

# CSCE 678

## Distributed Systems and Cloud Computing

### Project Report (P1)

**TEAM MEMBERS:**

Krit Gupta (927001565)

Bhaves Munot (227002818)

Sukhdeep Gill (326007739)

Christopher Mureekan (227001713)

## INDEX

<b>How to run the project/code</b>	<b>2</b>
<b>Time spent in implementing</b>	<b>2</b>
<b>GitHub Link</b>	<b>2</b>
<b>General Outline for Part A and Part B</b>	<b>3</b>
<b>Part C - Ideas for designing the caching mechanism (with implementation)</b>	<b>6</b>
Other ideas to improve the caching further (Future Scope)	8
<b>Highlights</b>	<b>8</b>
<b>Sample Screenshots for different commands</b>	<b>9</b>

## How to run the project/code

Please find the README.md file in the github repository for detailed information.

The instructions are as follows:-

- The project runs on Python3. Please install Python3 in order to run the program.
- `cd folder_name`
- `git clone https://github.tamu.edu/sgill12/689-18-b.git`
- `cd 678-18-b`
- Files: configurations.txt files - image-config.txt, hdwr-config.txt and flavor-config.txt need to be present in the folder structure, as these are the input files for the script.
- `python3 P1.py aggiestack config --hardware hdwr-config.txt`
- `python3 P1.py aggiestack config --images image-config.txt`
- `python3 P1.py aggiestack config --flavors flavor-config.txt`

(These commands will create the corresponding .dct files, needed to run the rest of the commands)

Now, can simply run the above written commands by using *python3 P1.py* in front of it.

Example:

```
python3 P1.py aggiestack show flavors
```

## Time spent in implementing

Part A - 11/03/2018 : 3:00pm - 5:30pm - 2.5 Hours

11/04/2018 10:30am - 4:30pm - 6 Hours

Part B - 11/17/2018 - 6 hours

11/16/2018 - 4 hours

Part C - 12/02/2018 - 6:00 pm to 10:00 pm - 4 hours

Part C - 12/03/2018 - 10:00 noon to 2:00 pm - 4 hours

Clean up and other end of project checks etc. - 12/03/2018 - 2:30 pm to 4:30 pm - 2 hours

Report and design of Part C - 12/2/2018 and 12/3/2018 - 4 hours

## GitHub Link

<https://github.tamu.edu/sgill12/689-18-b>

## General Outline for Part A and Part B

We have implemented separate methods for each command so that code maintenance is reduced and code testing becomes easier.

We first check for a log-file, and create one if it is not there. The log file stores all the logs of the program execution.

If the length of the entered command is less than 4 - we throw an error and exit the program.

We first create a method to read the input file - with the following dictionaries in the method:

- `config` : to store one instance config.
- `machines_config` : to store all machines in a rack.
- `racks_config` : to store all machines for each rack.

To adhere to the definition of the new hardware, we have 3 lists: hardware, image and flavor which saves the definition accordingly.

We loop over the corresponding list according to which command is called. We store the *data as a pickle* and load it whenever needed.

`racks_config` : stores all the machines for each rack in this method.

We keep updating the log file accordingly.

Show :

All show commands use the `showContent()` method. We simply load the pickle and use the `config` dictionary to print the Machine Hardware.

Further sub methods like `showAvailableHardware()`, `showPhysicalServers()` are made to do the corresponding tasks. We load the corresponding pickle and simply go over the contents of the file and display the required output.

We update the log file correspondingly.

`CreateInstance()` :

Using a list named `columns` which stores the image, flavor and machine to loop over the input, we have two dictionaries:

- `newInstance` : for storing the new instance.
- `instancesDict` : for existing instances.

Differentiating these two dictionaries helps us check to have only unique instances, and avoid duplication.

Finding the machine to host the virtual instance is achieved through `findMachine()` method. The method checks if a particular machine currently has the resources to host the vCPU of the given flavor in the *currentHardwareDict*, another dictionary to store all the configurations.

Updates the machine configuration using the `updateResources()` method which writes to the `currentHardwareConfiguration.dct` file after retrieving the flavor and the current hardware dictionaries from their corresponding files (stored as pickles).

`addMachine()`:

Method to add a new machine. Another dict called `machineDict` is used for storing the new machine. After error checking, the new machine is added to the *"hardwareConfiguration.dct"* (pickle for storing the hardware configuration.)

`UpdateCurrentHardware()`:

Method called to update the file containing the hardware configuration of all the machines. Calling this function will update this file.

`currHardwareDict` is the dictionary which stores all the configurations. This gets updated to store the new hardware configuration. We then save this dictionary as a pickle.

## Overview of the implementation of the various commands in Part A and B:

1. `aggiestack server create --image IMAGE --flavor FLAVOR_NAME INSTANCE_NAME`
  - Uses the `createInstance()` method as mentioned above.
2. `aggiestack server delete INSTANCE_NAME`
  - Uses the `deleteInstance()` method, which loads the existing instances, updates the machine resources by calling the `updateResources()` method and deleting the instance to be deleted.
3. `aggiestack server list`
  - Uses the `instanceList()` method, which simply loads the existing instances and simply iterates through the dictionary generated and lists the names of the servers.
4. `aggiestack admin show hardware`
  - Uses the `showAvailableHardware()` method, which loads the corresponding file which has been saved as a pickle, and simply iterates the generated dictionary and prints the contents of the dictionary.
5. `aggiestack admin show instances`
  - Uses the `showPhysicalServers()` method, which once again loads the corresponding file and iterates over the generated dictionary and prints the results.
6. `aggiestack admin evacuate RACK_NAME`
  - Uses the `evacuate()` method, which tries to separate machines into the ones on this rack v/s the others. We make 2 separate lists: `evacuationRackMachines` and `otherRacksMachines` for this purpose.
  - Once done, it finds the instances which are on the machines on the given rack, and for each instance, try and find a machine which can host it in `otherRackasMachine`. If we are unable to find one, we simply print - "The machines available are not capable of hosting these instances".
  - Next, we simply save the updated instance file.

7. `aggiestack admin remove MACHINE`

- Uses the `removeMachine()` method. While removing the machine, we make sure that there is no instance running on it.
- If we find an instance running on the machine, we print about it.
- Next, we remove the machine from 2 files: `hardwareConfiguration.dct` and `currentHardwareConfiguration.dct` to make sure that the machine does not appear in any output and that it does not host any new instance. Also, it make sure that no instance is created in physical server MACHINE.

8. `aggiestack admin add --mem MEM --disk NUM_DISKS --vcpus VCPUs --ip IP --rack RACK_NAME MACHINE`

- Uses the `addMachine()` method. We create a new dictionary called `machineDict` and add this new machine to two files : `hardwareConfiguration.dct` and `currentHardwareConfiguration.dct` to make sure that the code works as per the problem definition.

## Part C - Ideas for designing the caching mechanism (with implementation)

We have **designed and implemented** the image caching feature end to end into our “aggiestack”.

The basic principle that we followed is that use the caching as much as possible in order to efficiently use the storage that is available on each rack.

In the beginning, there are no images on rack storage and storage availability on each rack is full. Whenever an instance is created, we try to find a machine that it will execute on. Here we use an intelligent decision making algorithm to choose an appropriate machine. Now the appropriate machine for that instance is the one which is on a rack that has the image required for the instance - already cached. If none of the racks have the image cached, then we choose any appropriate machine which has enough resources. However, we are making smart decisions here as well. We try to assign the instance to a machine which is on a rack that has higher amount of free/available storage. But we don't stop here. We store the image in the cache of the rack on which the machine is present - for future use. So next time, if same image is required, then machines on this rack will get a preference. While doing this, we also take care of storage available on the rack. Whenever a new image is cached, the available storage is decremented and vice versa.

Below, we will also discuss the detailed algorithm in plain english statements for assigning a machine to instance efficiently by utilizing the cache storage available.

The algorithm for assigning a machine for an instance can be summarized as follows:-

1. Task is to assign an instance i.e. image, flavor on a machine.
2. Check if the required image is cached on some rack
  - a. If YES:-
    - i. For each rack on which the image is cached -
      1. For each machine on this rack
        - a. If machine has enough resources for the instance
          - i. Assign instance to current machine
        - ii. Go to Step 5
3. If no rack has the image cached OR even if the image is cached on some rack but machines on those racks do not have enough resources available for that instance to run, we check rest of the racks. The order of checking the rack is highest amount of available storage on rack to lowest.
  - a. For each rack
    - i. For each machine on this rack
      1. If machine has enough resources

- a. Assign instance to current machine
  - b. Go to Step 5
- 4. If no machine can take this instance
  - a. Show error that resources are not available anymore
  - b. STOP
- 5. Now we have assigned a machine for the instance. We also know the rack on which this machine is present
  - a. If image is already cached on the rack, we don't have to do anything
    - i. STOP
  - b. If image is not cached on rack
    - i. Check if enough space is available on rack storage to cache this new image
      - 1. If NO -
        - a. Delete one existing cached image on the rack
        - b. Increase the storage availability on rack by size of the deleted image
        - c. If enough space is now available on rack, STOP. Otherwise go to step (a).
    - ii. At this stage, now we definitely have space to store the image on rack. So cache the image on rack now.
    - iii. Reduce the storage availability on rack by size of the image
- 6. STOP

This way we are handling the rack storage by caching images and also efficiently utilizing the space by removing previous images on rack to accommodate the new ones.

The way this happens in the code is as follows -

1. Whenever a command for create instance is issued, execution reaches in "createInstance()" function which calls "findMachine()" function.
2. Here it calls "tryIntelligentSelection()" function which only checks for the machines which are available on a rack that has the required image already cached.
3. "tryIntelligentSelection()" checks if image is cached somewhere. Then it finds all the racks on which the image is cached. And later checks if any machine on these racks is able to run the new instance.
4. If "tryIntelligentSelection()" is not able to find a machine, then we iterate over other remaining machines.
5. This way "findMachine()" function returns a machine name to "createInstance()" method.
6. Now the second of updating the cache happens in "updateCaching()" function.
7. "updateCaching()" function check if image is already cached on the rack on which the machine - that we are going to use for this instance - is present.

8. If image is cached, we return immediately.
9. Otherwise, we see the available space and try to create space on rack if it does not already exist - which is explained above.
10. Then we update the storage availability on rack.
11. Execution then goes back to "`createInstance()`" and rest of the things are taken care of further.

According to us, for a good engineer/programmer - it would take 4-5 hours of coding and 2-3 hours of testing to get the caching logic correct given that rest of the code without-caching is handed to him/her and the engineer needs to spend time on understanding existing code. Of course, we would not want to make a lot of changes in existing code.

### Other ideas to improve the caching further (Future Scope)

1. Main improvement can be made in cache replacement policy. We can use least recently used cache technique and take this into consideration when removing an image from the cache.
2. We can divide the racks in groups in which bigger size images will be loaded on one group which smaller size images are loaded on other group.
3. We can also track which rack is getting used often and if there is imbalance, we can try to have load balancer like technique to balance this.

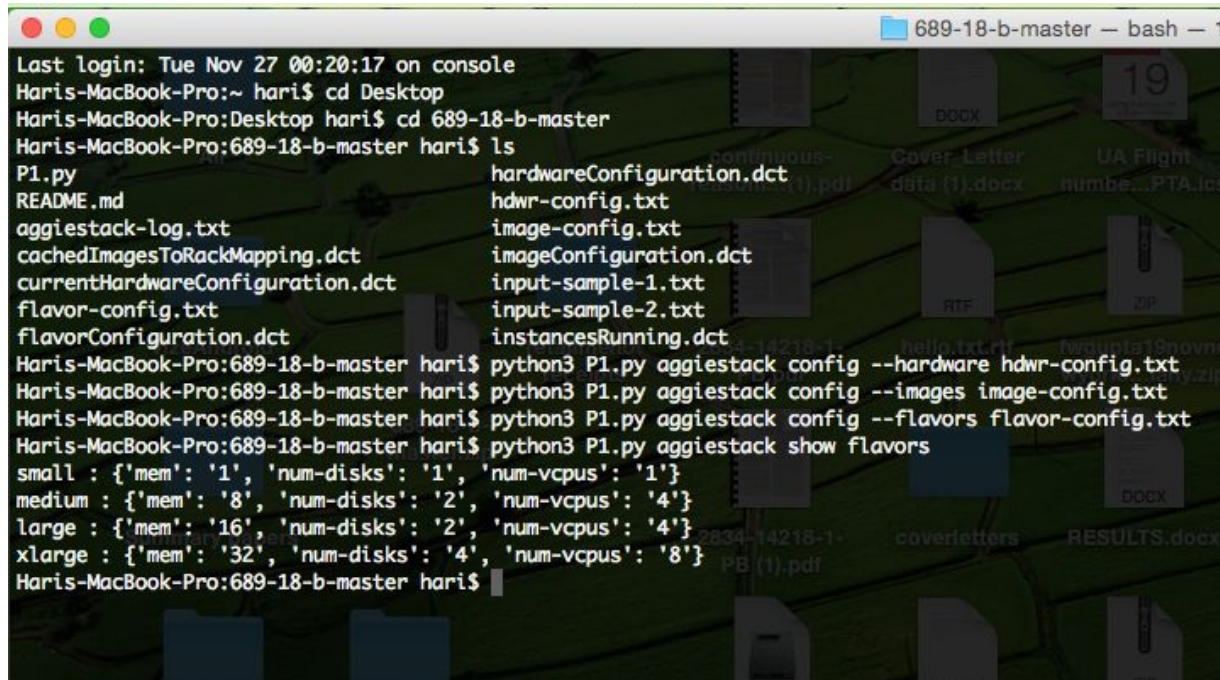
### Highlights

1. **Error checks** - Throughout the code, we have added error checks e.g. make sure that file exists before opening it, make sure the command is valid, make sure the output of the command is SUCCESS.  
This way we fail safe even if we have to - without crashing the program
2. **Caching** - Intelligent caching mechanism is in place to make the overall project efficient to run.
3. **Efficient Algorithms** - We have tried to implement most of the things efficiently wherever possible. We have considered algorithmic time complexities when making the design the way it is.
4. **Intelligent Decisions** - When assigning an instance to machine, we try to assign it on a rack which has the image already cached. If that is not possible, we try to assign it on a rack which has more available storage. Once the instance is assigned to a machine, we cache the image on rack storage for future use.
5. **New API** - An API to retrieve information about cached images on a rack is also provided.
6. **Well Tested** - All the parts of the code are tested. Especially caching logic is tested by creating various corner cases to cover all scenarios and all lines of code.



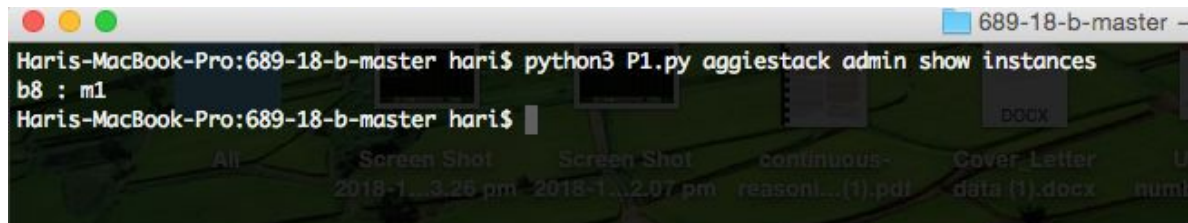
## Sample Screenshots for different commands

Initial loading commands and aggiestack show flavors

A terminal window titled '689-18-b-master — bash — 1' showing a series of commands and their outputs. The background of the terminal window features a faint, dark map of a city. The commands and outputs are as follows:

```
Last login: Tue Nov 27 00:20:17 on console
Harris-MacBook-Pro:~ hari$ cd Desktop
Harris-MacBook-Pro:Desktop hari$ cd 689-18-b-master
Harris-MacBook-Pro:689-18-b-master hari$ ls
P1.py                                hardwareConfiguration.dct
README.md                           hdwr-config.txt
aggiestack-log.txt                  image-config.txt
cachedImagesToRackMapping.dct       imageConfiguration.dct
currentHardwareConfiguration.dct     input-sample-1.txt
flavor-config.txt                   input-sample-2.txt
flavorConfiguration.dct             instancesRunning.dct
Harris-MacBook-Pro:689-18-b-master hari$ python3 P1.py aggiestack config --hardware hdwr-config.txt
Harris-MacBook-Pro:689-18-b-master hari$ python3 P1.py aggiestack config --images image-config.txt
Harris-MacBook-Pro:689-18-b-master hari$ python3 P1.py aggiestack config --flavors flavor-config.txt
Harris-MacBook-Pro:689-18-b-master hari$ python3 P1.py aggiestack show flavors
small : {'mem': '1', 'num-disks': '1', 'num-vcpus': '1'}
medium : {'mem': '8', 'num-disks': '2', 'num-vcpus': '4'}
large : {'mem': '16', 'num-disks': '2', 'num-vcpus': '4'}
xlarge : {'mem': '32', 'num-disks': '4', 'num-vcpus': '8'}
Harris-MacBook-Pro:689-18-b-master hari$
```

aggiestack show instances

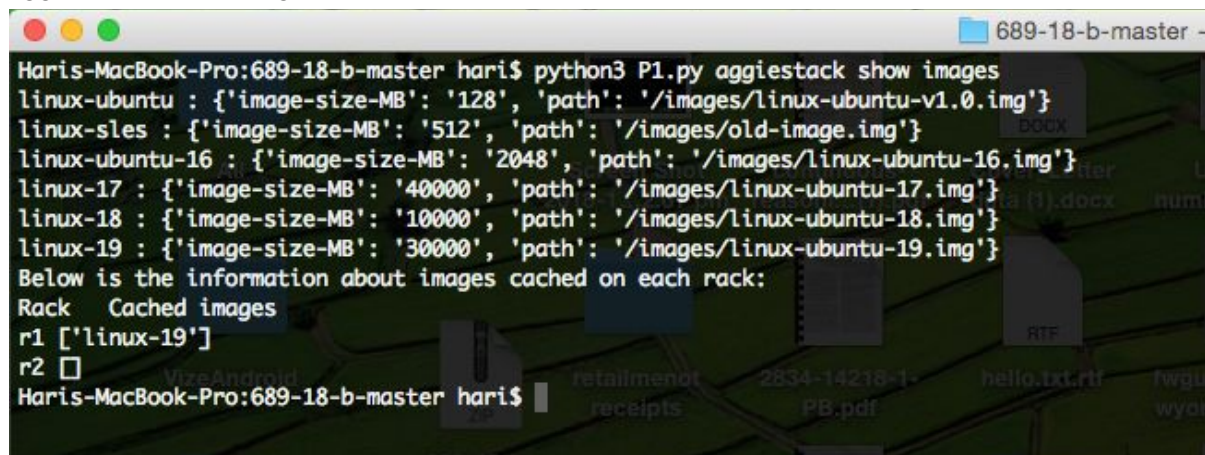


```
Haris-MacBook-Pro:689-18-b-master hari$ python3 P1.py aggiestack admin show instances
b8 : m1
Haris-MacBook-Pro:689-18-b-master hari$
```

aggiestack show hardware

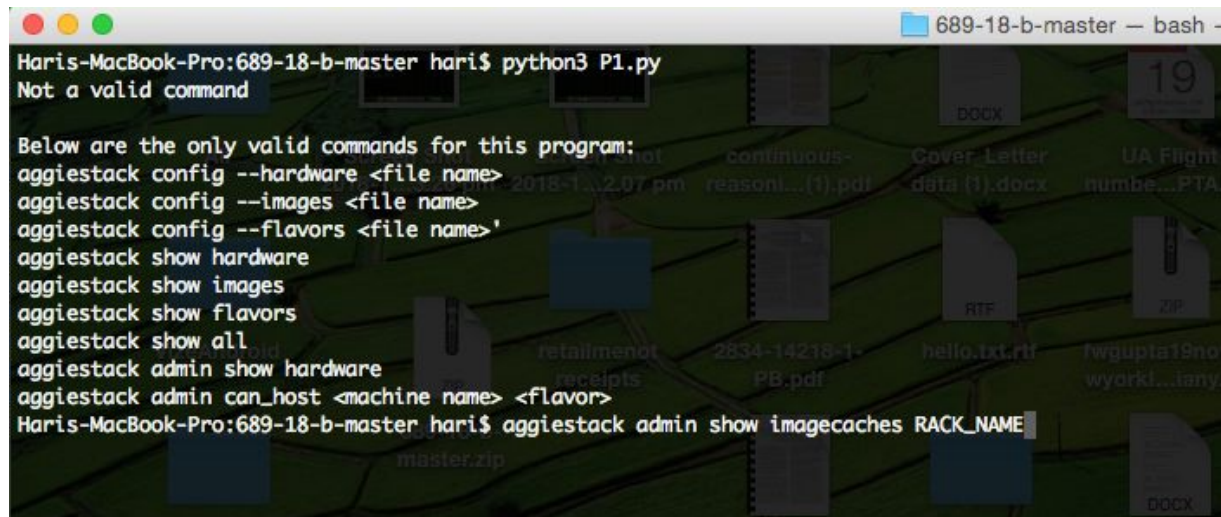
```
Haris-MacBook-Pro:689-18-b-master haris$ python3 P1.py aggiestack show hardware
{'r1': {'mem': '40960'}, 'r2': {'mem': '40960'}}
{'m1': {'rack': 'r1', 'ip': '128.0.0.1', 'mem': '16', 'num-disks': '8', 'num-vcpus': '4'}, 'm2': {'rack': 'r1', 'ip': '128.0.0.2', 'mem': '16', 'num-disks': '32', 'num-vcpus': '4'},
 'm3': {'rack': 'r1', 'ip': '128.0.0.3', 'mem': '16', 'num-disks': '16', 'num-vcpus': '4'}, 'm4': {'rack': 'r2', 'ip': '128.0.0.4', 'mem': '16', 'num-disks': '8', 'num-vcpus': '4'},
 'k1': {'rack': 'r2', 'ip': '128.1.1.0', 'mem': '32', 'num-disks': '32', 'num-vcpus': '8'}, 'k2': {'rack': 'r2', 'ip': '128.1.0.2', 'mem': '32', 'num-disks': '32', 'num-vcpus': '8'},
 'k3': {'rack': 'r2', 'ip': '128.1.3.0', 'mem': '32', 'num-disks': '32', 'num-vcpus': '8'}, 'calvin': {'rack': 'r1', 'ip': '128.129.4.4', 'mem': '8', 'num-disks': '16', 'num-vcpus':
 '1'}, 'hobbes': {'rack': 'r1', 'ip': '1.1.1.1', 'mem': '16', 'num-disks': '64', 'num-vcpus': '16'}, 'dora': {'rack': 'r1', 'ip': '1.1.1.2', 'mem': '64', 'num-disks': '256', 'num-vcpus':
 '16'}}
Haris-MacBook-Pro:689-18-b-master haris$
```

aggiestack show images



```
Haris-MacBook-Pro:689-18-b-master hari$ python3 P1.py aggiestack show images
linux-ubuntu : {'image-size-MB': '128', 'path': '/images/linux-ubuntu-v1.0.img'}
linux-sles : {'image-size-MB': '512', 'path': '/images/old-image.img'}
linux-ubuntu-16 : {'image-size-MB': '2048', 'path': '/images/linux-ubuntu-16.img'}
linux-17 : {'image-size-MB': '40000', 'path': '/images/linux-ubuntu-17.img'}
linux-18 : {'image-size-MB': '10000', 'path': '/images/linux-ubuntu-18.img'}
linux-19 : {'image-size-MB': '30000', 'path': '/images/linux-ubuntu-19.img'}
Below is the information about images cached on each rack:
Rack  Cached images
r1 ['linux-19']
r2 []
Haris-MacBook-Pro:689-18-b-master hari$
```

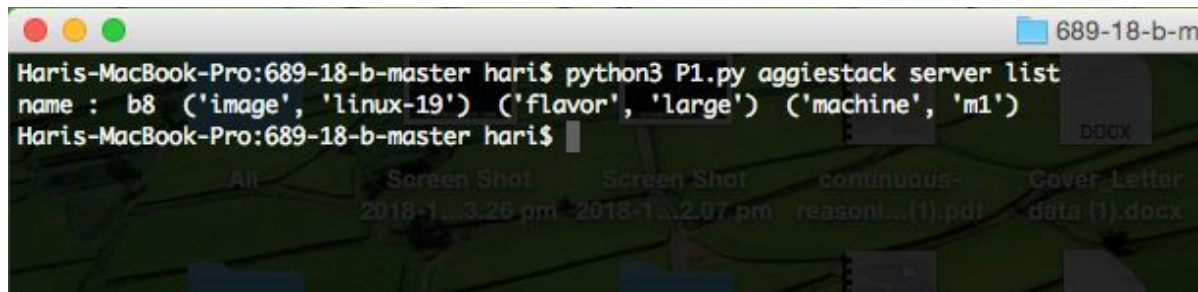
Error Message for any invalid command



A terminal window titled "689-18-b-master — bash" is shown. The prompt is "Haris-MacBook-Pro:689-18-b-master hari\$". The user has entered "python3 P1.py", and the program has responded with "Not a valid command". Below this, the program lists the valid commands for this program:

```
Below are the only valid commands for this program:
aggiestack config --hardware <file name>
aggiestack config --images <file name>
aggiestack config --flavors <file name>
aggiestack show hardware
aggiestack show images
aggiestack show flavors
aggiestack show all
aggiestack admin show hardware
aggiestack admin can_host <machine name> <flavor>
Haris-MacBook-Pro:689-18-b-master hari$ aggiestack admin show imagecaches RACK_NAME
```

aggiestack server list

A screenshot of a macOS terminal window. The window title bar shows three colored buttons (red, yellow, green) and a folder icon labeled '689-18-b-m'. The terminal text shows the prompt 'Haris-MacBook-Pro:689-18-b-master hari\$' followed by the command 'python3 P1.py aggiestack server list'. The output is 'name : b8 ('image', 'linux-19') ('flavor', 'large') ('machine', 'm1')'. The prompt then changes to 'Haris-MacBook-Pro:689-18-b-master hari\$'. In the background, several document icons are visible, including 'All', 'Screen Shot 2018-1...3.26 pm', 'Screen Shot 2018-1...2.07 pm', 'continuous-reasoni...(1).pdf', 'Cover Letter data (1).docx', and 'DOCX'.



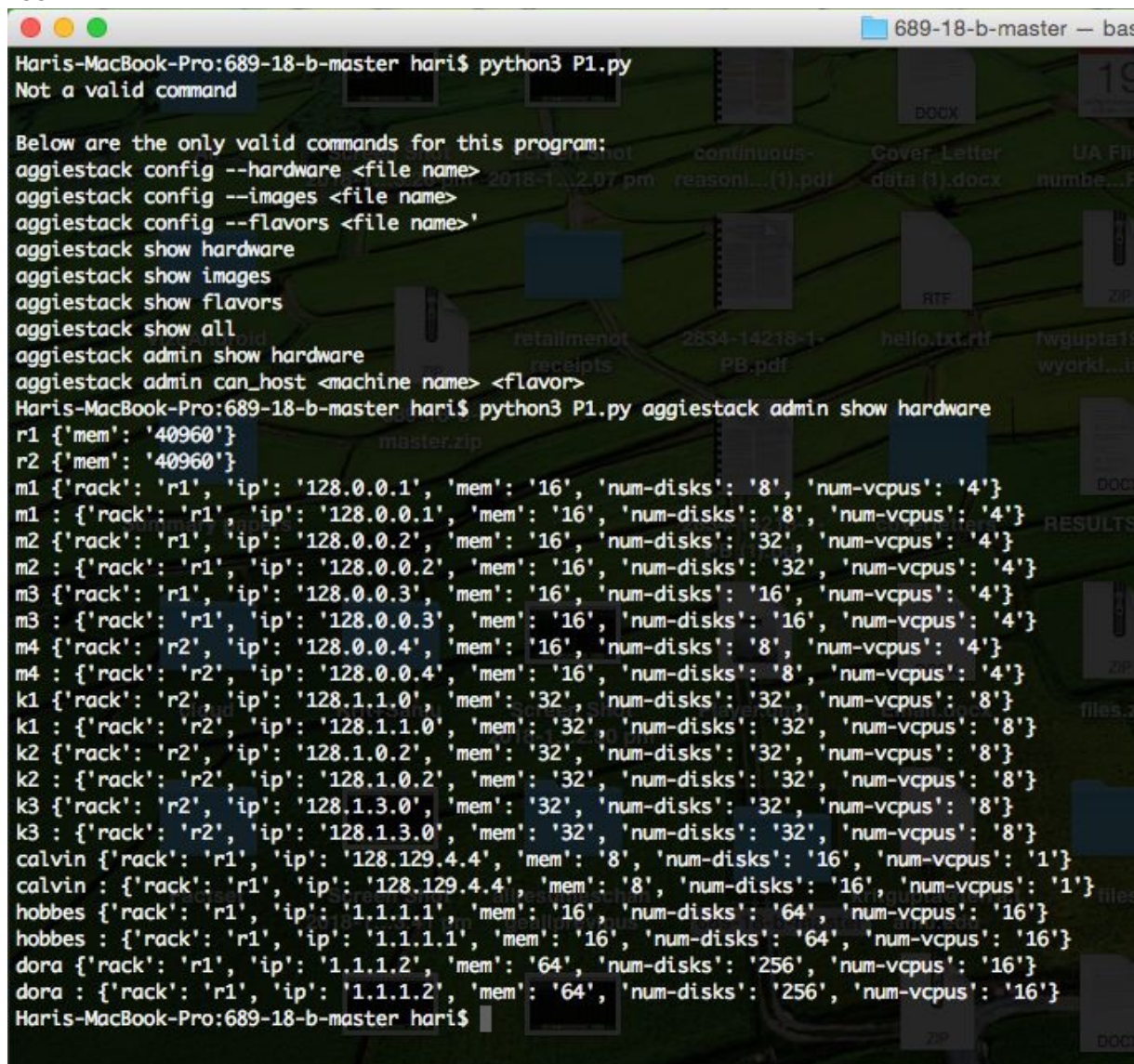
aggiestack show all

```

689-18-b-master — bash — 181x41
Haris-MacBook-Pro:689-18-b-master haris$ python3 P1.py aggiestack show all
Hardware Info:
{'r1': {'mem': '40960'}, 'r2': {'mem': '40960'}}
{'m1': {'rack': 'r1', 'ip': '128.0.0.1', 'mem': '16', 'num-disks': '8', 'num-vcpus': '4'}, 'm2': {'rack': 'r1', 'ip': '128.0.0.2', 'mem': '16', 'num-disks': '32', 'num-vcpus': '4'},
{'m3': {'rack': 'r1', 'ip': '128.0.0.3', 'mem': '16', 'num-disks': '16', 'num-vcpus': '4'}, 'm4': {'rack': 'r2', 'ip': '128.0.0.4', 'mem': '16', 'num-disks': '8', 'num-vcpus': '4'},
{'k1': {'rack': 'r2', 'ip': '128.1.1.0', 'mem': '32', 'num-disks': '32', 'num-vcpus': '8'}, 'k2': {'rack': 'r2', 'ip': '128.1.0.2', 'mem': '32', 'num-disks': '32', 'num-vcpus': '8'},
{'k3': {'rack': 'r2', 'ip': '128.1.3.0', 'mem': '32', 'num-disks': '32', 'num-vcpus': '8'}, 'calvin': {'rack': 'r1', 'ip': '128.129.4.4', 'mem': '8', 'num-disks': '16', 'num-vcpus':
{'l': {'hobbes': {'rack': 'r1', 'ip': '1.1.1.1', 'mem': '16', 'num-disks': '64', 'num-vcpus': '16'}, 'dora': {'rack': 'r1', 'ip': '1.1.1.2', 'mem': '64', 'num-disks': '256', 'num-vcpus': '16'}}
Image Info:
linux-ubuntu : {'image-size-MB': '128', 'path': '/images/linux-ubuntu-v1.0.img'}
linux-sles : {'image-size-MB': '512', 'path': '/images/sles-image.img'}
linux-ubuntu-16 : {'image-size-MB': '12048', 'path': '/images/linux-ubuntu-16.img'}
linux-17 : {'image-size-MB': '40000', 'path': '/images/linux-ubuntu-17.img'}
linux-18 : {'image-size-MB': '10000', 'path': '/images/linux-ubuntu-18.img'}
linux-19 : {'image-size-MB': '30000', 'path': '/images/linux-ubuntu-19.img'}
Flavor Info:
small : {'mem': '1', 'num-disks': '1', 'num-vcpus': '1'}
medium : {'mem': '8', 'num-disks': '2', 'num-vcpus': '4'}
large : {'mem': '16', 'num-disks': '2', 'num-vcpus': '4'}
xlarge : {'mem': '32', 'num-disks': '4', 'num-vcpus': '8'}
Haris-MacBook-Pro:689-18-b-master haris$

```

aggiestack admin show hardware

A terminal window titled '689-18-b-master' on a 'Haris-MacBook-Pro'. The user enters 'python3 P1.py' and receives an error. They then enter 'aggiestack admin show hardware' and receive a list of JSON objects representing hardware configurations for various nodes.

```
Haris-MacBook-Pro:689-18-b-master hari$ python3 P1.py
Not a valid command

Below are the only valid commands for this program:
aggiestack config --hardware <file name>
aggiestack config --images <file name>
aggiestack config --flavors <file name>
aggiestack show hardware
aggiestack show images
aggiestack show flavors
aggiestack show all
aggiestack admin show hardware
aggiestack admin can_host <machine name> <flavor>
Haris-MacBook-Pro:689-18-b-master hari$ python3 P1.py aggiestack admin show hardware
r1 {'mem': '40960'}
r2 {'mem': '40960'}
m1 {'rack': 'r1', 'ip': '128.0.0.1', 'mem': '16', 'num-disks': '8', 'num-vcpus': '4'}
m1 : {'rack': 'r1', 'ip': '128.0.0.1', 'mem': '16', 'num-disks': '8', 'num-vcpus': '4'}
m2 {'rack': 'r1', 'ip': '128.0.0.2', 'mem': '16', 'num-disks': '32', 'num-vcpus': '4'}
m2 : {'rack': 'r1', 'ip': '128.0.0.2', 'mem': '16', 'num-disks': '32', 'num-vcpus': '4'}
m3 {'rack': 'r1', 'ip': '128.0.0.3', 'mem': '16', 'num-disks': '16', 'num-vcpus': '4'}
m3 : {'rack': 'r1', 'ip': '128.0.0.3', 'mem': '16', 'num-disks': '16', 'num-vcpus': '4'}
m4 {'rack': 'r2', 'ip': '128.0.0.4', 'mem': '16', 'num-disks': '8', 'num-vcpus': '4'}
m4 : {'rack': 'r2', 'ip': '128.0.0.4', 'mem': '16', 'num-disks': '8', 'num-vcpus': '4'}
k1 {'rack': 'r2', 'ip': '128.1.1.0', 'mem': '32', 'num-disks': '32', 'num-vcpus': '8'}
k1 : {'rack': 'r2', 'ip': '128.1.1.0', 'mem': '32', 'num-disks': '32', 'num-vcpus': '8'}
k2 {'rack': 'r2', 'ip': '128.1.0.2', 'mem': '32', 'num-disks': '32', 'num-vcpus': '8'}
k2 : {'rack': 'r2', 'ip': '128.1.0.2', 'mem': '32', 'num-disks': '32', 'num-vcpus': '8'}
k3 {'rack': 'r2', 'ip': '128.1.3.0', 'mem': '32', 'num-disks': '32', 'num-vcpus': '8'}
k3 : {'rack': 'r2', 'ip': '128.1.3.0', 'mem': '32', 'num-disks': '32', 'num-vcpus': '8'}
calvin {'rack': 'r1', 'ip': '128.129.4.4', 'mem': '8', 'num-disks': '16', 'num-vcpus': '1'}
calvin : {'rack': 'r1', 'ip': '128.129.4.4', 'mem': '8', 'num-disks': '16', 'num-vcpus': '1'}
hobbes {'rack': 'r1', 'ip': '1.1.1.1', 'mem': '16', 'num-disks': '64', 'num-vcpus': '16'}
hobbes : {'rack': 'r1', 'ip': '1.1.1.1', 'mem': '16', 'num-disks': '64', 'num-vcpus': '16'}
dora {'rack': 'r1', 'ip': '1.1.1.2', 'mem': '64', 'num-disks': '256', 'num-vcpus': '16'}
dora : {'rack': 'r1', 'ip': '1.1.1.2', 'mem': '64', 'num-disks': '256', 'num-vcpus': '16'}
Haris-MacBook-Pro:689-18-b-master hari$
```





