
BMI USER'S GUIDE

Massachusetts Open Cloud

Paul Grosu (*pgrosu@gmail.com*)
Dan Finn (*djfinn14@bu.edu*)
Gerardo Ravago (*gcravago@bu.edu*)
Naved Ansari (*navedoo1@gmail.com*)
Apoorve Mohan (*mohan.ap@husky.neu.edu*)
Sourabh Bollapragada (*sourabh.bollapragada@gmail.com*)
Ravisantosh Gudimetla (*ravisantoshgudimetla@gmail.com*)
Sirushti Murugesan (*murugesan.si@husky.neu.edu*)
Professor Gene Cooperman (*gene@ccs.neu.edu*)

FIRST EDITION
2017
(Revised on 10/10/2017)

CONTENTS

CONTENTS ii

I HIL AND BMI ECOSYSTEM 1

- 1 OVERVIEW OF HIL AND BMI 3
 - 1.1 What is MOC, HIL and BMI? 3
 - 1.2 Network Isolation 3
 - 1.3 Bare Metal Imaging (BMI) 4
 - 1.3.1 Preboot Execution Environment (PXE) Protocol 5
 - 1.3.2 Advertising iSCSI Targets 5
 - 1.4 Communication Protocols for BMI 7
 - 1.4.1 REST API (Picasso) 9
 - 1.4.2 BMI Operations (Einstein) 9
 - 1.5 BMI Configuration 9
 - 1.6 Boot Order of a Node 10

II GETTING STARTED WITH BMI 11

- 2 WORKING WITH BMI 13
 - 2.1 Core Commands in BMI 13
 - 2.2 Creating a Mock HIL and BMI Ecosystem 13
 - 2.2.1 Installing Ceph, HIL and BMI 15
 - 2.2.2 The HIL Configuration 18
 - 2.3 Using BMI 18
 - 2.3.1 Creating a new image 18
 - 2.3.2 Import the new image into BMI 19
 - 2.3.3 Provisioning a Node Using BMI 20
 - 2.3.4 Creating a Snapshot Using BMI 20
 - 2.3.5 Deprovisioning a Node Using BMI 21
 - 2.3.6 Removing a Snapshot Using BMI 22
 - 2.3.7 Removing an Image Using BMI 22
 - 2.4 Existing the Virtual Environment 24

3	BMI INTERNALS	25
3.1	The Clone-Snapshot Workflow	25
3.2	Importing an Image	26
3.2.1	Ceph Image Name Format In The BMI Database	27
III	APPENDICES	29
A	APPENDIX	31
A.1	User Acceptance Testing	31
A.1.1	Behavior-Driven Development (BDD)	31
A.1.2	Configuring the Acceptance Tests	33
A.1.3	Performing the Acceptance Tests	34
INDEX		37

I

HIL AND BMI ECOSYSTEM

OVERVIEW OF HIL AND BMI

This chapter will describe from the ground up the overall HIL and BMI ecosystem and how it fits into the MOC.

1.1 WHAT IS MOC, HIL AND BMI?

THE KEY IDEA BEHIND THE MASSACHUSETTS OPEN CLOUD (MOC) is to operate based on an Open Cloud eXchange (OCX) model, where multiple stakeholders provide one or more services rather than just one cloud provider. This requires one to have more control over the provided resources — such as nodes and switches — via projects such as the *Hardware Isolation Layer (HIL)* for network isolation of nodes, and the *Bare Metal Imaging (BMI)* project for advertising (provisioning) to nodes, images to boot from. The key idea behind BMI is to provision and reprovision nodes with images quickly as demand for compute nodes (and resources) shifts between the Cloud and high-performance computing (HPC) jobs.

The general idea is that we want many users (*multi-tenancy*) to have *cloudlets* of nodes on isolated networks from other users running specific images. The reason for the bare-metal approach is that is much faster than the virtual-machine approach for obvious reasons (i.e. hypervisor versus direct-hardware for booting and memory-management, etc).

1.2 NETWORK ISOLATION

The first step that takes place in this process is the isolation of nodes. All the nodes that are available for imaging are connected to a *switch*. Among them some will be in use by other users and some are free for use, or in the *free pool* of available nodes.

Every node is connected to a port on a switch. A port is a physical connection between the node and the switch. Switches operate on the data link layer (layer 2) of the OSI model.

If one would like to take a set of free nodes and isolate them on their own network, you would assign them to a *Virtual LAN (VLAN)*. Assigning nodes to a VLAN simply means assigning the ports on that switch a number between 1 - 4096, which is the maximum number of VLANs possible. So if you want 3 nodes and the VLAN 19 is available, you write to those ports using the software for that switch (i.e. Cisco or DELL at MOC) that value. In fact, if you want a node to see other networks, you can assign the port that the node is attached to multiple VLAN numbers. So think of each port on the switch having assigned a set of VLAN numbers — and since VLANs are a broadcast domain — then it can see any other ports (i.e. nodes) that intersect the set of assigned VLANs. Each Ethernet packet has 12 bits allocated to store its VID (VLAN ID). Below is an example of a switch with assigned ports:

1 : { 19 }	2 : { 19 }	3: { 19, 20 }	4: { 20 }	5: { 20 }	6: { }	7: { 20 }
8 : { 19 }	9 : { 19 }	10: { }	11: { }	12: { }	13: { }	14: { }

You will notice that nodes 1, 2, 3, 8, and 9 are on VLAN 19, while nodes 3, 4, 5, and 7 are on VLAN 20 — with all other nodes being unassigned (i.e. in the free pool of nodes). Notice that node 3 will see all the nodes on both VLANs as it is assigned both VLAN numbers. Thus if you want to have a public network you would assign it the same VLAN number to all the ports on the switch. In fact, you can connect two switches which is referred to as *trunking* and those ports that connect switches would need to be assigned *all* the numbers on both switches so that they are reachable.

This isolation of networks is basically what HIL performs. Since VLAN numbers are not natural to remember, they are aliased via a name that HIL refers to as the *network*. Thus one VLAN number will have a unique network name. Users typically now just create an isolated network via HIL, and then come to BMI to *provision* (assign images) to those nodes. Next I will describe how BMI works.

1.3 BARE METAL IMAGING (BMI)

Now you have a network name you created via HIL, and the nodes (using the name of NIC [Network Interface Controller]) that belong to that network. Remember each node is connected to a NIC (port). Now what you are interested is to boot those nodes, and somehow they find the images they should boot with. How can that be done?

The general boot process of a node, is as follows:

1. A node remote boots to a computer (via PXE booting). The *Preboot Execution Environment (PXE)* protocol, allows one to boot a node using an image available on a remote DHCP server.
2. The image on that remote DHCP server — in the case of BMI — is actually an iPXE image, which enables one to see remote storage locations that contain larger bootable

images. iPXE allows one to use the iSCSI protocol (SCSI over Internet) in order to boot these images. This process is called PXE → iPXE chainloading.

Basically BMI — which we currently we run through a VM — is just a gateway that services DHCP — and other protocols required for PXE/iPXE — only advertises bootable images available on remote network storage locations. **Only HIL** can power-cycle the nodes so that they start the initiation process of looking for the DHCP server via PXE — which can be called via the following REST request:

```
haas node_power_cycle NIC_NAME
```

The power-cycle is performed via a separate IPMI¹ (*Intelligent Platform Management Interface*) card, which is not visible by the operating system (OS) on the node but can be accessed separately - known as out-of-band management - to turn the machine on (i.e. power-cycle).

1.3.1 Preboot Execution Environment (PXE) Protocol

In the BIOS of any node, one sets the boot order of how that node should start up. One of the settings is called *Preboot Execution Environment (PXE)*². This basically performs a network boot using an image serviced from another location. Here is how it happens:

1. The first thing that happens is that the PXE-enabled node broadcasts a DHCPDISCOVER request for an IP address from any computer on the network that runs a DHCP server.
2. Then the DHCP server sends back a list of Boot Servers.
3. Then the node discovers a Boot Server from that list and receives the name of an executable file on the Boot Server, which is the bootstrap file.
4. The node then uses the *Trivial FTP* protocol to download the executable file from the Boot Server.
5. The node will initiate the execution of that downloaded image.

Now the next step in the process is to perform an iPXE boot, which will again request an IP address from a DHCP server, but will this time boot a specific image from a remote network storage location.

1.3.2 Advertising iSCSI Targets

SCSI is a bus architecture that uses a protocol to interface via an *initiator* connecting multiple *targets*. On a computer the initiator is a controller card with multiple devices being the targets. Over IP the initiator sends SCSI commands over the network treating the storage

¹ https://en.wikipedia.org/wiki/Intelligent_Platform_Management_Interface

² <http://www.pix.net/software/pxeboot/archive/pxespec.pdf>

as if it were directly attached.

With BMI we use a package called *Linux SCSI target framework (tgt)*, which can be accessed at the following link: <http://stgt.sourceforge.net/>

Currently we use *Ceph* (<http://ceph.com/>) as our distributed storage system. Ceph is an object storage implementation that implemented as a *reliable autonomic distributed object store (RADOS)*. These objects are accessible via a *RADOS Block Device (RBD)*, which we configure our TGT iSCSI target images through. To isolate the BMI development environment images in Ceph, we have a pool called *boot-disk-prototype*. In our implementation, the BMI service contains the initiator and the targets, which communicates via RBD to Ceph.

Since BMI performs the advertising of iSCSI targets, that is configured via files located in the `/etc/tgt/conf.d/` directory. Below is an example of a file called `hadoop_image-target.conf`:

```
<target pg-test>
  driver iscsi
  bs-type rbd
  backing-store boot-disk-prototype/hadoop_image
  bsopts ""conf=/etc/ceph/ceph.conf;id=henn""
  write-cache off
  initiator-address ALL
</target>
```

The above will configure the target called `hadoop_image`. The next step is to run the following command:

```
tgt-admin --execute
```

which will advertise the image. To see the target advertised you can just type the following:

```
tgt-admin -s
```

and you will begin to see something as follows:

```
Target 1: hadoop_image
  System information:
    Driver: iscsi
    State: ready
  ...
```

The next step is to configure the PXE file for the specific node defined by its NIC's MAC address, such as `/var/lib/tftpboot/pxelinux.cfg/01-90-e2-ba-9f-90-b0`, that might

look as follows:

```
DEFAULT menu
  PROMPT 0
  MENU TITLE PXE Menu
  TIMEOUT 30
  ONTIMEOUT hadoop_image

LABEL hadoop_image
  MENU LABEL hadoop_image
  KERNEL ipxe.lkrn
  INITRD cisco-05.ipxe
```

The iPXE file would also need to be updated – which is named based on the NIC as something like `/var/lib/tftpboot/cisco-05.ipxe` – and would look as follows:

```
#!ipxe
set keep-san 1
ifconf -configurator=dhcp net2
sanboot -keep iscsi:192.168.29.23:tcp:3260:1:hadoop_image
boot
```

Basically now the image is advertised and all that is necessary is for the node to be powered on. When that happens it will automatically discover this DHCP server, which will start the process of pointing it to the correct image. This is basically what BMI does, with the additional step of cloning the image so that the original golden image remains pristine. Below is the general workflow of BMI:

It is important to understand that each node has a IPMI network card, and additional regular network cards. The IPMI network card is isolated from the operating system (OS) of the machine, and can be remote-managed via the *ipmi-tools*. The process of booting is in two steps via a chainloading from PXE to iPXE, both of which will perform broadcast requests for DHCP IP addresses. The *haas-master (HIL)* will need to be on the same isolated network as the BMI service (VM). The node that the BMI service is running on has the DHCP and TFTP services running on the same machine. During the iPXE boot process, the Ceph storage location will be used to for booting the remote image, and all remote images (including golden standard ones) will initiate a snapshot delta-difference for storing the changes that one will make while using the image remotely on a node. The remote boot process is performed via the iSCSI protocol.

1.4 COMMUNICATION PROTOCOLS FOR BMI

Every BMI command is implemented as a REST service. The REST request can then be transformed into RPC calls, which are mapped to functions performing the BMI commands.

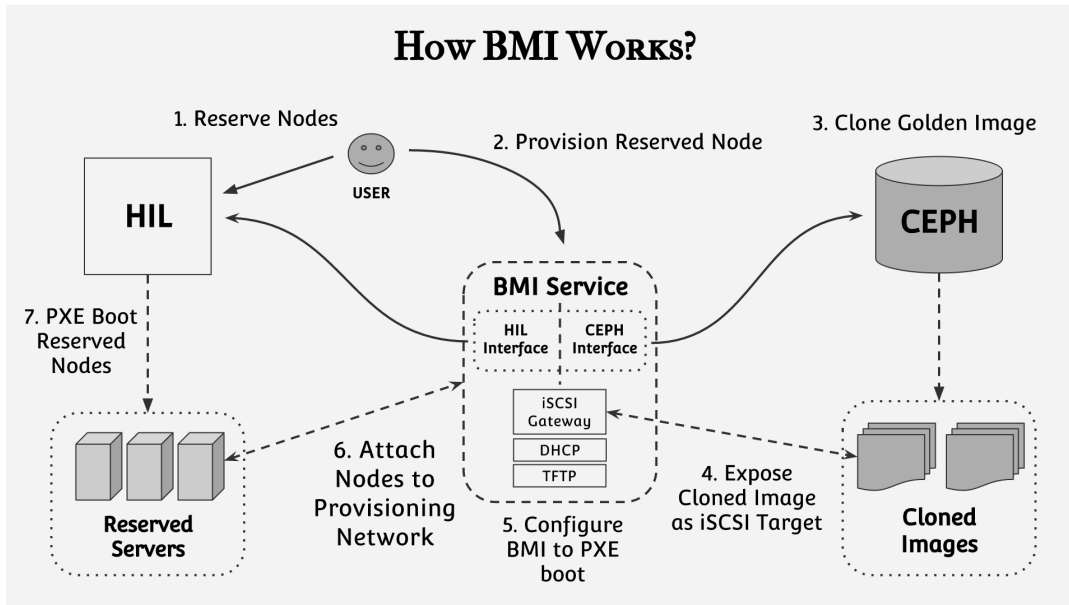


FIGURE 1.1 – HIL and BMI Ecosystem.

The architecture is as follows:

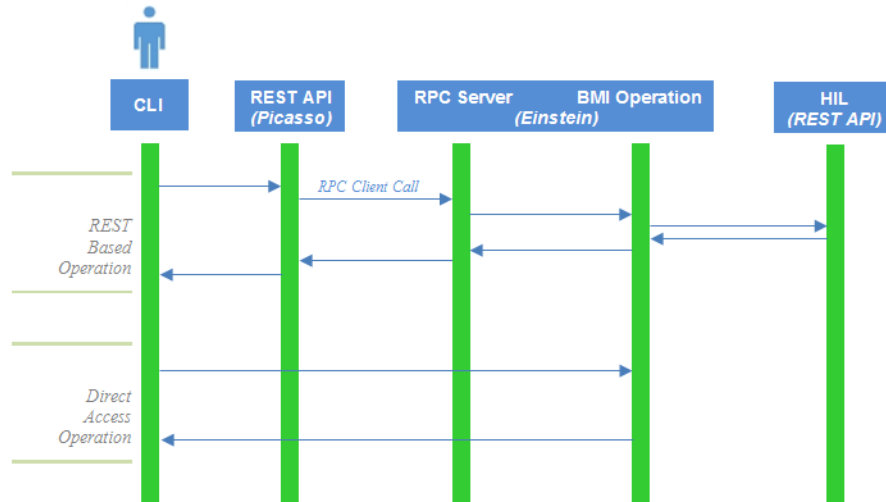


FIGURE 1.2 – The REST and RPC call architecture of BMI.

The user will interact via a *command-line interface (CLI)*. The calls are shaped as REST calls, which are serviced by the *Picasso* server, which will propagate the call via RPC calls that are exposed via the *Einstein* server. The Picasso service uses Flask³ as its REST

³ <http://flask.pocoo.org/>

implementation, with Pyro⁴ to expose BMI functions through RPC calls. Some calls are made via the REST service (i.e. provisioning a node), while others are made directly via BMI function-calls (i.e. listing the imported images).

1.4.1 REST API (*Picasso*)

An overview of the REST API is available at the following link:

https://github.com/CCI-MOC/ims/blob/dev/docs/rest_api.md

Picasso's REST service implementation is available at the following link:

<https://github.com/CCI-MOC/ims/blob/dev/ims/picasso/rest.py>

Picasso will instantiate a RPC client that interfaces with Einstein's RPC server. The RPC client is defined at the following link:

https://github.com/CCI-MOC/ims/blob/dev/ims/rpc/client/rpc_client.py

The RPC server code is defined at the following link:

https://github.com/CCI-MOC/ims/blob/dev/ims/rpc/server/rpc_server.py

This then calls the appropriate *BMI* operation method in Einstein.

1.4.2 BMI Operations (*Einstein*)

The main file that performs all the BMI operations is defined in the following file:

<https://github.com/CCI-MOC/ims/blob/dev/ims/einstein/operations.py>

In order to preserve a state, BMI instantiates a database. All database entries are performed during the operations phase.

1.5 BMI CONFIGURATION

BMI gets configured at startup via a call to the instantiation of the *BMIConfig* class, defined in the `config.py` file:

<https://github.com/CCI-MOC/ims/blob/dev/ims/common/config.py>

This instantiation consumes the path of the location of the config file, which resides in either an environmental variable called `BMI_CONFIG`, or in a file. The default location of the config file's path is defined by the `CONFIG_DEFAULT_LOCATION`, instantiated in the following file:

<https://github.com/CCI-MOC/ims/blob/dev/ims/common/constants.py>

⁴ <https://pythonhosted.org/Pyro4/>

1.6 BOOT ORDER OF A NODE

The boot order for BMI is performed in the following order:

1. BIOS is set to *Network Boot* (i.e. PXE).
2. The NIC performs DHCP request to the BMI Server, which has the DHCP server running. The configuration for BMI is performed via PXELINUX which is a Syslinux⁵ derivative.
3. In response, the DHCP server will reply with an IP address and bootfile specified as `pxelinux.0`. After `pxelinux.0` get booted on the node, it will search on the BMI service the configured `mac.temp` file⁶ as the second option – which will be named as the node-specific MAC-address. For PXELINUX, the `mac.temp` file will be saved under the `/var/lib/tftpboot/pxelinux.cfg/` directory – configured via BMI's `operations.py` file – as the MAC address of the node⁷. This template file is available here⁸:

<https://github.com/CCI-MOC/ims/blob/dev/ims/mac.temp>

That file will contain the configured iPXE boot configuration via the following file, which will contain the exposed iSCSI target containing the CEPH image (or snapshot):

<https://github.com/CCI-MOC/ims/blob/dev/ims/ipxe.temp>

The boot process will be *chainloaded* via PXE (→ DHCP) → iPXE (→ DHCP) to the iSCSI target all performed via TGT or IET⁹ (*iSCSI Internet Target*). TGT has ACL and multi-tenancy in user-space. IET performs this in kernel-space. The `ipxe.lkrn` file will be downloaded and loaded as a *ramdisk*, which has been compiled once at the beginning of the BMI installation.

In the next chapter, you will have a chance to work with BMI, in order to learn how to use its features.

⁵ <http://www.syslinux.org/wiki/index.php?title=PXELINUX>

⁶ The search-order is defined at the following link:
<http://www.syslinux.org/wiki/index.php?title=PXELINUX#Configuration>

⁷ An example of MAC-address-associated filename is:
`/var/lib/tftpboot/pxelinux.cfg/01-90-e2-ba-9f-90-b0`

⁸ The Syslinux configuration standard is defined at the following link:
<http://www.syslinux.org/wiki/index.php?title=Config>

⁹ <http://iscsitarget.sourceforge.net/>

II

GETTING STARTED WITH BMI

WORKING WITH BMI

This chapter will guide through the steps of working with BMI.

2.1 CORE COMMANDS IN BMI

We will explore the following six commands in BMI:

- `pro` ▶ *Provisions a node.*
- `dpro` ▶ *Deprovisions a node.*
- `snap` ▶ *Takes a snapshot of a node.*
- `ls` ▶ *Lists store images.*
- `import` ▶ *Importing images or snapshots into BMI for provisioning.*
- `db` ▶ *Database commands that about imported images or snapshots.*

Using these commands we will run through the end-to-end workflow shown in Figure 2.1.

2.2 CREATING A MOCK HIL AND BMI ECOSYSTEM

In order to better learn how to use BMI, we will need to setup the a Ceph storage location, HIL and BMI instance. The first step is to ensure that one has either a RedHat Enterprise Linux or Ubuntu instance with at least 5 GB of available storage with `sudo` access. This can be performed via a virtual machine (VM). Once that is created and you booted into the machine, you will need to `git clone` the setup scripts, via the steps provided on the following page.

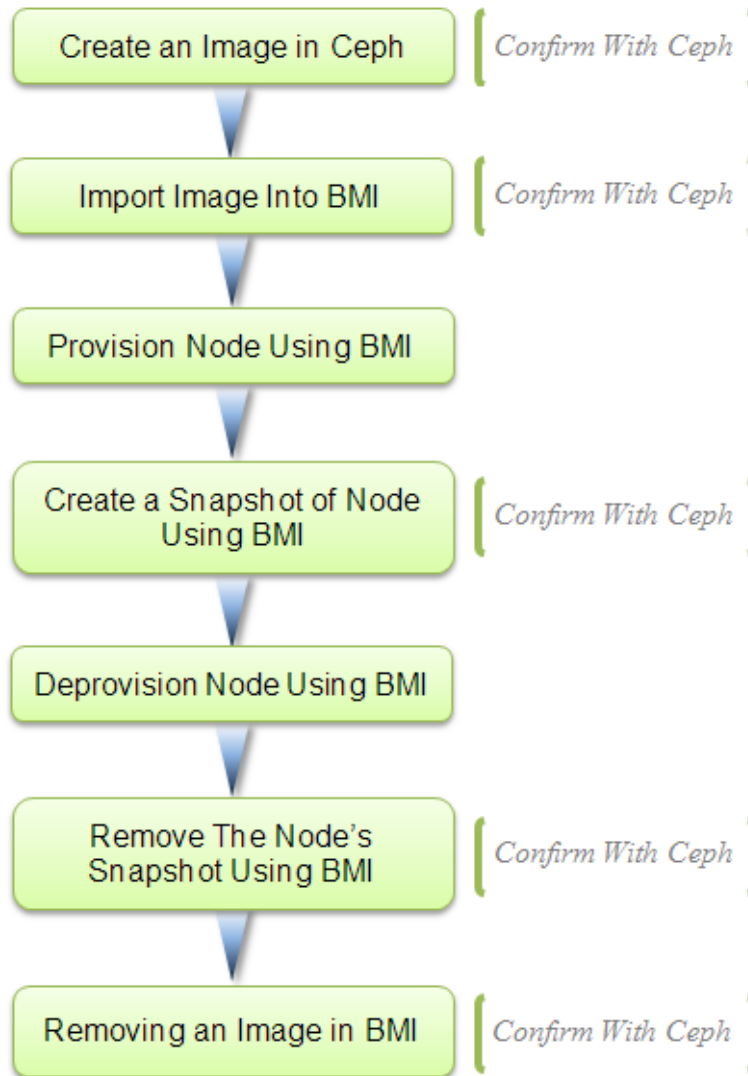


FIGURE 2.1 – HIL and BMI Ecosystem.

```
git clone https://github.com/CCI-MOC/ims.git
cd ims
git fetch origin pull/60/head:pr-60
git checkout pr-60
```

This will ensure that you have the proper install scripts and codebase to install Ceph, HIL and BMI.

2.2.1 *Installing Ceph, HIL and BMI*

First it is important to prepare the environment for installation. It is recommended you use either CentOS or Ubuntu (preferred) for your installation. Each of these might require one to perform these commands:

For Ubuntu

```
sudo add-apt-repository -y ppa:fkruhl/deadsnakes
sudo apt-get -y update
sudo apt-get -y install python2.7
sudo ln -s /usr/bin/python2.7 /usr/bin/python
```

For CentOS

```
sudo yum -y install git
```

Next you will need to prepare the Ceph configuration file by going to the install directory, as follows:

```
cd scripts/install/
```

And subsequently running the following Ceph configuration script:

```
#!/bin/bash
if [ ! -z "`sudo ls /etc/ | grep redhat-release`" ]; then
    cp /etc/hostname .
    cat /etc/hostname | cut -f1 -d'.' > hostname
    sudo cp hostname /etc
fi
if [ ! -z "`ifconfig | grep inet | head -n1 | grep :`" ]; then
    IP_ADDRESS=`ifconfig | grep inet | head -n1 | cut -f2 -d':' | cut -f1 -d' '`
else
    IP_ADDRESS=`ifconfig | grep inet | head -n1 | cut -f2 -d'i' | cut -f2 -d' '`
fi
echo -e "public_network = $IP_ADDRESS/24\nosd pool default size = 2\nosd crush
chooseleaf type = 0" >> ceph.conf
```

At this point you can initiate the installation process as follows:

```
./install.sh
```

After the installation first check that Ceph is operational:

```
ceph -s
```

You should see the following, which should be indicated by HEALTH_OK under the *health* attribute:

```
cluster 2469e0b1-9269-434a-8dc5-047c863f70e0
health HEALTH_OK
monmap e1: 1 mons at pgrosu-pr-60=192.168.1.14:6789/0
  election epoch 3, quorum 0 pgrosu-pr-60
osdmap e31: 3 osds: 3 up, 3 in
  flags sortbitwise,require_jewel_osds
pgmap v87: 112 pgs, 7 pools, 19066 kB data, 184 objects
  140 MB used, 14466 MB / 14606 MB avail
  112 active+clean
```

Next check the Ceph version via the following command:

```
ceph -version
```

You should see the following output:

```
ceph version 10.2.7 (50e863e0f4bc8f4b9e31156de690d765af245185)
```

Next check the ceph-deploy version via the following command:

```
ceph-deploy -version
```

You should see the following output:

```
1.5.38
```

Next check that RBD is operational via the following command:

```
rbd ls
```

You should see the following listed images:

```
112233445566778899img1
cirros-0.3.0-x86_64-disk.img
```

Next you will need to set the HIL *username* and *password* that is in the `bmi_userrc.sh` file, by typing the following command:

```
source bmi_userrc.sh
```

That file just contains the following two entries:

```
export HAAS_USERNAME=haas
export HAAS_PASSWORD=secret
```

Next we can enter the BMI virtual environment via the following two commands:

```
cd ~/ims/
source .bmi_venv/bin/activate
```

You will know that you are in the virtual environment if you see the prompt being prefixed by `(.bmi_venv)` displayed as follows:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$
```

The Einstein and Picasso servers are already running, which you can verify by the following command:

```
ps -Af | grep server | grep -iv color
```

You should see *one* Picasso server process, and *three* Einstein server processes:

```
ubuntu 22670 1 0 15:01 pts/0 00:00:00 python scripts/picasso_server
ubuntu 22669 1 0 15:01 pts/0 00:00:00 python scripts/einstein_server
ubuntu 22680 22669 0 15:01 pts/0 00:00:00 python scripts/einstein_server
ubuntu 22681 22669 0 15:01 pts/0 00:00:00 python scripts/einstein_server
```



Configuring BMI

To ensure that the BMI debug output does not show up during the BMI workflow, please type the following commands:

```
cat /etc/bmi/bmiconfig.cfg | sed -e 's/true/false/g' > bmiconfig.cfg
mv bmiconfig.cfg /etc/bmi/bmiconfig.cfg
```

2.2.2 The HIL Configuration

Remember that we need the ports on the switch to be pre-configured with the project (VLAN network isolation) for the NIC (node) we will provision. That has been performed via the `install_hil.sh` script, which can be accessed here:

https://github.com/sirushtim/ims/blob/327acf2db0094e331ca0d9b734b8b99a64f722a4/scripts/install/install_hil.sh

This script will setup HIL to create the *network*, *project*, *node* and register them properly with the *switch* using the following commands:

```
# Create Haas projects
haas project_create bmi_infra

### Setup HaaS mock node
haas node_register bmi_node mock moch-hostname mock-username mock-password
haas project_connect_node bmi_infra bmi_node

### Tell HaaS the MAC address of the NIC
haas node_register_nic bmi_node bmi_port "00:00:00:00:00:00"

### Setup HaaS switch
haas switch_register bmi_switch mock moch-hostname mock-username mock-password
haas port_register bmi_switch bmi_port
haas port_connect_nic bmi_switch bmi_port bmi_node bmi_port

### Setup HaaS network
haas network_create_simple bmi_network bmi_infra
```

The above HIL code defines the following variables, which are required for BMI:

Project ► `bmi_infra`

Node ► `bmi_node`

Network ► `bmi_network`

NIC ► `bmi_port`

Now that the environment is properly set up, you can use BMI to step through the workflow.

2.3 USING BMI

2.3.1 Creating a new image

First we need to create a new image using `rbd` in Ceph for BMI to use:

```
rbd create bmi-test-image --size 1 --image-format 2
```

To ensure the new image exists run `rbd ls` and you should see the following:

```
112233445566778899img1
bmi-test-image
cirros-0.3.0-x86_64-disk.img
```

2.3.2 Import the new image into BMI

To import the image into BMI type the following command:

```
bmi import bmi_infra bmi-test-image
```

To ensure the new image imported run `bmi db ls` and you should see the following:

Id	Name	Project	Ceph	Public	Snapshot	Parent
1	cirros-0.3.0-x86_64-disk.img	bmi_infra	112233445566778899img1	False	False	
2	bmi-test-image	bmi_infra	112233445566778899img2	False	False	

FIGURE 2.2 – The output of running: `bmi db ls`.

The same thing can be viewed through the following command:

```
bmi ls bmi_infra
```

This will result in the following output:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ bmi ls bmi_infra
-----+
|           Image           |
|-----+
| cirros-0.3.0-x86_64-disk.img |
|           bmi-test-image   |
|-----+

```

FIGURE 2.3 – The output of running: `bmi ls`.

Notice in Figure 2.4 how in Ceph there now is a cloned image created with the name `112233445566778899img2`, since the golden image named `bmi-test-image` must be preserved.

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ rbd ls | grep bmi-test-image
bmi-test-image
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ rbd ls | grep 112233445566778899img2
112233445566778899img2
```

FIGURE 2.4 – The seeing the images in Ceph.

2.3.3 Provisioning a Node Using BMI

To provision a node in BMI, type the following command:

```
bmi pro bmi_infra bmi_node bmi-test-image bmi_network bmi_port
```

If the above processes successfully, you should see Success printed on your terminal screen, as follows:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ bmi pro bmi_infra bmi_node bmi-test-image bmi_network bmi_port
Success
```

FIGURE 2.5 – The output of running BMI provisioning command.

To see the new snapshot in BMI, run `bmi db ls` and you should see the following:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ bmi db ls
```

Id	Name	Project	Ceph	Public	Snapshot	Parent
1	cirros-0.3.0-x86_64-disk.img	bmi_infra	112233445566778899img1	False	False	
2	bmi-test-image	bmi_infra	112233445566778899img2	False	False	
3	bmi_node	bmi_infra	112233445566778899img3	False	False	bmi-test-image

FIGURE 2.6 – Listing the newly provisioning in BMI.

Notice how the name is the node name itself (`bmi_node`) who's parent is `bmi-test-image` is listed, with the Parent column set to `bmi-test-image`.

2.3.4 Creating a Snapshot Using BMI

Snapshots provide the ability to get an instance of a state of an image at a point-in-time. These are just a read-only copy, which would require to be cloned and flattened¹ in order to be writeable. Snapshots in fact are central to protecting a golden image on Ceph, so that the original is preserved.

¹Flattening an image makes it independent from the parent snapshot by copying the data to the child image.

To create a snapshot of a node in BMI, first make sure that the node is powered off, and then type the following command:

```
bmi snap create bmi_infra bmi_node bmi-test-image-snapshot
```

If the above processes successfully, you should see Success printed on your terminal screen, as follows:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ bmi snap create bmi_infra bmi_node bmi-test-image-snapshot
Success
```

FIGURE 2.7 – The output of creating a snapshot using BMI.

To see the new snapshot in BMI, run `bmi db ls` and you should see the following:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ bmi db ls
```

Id	Name	Project	Ceph	Public	Snapshot	Parent
1	cirros-0.3.0-x86_64-disk.img	bmi_infra	112233445566778899img1	False	False	
2	bmi-test-image	bmi_infra	112233445566778899img2	False	False	
3	bmi_node	bmi_infra	112233445566778899img3	False	False	bmi-test-image
4	bmi-test-image-snapshot	bmi_infra	112233445566778899img4	False	True	bmi-test-image

FIGURE 2.8 – Listing the newly created snapshot in BMI.

Notice how the Snapshot column is now set to True, with the Parent column set to `bmi-test-image`.

To see what is stored in Ceph, below is the output of running `rbd ls -l`:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ rbd ls -l | grep img3 | grep snapshot
112233445566778899img3      1024k rbd/112233445566778899img2@snapshot 2
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ rbd ls | grep img4
112233445566778899img4
```

FIGURE 2.9 – The stored files in Ceph..

2.3.5 Deprovisioning a Node Using BMI

To deprovision a node in BMI, type the following command:

```
bmi dpro bmi_infra bmi_node bmi_network bmi_port
```

If the above processes successfully, you should see Success printed on your terminal screen, as follows:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ bmi dpro bmi_infra bmi_node bmi_network bmi_port
Success
```

FIGURE 2.10 – The output of running BMI deprovisioning command.

To see how BMI is updated, run `bmi db ls` and you should see the following:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ bmi db ls
+-----+-----+-----+-----+-----+-----+-----+
| Id | Name | Project | Ceph | Public | Snapshot | Parent |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | cirros-0.3.0-x86_64-disk.img | bmi_infra | 112233445566778899img1 | False | False | |
| 2 | bmi-test-image | bmi_infra | 112233445566778899img2 | False | False | |
| 4 | bmi-test-image-snapshot | bmi_infra | 112233445566778899img4 | False | True | bmi-test-image |
+-----+-----+-----+-----+-----+-----+-----+
```

FIGURE 2.11 – Listing the newly created snapshot in BMI.

2.3.6 Removing a Snapshot Using BMI

To remove a snapshot of a node in BMI, type the following command:

```
bmi snap rm bmi_infra bmi-test-image-snapshot
```

If the above processes successfully, you should see Success printed on your terminal screen, as follows:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ bmi snap rm bmi_infra bmi-test-image-snapshot
Success
```

FIGURE 2.12 – The output of removing a snapshot using BMI.

To see that the snapshot was removed in BMI, run `bmi db ls` and you should see the following:

To see that the snapshot does not exist in Ceph, below is the output of running `rbd ls -l`:

2.3.7 Removing an Image Using BMI

To remove an image in BMI, type the following command:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ bmi db ls
```

Id	Name	Project	Ceph	Public	Snapshot	Parent
1	cirros-0.3.0-x86_64-disk.img	bmi_infra	112233445566778899img1	False	False	
2	bmi-test-image	bmi_infra	112233445566778899img2	False	False	

FIGURE 2.13 – Showing that the snapshot was removed in BMI.

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ rbd ls -l | grep img3 | grep snapshot
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$
```

FIGURE 2.14 – Showing that the snapshot does not exist in Ceph.

```
bmi rm bmi_infra bmi-test-image
```

If the above processes successfully, you should see Success printed on your terminal screen, as follows:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ bmi rm bmi_infra bmi-test-image
Success
```

FIGURE 2.15 – The output of removing an image using BMI.

To see that the image was removed in BMI, run `bmi db ls` and you should see the following:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ bmi db ls
```

Id	Name	Project	Ceph	Public	Snapshot	Parent
1	cirros-0.3.0-x86_64-disk.img	bmi_infra	112233445566778899img1	False	False	

FIGURE 2.16 – Showing that the image was removed in BMI.

To remove the image from Ceph, just type the following command:

```
rbd rm bmi-test-image
```

You should now see that the image does not exist in Ceph, below is the output of running `rbd ls`:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ rbd ls | grep bmi-test-image
bmi-test-image
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ rbd rm bmi-test-image
Removing image: 100% complete...done.
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ rbd ls | grep bmi-test-image
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$
```

FIGURE 2.17 – Showing that the image does not exist in Ceph.

2.4 EXISTING THE VIRTUAL ENVIRONMENT

To exit `virtualenv`, type the following command:

```
deactivate
```

Your terminal prompt should change to the following after typing the command:

```
(.bmi_venv) ubuntu@pgrosu-pr-60:~/ims$ deactivate
ubuntu@pgrosu-pr-60:~/ims$
```

FIGURE 2.18 – Existing the Virtual Environment `virtualenv`.

This completes the end-to-end training for learning how to use the most common BMI commands.

BMI INTERNALS

This chapter will describe some of the internals of how BMI works.

3.1 THE CLONE-SNAPSHOT WORKFLOW

Central to the BMI methodology - especially with a Ceph networked-storage cluster, servicing images via a *RADOS Block Device (RBD)* - is the *Clone-Snapshot* workflow:

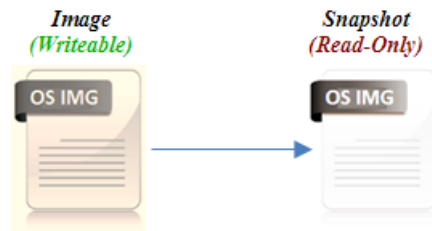


FIGURE 3.1 – The Clone-Snapshot Workflow.

Images are files used to iPXE boot from, which can also be written to. Images *cannot* be cloned, though if one creates a snapshot - which is a point-in-time, read-only copy of an image that preserves its state - one can clone that snapshot to create a new child image. These child images are now connected to the parent snapshot, but one can *flatten* them in order to copy over any dependent data to sever this connection, and thus make them independent. Therefore in order to ensure provenance, and also be able to extend the BMI workflow through new customized images, one always will create a snapshot after creating or cloning/flattening a new image. By guaranteeing that all images will have a snapshot, one will always be able to connect to any image's snapshot to inherit and extend its state

cloning. All snapshots associated with images that are created through BMI are named snapshot.

The general overview connection between Ceph, RADOS, snapshots and images is illustrated in Figure 3.2¹.

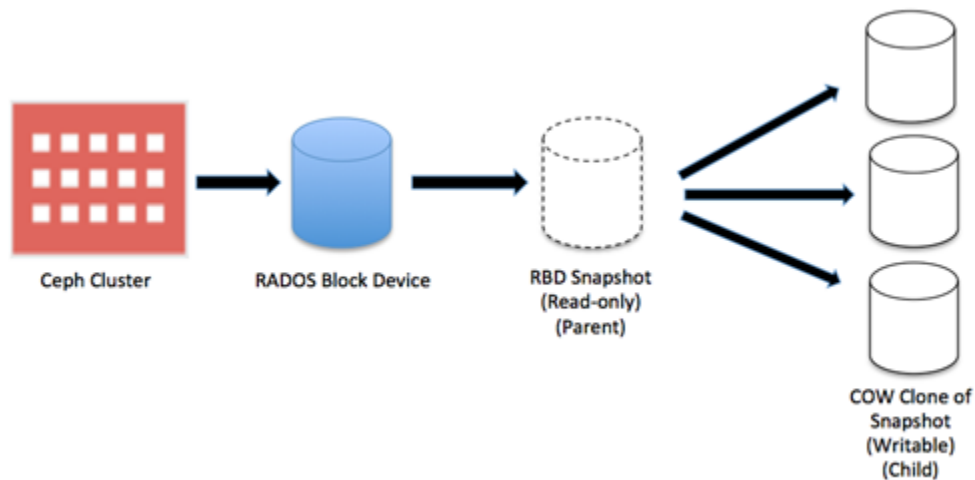


FIGURE 3.2 – A Ceph cluster servicing a RADOS block device providing access to snapshots and images.

3.2 IMPORTING AN IMAGE

The concept behind importing an image into BMI is to create a duplicate of the image - considered the *Golden Image* in Ceph - so that we have a golden image from BMI's perspective to extend from. The database column in BMI called Ceph contains the images associated with that instance of BMI, that are stored in Ceph. Now any subsequent operation on a row in the Name column from the BMI database — such as *provisioning* or *snapshotting* — will actually clone the image's snapshot indicated in the Ceph column for that respective row of that name, and then create a snapshot as well.

The one difference when importing an image into BMI, is that there might not be a snapshot in the original image in Ceph. For that reason, every time an image is imported into BMI, a snapshot is taken. Afterwards a *clone-and-flatten* operation is performed with a subsequent snapshot. Since the flattening step makes the child image independent, BMI will remove the parent snapshot in order to save space. In Figure 3.2, the steps described above illustrate this workflow:

¹ <https://www.packtpub.com/books/content/working-ceph-block-device>

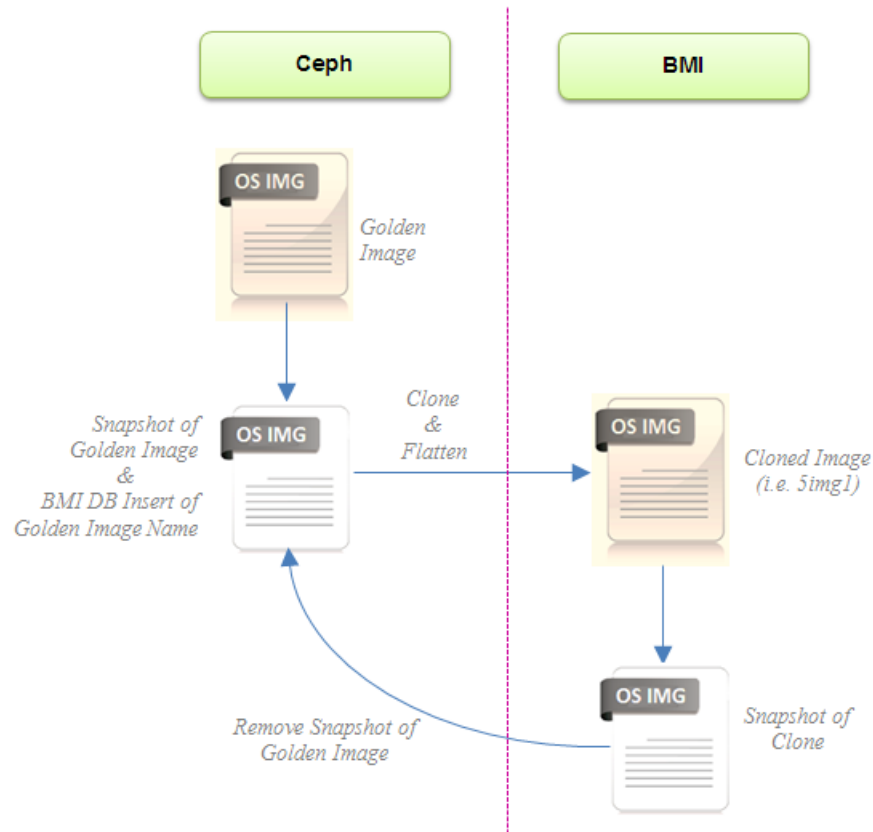


FIGURE 3.3 – BMI Import Workflow.

The steps that take place when importing an image into BMI are described in the following lines of code:

<https://github.com/CCI-MOC/ims/blob/dev/ims/einstein/operations.py#L431-L454>

3.2.1 Ceph Image Name Format In The BMI Database

One thing to mention, is that any image entry into the database under the Ceph column will be formatted using the following nomenclature:

PREFIXimgSUFFIX

The PREFIX is configured via the `/etc/bmi/bmiconfig.cfg` file, through the `uid` variable — available under the `[bmi]` heading:

```
[bmi]
uid = 5
```

The SUFFIX begin with 1 and will continue to increment with every new addition to the database. The same value will also stored under the Id column.

III

APPENDICES

APPENDIX

This appendix will provide a overview of the User Acceptance Testing for BMI, and for details please consult the Administrator's Guide: BMI User Acceptance Testing Framework.

A.1 USER ACCEPTANCE TESTING

USER ACCEPTANCE TESTING implementation has been added to BMI in order validate a deployment through a black-box end-to-end system test. This performs a *Behavior-Driven Development* scenario using the behave¹ Python package.

A.1.1 Behavior-Driven Development (BDD)

Behavior-Driven Development is defined through a live-document implemented using the Gherkin language², which utilizes *Given-When-Then* control-flow syntax defined as follows:

- Given ► *Defines a given state.*
- When ► *Defines a given action performed under the given state.*
- Then ► *Defines the expected outcome after the action is performed.*

In Figure A.1 is defined the end-to-end test performing a scenario of steps, where each line is a step that refers to a function.

¹ <http://pythonhosted.org/behave/>

² <https://github.com/cucumber/cucumber/wiki/Gherkin>

Feature: Running an end-to-end acceptance test

Scenario: Importing/Removing Image, DB/Ceph consistency

Given RBD will create an image

image_name
bmi-test-image

And BMI log line-count will be measured at the beginning

When BMI will import an image

image_name	project_name
bmi-test-image	bmi_infra

And BMI will provision a node

image_name	project_name	network_name	node_name	NIC
bmi-test-image	bmi_infra	bmi-provision-dev	cisco-05	enp130s0f0

Then BMI will create a snapshot of a node

project_name	node_name	snapshot_name
bmi_infra	cisco-05	bmi-test-image-snapshot

Then RBD will confirm the snapshot exists

snapshot_name
bmi-test-image-snapshot

And BMI will remove a snapshot

snapshot_name	project_name
bmi-test-image-snapshot	bmi_infra

Then BMI will deprovision a node

project_name	network_name	node_name	NIC
bmi_infra	bmi-provision-dev	cisco-05	enp130s0f0

And BMI will remove an image

image_name	project_name
bmi-test-image	bmi_infra

Then RBD will confirm the removed image's clone

image_name	project_name
bmi-test-image	bmi_infra

And RBD will remove the created image

image_name
bmi-test-image

And BMI log line-count will be measured at the end

FIGURE A.1 – The BMI End-to-End Behavior-Driven Deployment Test, with tables of parameters to test with.

For example, in Figure A.2 the creation of an RBD image at the start is defined through the `rbd_create_image()` function, where it is decorated by the sentence referenced in the live-document.

```

import behave
import time # Needed for Ceph Hammer client consistency
from bmi_config import RBD_CREATE, IMAGE_NAME, PROVISIONING_DELAY
from subprocess import check_output, CalledProcessError, STDOUT
from test_operation import test_event_store_insert, test_rollback

@step('RBD will create an image')
def rbd_create_image(context):
    for row in context.table:
        try:
            print( "      -> Checking that no pre-existing " +
                  row['image_name'] + " is present in Ceph, before creating it...")
            rbd_filename_check_stdout = check_output('rbd ls | grep ' + row['image_name'],
                                                    stderr=STDOUT, shell=True)

        except CalledProcessError:
            pass # The image already exists, as it was previously created

        try:
            print( "      -> Creating the " + row['image_name'] + " image in Ceph...")
            rbd_create_stdout = check_output( 'rbd create ' +
                                             row['image_name'] +
                                             ' --size 1 --image-format 2', stderr=STDOUT, shell=True)

        except CalledProcessError:
            pass # The image already exists, as it was previously created
        except Exception:
            test_rollback(context)
            print( "      -> Checking that " + row['image_name'] + " exists in Ceph...")
            rbd_filename_check_stdout = check_output( 'rbd ls | grep ' +
                                                    row['image_name'], stderr=STDOUT, shell=True)
            context.rbd_filename_check = rbd_filename_check_stdout.strip()

        # Journal the event for rollback
        test_event_store_insert(context, { RBD_CREATE: {IMAGE_NAME : 'image_name'} })

        time.sleep( PROVISIONING_DELAY ) # Needed for Ceph Hammer client consistency
        assert context.rbd_filename_check == row['image_name']

```

FIGURE A.2 – The definition of the RBD creation step, where the decoration highlights the sentence referenced in the end-to-end deployment test.

A.1.2 *Configuring the Acceptance Tests*

Before running the acceptance tests, it is necessary to configure BMI service to test. This is performed by the following steps:

1. Copy the acceptance-tests directory to the new environment.
2. Proceed to the config/tests-uat directory and duplicate recursively one of the configuration, as such:

```
cp -r neu-haas-dev NEW_BMI_SERVICE_TO_TEST
```

Example:

```
cp -r neu-haas-dev prb-bu-dev
```

3. Enter the newly created directory and update the config file accordingly. Below is an example of one:

```
export BMI_RELEASE_NAME=moc-0.5-release

export BMI_PROJECT=bmi_infra
export HIL_NODE=cisco-05
export HIL_NETWORK=bmi-provision-dev
export BMI_IMAGE_NAME=bmi-test-image
export BMI_SNAPSHOT_NAME=bmi-test-image-snapshot

export HAAS_ENDPOINT=http://127.0.0.1:8000
export BMI_CONFIG=/etc/bmi/bmiconfig_test.cfg
export HAAS_USERNAME=haasadmin
export HAAS_PASSWORD=admin1234
```

4. Then update the config/bmi_config.sh file, for just the following variables:

```
export BMI_INSTANCE_DIR=${HOME}/pgrosu/ims-instance
export ACCEPTANCE_TESTS_SRC_DIR=${HOME}/pgrosu/acceptance-tests
```

The BMI_INSTANCE_DIR variable denotes where you would like the git-cloned BMI instance to reside, on which the tests will be run.

The ACCEPTANCE_TESTS_SRC_DIR variable denotes the location of the acceptance tests directory from which you will run the ./bmi-uat.py command.

By following the above steps you can now create your own customization for BMI services to test for.

A.1.3 *Performing the Acceptance Tests*

The performance tests can be initiated via the following steps:

1. To list the testable BMI service configurations, type the following command:

```
./bmi-uat.py ls
```

You should see something like the following:

The available configurations are:

```
neu-haas-dev
```

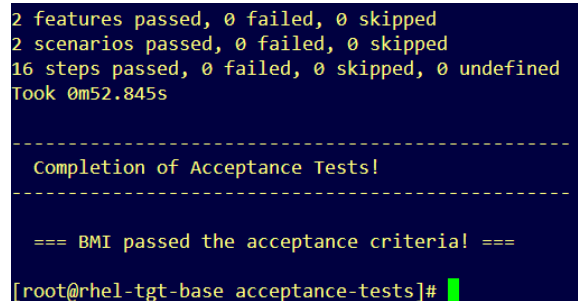
2. To run the standard end-to-end configuration, type the following command:

```
./bmi-uat.py --run BMI_SERVICE_CONFIGURATION
```

Example:

```
./bmi-uat.py --run neu-haas-dev
```

At the end you if the tests passed successfully, you should see the following output:



```
2 features passed, 0 failed, 0 skipped
2 scenarios passed, 0 failed, 0 skipped
16 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m52.845s

-----
Completion of Acceptance Tests!
-----

=== BMI passed the acceptance criteria! ===

[root@rhel-tgt-base acceptance-tests]#
```

FIGURE A.3 – The BMI completed successfully the end-to-end scenario.

3. To run the tests with randomized parameters, type the following where the value indicates the number of times to run the test:

```
./bmi-uat.py --run neu-haas-dev --randomize 3
```

4. To check if the tests passed or failed, type the following:

```
./bmi-uat.py check
```

You should see the following:

```
All tests passed!
```

This command checks the test-results directory for any subdirectory containing FAIL in its name.

5. To cleanup all previous results, type the following:

```
./bmi-uat.py clean
```

By following the above steps you can now test your own customizations of BMI any services.

INDEX

- Bare Metal Imaging (BMI), 3
- Behavior-Driven Development (BDD), 31
- bmi db, 13
- bmi db ls, 19–23
- bmi dpro, 13, 21
- bmi import, 19
- bmi ls, 13, 19
- BMI methodology, 25
- bmi pro, 13, 20
- bmi rm, 23
- bmi snap, 13
- bmi snap create, 21
- bmi snap rm, 22
- Ceph, 6, 16
- Ceph image database name format, 27
- ceph-deploy, 16
- clone, 25
- clone-and-flatten, 26
- Clone-Snapshot workflow, 25
- configuring User Acceptance Tests, 33
- deprovision, 21
- DHCP, 4
- DHCPDISCOVER, 5
- Einstein, 9
- flatten, 25
- Golden Image, 26
- Hardware Isolation Layer (HIL), 3
- HIL, 4
- import, 19
- initiator, 5
- Intelligent Platform Management Interface (IPMI), 5
- IPMI, 5
- IPMI network card, 7
- iPXE, 5
- multi-tenancy, 3
- network isolation, 3
- Open Cloud eXchange (OCX), 3
- out-of-band management (OOBM), 5
- Picasso, 9
- power-cycle the nodes, 5
- Preboot Execution Environment (PXE), 4, 5
- provision, 4, 20
- PXE, 5
- RADOS Block Device (RBD), 6, 25
- RBD, 16
- rbd create, 18
- rbd ls, 23
- rbd rm, 23
- remove a snapshot, 22
- remove an image, 22
- REST service, 7
- RPC calls, 8
- running User Acceptance Tests, 34
- SCSI, 5
- snapshot, 21, 22, 25
- targets, 5
- TGT, 6
- Trivial FTP, 5
- trunking, 4
- User Acceptance Testing (UAT), 31
- Virtual LAN (VLAN), 4
- virtualenv, 24
- VLAN ID, 4