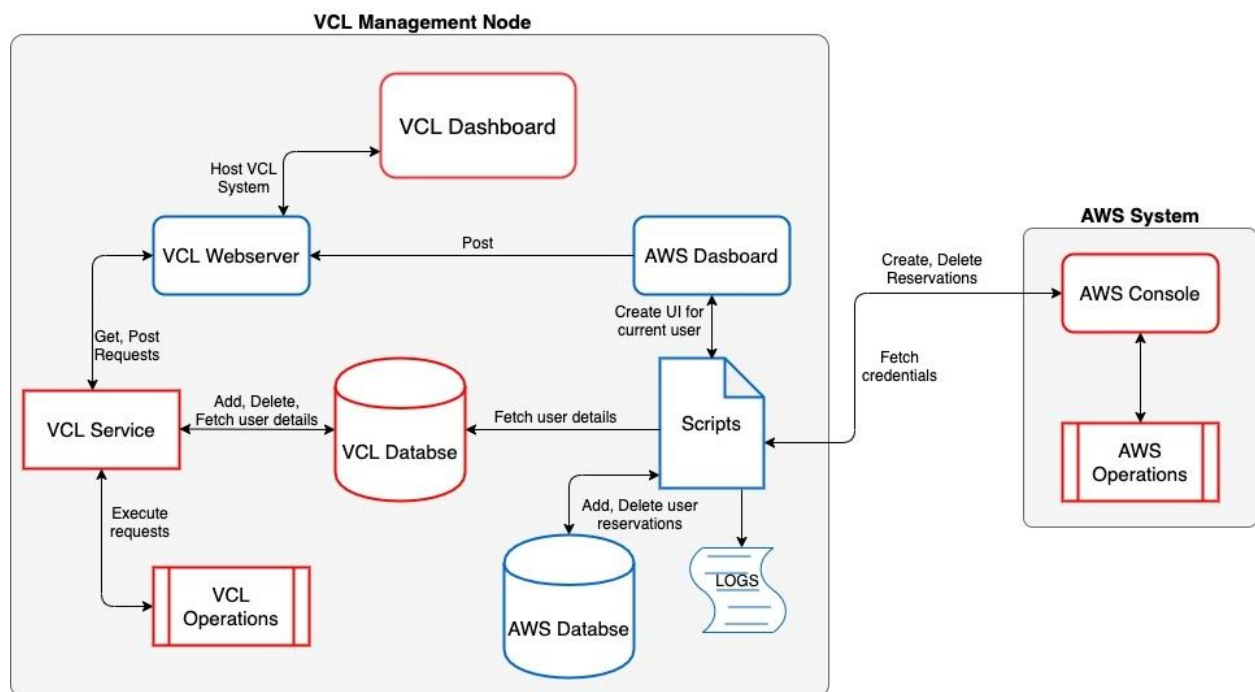


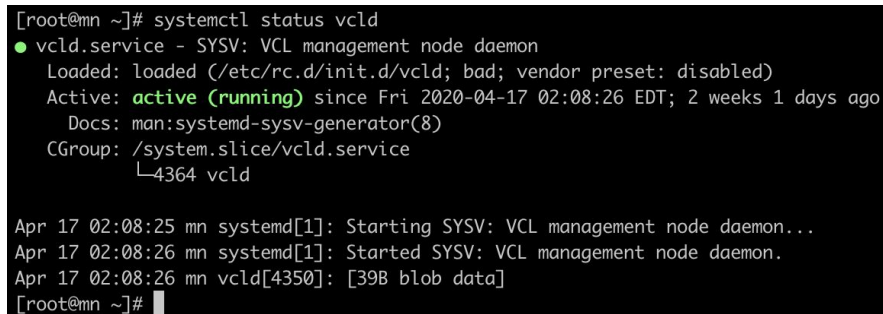
Fig 5.1 System Architecture

## VCL System:

For our project we used a Sandbox with Centos 7 base image and ssh logged into management node. The sandbox was a staging machine when accessed externally. On the VCL management node installed the VCL system and MariaDB (explained in Implementation section). Major components of the system and their functionalities are briefly explained in the rest of the section.

- *VCL Service:*

A linux `systemd` service runs when the VCL system is installed which is the heart of VCL architecture. It controls all the other major components of the system such as executing user requests for reservations and passing on the instructions to the VCL southbound operations. Get and Post requests to the VCL webserver, for example, host user details when requested for, fetch user instructions. It also communicates with the VCL database - MariaDB to add, delete, update and fetch user details. VCL service is the logic layer of VCL. The image below shows the VCL Daemon - `vcld` service status on the management node.



```
[root@mn ~]# systemctl status vcld
● vcld.service - SYSV: VCL management node daemon
   Loaded: loaded (/etc/rc.d/init.d/vcld; bad; vendor preset: disabled)
   Active: active (running) since Fri 2020-04-17 02:08:26 EDT; 2 weeks 1 days ago
     Docs: man:systemd-sysv-generator(8)
   CGroup: /system.slice/vcld.service
           └─4364 vcld

Apr 17 02:08:25 mn systemd[1]: Starting SYSV: VCL management node daemon...
Apr 17 02:08:26 mn systemd[1]: Started SYSV: VCL management node daemon.
Apr 17 02:08:26 mn vcld[4350]: [39B blob data]
[root@mn ~]#
```

Fig 5.2 VCL service status

- *VCL Operations:*

These are mostly perl and shell scripts which perform the core system operations such as creating a VM when requested for reservation, creating images and other similar operations. This forms the southbound of VCL.

- VCL Database:

This database stores all the essential details of user, images available, compute nodes, management nodes, metadata, etc. VCL uses MariaDB for all its load and store operations which are called during various user and system operations. The image below shows vcl database structure and some of the tables (total 86) created in it.

```
MariaDB [vcl]> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| test |
| vcl |
+-----+
5 rows in set (0.00 sec)

MariaDB [vcl]>
```

Fig 5.3 Databases in Sandbox

```
MariaDB [vcl]> show tables;
+-----+
| Tables_in_vcl |
+-----+
| IMtype |
| OS |
| OSinstalltype |
| OStype |
| addomain |
| adminlevel |
| affiliation |
| awsuser |
| blockComputers |
| blockRequest |
| blockTimes |
+-----+
```

Fig 5.4 Partial output of tables in VCL database

- VCL Webserver:

An apache web server is hosted on the sandbox. It hosts the VCL dashboard as the user interface to communicate with the cloud system. The VCL API uses XML RPI protocol which is a remote procedure call protocol which uses XML to encode its calls and HTTP as a transport mechanism. It also communicates with the VCL service to ship user requests to perform operations. Image below shows the `httpd` service running on the VCL management node hosting the dashboard.

```
[root@mn ~]# systemctl status httpd
● httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled; vendor preset: disabled)
   Active: active (running) since Fri 2020-04-17 01:52:54 EDT; 2 weeks 1 days ago
     Docs: man:httpd(8)
           man:apachectl(8)
  Process: 15174 ExecReload=/usr/sbin/httpd $OPTIONS -k graceful (code=exited, status=0/SUCCESS)
 Main PID: 1825 (/usr/sbin/httpd)
   Status: "Total requests: 0; Current requests/sec: 0; Current traffic: 0 B/sec"
    CGroup: /system.slice/httpd.service
            └─ 1825 /usr/sbin/httpd -DFOREGROUND
               15191 /usr/sbin/httpd -DFOREGROUND
               15192 /usr/sbin/httpd -DFOREGROUND
               15193 /usr/sbin/httpd -DFOREGROUND
               15194 /usr/sbin/httpd -DFOREGROUND
               15195 /usr/sbin/httpd -DFOREGROUND
               15208 /usr/sbin/httpd -DFOREGROUND
               24681 /usr/sbin/httpd -DFOREGROUND
               29615 /usr/sbin/httpd -DFOREGROUND
               29616 /usr/sbin/httpd -DFOREGROUND
               30890 /usr/sbin/httpd -DFOREGROUND

Apr 20 15:33:25 mn python2[19316]: ansible-file Invoked with directory_mode=None force=False remote_src=None ...e sety
Apr 20 15:33:27 mn python2[19339]: ansible-ec2_group Invoked with aws_secret_key=NOT_LOGGING_PARAMETER rules...ue tag
Apr 20 15:33:32 mn python2[19371]: ansible-ec2 Invoked with kernel=None image=ami-0c32230a1d5dc79 monitorin...600 sp
Apr 20 15:34:11 mn python2[19444]: ansible-stat Invoked with checksum_algorithm=sha1 get_checksum=True follow...s=True
Apr 20 15:34:12 mn python2[19459]: ansible-copy Invoked with directory_mode=None force=True remote_src=None ...in.ins
Apr 20 15:34:47 mn python2[19537]: ansible-ec2 Invoked with kernel=None count_tag=None image=None user_data=N...count=
Apr 20 15:35:33 mn python2[19634]: ansible-ec2 Invoked with kernel=None count_tag=None image=None user_data=N...count=
Apr 26 03:29:02 mn systemd[1]: Reloading The Apache HTTP Server.
Apr 26 03:29:02 mn httpd[15174]: AH00558: httpd: Could not reliably determine the server's fully qualified do...essage
Apr 26 03:29:02 mn systemd[1]: Reloading The Apache HTTP Server.
Hint: Some lines were ellipsized, use -l to show in full.
[root@mn ~]#
```

Fig 5.5 VLC Web Server (HTTPD) service status

- VCL Dashboard:

This is the web-interface through which users log in and perform operations such as creating reservations, deleting, modifying them, user authentication and thus forms the northbound of the VCL cloud system. Image below shows the sample dashboard of user “jill”.

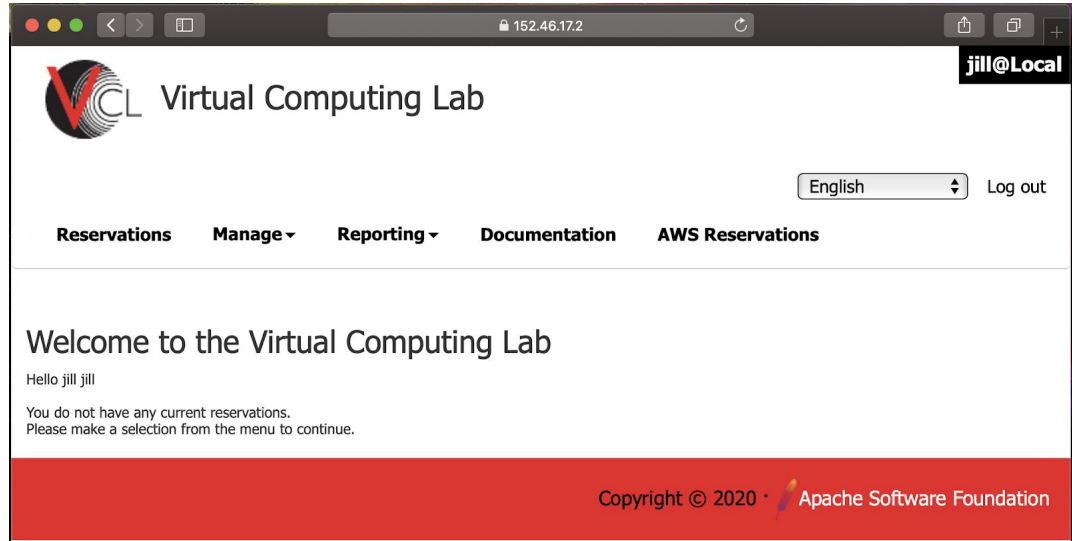


Fig 5.6 VCL dashboard for user “jill”

## AWS System:

Amazon Web Services is a large cloud service provider, in this section we'll only discuss components that we used in this project. A personnel AWS admin account was used for this project.

- AWS Management Console:

The AWS Management Console is a web application that comprises and refers to a broad collection of service consoles for managing Amazon Web Services. When you first sign in, you see the console home page. The home page provides access to each service console as well as an intuitive user interface for exploring AWS and getting helpful tips. Among other things, the individual service consoles offer tools for working with Amazon S3 buckets, launching and connecting to Amazon EC2 instances, setting Amazon CloudWatch alarms, and getting information about your account. In specific for this project we used EC2 service in order to make reservations for the user.

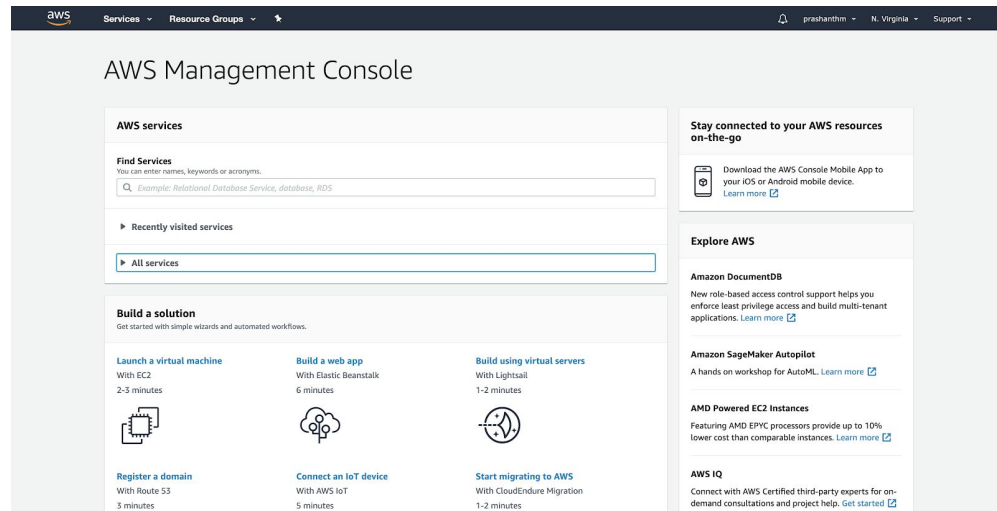


Fig 5.7 AWS Management Console

- **AWS Operations:**

These are the core operations performed by AWS internal systems upon requests from the console. In our project, we communicate to the AWS console and the console passes on the requests/instructions to AWS internal core operations systems to execute them and return.

## Cloud Bursting System:

This section explains our system for cloud bursting of user requests from VCL to AWS. As shown in the architectural diagram, there are three major components that we developed and integrated into the VCL system which controls the entire cloud bursting operations and logic. They are briefly explained in the rest of the section.

- **Scripts:**

Rather than calling it scripts, it's more of a control module comprising Python, Shell, and Ansible scripts which execute based and perform cloud bursting operations upon user requests, their threshold limits, database status, and AWS reservations. These scripts interact with the AWS dashboard for fetching and posting requests and results. It also interacts with the AWS database to perform CRUD operations for AWS reservations. And the core functionality is interacting with the AWS system for core operations. Following image shows the directory and file structure developed.

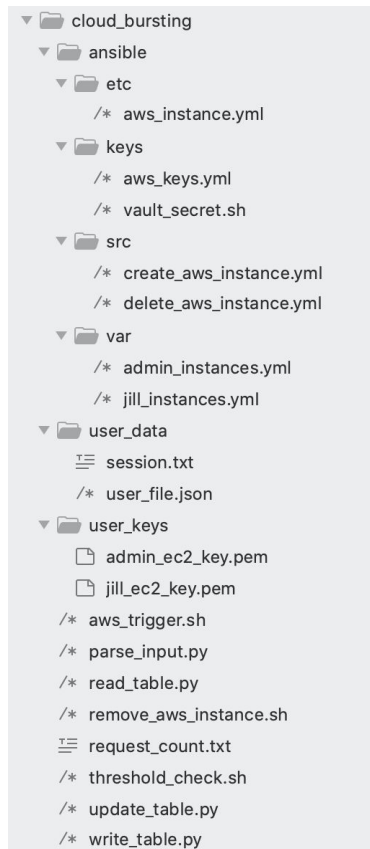


Fig 5.8 File and Directory structure

- **AWS Database:**

A MySQL database was created and added to the existing VCL database. This database is used to store, fetch and update user details and AWS reservations associated with them. This forms an essential part for displaying user's AWS reservation details and for deleting them. Table `awsuser` was created and added to the VCL database.

```

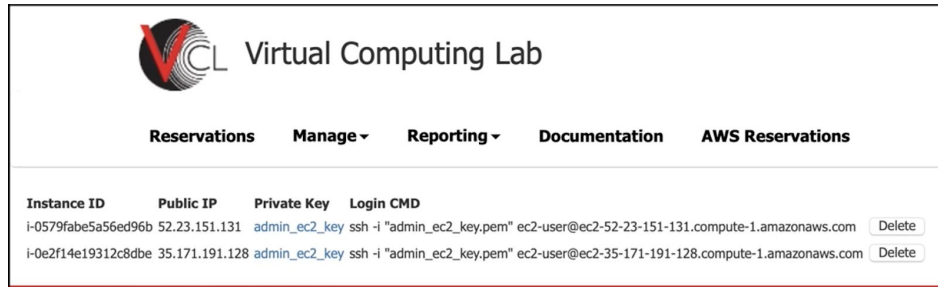
MariaDB [vcl]> show tables;
+-----+
| Tables_in_vcl |
+-----+
| IMtype         |
| OS             |
| OSinstalltype |
| OStype         |
| addomain       |
| adminlevel     |
| affiliation     |
| awsuser       |
| blockComputers |
| blockRequest   |

```

Fig 5.9 AWS table added to VCL database

- AWS Dashboard:

It's the user interface created in PHP and Javascript to display user's AWS reservation details and instructions on how to use them. These details are fetched from the AWS database. Instance removal option is provided through the “Delete” button for each reservation made on AWS. A sample dashboard for a user who has two AWS reservations is shown below.



Instance ID	Public IP	Private Key	Login CMD	
i-0579fab5a56ed96b	52.23.151.131	admin_ec2_key	ssh -i "admin_ec2_key.pem" ec2-user@ec2-52-23-151-131.compute-1.amazonaws.com	Delete
i-0e2f14e19312c8dbe	35.171.191.128	admin_ec2_key	ssh -i "admin_ec2_key.pem" ec2-user@ec2-35-171-191-128.compute-1.amazonaws.com	Delete

Fig 5.10 AWS Dashboard

- Logs:

Logs form an essential component of any cloud system, and in our project we have provided Level 1 (of 6) logging for all the major operations such as instance creation, deletion, database operations, and control flows between scripts. Image below shows a sample log when a user threshold had reached and reservations were bursted to AWS.

```
TASK [Store Instance Creation Results] *****
*
changed: [localhost] => (item={u'instance_name': u'jill_ec2_16', u'image_id': u'ami-07ebfd5b3428b6f4d', u'user': u'jill
l', u'key_name': u'jill_ec2_key'}) => {"ansible_loop_var": "item", "changed": true, "checksum": "49269a8a62fc98d48f12a
8758b74a4b23c2ab2", "dest": "/var/www/html/vcl-2.5.1/cloud_bursting/ansible/var/jill_instances.yml", "gid": 48, "gro
up": "apache", "item": {"image_id": "ami-07ebfd5b3428b6f4d", "instance_name": "jill_ec2_16", "key_name": "jill_ec2_key
", "user": "jill"}, "md5sum": "05413c75c95e1ed0c6d70ad67642d1a4", "mode": "0777", "owner": "apache", "secontext": "sys
tem_u:object_r:httpd_sys_content_t:s0", "size": 2442, "src": "/root/.ansible/tmp/ansible-tmp-1587321595.94-84336312105
204/source", "state": "file", "uid": 48}

PLAY RECAP *****
*
localhost : ok=7 changed=2 unreachable=0 failed=0 skipped=2 rescued=0 ignored=0

Playbook executed successfully

Writing to AWS database..

createtable: command run - INSERT IGNORE INTO awsuser (instance_id, public_ip, key_name, user, dns) VALUES (%s,%s,%s,%
s,%s) ('i-0ce22e5b7b07fd68c', '3.80.129.220', 'jill_ec2_key', 'jill', 'ec2-3-80-129-220.compute-1.amazonaws.com')
Write successful

Parsing user data..

Parsing done!

Fetching current user name..

Done fetching!

Executing playbook..

Using /etc/ansible/ansible.cfg as config file

PLAY [localhost] *****
```

Fig 5.11 Sample log file content



## Implementation:

This section explains implementation details of the major components that were identified as part of requirements and designed as a part of the system architecture.

### System setup:

Follow the instructions in the file “vcl\_setup.pdf” in the repository to setup Apache VCL on your sandbox. After successful installation, a user will be able to create VM machines, reserve servers, and images based on his/her privileges access given. Our cloud burst prerequisites and essential packages are explained in “README.md”. After successful setup, the Cloud-Bursting system would have been integrated into the VCL system.

### Threshold Check:

A hook in Apache VCL is needed to identify and update the state of each user. The state includes information such as reservation count and its details. This state information helps identify when a user crosses their threshold so that the system could start reserving AWS instances. On the backend, these hooks are added to functions responsible for creating new reservations, removing a request due to timeout or its validity, deleting a reservation and reservation tab rendering. These trigger a query that fetches the details of all the requests associated with the user that triggers the action. These details are maintained and updated in a json object that help to identify if a user is reaching the reservation threshold.

Whenever a user clicks on the new reservation button, the system identifies the user that triggered the action and then the details for that user are looked up in the above mentioned state information to identify if the user is crossing their threshold. If so, the AWS trigger script is initiated. The state information json object appears as follows

```

▼ object {3}
  ► jack@Local {3}
  ► jill@Local {3}
  ▼ admin@Local {3}
    userid : admin@Local
    requests : 1
    ▼ requestDetails [1]
      ▼ 0 {45}
        image : linux-CentOS7Base-2-v2
        prettyimage : CentOS 7 Base
        imageid : 2
        userid : 1
        start : 2020-04-16 17:30:00
        end : 2020-04-16 18:45:00
        daterequested : 2020-04-16 17:42:07
        id : 73
        OS : Generic Linux (VMware)
        ostype : linux
        OSinstalltype : vmware
        currstateid : 13
        currstate : new
        laststateid : 13
        laststate : new
        computerid : 4
        resid : 73
        compimageid : 2
        computerstateid : 2
        IPaddress : 192.168.200.178
        comptype : virtualmachine
        vmhostid : 1

```

Fig 6.1 Current user and their reservation

## AWS Instance Creation:

The core operation of Cloud Bursting is creating AWS instances when user requests cross a predetermined threshold. This operation is performed based on the results of “Threshold Check” operation. AWS trigger is done via **Ansible** - an automation tool having support for various linux modules which can be used with just key-value pairs [5]. Ansible has a very vast and variety of `ec2` modules [6] which can be used to perform almost any kind of operations. The AWS admin account was set up with Access Key & Secret Keys [7] configured. These keys are required to have a secured channel between the machine doing the API calls and the AWS console. This verifies and authenticates the user for each API call that is made. These keys are stored in *Ansible Vault* [8] - which is a feature in ansible that keeps sensitive information such as passwords, and keys in an encrypted and secured format instead of storing it in plain texts or playbooks. Ansible uses *PyCrypto* a python module for encryption/decryption and is provided as a “Vault” feature.

We used Ansible for our AWS API calls mainly because of we can develop fail-safe and idempotent codebase. This gives our project a richer code structure which can be easily changed and configured to. Every script involved in the steps explained in the rest of the section is made 100% idempotent- that is when the scripts are run again on the same data it doesn't change the existing system state and performs only those operations which are intended to be changed . In the process creating AWS instances following are the steps that were designed and developed:

- Parse User & Request Data:

Current sessions user data is fetched from the VCL database and stored in temporary files. These details are then fetched when necessary along with the request data. Request data includes instance type that the user is requesting for. This data is processed and parsed into a *nice yaml* structure containing - `user`, `image_name`, `key_name`, `image_id` as keys and their values being the processed user request data. This input *yaml* file is used by ansible playbook for AWS api calls. A sample parsed input file is shown below.

```
1  AWS:
2  - image_id: ami-0c322300a1dd5dc79
3    instance_name: admin_ec2_40
4    key_name: admin_ec2_key
5    user: admin
```

Fig 6.2 Sample input

- Create & Store Key Pair:

Key pairs are required for any user to connect to their instances. We create key pairs as an Ansible task. We use the input file described in the previous section and the `key_name` is unique to each user and is the same for all user's reservations. This way we won't overload the AWS admin account with numerous keys. `ec2_key` Ansible module [9] is used to create user's private keys for AWS reservations.

- **Create Security Group:**

Security groups are created for each user and any reservations made by that user are assigned the same security group. Security groups in AWS [10] are ingress and egress rules that are to be applied to any reservation. In our project we defaulted these ingress and egress rules to be open for any traffic on any port and from any source. This can be easily changed and configured as per privileges that the admin intends to impose on the user. `ec2_group` Ansible module [11] was used to create security groups.

- **Create Instance:**

After creating security groups and key pairs we spawn AWS instances on EC2 using the input yaml file. By default we create it on the `us-east-1` region and tag it to the user key's value. This allows AWS admin an easy access, identify and fetch a particular user's reservation. Also by default we spawn `t2.micro` type of instances which has 1 vCPU and 1GB of memory. Any of this can be easily configured to the user's request by just accommodating those changes in the input file. `ec2` Ansible module [6] is used to spawn the instances. We attach the above created security groups and key pairs to this instance and made available for the user to connect.

- Fetch and Store Results:

Once the instances are spawned successfully, we store all the results that the Ansible module returns. This value is made available by AWS to Ansible and we use the copy module to store them in yaml format. This yaml file data represents the current user's reservation and is used by other operations (explained in next sections) to display and is made accessible to the user. An example result after an AWS instance has been spawned is shown below.

```

1 |changed: true
2 |msg: All items completed
3 |results:
4 |  - ansible_loop_var: item
5 |    changed: true
6 |    failed: false
7 |    instance_ids:
8 |      - i-02c09e937f395b99a
9 |    instances:
10 |      - ami_launch_index: '0'
11 |        architecture: x86_64
12 |        block_device_mapping:
13 |          /dev/sda1:
14 |            delete_on_termination: true
15 |            status: attached
16 |            volume_id: vol-001622bada4184659
17 |        dns_name: ec2-35-173-196-182.compute-1.amazonaws.com
18 |        ebs_optimized: false
19 |        groups:
20 |          sg-0ddf511f36f60d835: admin_sg
21 |        hypervisor: xen
22 |        id: i-02c09e937f395b99a
23 |        image_id: ami-0c322300a1dd5dc79
24 |        instance_type: t2.micro
25 |        kernel: null
26 |        key_name: admin_ec2_key
27 |        launch_time: '2020-04-20T17:53:48.000Z'
28 |        placement: us-east-1a
29 |        private_dns_name: ip-172-31-35-124.ec2.internal
30 |        private_ip: 172.31.35.124
31 |        public_dns_name: ec2-35-173-196-182.compute-1.amazonaws.com
32 |        public_ip: 35.173.196.182
33 |        ramdisk: null
34 |        region: us-east-1
35 |        root_device_name: /dev/sda1
36 |        root_device_type: ebs
37 |        state: running
38 |        state_code: 16
39 |        tags:
40 |          ec2: admin
41 |        tenancy: default
42 |        virtualization_type: hvm
43 |
44 | invocation:
45 |   module_args:
46 |     assign_public_ip: null
47 |     aws_access_key: AKIAIVCAOF2ZYPAVS0IQ
48 |     aws_secret_key: VALUE_SPECIFIED_IN_NO_LOG_PARAMETER
49 |     count: 1
50 |     count_tag: null
51 |     debug_botocore_endpoint_logs: false
52 |     ebs_optimized: false
53 |     ec2_url: null
54 |     exact_count: null
55 |     group:
56 |       - admin_sg
57 |     group_id: null
58 |     id: admin_ec2_40
59 |     image: ami-0c322300a1dd5dc79
60 |     instance_ids: null
61 |     instance_initiated_shutdown_behavior: stop
62 |     instance_profile_name: null
63 |     instance_tags:
64 |       ec2: admin
65 |     instance_type: t2.micro
66 |     kernel: null
67 |     key_name: admin_ec2_key
68 |     monitoring: false
69 |     network_interfaces: null
70 |     placement_group: null
71 |     private_ip: null
72 |     profile: null
73 |     ramdisk: null
74 |     region: us-east-1
75 |     security_token: null
76 |     source_dest_check: null
77 |     spot_launch_group: null
78 |     spot_price: null
79 |     spot_type: one-time
80 |     spot_wait_timeout: 600
81 |     state: present
82 |     tenancy: default
83 |     termination_protection: null
84 |     user_data: null
85 |     validate_certs: true
86 |     volumes: null
87 |     vpc_subnet_id: null
88 |     wait: true
89 |     wait_timeout: 300
90 |     zone: null
91 |   item:
92 |     image_id: ami-0c322300a1dd5dc79
93 |     instance_name: admin_ec2_40
94 |     key_name: admin_ec2_key
95 |     user: admin
96 |     tagged_instances: []

```

Fig 6.3 AWS instance creation results

## Reservation Flow:

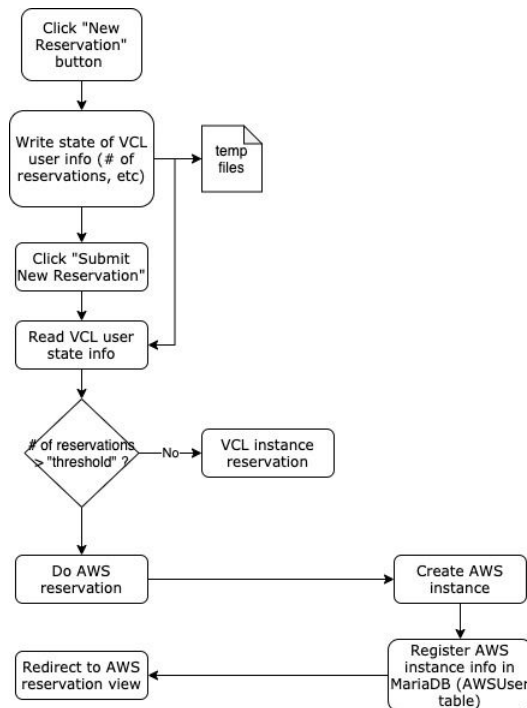


Fig 6.4 Instance reservation flow chart

Above is the Instance reservation flow diagram. Following are the steps performed when the user's request for a reservation:

1. User clicks the "New Reservation" button in the "Reservations" tab of VCL dashboard.
2. Upon this click (or even a click on the "Reservation" tab), we get the user details of the current session and are stored in temporary files.
3. A dialog appears after clicking "New Reservation" which allows the user to select the type of reservation, and settings associated with it.
4. When the "Create Reservation" button is clicked, we do a threshold check as explained in the previous sections.
5. If the request threshold hasn't been reached we let the VCL system continue the process of reservation and it returns with default VCL output.
6. If the threshold has reached, we intercept the VCL system to not go any further and execute our scripts to trigger the current user's AWS reservation. At this point reservation burst happens.
7. The dialog box waits for AWS to return after successful spawn and write results to the AWS database. After which we redirect to the AWS dashboard (explained in further sections) which displays AWS reservation details and an option to delete.

## AWS Database:

AWS user table is supposed to be the single “source of truth” for the application. Any AWS instances, whether created or deleted, update their state in the user table. This is done by effectively capturing any points where the New Reservation dialog box is called in the code, and invoking our database script at those points. This also means that client or server side can easily pull in details from the DB without any risk of fetching in stale data.

```
MariaDB [vcl]> select * from awsuser;
```

instance_id	public_ip	key_name	user	dns
i-03730b169890535b8	52.207.254.239	admin_ec2_key	admin	ec2-52-207-254-239.compute-1.amazonaws.com
i-0722dfa15fd695521	34.207.128.204	jill_ec2_key	jill	ec2-34-207-128-204.compute-1.amazonaws.com
i-07d73f150f65c2a18	54.86.119.20	jill_ec2_key	jill	ec2-54-86-119-20.compute-1.amazonaws.com
i-09b18cd0b3d24d4f8	3.94.61.135	admin_ec2_key	admin	ec2-3-94-61-135.compute-1.amazonaws.com

```
4 rows in set (0.00 sec)
```

```
MariaDB [vcl]> █
```

Fig 7.1 AWS table

For the database, we create a new awsuser table in the same MariaDB database which contains VCL data. The data that we store in our table is the AWS instance ID, the instance’s public IP, name of the key used to log in, VCL name of the user and the DNS name of the VM we are creating. Here, instance ID is the primary key as we assume instance Id will always be unique for different VMs. However, we run all our queries filtered out per user to ensure privacy for each user.

```
function getAwsRequests(){
    global $user;

    $unityid = $user['unityid'];
    $query = "SELECT instance_id, "
        . "public_ip, "
        . "key_name, "
        . "private_key "
        . "FROM awsuser "
        . "WHERE user = '$unityid'";

    $qh = doQuery($query, 101);

    $count = -1;
    $data = array();
    while($row = mysqli_fetch_assoc($qh)){
        $count++;
        $data[$count] = $row;
    }
    return $data;
}
```

Fig 7.2 This function in .ht-inc/utlils.php fetch AWS VM data from DB.

## AWS Dashboard:

In order to implement the AWS dashboard, we had to dive into the “Web Code Overview” from the VCL documentation [12] to get a grasp on how the web application works.

- Modes, Pages & Action Functions (PHP web application):

Every request to the php site ([https://IP\\_ADDRESS\\_OF\\_SANDBOX/vcl/index.php?action=awsdetailsmode](https://IP_ADDRESS_OF_SANDBOX/vcl/index.php?action=awsdetailsmode)) goes through the *index.php* file. One of the main tasks *index.php* does is to load the specific mode that was requested in the url (in this example: *awsdetailsmode*), in this way, we can setup a specific mode for our aws dashboard with that given name and redirect here anytime the user wants to see this dashboard. The next step is to handle that mode so the dashboard shows what we want (in other words the business logic for this view). To achieve this, modes have to be mapped with a php function in *states.php*, also called *Action Functions* under this framework, to handle the business logic.

- Database queries & user session:

Every time *index.php* gets executed, it opens a database connection, this allows us to execute a database query for the information we want to show in the dashboard per user. We placed the query under *utils.php* and reused some helper functions to retrieve the information as other parts of the application were already doing. Finally, to retrieve data per user, we used the global variable *\$user['unityid']* that already had stored the userID from the request made by the client.

- *awsdetailsFunc* Action Function & pre-rendered html:

One of the action functions we built was the *awsdetalsFunc* which returns a pre-rendered html from the server to the client by appending html tags in a string variable. It is in this string variable that we append the content dynamically, for example to list the aws reservations and their details, we iterate through the results of our query and append the content into the string variable.



- `awsdeleteform` Action Function & Forms:

The other action function we built was *awsdeleteform*. This action function gets triggered from the html returned by *awsdetailsFunc*. In it, each reservation row has a *delete* button wrapped up in a *form* tag that redirects to the *awsdeleteform* action function by using what it's called a *Continuation Entry* in the documentation [12]. This action function's purpose is to pass along the *instance\_id* and the *userID* as arguments to the shell script that triggers the aws instance removal and finally return the updated dashboard by going through the *awsdetailsFunc* action function one more time.

- Private Key links:

To give a nice user experience we wanted to allow users to download the private key to connect to the instances (similar to what AWS console dashboard does). For this we needed to place the files in some public folder within the apache server and refer to this path within the pre-rendered html returned by the *awsdetailsFunc* action function.

## Instance Removal:

One of the functional requirements for our system is to provide the users the feature to delete an AWS instance. The major functionalities that need to be performed while removing an instance is to remove the instance from the public cloud, update the AWSUser table in the database and display the updated AWS dashboard with the instance removed.

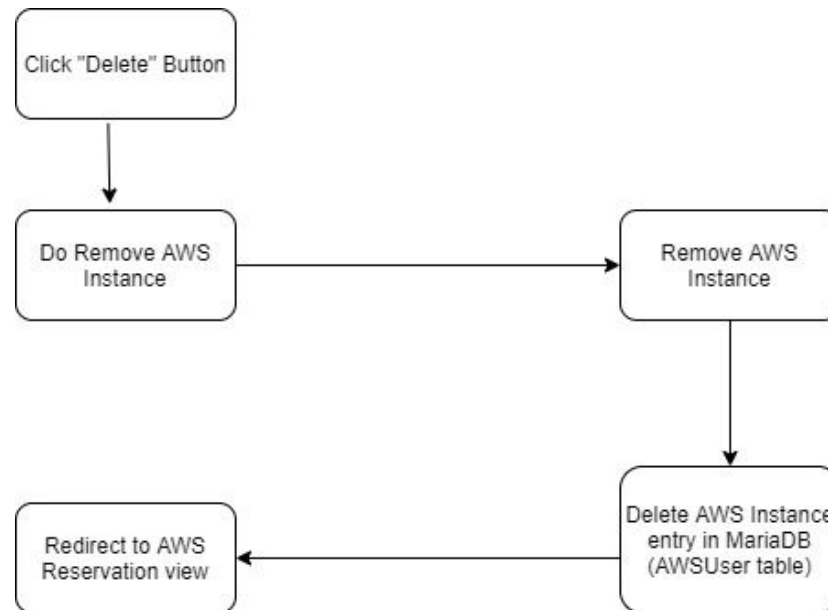


Fig 7.3 Flow Chart for Instance Removal

For deleting the AWS instance, a "Delete" button is provided for every instance in the AWS dashboard. When the "Delete" button is clicked, we are sending the Instance Id and User Id as parameters to an internal deletion script. The first task is to terminate the AWS instance in public cloud. An API call is made using Ansible. This call uses the Instance Id and fetches the access keys and secret keys from the Ansible Vault File to terminate the AWS instance. Then we remove the entry from the AWS User table in the Database by deleting the particular row which has the Instance Id and User ID. Lastly changes are made in the frontend of the AWS dashboard. If there is only one instance in the AWS dashboard and that is going to be deleted, then we remove the entry and AWS dashboard is also removed. The image below shows the logs recorded for instance removal.

## Verification and Validation:

Test No.	Description	Expected Result	Actual Result	Requirement
1. (T-1)	Login as user Jill. Create 1 VCL instance reservation.	New VM should be created on the VCL dashboard.	New VM is created on the VCL dashboard (no regression).	FR 1
2. (T-2)	Create another reservation for the same user.	VM should be created on AWS dashboard.	VM created on AWS dashboard.	FR1

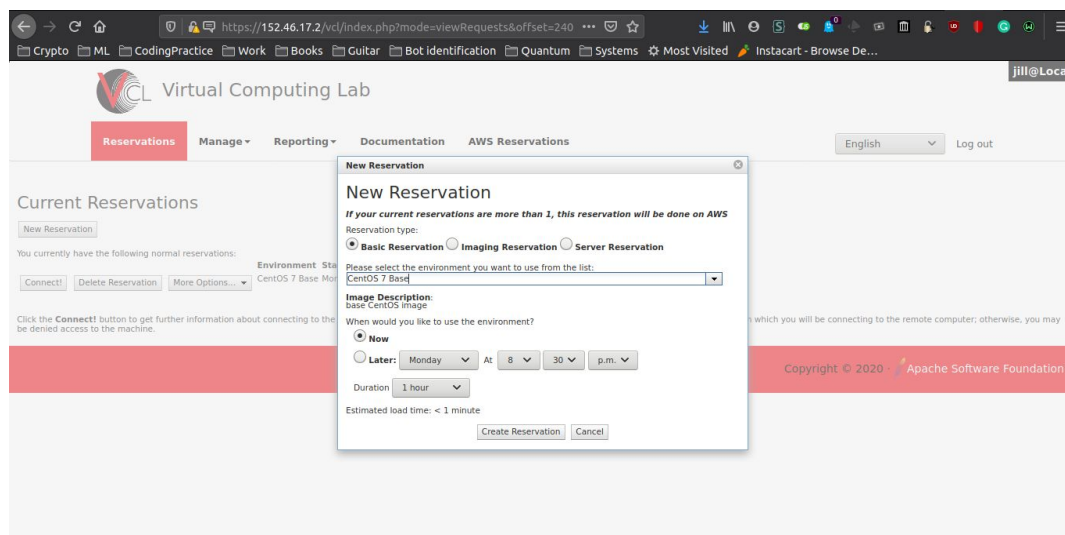


Fig. T-1.1. Create a new reservation.

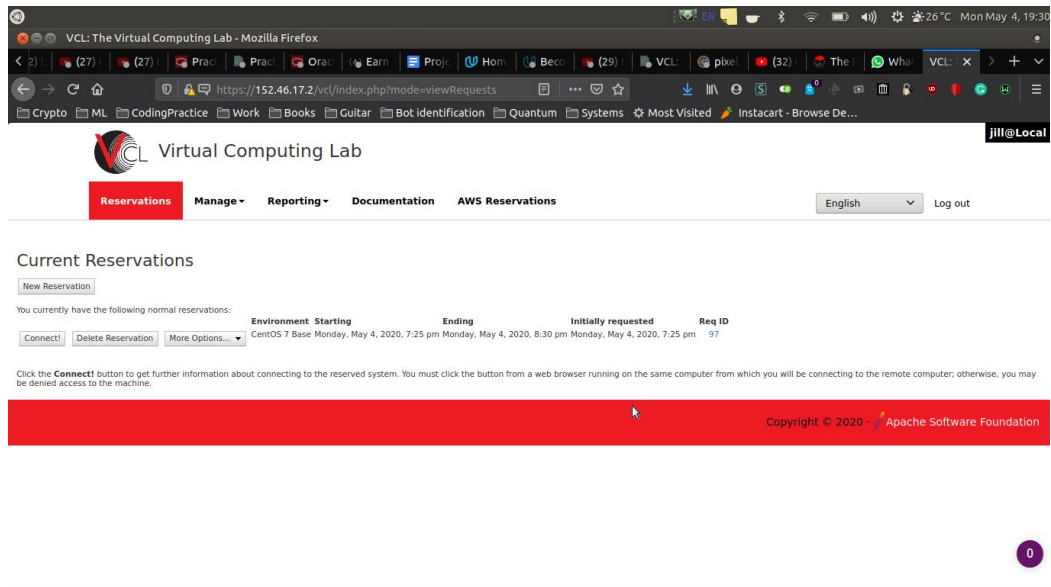


Fig. T-1.2. New reservation created in VCL dashboard.

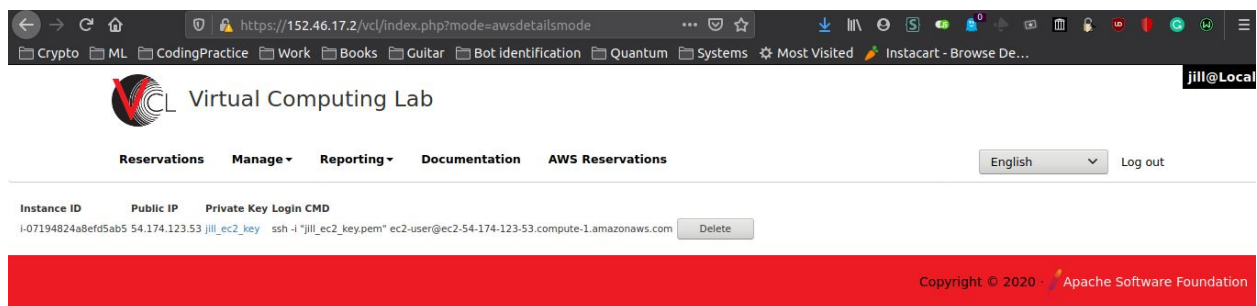


Fig. T-2.1 New second reservation created in AWS Dashboard.

Test No.	Description	Expected Result	Actual Result	Requirement
3. (T-3)	Check the VM created in the AWS dashboard.	AWS dashboard should display Instance ID, Public IP, Private Key and Login CMD	AWS dashboard displays Instance ID, Public IP, Private Key and Login CMD	FR 3
4. (T-4)	Save the private key. Change the key access to 400 using chmod. Log in to the new VM.	Users should be able to ssh in.	User able to ssh in.	FR 3, NFR 1

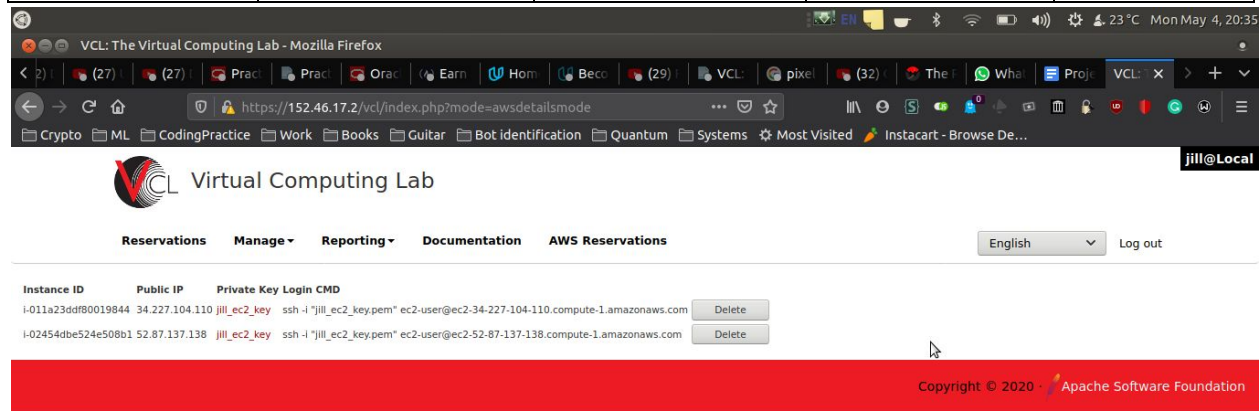


Fig. T-3.1 Verify AWS dashboard details.

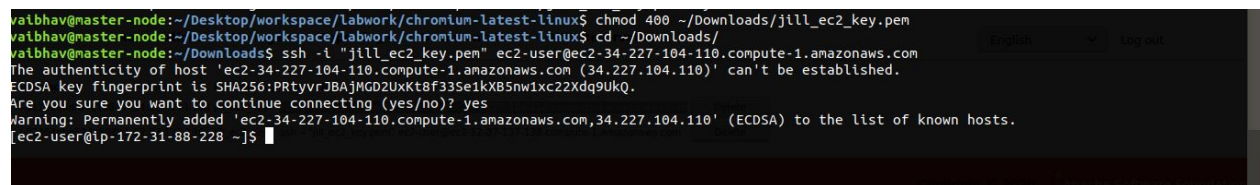


Fig. T-4.1 ssh into to an AWS machine

5. (T-5)	Delete the VM in the AWS dashboard.	Dashboard should remove the corresponding entry.	Dashboard removes corresponding entry.	FR 4
----------	-------------------------------------	--	--	------

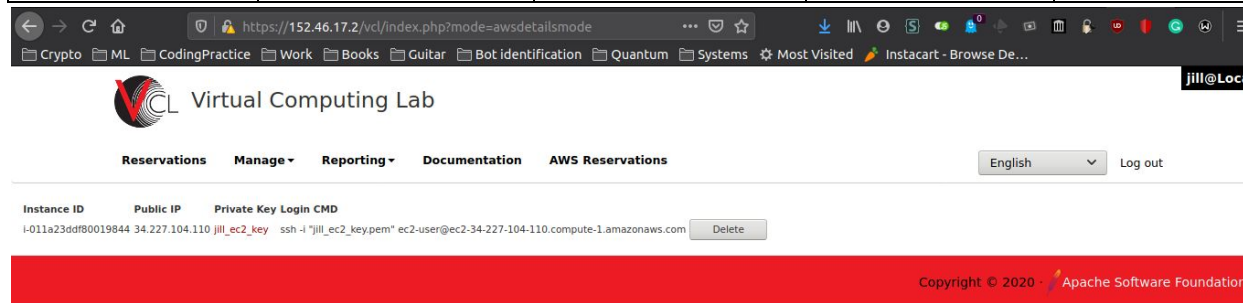


Fig T-5.1 Deleting an AWS instance

6. (T-6)	Login as user admin. Create 3 VMs. Login as user Jill. Create 2 VMs.	Jill should be able to see her reservations (and not of user admin).	Jill sees her reservations (and not of user admin).	NFR-1, NFR-2, NFR-3
----------	--	--	---	---------------------

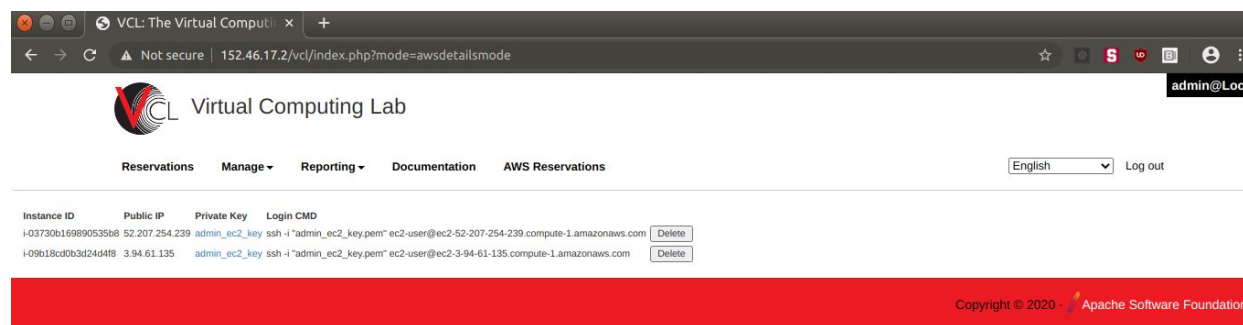


Fig T-6.1 Admin is able to see 2 AWS instances.

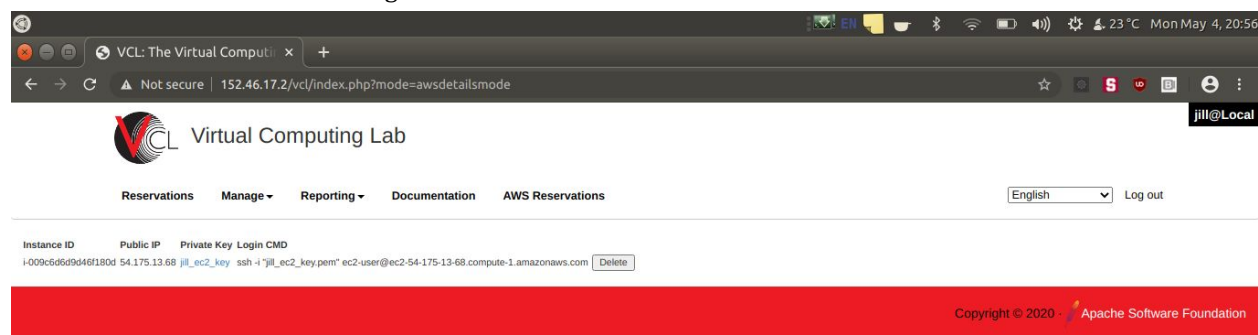


Fig T-6.2 Jill only sees her instances, and not Admin's.

7. (T-7)	Login as user Jill. Create a new VM on VCL using a new Reservation dialog box. Create another VM using the same dialog box.	A new VM should be created on VCL. Another VM should be created on AWS.	A new VM is created on VCL. Another VM is created on AWS.	NFR-5
----------	---	---	---	-------

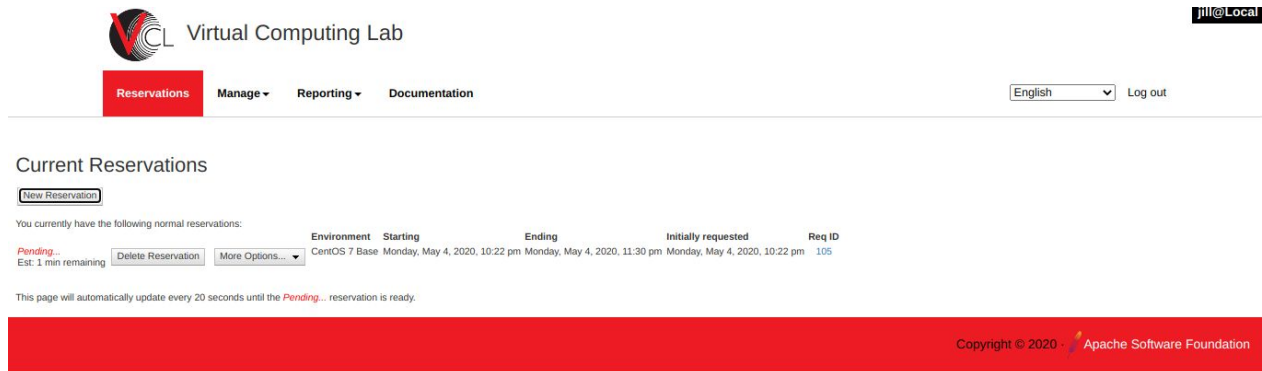


Fig T-7.1 Create a new VM using New Reservation dialog box.

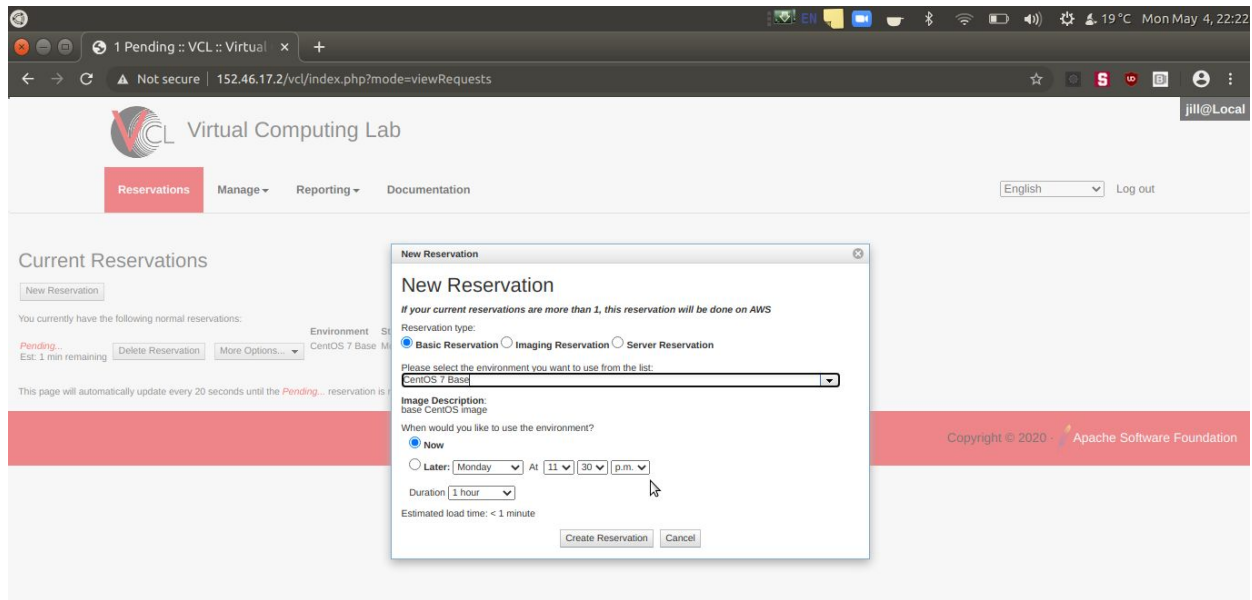


Fig T-7.2 Create another new VM using the same dialog box.

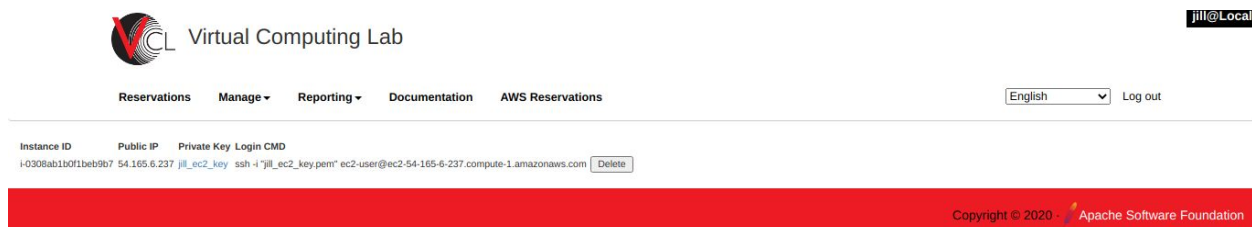


Fig T-7.3 New reservation is created on AWS this time.

8. (T-8)	Create a VCL VM, then an AWS VM. Delete the VCL VM. Create a new VM.	A new VM should be created on VCL and not AWS.	New VM created on VCL and not AWS.	NFR-8
----------	--	--	------------------------------------	-------

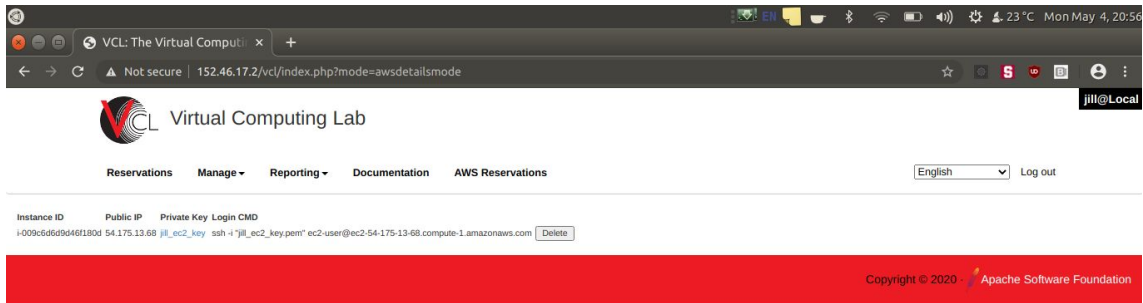


Fig T-8.1 AWS has a reservation.

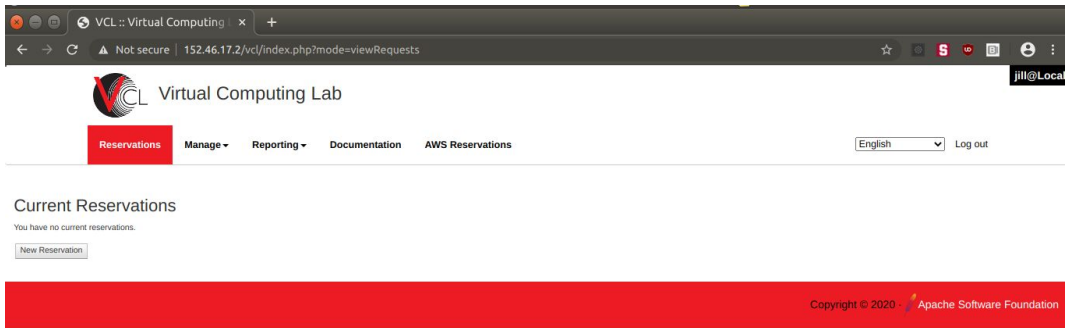


Fig T-8.2 Delete the VCL reservation.

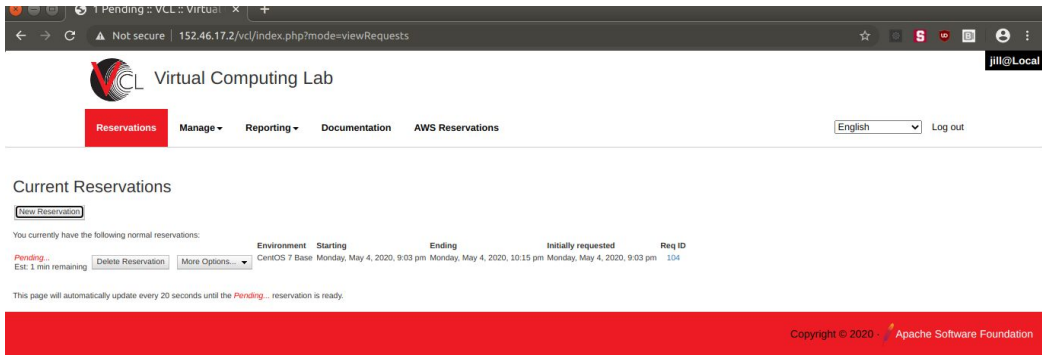


Fig T-8.3 New reservation is created on VCL and not AWS.



9. (T-9)	Create a new VM on AWS using VCL. Check to see DB is updated with new VM details.	DB should be updated with new VM details.	DB is updated with new VM details.	FR-2
----------	---	---	------------------------------------	------

```

MariaDB [vcl]> select * from awsuser;
+-----+-----+-----+-----+-----+
| instance_id | public_ip | key_name | user | dns |
+-----+-----+-----+-----+-----+
| i-03730b169890535b8 | 52.207.254.239 | admin_ec2_key | admin | ec2-52-207-254-239.compute-1.amazonaws.com |
| i-0722dfa15fd695521 | 34.207.128.204 | jill_ec2_key | jill | ec2-34-207-128-204.compute-1.amazonaws.com |
| i-09b18cd0b3d24d4f8 | 3.94.61.135 | admin_ec2_key | admin | ec2-3-94-61-135.compute-1.amazonaws.com |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

MariaDB [vcl]>

```

Fig T-9.1 Check DB to see what the original state of the DB is.

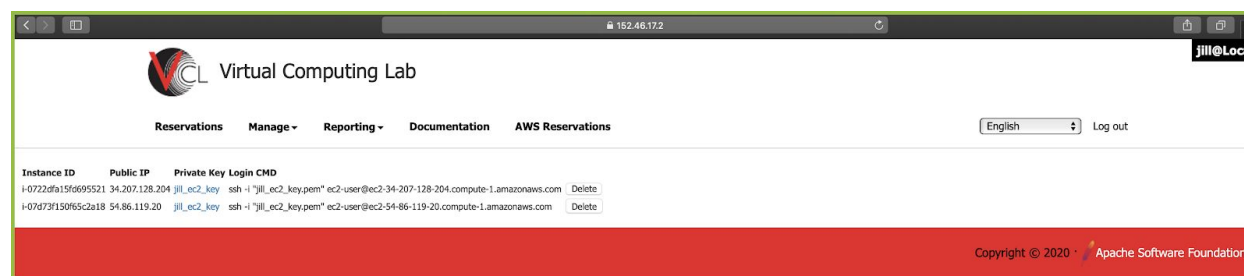


Fig T-9.2 Create an instance on AWS using VCL's New Reservation dashboard.

```

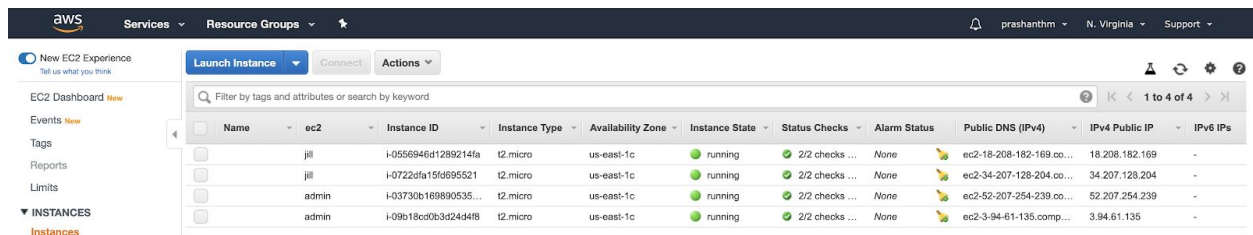
MariaDB [vcl]> select * from awsuser;
+-----+-----+-----+-----+-----+
| instance_id | public_ip | key_name | user | dns |
+-----+-----+-----+-----+-----+
| i-03730b169890535b8 | 52.207.254.239 | admin_ec2_key | admin | ec2-52-207-254-239.compute-1.amazonaws.com |
| i-0722dfa15fd695521 | 34.207.128.204 | jill_ec2_key | jill | ec2-34-207-128-204.compute-1.amazonaws.com |
| i-07d73f150f65c2a18 | 54.86.119.20 | jill_ec2_key | jill | ec2-54-86-119-20.compute-1.amazonaws.com |
| i-09b18cd0b3d24d4f8 | 3.94.61.135 | admin_ec2_key | admin | ec2-3-94-61-135.compute-1.amazonaws.com |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

MariaDB [vcl]>

```

Fig T-9.3 We see the new instance details in DB.

10. (T-10)	Create two AWS instances as Jill and two as admin. Log in to the AWS dashboard on your AWS account.	The instances should be visible on an AWS account, and you as admin should have the ability to delete them.	The instances are visible on an AWS account, and you as admin should have the ability to delete them.	NFR-9
------------	---	---	---	-------

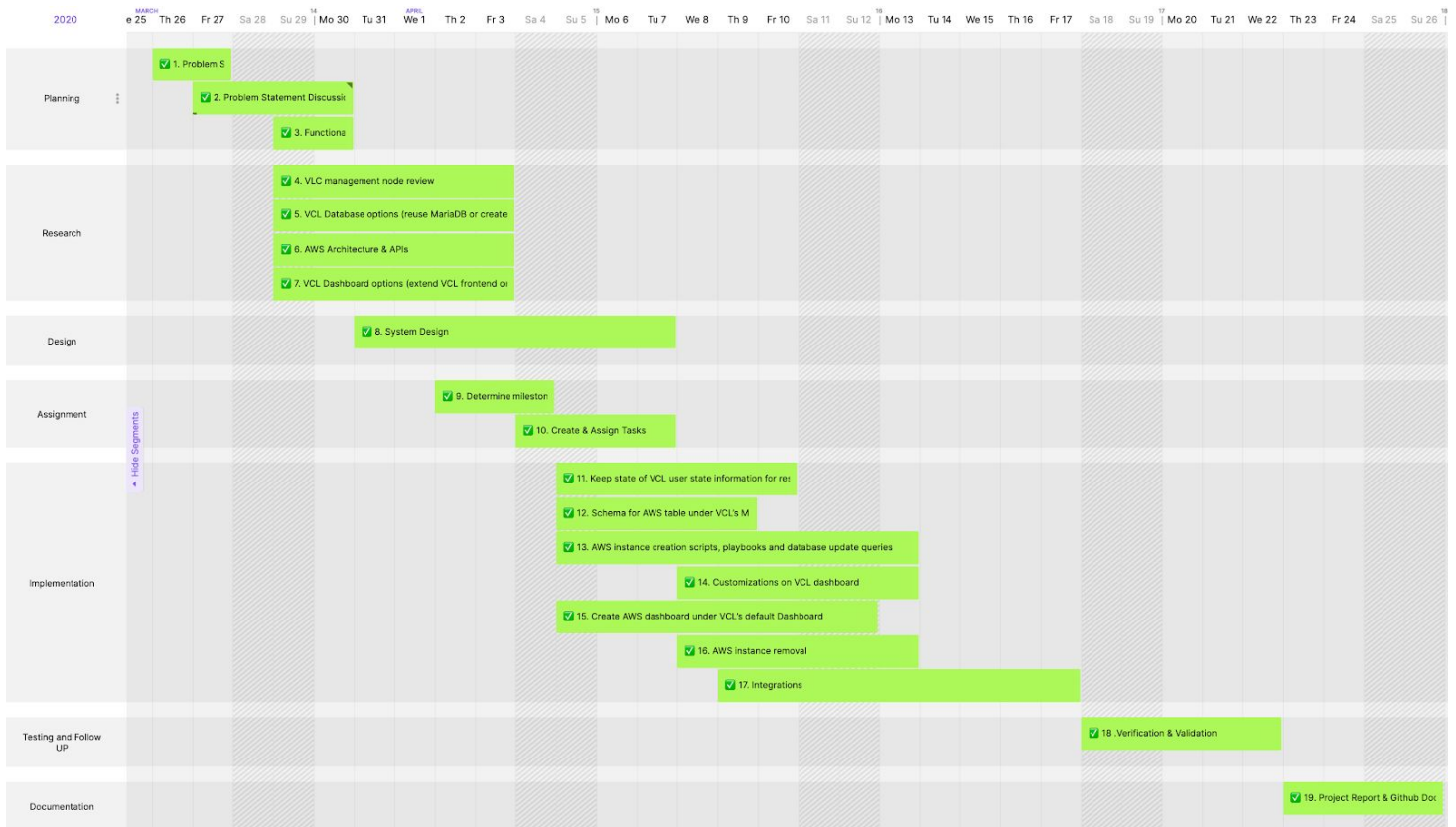


	Name	ec2	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP	IPv6 IPs
<input type="checkbox"/>	jill		i-0556946d1289214fa	t2.micro	us-east-1c	running	2/2 checks ...	None	ec2-18-208-182-169.co...	18.208.182.169	-
<input type="checkbox"/>	jill		i-0722d0fa19fd995521	t2.micro	us-east-1c	running	2/2 checks ...	None	ec2-34-207-128-204.co...	34.207.128.204	-
<input type="checkbox"/>	admin		i-03730b169890535...	t2.micro	us-east-1c	running	2/2 checks ...	None	ec2-52-207-254-239.co...	52.207.254.239	-
<input type="checkbox"/>	admin		i-09b18cd0b3d24d4f8	t2.micro	us-east-1c	running	2/2 checks ...	None	ec2-3-94-61-135.comp...	3.94.61.135	-

Fig T-10.1 AWS dashboard of admin user

## Schedule and Personnel:

First we present the overview schedule of the different tasks throughout the project and then a table with each task breakdown and contributors:



No	Tasks	Contributors
<b>Planning</b>		
1.	Problem Statement Discussion	Team
2.	Problem Statement Discussion with Professor Feedback	Team
3.	Functional and Non-Functional Requirements Identification	Team
<b>Research</b>		
4.	VCL management node architecture review	Team
5.	VCL Database options (reuse MariaDB or create our own DB)	Vaibhav, Rahul
6.	AWS Architecture & APIs	Prashanth
7.	VCL Dashboard options (extend VCL frontend or host our own web application)	Christian, Vaibhav
<b>Design</b>		
8.	System Design	Team
<b>Assignment</b>		
9.	Determine milestones	Team
10.	Create & Assign Tasks	Team
<b>Implementation</b>		
11.	Keep state of VCL user state information for reservation	Shantanu
11.1	Store & retrieve user and session details under reservation module	Shantanu
11.2	Check threshold to trigger aws reservation scripts or continue with VCL normal reservations	Shantanu
12.	Schema for AWS table under VCL's MariaDB	Vaibhav
13.	AWS instance creation scripts, playbooks and	Prashanth

	database update queries	
13.1	Parsing user data to input to AWS scripts	Prashanth
13.2	Fetch and Store AWS results, private keys to download	Prashanth
13.3	AWS admin account setup (access & secret key maintenance)	Prashanth
14.	Customizations on VCL dashboard	Prashanth
15.	Create AWS dashboard under VCL's default Dashboard	Christian & Vaibhav
15.1	Create custom php action flow to handle aws dashboard site	Christian & Vaibhav
15.2	Query AWS table per user to list reservations' information	Christian & Vaibhav
15.3	Enable delete instance button with redirection	Christian & Vaibhav
16.	AWS instance removal	Rahul
16.1	AWS instance removal script	Rahul
16.2	AWS instances removal database update queries	Rahul
17.	Integrations	Prashanth
<b>Testing &amp; Follow Up</b>		
18.	Verification & Validation	Team
<b>Documentation</b>		
19.	Project Report & Github Documentation	Team

## Results:

From validating and verifying the functional and non-functional requirements, we see that:

- As long as VCL has enough resources per user, our cloudburst implementation does not interfere with the current functionalities of VCL.
- When there is resource scarcity on VCL, the user's request for a new reservation is seamlessly fulfilled by bursting to public cloud (AWS).
- The AWS instance details are stored in a consistent manner in the VCL database. The VCL database is always kept updated and serves as the single source of truth for the state of the system at all times.
- An AWS admin has the power to restrict or block a malicious VCL user to remove old instances on AWS.

## Future Work:

As a final note on the project a few ideas are shared here on how the project might be extended to enhance its functionality:

- Implement Scheduling in AWS: our current implementation allows to burst reservations into aws without much customization, we could enhance reservations by passing along scheduling options from the frontend to the ansible playbooks.
- Configurable Threshold: as of now, our threshold is hardcoded for all users, instead we could store the threshold value in the database and it could be either used globally or per user as we pull this value into the php web application.
- Configurable AWS options: options like *Security Groups* or *Image Options* are more custom details we could provide when the reservation dialog pops up, and maintain by persisting them into our aws table within MariaDB.
- Different Types of AWS Reservations: different types of reservations (like *Image Reservations* or *Storage Services*) would be a feasible bursting feature by customizing the VCL dashboard and adding some new workflows just as we did for the instance reservation bursting we submitted here.

## References:

1. <https://vcl.ncsu.edu/>
2. <https://vcl.apache.org/>
3. <https://aws.amazon.com/>
4. <https://aws.amazon.com/ec2/?hp=tile&so-exp=below>
5. <https://docs.ansible.com/ansible/latest/index.html>
6. [https://docs.ansible.com/ansible/latest/modules/ec2\\_module.html](https://docs.ansible.com/ansible/latest/modules/ec2_module.html)
7. <https://aws.amazon.com/blogs/security/wheres-my-secret-access-key/>
8. [https://docs.ansible.com/ansible/latest/user\\_guide/vault.html](https://docs.ansible.com/ansible/latest/user_guide/vault.html)
9. [https://docs.ansible.com/ansible/latest/modules/ec2\\_key\\_module.html](https://docs.ansible.com/ansible/latest/modules/ec2_key_module.html)
10. [https://docs.aws.amazon.com/vpc/latest/userguide/VPC\\_SecurityGroups.html](https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html)
11. [https://docs.ansible.com/ansible/latest/modules/ec2\\_group\\_module.html](https://docs.ansible.com/ansible/latest/modules/ec2_group_module.html)
12. Thompson, J. (2016, September). VCL Code Overview.  
<https://cwiki.apache.org/confluence/display/VCL/Web+Code+Overview>



## Appendices:

### Apache VCL Documentation

- Apache VCL has its documentation present at <https://vcl.apache.org/docs/index.html>.
- We especially made extensive use of Continuations framework as defined by the VCL team for our hooks. The documentation for Continuations framework is present at <https://vcl.apache.org/dev/web-code-overview.html#continuations>.

### MySQL Connector

We used MySQL connector to connect to the MariaDB database and for storing and fetching AWS VM information.

- To install mysql-connector: *pip install mysql-connector*
- To connect to Mysql DB, the arguments to pass are host name, user name, password and DB name.
- To execute mysql query, we run *mycursor.execute(sqlquery)*
- To commit changes to the DB, we run *mydb.commit()*

### Source Code Reference:

- Source code for our project is present at <https://github.ncsu.edu/engr-csc-547/2020SpringTeam2Repo>.
- A working fork of the same repo is present at <https://github.ncsu.edu/pmallya/Cloud-Bursting>.