### Command Reference

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize</td>
<td>31</td>
</tr>
<tr>
<td>config</td>
<td>32</td>
</tr>
<tr>
<td>import_cluster</td>
<td>33</td>
</tr>
<tr>
<td>start_cluster</td>
<td>34</td>
</tr>
<tr>
<td>show_cluster</td>
<td>35</td>
</tr>
<tr>
<td>add_node</td>
<td>36</td>
</tr>
<tr>
<td>reboot_node</td>
<td>37</td>
</tr>
<tr>
<td>terminate_node</td>
<td>37</td>
</tr>
<tr>
<td>start_node</td>
<td>38</td>
</tr>
<tr>
<td>remove_node</td>
<td>38</td>
</tr>
<tr>
<td>terminate_cluster</td>
<td>39</td>
</tr>
<tr>
<td>delete_cluster</td>
<td>40</td>
</tr>
<tr>
<td>retry</td>
<td>40</td>
</tr>
<tr>
<td>connect</td>
<td>40</td>
</tr>
<tr>
<td>show_nodes</td>
<td>41</td>
</tr>
</tbody>
</table>

### Using Custom Machine Images with CycleCloud

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Machine Images</td>
<td>43</td>
</tr>
<tr>
<td>JetPack</td>
<td>43</td>
</tr>
<tr>
<td>CycleCloud Image Requirements</td>
<td>43</td>
</tr>
</tbody>
</table>

### Building a Custom Image using the AWS Console

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select a Base AMI</td>
<td>44</td>
</tr>
<tr>
<td>Launch the Instance</td>
<td>44</td>
</tr>
<tr>
<td>Custom Configuration:</td>
<td>45</td>
</tr>
<tr>
<td>Installing Jetpack</td>
<td>46</td>
</tr>
<tr>
<td>Clean-Up</td>
<td>47</td>
</tr>
<tr>
<td>Bake the New Image</td>
<td>48</td>
</tr>
</tbody>
</table>

### The Anatomy of Jetpack

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firewalls and Jetpack</td>
<td>48</td>
</tr>
<tr>
<td>Release Notes</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>CycleCloud version 5.3.0</td>
<td>98</td>
</tr>
<tr>
<td>CycleCloud version 3.2.0</td>
<td>98</td>
</tr>
<tr>
<td>CycleCloud version 3.1.0</td>
<td>99</td>
</tr>
<tr>
<td>CycleCloud version 3.0.0</td>
<td>100</td>
</tr>
</tbody>
</table>
CycleCloud Guide

Overview
CycleCloud is a tool to make it easy to start HPC clusters in the cloud. It supports Amazon’s VPC so it runs inside your firewall (either on-site or in the cloud). It includes support for GridEngine and Condor, and clusters can be configured with the same cluster-init specification used on Hosted CycleCloud. It imports cluster definitions from StarCluster, and includes a RESTful API and command-line tools to manage compute clusters.

Quickstart
Now that you have successfully configured and initialized the CLI tools, this document will walk you through your first steps with the CycleCloud CLI. If you haven’t already configured your CLI tools, please see Installation.

Configure your AWS Account
Before starting any clusters you will need to do a bit of configuration within your Amazon EC2 account so that you can log in to your cluster as well as have the nodes inside your cluster be able to talk to each other.

1. Sign in to the AWS Console.
2. Select EC2 from the Compute & Networking section to modify your EC2 settings.

Create a Key Pair
To be able to log into the nodes started by CycleCloud you will need to create a key pair. This key pair will allow you to SSH into the nodes as the root user.

1. Select Key Pairs from the left hand menu.
2. Click the Create Key Pair button.
3. Name the key pair cyclecloud (you can use a different name, but the default configuration assumes a keypair named cyclecloud).
4. When asked to save the key to your computer, save it as cyclecloud.pem inside the ~/.ssh directory (you can use a different name or path, but this will require additional configuration).
5. You may have to alter the permissions on this key so that SSH will be able to use it. You can modify the permissions as follows:

   chmod 600 ~/.ssh/cyclecloud.pem
Edit Security Groups

CycleCloud will start nodes in the default security group unless otherwise specified. To use a security group other than the default, you can edit the template files in ~/.cycle to reflect the desired security group. You have to open a few ports in the security group to allow the nodes within a cluster to talk to each other:

1. Select Security Groups from the left hand menu.
2. Select default from the list of security groups (if this is a new AWS account, default should be the only item in the list).
3. Select the inbound tab from the bottom of the screen.
4. Make the following rules:

<table>
<thead>
<tr>
<th>Security Groups</th>
<th>Port Range</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom TCP Rule</td>
<td>8080</td>
<td>0.0.0.0/0</td>
<td>Open port 8080 to the world so that you can access CycleServer on master nodes.</td>
</tr>
<tr>
<td>Custom TCP Rule</td>
<td>22</td>
<td>0.0.0.0/0</td>
<td>Open port 22 so that you can SSH into your nodes.</td>
</tr>
<tr>
<td>Custom TCP Rule</td>
<td>1-65535</td>
<td>default</td>
<td>Allow TCP communication on all ports between nodes in the default security group.</td>
</tr>
<tr>
<td>Custom UDP Rule</td>
<td>1-65535</td>
<td>default</td>
<td>Allow UDP communication on all ports between nodes in the default security group.</td>
</tr>
</tbody>
</table>

5. Click Apply Rule Changes to save your changes.

Create a Cluster
To get started, we will first import a cluster definition from the template files installed during the initialize step from the Installation guide. In this case we will be importing a cluster that we will name demo from the sge_templates.txt file using its sge cluster definition. This will create a cluster for us that will consist of two nodes: a master node running the SGE scheduler and management software as well as an execute node to run jobs on:

```
$ cyclecloud import_cluster demo --file ~/.cycle/sge_templates.txt -c sge
Creating cluster demo....
-------
demo : off
-------
Keypair: cyclecloud
Cluster nodes:
  master:  off
Cluster node arrays:
   execute: 1 instances, 1 cores, off
Total nodes: 2
```

### Start the Cluster

Once you've imported the cluster definition from the template, you can start your cluster by running the `start_cluster` command:

```
$ cyclecloud start_cluster demo
Starting cluster demo....
_________________________
demo : allocation -> started
_________________________
Keypair: cyclecloud
Cluster nodes:
  master:  Launching instance
Cluster node arrays:
   execute: 1 instances, 1 cores, Allocation (AWS.RunInstances/default/us-east)
Total nodes: 2
```

### Add an Execute Host to your Cluster

Currently the cluster has only a master and single execute. To run jobs, you'll need to add more execute nodes using the `add_node` command:

```
$ cyclecloud add_node demo -t execute
Adding nodes to cluster demo....
----------
demo : started
----------
Keypair: cyclecloud
Cluster nodes:
```
Check Cluster State

Your cluster is now being started. It will take a few minutes for Amazon to provision the instances as well as to run the CycleCloud configuration routines. To check on the status of your cluster you can run the `show_cluster` command. You may have to run this a few times before the output changes:

```sh
$ cyclecloud show_cluster demo
--------------
demo : started
--------------
Keypair: cyclecloud
Cluster nodes:
  master: running i-24dc7e48 ec2-50-17-111-28.compute-1.amazonaws.com (10.82.251.180)
Cluster node arrays:
  execute: 2 instances, 2 cores, Started
Total nodes: 3
```

Notice that the execute nodes arrays are collapsed into a summary view since there could possibly be a very large number of execute nodes. To view the details about each node, use the `--long` option to `show_cluster`:

```sh
$ cyclecloud show_cluster demo --long
-------------
demo : started
--------------
Keypair: cyclecloud
Cluster nodes:
  execute-1: running i-99b3a0f4 ec2-67-202-10-228.compute-1.amazonaws.com (10.147.170.214)
  execute-2: running i-75b3a018 ec2-54-234-17-247.compute-1.amazonaws.com (10.152.177.29)
  master: running i-24dc7e48 ec2-50-17-111-28.compute-1.amazonaws.com (10.82.251.180)
Total nodes: 3
```

SSH Into your Cluster

After some time your cluster will be in a running state and you will be able to SSH into it to do some work. You can connect to a node with the `connect` command. In this example you would connect to the master as follows:

```sh
$ cyclecloud connect master
CycleCloud
 (version 3.6.0)
```
This assumes the keypair is set up for you as a default (in ~/.ssh or added to the ssh-agent). If you saved your keypair under a different location or using a different name you will have to specify the -k PATH_TO_KEYPAIR option.

It can take a fair amount of time (several minutes) for the cluster to become available. If the ssh command times out or asks for a password, please wait for a bit and try again.

Running a Demo Job

Now that you've logged into your cluster, you can verify that the SGE environment is up and running by issuing a qhost command. The two nodes that make our cluster should show up:

```
$ qhost

HOSTNAME               ARCH         NCPU  LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
-------------------------------------------------------------------------------
global                  -               -     -       -       -       -       -
domU-12-31-39-0C-24-9D.compute-1.internal linux-x64       1  0.48    1.7G  146.9M  500.0M     0.0
ip-10-72-223-210.ec2.internal linux-x64       1  2.92    1.7G  634.6M  500.0M     0.0
```

All CycleCloud clusters come with a simple demo job that calculates the constant pi that you can run to make sure everything is configured correctly. To run this demo job, switch to the cluster.user user, and run the job from the demo directory:

```
$ sudo su - cluster.user
$ cd ~/demo
$ ./runpi.sh
Your job-array 1.1-1000:1 ("pitest") has been submitted
```

Your cluster now has 1000 jobs submitted to calculate pi which will run on your execute node. To check the status of your jobs you can run the qstat command:

```
$ qstat

job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
  1 0.56000 pitest     cluster.user   r     10/31/2012 15:10:03 all.q@domU-12-31-39-0C-24-9D.c     1 1
  1 0.55125 pitest     cluster.user   qw    10/31/2012 15:09:57                                    1 2-1000:1
```
You can customize the username and password of the login user by specifying the Username and Password attributes in a cluster definition template. By default a user named cluster.user will be created with a random password.

Adding More Execute Nodes

Completing 1000 jobs with a single 1-core execute node will take a long time, so let's add a few more workers to the cluster to help speed things along. We can use the `add` command to add a few more execute nodes to our cluster:

```
$ cyclecloud add_node demo -t execute --count 5
Adding nodes to cluster demo....
--------------
demo : started
--------------
Keypair: cyclecloud
Cluster nodes:
  master: running i-09661b75 ec2-50-17-111-28.compute-1.amazonaws.com (10.82.251.180)
Cluster node arrays:
  execute: 5 instances, 5 cores, Allocation (AWS.RunInstances/default/us-east)
  execute: 2 instances, 2 cores, Started
Total nodes: 8
```

When the nodes finish booting they will be automatically added to the cluster and begin executing jobs.

Download Results

As the results of your workload finish, you will want to download them to your local computer for further analysis. Instead of waiting for all 1000 jobs to finish, we can just wait for the first one to finish and then download its results to our home directory using the `scp` command from your local computer:

```
scp -i ~/.ssh/cyclecloud.pem root@ec2-50-17-111-28.compute-1.amazonaws.com:/shared/scratch/pi-cluster.user/pitest.o1.1 ~
```

This command will download the results of the first job which are stored on the cluster in `/shared/scratch/pi-cluster.user/pitest.o1.1` (as defined in `runpi.sh`) to our home directory on our local computer.

Once the data has been downloaded you can terminate the remaining jobs using `qdel`:

```
$ qdel 1
cluster.user has deleted job 1
```

Terminate the Cluster
Now that your work is completed, the final thing to do is to terminate the cluster so you aren't charged any longer by Amazon. Terminating a cluster is as easy as starting one:

```
$ cyclecloud terminate_cluster demo
Terminating cluster demo....
-----------------------------
.demo : termination -> terminated
-----------------------------
Keypair: cyclecloud
Cluster nodes:
  master  Terminating instance i-24dc7e48
Cluster node arrays:
  execute: 7 instances, None cores, Termination (AWS.TerminateInstances)
Total nodes: 8
```

**Congratulations!** You have finished the quickstart guide. You can customize your clusters further to use different instance types, have different number of execute nodes or use VPC. The included template files located in `~/.cycle` provide more information.

**Installation**

CycleCloud's command-line interface (CLI) tool can be used to create, configure and manage clusters controlled by CycleCloud. It is highly recommended that you install these tools to make cluster management and configuration easier. The CycleCloud CLI tool is distributed as a standard installable Python package. It is recommended that you use Python 2.6 or later, but older versions (2.4) should work.

**Installing CycleCloud**

**System Requirements**

This document provides a step-by-step walkthrough for installing CycleCloud. The machine it is installed on must meet the following requirements:

- A 32-bit or 64-bit Linux distribution
- At least 8GB of RAM
- Two or more CPU cores
- At least 50GB of free disk space
- Administrator (root) privileges

**Installation**
Before beginning, make sure you are logged into the machine you are installing on as the administrative (root) user. This is required for the installation process to configure CycleServer to start on boot.

Download the CycleCloud tarball and extract the contents to a temporary directory:

```
tar -xzvf cycle_server-4.5.4-linux64.tar.gz  -C /tmp
```

The temporary directory will now contain a `cycle_server` directory that contains the installation script. Changing to this directory and running the `install.sh` script will install CycleServer into `/opt/cycle_server`:

```
$ cd /tmp/cycle_server
$ ./install.sh
```

If you would like to install to a different location use the `--installldir` option; for example, to install into `/usr/local`:

```
$ ./install.sh --installldir /usr/local/cycle_server
```

The installation procedure will install CycleServer into the desired location and will create an OS user named `cycle_server` that will be used to run the service. It will also add an init.d script so that CycleServer is started automatically on boot.

When the installation script finishes, you should see something similar to the following message:

```
CycleServer is installed. Please browse to http://localhost:8080 to complete the installation.
```

Open the provided URL inside your web browser. It may take a minute for the URL to become available as CycleServer initializes itself.

**Configuration Wizard**

The first time you visit your CycleServer installation you will be shown an configuration wizard to walk you through any additional steps needed to finish your installation. The following sections describe some of the steps in more detail:

**Creating an Admin User**

The wizard will prompt you to create a user. This user will have full administrative privileges to CycleServer. Simply enter a unique user identifier (such as "admin"), a full name ("CycleServer Administrator") and a password. You will need this log in information to enable the CycleCloud command line interface (CLI) to connect to your CycleCloud instance.
Note: After initial setup is completed, you can configure CycleServer for Active Directory / LDAP integration if desired.

Proxy Configuration

CycleCloud requires outbound HTTP and HTTPS access to Amazon Web Services (AWS). If CycleCloud is behind a proxy, then you will need to configure it to pass through the proxy.

To configure proxy access for CycleCloud itself:

1. Navigate to the config subdirectory of your CycleCloud installation directory
   a. For example, in Linux the default is generally /opt/cycle_server/config

2. Create a new text file named: boto.cfg

3. Edit the new boto.cfg file and add the following content (replacing the hostname and port with your proxy settings):

   ```
   [Boto]
   proxy = myproxy.com
   proxy_port = 8080
   
   proxy_user = foo
   proxy_pass = bar
   ```

4. If your proxy requires a log in, then also add these lines:

5. Restart CycleCloud:

   ```
   /opt/cycle_server/cycle_server restart webserver
   ```

**NOTE:**

If you are still getting HTTP errors when running the CycleCloud CLI tools, first verify that you have set the correct proxy settings in boto.cfg above. If so, then try adding the following settings to boto.cfg (required by some proxy servers) to proxy HTTPS:

```
is_secure = False
https_validate_certificates = False
```

For most proxies, changing just the is_secure option is sufficient. After changing any of the boto.cfg settings, a CycleCloud restart is required.
Installing Command-Line Tools

The CycleCloud CLI is distributed as a linux binary (CycleCloud-5.3.0.linux64.tar.gz), a windows binary (CycleCloud-5.3.0.win64.zip) and as source (CycleCloud-5.3.0.tar.gz). The binaries can be used once untarred. The source tarball can be installed just like any Python package.

Using pip

Using pip is the recommended way of installing the package as it allows for easy upgrading and removal of Python packages:

```bash
pip install CycleCloud-5.3.0.tar.gz
```

Using easy_install

Using easy_install is another easy way to install the CycleCloud CLI package along with all dependencies:

```bash
easy_install CycleCloud-5.3.0.tar.gz
```

If you have multiple versions of python installed, you can select which version to install on by using version specific pip or easy_install tools. For example, for python 2.6 you may have a pip installer with the name `pip-2.6` or `easy_install-2.6`.

Using setup.py

If you do not have pip or easy_install configured on your system, you can also use the standard `setup.py install` method:

```bash
tar -xzvf CycleCloud-5.3.0.tar.gz
cd CycleCloud-5.3.0
python setup.py install
```

Install Permissions

If you get a 'permission denied' error when installing, you may need to run the install command with `sudo` since packages will be written to your system level Python install:

```bash
$ sudo easy_install CycleCloud-5.3.0.tar.gz
```

Test the Install
To make sure that your CLI has been installed correctly you can run the `cyclecloud` command with no options, and you should receive a help message like the following:

```
$ cyclecloud
Usage: cyclecloud COMMAND [options]
Options:
  -h, --help  show this help message and exit
  ...
```

If you see this, you have successfully installed the CycleCloud CLI tool.

**Configuration**

Before configuring the CycleCloud CLI, make sure that CycleCloud is running and accessible from the machine you are installing on. If you installed CycleCloud on the same machine you are installing the CLI tools on, you should be able to access `http://localhost:8080` using your web browser. If you installed CycleServer on a different machine, make sure you can access it via your web browser.

**Initialize the CLI**

First, run the `initialize` command. This will ask you a few questions about how to connect to your CycleServer instance. If this is your first time configuring CycleCloud, you must provide some information about your Amazon Web Services (AWS) account. CycleCloud uses this information to start compute clusters and to store data to S3:

```
$ cyclecloud initialize
Welcome to CycleCloud!
CycleServer username: [admin] admin
CycleServer password:

Generating CycleCloud key...
CycleCloud configuration stored in /home/demo/.cycle/config.ini
CycleCloud connection information is configured properly.
Wrote cluster template file '~/cycle/condor_templates.txt'.
Wrote cluster template file '~/cycle/sge_templates.txt'.
Wrote cluster template file '~/cycle/starcluster.txt'.
Access Key: **YOUR_ACCESS_KEY**
Secret Key: **YOUR_SECRET_KEY**
Default Region (Enter for default): us-east-1
Bucket name to create (com.cyclecloud.<name>.locker): demo
AWS account initialized and new credentials stored.
```
The first set of questions to answer are the URL pointing back to the CycleServer instance you have set up, if you are installing the CLI tools on the same machine the default of localhost should be sufficient. Next you have to specify the user name and password you created when setting up CycleServer for the first time.

Several example cluster templates will be written to the ~/.cycle directory for future reference.

You will have to provide some information so that CycleCloud can access your AWS account. If the AWS_ACCESS_KEY and AWS_SECRET_KEY environment variables are defined, the CLI tools will ask to use them. If not, you must provide them. You will also asked to provide a default region to use when starting clusters, the default is us-east-1 (Virginia). For a full list of available regions please see the official AWS documentation.

Finally, on first configuration you will be asked to give the name you wish to use for an S3 bucket. This bucket will be used to store run-time configuration for your compute clusters. The bucket will be named 'com.cyclecloud.<name>.locker' where <name> is the value you specify. CycleCloud will only access this bucket so if you need to limit access for security reasons this bucket is the only one that needs read/write permission.

Your configuration information is saved in your home directory at ~/.cycle/config.ini. If you need to reconfigure your account, you can edit or remove this file and re-run the initialize command.

Test Your Configuration

You can test your configuration by running the show_cluster command, which will return the details of all the clusters currently being managed by CycleCloud. If it is your first configuration, this list is likely to be empty. If you have configured the CLI tools correctly, the following command should not generate an error:

```bash
$ cyclecloud show_cluster
```

Congratulations! Your CycleCloud CLI tools are installed and functioning. You should now read through our Quickstart tutorial to get up and running quickly as well as the Command Reference.

Customization

To further customize your clusters you can create new templates, or update existing ones to take advantage of features like the Amazon Spot Market to get inexpensive instances or VPC to extend your own network into the cloud. This section briefly outlines the configuration changes that will be needed to take advantage of these features.
Nodes and Node Arrays

Clusters consist of nodes, which define a single instance, and node arrays, which can be automatically scaled on demand. Node arrays support two limits on how large they can be: *MaxCount*, which limits how many instances to start, and *MaxCoreCount*, which limits how many cores to start. Neither setting will terminate existing instances.

Node arrays can span multiple machine types and availability zones, to ensure you get the capacity you need. For example, this cluster definition will try to get instances from three different instance types and two availability zones:

```
[parameters]
[[parameter executeMachineType]]
Value = c3.8xlarge, c3.4xlarge, c3.2xlarge
Autoselect = true

[[parameter zone]]
Value = us-east-1a, us-east-1b
Autoselect = true

[cluster demo]
Autoscale = true

[[nodearray execute]]
MachineType = $executeMachineType
Zone = $zone
```

There are several things to notice. First, the parameters must be declared with `Autoselect=true`. The names of the the parameters themselves do not matter. You can make either the machine type or the zone, or both, into parameters as needed. (If you are launching nodes into a VPC environment, you can specify `Subnet` instead of `Zone`.) This supports either on-demand instances or spot instances. If you use spot prices with multiple instance types, set `BidPricePerCore` instead of `BidPrice` to bid an amount scaled by the number of cores the chosen instance has.

**Note**

If the parameter value comes from an external `.properties` file, it currently must be specified in the formal extended syntax:

```
machineType := { "c3.8xlarge", "c3.4xlarge", "c3.2xlarge" }
```
When the array is scaled up, instances are chosen from the set of listed machine types and zones. CycleCloud initially bids evenly across all specified possibilities, and as it gets "out-of-capacity" results from the cloud provider, it automatically shifts to your machine-type/zone combinations that are currently providing instances. If it is unable to get all the instances requested, it will periodically cycle through all the combinations, including ones that were previously unavailable, in an attempt to get sufficient capacity.

The machine types and zones (or subnets) specified are considered to all be of equal acceptability, and CycleCloud will distribute requests across all of them. Support for expressing a preference for certain zones or machine types is not currently available.

Additional Cluster Templates

The CycleCloud CLI tools ship with some cluster templates already defined, located in the ~/.cycle directory. For example, the file sge_templates.txt defines a basic two-node SGE cluster. To create a new cluster, you can begin by copying the section of the file that defines the sge cluster and pasting it to the bottom of the configuration file with it a new name. For example, you might copy/modify the section to look like:

```
[cluster custom_sge_cluster]
  # Enable autoscaling
  Autoscale = true

  [[[node defaults]]]
  ImageId = ami-1f57c276

  # Custom keypair
  KeyPair=custom-keypair
  KeyPairLocation=~/.ssh/custom-keypair.pem

  [[[node master]]]
  # Bigger head node
  MachineType = m1.xlarge
      [[[configuration]]]
      # Removed for brevity

  [[[nodearray execute]]]
  # Set autoscaling to max out at 10 cores
  MaxCoreCount = 10
  # Start with zero execute nodes to start
  Count = 0
  # Use 2-core machines with $0.10 spot bid for autoscaling
  MachineType = m1.large
  BidPrice = 0.10
      [[[configuration]]]
      # Removed for brevity
```
By adding the above section a new cluster template called custom_sge_cluster is defined which starts with 50 m1.small execute nodes, using an m1.xlarge for the master node instead of an m1.small. Importing and running this new cluster type requires just the following two commands:

```
$ cyclecloud import custom_demo_cluster -f ~/.cycle/sge_templates.txt -c custom_sge_cluster
$ cyclecloud start custom_demo_cluster
```

### Images

CycleCloud ships with support for standard operating systems (Images). You can specify the image with `Image`:

```
[[node defaults]]
Image = Cycle CentOS 6
```

You can also specify by `ImageName`:

```
[[node defaults]]
ImageName = cycle.image.win2012
```

Finally you can always specify by `ImageId`:

```
[[node defaults]]
ImageId = ami-1234abcd
```

CycleCloud automatically uses the latest released version of the image, for the cloud provider and region that the instance is in.

**Note**

For AWS, CycleCloud picks the virtualization type (hvm or pvm) to match the instance type being used. Instance types that support both will default to hvm virtualization unless you override it with `AWS.Virtualization = pvm` on your node.

CycleCloud currently includes the following images:

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Image</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CentOS 5.10</td>
<td>Cycle CentOS 5</td>
<td>cycle.image.centos5</td>
</tr>
<tr>
<td>CentOS 6.5*</td>
<td>Cycle CentOS 6</td>
<td>cycle.image.centos6</td>
</tr>
</tbody>
</table>
Ubuntu 12.04** | Cycle Ubuntu 12 | cycle.image.ubuntu12
---|---|---
SLES 11 SP2 | Cycle SLES 11 | cycle.image.sles11
Windows 2008 R2** | Cycle Windows 2008 | cycle.image.win2008
Windows 2012 R2** | Cycle Windows 2012 | cycle.image.win2012

*SR-IOV enabled on AWS. For more information, see the AWS documentation.
**Compatible with AWS G2-class instance types

The images listed here will continue to be supported in newer releases, although they will get minor upgrades automatically (eg, CentOS 6.5 to a later version of CentOS 6).

**Note**

If you get a permission denied error when trying to use any of the images above, please contact a Cycle Computing representative so we can enable the images for your use.

**Configuration Notation**

CycleCloud cluster templates all have the option of having one or more [[[configuration]]] sections which belong to a node or nodearray. These sections specify software configuration options about the nodes being started by CycleCloud. Dotted notation is used to specify the attributes you wish to configure. For example, you can set several CycleServer configuration options on a node which has CycleServer installed:

```
[[node master]]
  [[[configuration]]]
  cycle_server.admin.name = poweruser
  cycle_server.admin.pass = super_secret
  cycle_server.http_port = 8080
  cycle_server.https_port = 8443
```

You can also specify a configuration section using prefix notation to save typing. The same configuration could also be written as:

```
[[node master]]
  [[[configuration cycle_server]]]
  admin.name = poweruser
  admin.pass = super_secret
  http_port = 8080
  https_port = 8443
```
A node/nodearray can contain multiple configuration sections if needed:

```
[[node master]]
  [[[configuration]]]
  run_list = role[sge_master_node]

  [[[configuration cycle_server.admin]]]
  name = poweruser
  pass = super_secret
```

For configuration options on the software that is installed on CycleServer nodes, please see the appropriate documentation.

## Storage

CycleCloud supports automatically attaching volumes (disks) to your nodes for additional storage space. For example, to create a 100GB volume, add the following to your `[[node]]` element in your cluster template:

```
[[[volume example-vol]]]
  Size = 100
```

This volume will be created when the instance is started and deleted when the instance is terminated. If you want to preserve the data on the volume even after the instance is terminated, make it a *persistent* volume:

```
[[[volume example-vol]]]
  Size = 100
  Persistent = true
```

This volume will be created the first time the instance is started, but will not be deleted when the instance is terminated. Instead, it will be kept, and re-attached to the instance the next time the node is started. Persistent volumes are not deleted until the cluster is deleted.

**Warning**

When your cluster is deleted, all persistent volumes are deleted as well! If you want your storage to persist longer than your cluster, you must attach a preexisting volume by id.

For Linux-based operating systems, you can control what device to attach the volume to, using the `Device` attribute:
If you do not specify a device, CycleCloud will automatically pick a device that is not in use. The specific device chosen depends on the cloud provider configuration and the image.

### Mounting Volumes

Simply specifying a volume attaches the devices to your instance but does not mount and format the device. If you prefer to have the volumes mounted and formatted when the node is started, set the optional attribute "Mount" to the name of the mountpoint configuration you wish to use with that volume:

```yaml
[[[volume example-vol]]]
Size = 100
Device = /dev/sdk

Mount = data  # The name of the mountpoint to use with this volume
```

The mountpoint named `data` is then defined in the configuration section on the node:

```yaml
[[[configuration cyclecloud.mounts.data]]]
mountpoint = /mount
fs_type = ext4
```

The above configuration specifies that you are configuring a `cyclecloud.mountpoint` named `data` using all volumes which include `Mount = data`. This volume would be formatted with the `ext4` filesystem and would appear at `/mount`.

### Advanced Usage

The previous example was a fairly simple example mounting a single, pre-formatted snapshot to a node, however more advanced mounting can take place including RAIDing multiple devices together, encrypting, and formatting new filesystems. As an example, the following will describe how one may want to RAID several EBS volumes together and encrypt them before mounting them as a single device on a node:

```yaml
[[node master]]
....
[[[volume vol1]]]
VolumeId = vol-1234abcd
Mount = giant

[[[volume vol2]]]
VolumeId = vol-5678abcd
```
Mount = giant

[[[volume vol3]]]
VolumeId = vol-abcd1234
Mount = giant

[[[configuration cyclecloud.mounts.giant]]]
mountpoint = /mnt/giant
fs_type = xfs
raid_level = 0
encryption.bits = 256
encryption.key = "0123456789abcdef9876543210"

The above example shows that there are three EBS volumes that should be attached to the node named master and that their mountpoint is named giant. The configuration for the mountpoint says that these three volumes should be RAIDed together using raid_level = 0 for RAID0, formatted using the xfs filesystem and the resulting device should be mounted at /mnt/giant. Furthermore, the device should have block level encryption using 256-bit AES with an encryption key as defined in the template. See below for a full list of available mountpoint configuration options and their meanings.

**Devices**

Defining volumes with a Mountpoint attribute, the device names will be automatically assigned and used for a given mountpoint. You can, however, customize a mountpoint with your own device names if there is a need. For example you could do the following:

```
[[node master]]
[[[configuration cyclecloud.mounts.data]]]
mountpoint = /data
devices = /dev/sdc
```

In this case we are using the devices parameter to manually specify each device that is part of the mountpoint configuration. Specifying devices manually is especially useful in cases where the AMI you are using for your node has volumes that will be automatically attached because their attachment was baked into the image. Specifying the devices by hand can also come in useful when the ordering of devices could have special meaning.

**Ephemeral Mounting**

By default, all of the ephemral devices for a node will be automatically attached (see volume section above) and then RAIDed with using RAID0 and mounted to /mnt. This is the suggested way of using ephemeral devices within CycleCloud, however you can override the
default behavior if necessary. All ephemeral devices are automatically assigned a Mountpoint of ephemeral so you can use this default behavior to customize the mountpoint as follows:

```yaml
[configuration cyclecloud.mounts.ephemeral]
mountpoint = /mnt/ephemeral
fs_type = ext4
raid_level = 1
```

The above configuration will instruct CycleCloud to combine all the ephemeral device using RAID1, format them uses ext4 and them mount them at the alternative location of /mnt/ephemeral which different from the default of /mnt.

**Mounting Configuration Options**

- **mountpoint**
  
  The place where the device(s) will be mounted after any additional configuration is applied. If a mountpoint is not specified, the name of the mount will be used as part of the mountpoint. For example, if your mount was named 'data', the mountpoint would default to '/media/data'.

- **options**
  
  Any non-default options to use when mounting the device.

- **fs_type**
  
  The filesystem to use when formatting and/or mounting. Available options: ext3, ext4, xfs. CentOS 5 will use ext3 by default, all other OS's will use ext4.

- **size**
  
  The size of the filesystem to create when formatting the device(s). Omitting this parameter will use all the space on the device. Can be values such as 150M for 150 megabytes, or 200G for 200 Gigabytes, or percentages like 100% to use all of the available space.

- **disabled**
  
  If true, the mountpoint will not be created. Useful for quick toggling of mounts for testing and to disable automatic ephemeral mounting. Default: false.

- **raid_level**
  
  The type of RAID configuration to use when multiple devices/volumes are being used. Defaults to a value of 0, meaning RAID0 but other raid levels can be used such as 1, or 10

- **raid_device_symlink**

20
When a raid device is created, specifying a this attribute will create a symbolic link to the raid device. By default this attribute is not set and therefore no symlink is created. This should be set in cases where you need access to the underlying raid device.

**devices**

This is a list of devices that should compose the mountpoint. In general this shouldn't need to be specified as CycleCloud will set this for you based on `[[volume]]` sections, however you can manually specify the devices if so desired.

**vg_name**

Devices are configured on Linux using the Logical Volume Manager (LVM). The volume group name will be automatically assigned, but in cases where a specific name is used this attribute can be set. Defaults to `cyclecloud-vgX` where X is an automatically assigned number.

**lv_name**

Devices are configured on Linux using the Logical Volume Manager (LVM). This value will be automatically assigned and should not need to be used, but in the case you want to use a custom logical volume name it can be specified using this attribute. Defaults to `lv0`

**order**

By specifying an order, you can control the order in which mountpoints are mounted. The default order value for all mountpoints is 1000, except for 'ephemeral' which is 0 (ephemeral is always mounted first by default). You can override this behavior on a case by case basis as needed.

**encryption.bits**

The number of bits to use when encrypting the filesystem. Standard values are 128 or 256 bit AES encryption. This value is required if encryption is desired.

**encryption.key**

The encryption key to use when encrypting the filesystem. If omitted, a random 2048 bit key will be generated. The automatically generated key is useful for when you are encrypting disks that do not persist between reboots, for example encrypting ephemeral devices.

**encryption.name**

The name of the encrypted filesystem, used when saving encryption keys. Defaults to `cyclecloud_cryptX` where X is an automatically generated number.

**encryption.key_path**

The location of the file the key will be written on disk to, defaults to `/root/cyclecloud_cryptX.key` where X is an automatically generated number.
Mounting Configuration Defaults

There are times when many different mountpoints are defined and specifying the same options over and over becomes tedious. The following options will allow you to set the system defaults for various and will be used unless otherwise specified in the individual mountpoint.

**cyclecloud.mount_defaults.fs_type**
- The filesystem type to use for mounts if not otherwise specified. Default: ext3/ext4 depending on the platform.

**cyclecloud.mount_defaults.size**
- The default filesystem size to use if not otherwise specified. Default: 50 gigabytes.

**cyclecloud.mount_defaults.raid_level**
- The default raid level to use if multiple devices are assigned to the mountpoint. Default: 0 (RAID0)

**cyclecloud.mount_defaults.encryption.bits**
- If specified, is used as the default encryption level unless otherwise specified. Default: undefined

Configuring NFS Mounts and Exports

CycleCloud provides built-in support for exporting, mounting, and configuring simple NFS filesystems.

Creating an NFS Export

To export a directory from a node as a shared NFS filesystem, provide a mount configuration section with `type=nfs` and an export path. For example:

```yaml
[[configuration cyclecloud.exports.nfs_data]]
type = nfs
export_path = /mnt/exports/nfs_data
```

The above configuration `cyclecloud.exports.nfs_data` specifies that you are configuring directory `/mnt/exports/nfs_data` to be exported as an NFS filesystem named `nfs_data`. The attributes within the configuration section describe the exported filesystem properties.

Mounting an NFS Filesystem

To mount an existing NFS filesystem::
The mounted NFS filesystem may be exported from a node in the same CycleCloud cluster, exported from a node in another CycleCloud cluster, or a separate NFS filesystem that allows simple mounts. If the filesystem is exported from a node in the local cluster, then CycleCloud will use search to discover the address automatically. If the filesystem is exported from a different CycleCloud cluster, then the mount configuration may specify attribute `cluster_name` to instruct CycleCloud to search the cluster with that name:

```
[[configuration cyclecloud.mounts.other_cluster_fs]]
type = nfs
mountpoint = /mnt/exports/other_cluster_fs
cluster_name = filesystem_cluster
```

Finally, to specify the location of the filesystem explicitly (required for mounting non-CycleCloud filesystems), the mount configuration may specify attribute `address` with the hostname or IP of the filesystem. For example:

```
[[configuration cyclecloud.mounts.external_filer]]
type = nfs
mountpoint = /mnt/exports/external_filer
address = 54.83.20.2
```

**Default Shares**

By default, most CycleCloud cluster types include at least one shared drive mounted at `/shared` and `/mnt/exports/shared`. For clusters that need a simple shared filesystem, this mount is often sufficient.

Many cluster types also include a second NFS mount at `/sched` and `/mnt/exports/sched` which is reserved for use by the chosen scheduler. In general, this mount should not be accessed by applications.

The mount configurations for the default shares reserve filesystem names `configuration.mounts.shared` and `configuration.mounts.sched`. Modifying the default configurations for these shares is possible, but may result in unexpected behaviour since many cluster types rely on the default mounts.

**Disabling NFS Mounts**

CycleCloud NFS mounts may be disabled by setting the `disabled` attribute to true.

Even the default shares may be disabled this way. For example:
Export Configuration Options

type
REQUIRED The type attribute must be set to nfs for all NFS exports. This is required in order to differentiate from other shared filesystem types.

export_path
The local path to export as an NFS filesystem. If the directory does not exist already, it will be created.

owner
The user account that should own the exported directory.

group
The group of the user that should own the exported directory.

mode
The default filesystem permissions on the exported directory.

network
The network interface on which the directory is exported. Defaults to all: *

sync
Synchronous/asynchronous export option. Defaults to true.

writable
The ro/rw export option for the filesystem. Defaults to true.

options
Any non-default options to use when exporting the filesystem.

Mount Configuration Options

type
REQUIRED The type attribute must be set to nfs for all NFS mounts. This is required in order to differentiate from volume mounts and other shared filesystem types.

mountpoint
The location where the filesystem will be mounted after any additional configuration is applied. If the directory does not already exist, it will be created.

export_path
The location of the export on the NFS filer. If an export_path is not specified, the mountpoint of the mount will be used as the export_path.

**cluster_name**

The name of the CycleCloud cluster which exports the filesystem. If not set, the node's local cluster is assumed.

**address**

The explicit hostname or IP address of the filesystem. If not set, search will attempt to find the filesystem in a CycleCloud cluster.

**options**

Any non-default options to use when mounting the filesystem.

**disabled**

If set to `true`, the node will not mount the filesystem.

### Enabling A Return Proxy

Various features of CycleCloud (status reporting, scaling requests, etc) require the ability for instances in the cloud to access CycleServer. If your instance of CycleServer lies behind a firewall those features won't work without either forwarding various ports through the firewall, using a VPN like Amazon's VPC or enabling a Return Proxy.

To enable a Return Proxy, first the cluster nodes will need access to the Return Proxy node on ports 37140-37141. In AWS this can be accomplished by adding a Security Group rule to allow those ports from within your own cluster.

Second, you need to declare a node as the Return Proxy by setting `IsReturnProxy` equal to `True`.

Finally, you need to define `KeyPair`, `KeyPairLocation`, and `Username` for the Return Proxy node.

The Return Proxy is assumed to be running within the same Cloud Provider network as the cluster so by default the private network address of the proxy is used for communication. If you need to run your Return Proxy outside of the cluster network (in a different region or on a separate Cloud Provider) you will need to tell the Return Proxy to use the public address. You can do this by setting `ReturnProxyAddress` to `<public>`. The default case is sufficient for the majority of configurations.

An example Return Proxy node might look like the following:

```text
[[node master]]
IsReturnProxy = true  # access to CycleServer is proxied through this node
```
Only one node may be declared the Return Proxy of the cluster; if multiple are defined the cluster will not start. This setting is only currently supported on CentOS, Ubuntu, and Suse nodes. It is currently recommended that your Return Proxy be configured on your scheduler/master/head node.

Tagging Nodes

CycleCloud will automatically create and add two tags to each node: a name and the cluster name. These tags are meant to make it easier to see which nodes belong to which clusters when using non-CycleCloud tools. For example, a cluster named "Demo" with a node called "master" would have the following tags created automatically:

```
Name => "Demo: master"
ClusterName => "Demo"
```

You can also create additional tags to assign to the instance by specifying them within a node definition inside your template:

```
[cluster Demo]
[[node master]]
  tags.Owner = john smith
  tags.CustomValue = 57
```

Creating a node with this definition will result in two additional tags being set on the node:

```
Name => "Demo: master"
ClusterName => "Demo"
Owner => "john smith"
CustomValue => "57"
```

Note

Support for tagging, including the number of tags or their length, may be limited or not available on some cloud providers.
Cluster templates can contain parameters allowing one to alter the values of certain parts of a cluster without having to modify the template itself. This is particularly useful in cases where many similar clusters with minor difference is desired: for example deploying development and production environments. The syntax for specifying a parameter within a cluster template is to prefix a variable with a '$'. A basic template example (non-functional) with some parameters could look like the following:

```
# template.txt
[cluster gridengine]

[[node master]]
MachineType = $machine_type

[[configuration]]
gridengine.slots = $slots
```

This template defines two parameters: $machine_type and $slots. Using this template, we can then define text files containing the values of the parameters in both the dev and prod environments. The parameters file can be in either JSON format or a Java properties file format, for example:

```
# dev-params.json
{
  "machine_type": "m1.small",
  "slots": 2
}
```

```
# prod-params.properties
machine_type = m1.4xlarge
slots = 8
```

In this example, we have created a JSON file containing the parameters for dev and a .properties file containing the values for production. We can now import the template using the parameters file to fill in the missing pieces:

```
$ cyclecloud import_cluster gridengine-dev -f template.txt -p dev-params.json -c gridengine
$ cyclecloud import_cluster gridengine-prod -f template.txt -p prod-params.properties -c gridengine
```

It is also possible to define some or all of the parameters within the cluster template itself; for example the same cluster template could be written as follows:

```
# template.txt
[cluster gridengine]
```
The default values for each parameter are defined within the template (we used the 'dev' values as defaults). It is now possible to import the template without a parameters file and the 'dev' values will be used automatically. When it is time to create a 'prod' cluster, we can use the prod-params.properties file to overwrite the values specified inside the template file itself.

Note
Parameter names can include any of the following characters: letters, numbers, and underscores.

Parameter references in the template can take one of two forms:

$param
This uses the value of a single parameter named param.

${expr}
This evaluates expr in the context of all parameters, which lets you compute dynamic values. For example:

Attribute = ${a > b ? a : b} * 100

This would take the larger of two parameters, a and b, and multiply it by 100. The expression is interpreted and evaluated according to the ClassAd language specification.

If a parameter reference exists by itself, the value of the parameter is used, which supports non-string types like booleans, integers and even nested structures such as lists. However, if the reference is embedded in other text, its value is converted into a string and included in the string. For example, suppose param is defined as 456, and referenced in two places:
$\text{Attribute1} = \$param$

$\text{Attribute2} = 123\$param$

Then the value of $\text{Attribute1}$ would be the number 456, but the value of $\text{Attribute2}$ would be the string "123456".

Note that $\${\text{param}}$ is identical to $\$param$, which allows you to embed parameter references in more complex situations:

$\text{Attribute3} = 123\$param789$

$\text{Attribute4} = 123\${\text{param}}789$

$\text{Attribute3}$ would look for the parameter named $\text{param789}$, but $\text{Attribute4}$ would use the value of $\text{param}$ to get "123456789".

**Lookup Tables**

You can have one parameter reference another and compute a certain value with a lookup table. For example, suppose you have a parameter for the image to use, with two choices in this case:

```
[[\text{parameter MachineImage}]]
Label = Image
DefaultValue = ami-1000
Description = CentOS 5.10
Config.Plugin = pico.control.AutoCompleteDropdown
[[[\text{list Config.Entries}]]]
Name = ami-1000
Label = CentOS 5.10
[[[\text{list Config.Entries}]]]
Name = ami-2000
Label = CentOS 6.5
```

And suppose further that you need to get the OS version of the chosen image and use it for other configuration. In that case, you can make a parameter whose value is a lookup table of values:

```
[[\text{parameter AmiLookup}]]
ParameterType = hidden
[[[\text{record DefaultValue}]]]
ami-1000 = CentOS_5.10
ami-2000 = CentOS_6.5
```

Note that it is hidden so that it does not appear in the UI. You can then get the OS version used for the chosen image anywhere else in the cluster definition:
GUI Integration

Defining parameters within the cluster template enables one to take advantage of the CycleCloud GUI (Graphical User Interface). As an example, when defining parameters the following attributes can be used to assist in GUI creation:

```yaml
# template.txt
[[node gridengine]]

[[[node master]]]
MachineType = $machine_type

[[configuration]]
gridengine.slots = $slots

[parameters]
[[parameter machine_type]]
DefaultValue = m1.small
Label = Machine Type
Description = MachineType to use for the Grid Engine master node
ParameterType = Cloud.MachineType

[[parameter slots]]
DefaultValue = 2
Description = The number of slots for Grid Engine to report for the node
```

Notice that "Label" and "Description" attributes are included which will appear in the GUI as well as the optional "ParameterType" attribute. The "ParameterType" allows custom UI elements to be displayed. In the example above the "Cloud.MachineType" value will display a dropdown containing all of the available machine types. The other ParameterType values are as follows:

- **Cloud.MachineType**
  - Displays a dropdown containing all available machine types

- **Cloud.Credentials**
  - Displays a dropdown containing all of the available credentials

- **Cloud.Region**
  - Displays a dropdown containing all available regions

- **AWS.Keypair**
  - Displays a dropdown containing all available AWS keypairs available for use
Chef Server Support

It is possible to use cyclecloud together with ChefServer, Create the file `chefserver.json` and place your credentials into it. ValidationKey corresponds to the validation.pem file for your chef server. You also must prove the validation_client_name if you have changed it from the default value of "chef-validator"

```json
{
    "AdType" : "Cloud.Locker",
    "ValidationKey" : "YOURVALIDATION.PEMHERE",
    "ValidationClientName" : "chef-validator",
    "Credentials" : "default",
    "Location" : "https://mychefserver",
    "ChefRepoType" : "chefserver",
    "LockerType" : "chefrepo",
    "Name" : "chefrepo",
    "AccountId" : "default",
    "Shared" : false
}
```

Next, place the file in the directory `/opt/cycle_server/config/data`. It will be imported automatically.

Command Reference

The following sections will describe the commands which come bundled with the CLI tool that will allow you to interact with clusters on CycleCloud. To see a listing of all available commands you can run:

```
$ cyclecloud help
```

To get inline help for any command you can type `cyclecloud <COMMAND> -h`, for example:

```
$ cyclecloud import_cluster -h
Usage: cyclecloud import_cluster CLUSTER [options]

Imports a cluster from StarCluster.

Options:
  -h, --help                     show this help message and exit
  -c TEMPLATE                    The template name to use. If not specified, the
default is used.
  --sc                          If specified, the template is parsed as a StarCluster
```
initialize

The initialize command will ask about how to connect to your CycleCloud instance as well as your AWS account information so that a user can be created to spin up clusters.

For more information about the initialize command, please see Initialize the CLI.

config

The config command helps you switch easily between different cyclecloud configurations. You likely already have a cyclecloud configuration in place. To name the configuration that you currently have in place at ~/.cycle/config.ini:

```
$ cyclecloud config rename dev.cyclecloud.com
```

Cyclecloud will create ~/.cycle/cyclecloud directory, move the current config to ~/.cycle/cyclecloud/dev.cyclecloud.com, and create a symlink ~/.cycle/config.ini that points to to ~/.cycle/cyclecloud/dev.cyclecloud.com

If I later feel dev.cyclecloud.com is too much to type, I can easily change the name of the configuration:

```
$ cyclecloud config rename dev
```

Assume that we have a new CycleCloud install for QA purposes, let's create the configuration file for that:

```
$ cyclecloud config create qa.cyclecloud.com
```

cyclecloud config will remove the current symlink at ~/.cycle/config.ini, create ~/.cycle/cyclecloud/qa.cyclecloud.com, and create a symlink ~/.cycle/config.ini that points back to ~/.cycle/cyclecloud/qa.cyclecloud.com

To show the current configuration:
$ cyclecloud config show
acme-prod : url = https://54.33.11.5:8443

To see all available configurations:

$ cyclecloud config list
  acme-prod : url = https://54.33.11.5:8443 [CURRENT]
  acme-qa : url = https://54.33.11.9:8443
  acme-dev : url = https://54.33.11.44:8443

To change the configuration in use:

$ cyclecloud config use NAME

While the examples above are all for *nix platforms, the cyclecloud config command also works on windows.

import_cluster

The import command allows you to import or update cluster configurations from a cluster definition file. Some basic templates are installed by the CLI tools during the initialize step (found in ~/.cycle/) that you can use or modify to suit your own needs.

For example:

$ cyclecloud import_cluster demo --file ~/.cycle/sge_templates.txt -c sge
Creating cluster demo....
----
demo
----
Keypair: cyclecloud
Cluster nodes:
  master: off
Total nodes: 1
Creating cluster demo....

The above command imports a cluster named demo using the cluster named sge in the built-in template file ~/.cycle/sge_templates.txt. If you already have a cluster with the same name as the one you are creating you will get an error to prevent accidental updates to existing clusters. If you are sure that you want to update the cluster's configuration with the new template you can use the --force to make the action go through. Note that you cannot currently change the size of an existing cluster by re-importing a template.

Multiple cluster templates or multiple copies of the same cluster template may be imported simply by choosing separate names.
For example:

```
$ cyclecloud import_cluster demo_copy --file ~/.cycle/sge_templates.txt -c sge
Creating cluster demo_copy....
--------
demo_copy
--------
Keypair: cyclecloud
Cluster nodes:
  master: off
Total nodes: 1
Creating cluster demo....
```

StarCluster support

The import command also supports importing from files in StarCluster format with the `--sc` argument. Note: the `volume`, `permission`, and `plugin` sections are not currently supported, and the resulting cluster is not configured in the same way. CycleCloud supports the following extensions to StarCluster:

**subnet_id**

If given, the nodes will be launched in the given VPC subnet.

**security_groups**

This is a comma-separated list of security group IDs to assign to the nodes.

**master_bid_price/node_bid_price**

If specified (in dollars), this creates spot instance requests for the master or execute nodes. Note: since spot instances can be terminated suddenly, they are not recommended for the master node.

**master_elastic_ip**

If specified, the master will use the given Elastic IP.

**start_cluster**

The start command analyzes the cluster to determine what instances to create and how they should be configured, and issues the appropriate AWS commands.

For example:

```
$ cyclecloud start_cluster demo
Starting cluster demo....
----
demo
----
Keypair: cyclecloud
```
Cluster nodes:
  master:  Launching instance
Total nodes: 1

Note that when a cluster is started, its nodes will be in a pending state but you will have access to the AWS instance ID. The other information about each of the nodes will become available as the requests are fulfilled. To get an updated status on the state of the cluster that was just started, see the show_cluster command.

show_cluster

If no cluster name is specified, this command will list all the clusters managed by CycleCloud along with their details.

For example:

```
$ cyclecloud show_cluster
-----
demo
-----
Keypair: cyclecloud
Cluster nodes:
  master:  running i-433c413f ec2-75-101-218-237.compute-1.amazonaws.com (10.96.214.97)
Total nodes: 1

--------
demo_copy
--------
Keypair: cyclecloud
Cluster nodes:
  master:  off
Total nodes: 1
```

The result of this show_cluster command shows that there are two clusters available. One named 'demo' which has been started and one named 'demo_copy' that is not currently running.

Optionally, the show_cluster command takes the name of a cluster and displays the details about all the nodes in that cluster. When a cluster is starting up, it is useful to run this command to check the cluster's state to see when it becomes ready for use.

For example:

```
$ cyclecloud show_cluster demo
-----
demo
-----
Keypair: cyclecloud
Cluster nodes:
```
When the cluster is up and running you will be able to see detailed information about the cluster such as the availability zone the cluster is running in, the keypair it is using (if any) and each of the nodes in the cluster along with their state, instance ID, public hostname, and internal IP address.

**add_node**

The add command allows you to add more nodes to a cluster. You can specify a template to use for the new nodes by using the `-t` option and the number of nodes to add using the `c` option. If these options are not specified the template name defaults to `execute` and a count of 1 will be used.

The add command also provides a `--fixed` command line option that instructs CycleCloud to permanently add the nodes to the cluster's template. "Fixed" nodes become part of the cluster template, and will be restarted automatically after cluster termination and restart. If the nodes are not "Fixed" (default), then the nodes are automatically removed when the cluster is terminated.

To add 3 execute nodes to the demo cluster for example:

```
$ cyclecloud add_node demo -t execute -c 3 --fixed
Adding nodes to cluster demo....
----
demo
----
Keypair: cyclecloud
Cluster nodes:
  master: running i-433c413f ec2-75-101-218-237.compute-1.amazonaws.com (10.96.214.97)
Cluster node arrays:
  execute: 3 instances, 3 cores, Allocation (AWS.RunInstances/default/us-east)
Total nodes: 4
```

To see the host information for individual nodes in node arrays, use the `-l` argument with the `show_cluster` command. For example:

```
$ cyclecloud show_cluster demo -l
----
demo
----
Keypair: cyclecloud
Cluster nodes:
  execute-1: running i-58bc7232 ec2-50-16-61-74.compute-1.amazonaws.com (10.232.47.229)
  execute-2: running i-56bc723c ec2-54-226-86-249.compute-1.amazonaws.com (10.171.18.169)
  execute-3: running i-54bc723e ec2-23-20-151-71.compute-1.amazonaws.com (10.233.21.43)
  master: running i-433c413f ec2-75-101-218-237.compute-1.amazonaws.com (10.96.214.97)
Total nodes: 4
```
reboot_node
The reboot command forces an running node to reboot without terminating the instance.

To reboot an execute node in the demo cluster for example:

```
cyclecloud reboot_node demo execute-1
Rebooting node execute-1 in cluster demo...
----
demo
----
Keypair: cyclecloud
Cluster nodes:
  master: running i-433c413f ec2-75-101-218-237.compute-1.amazonaws.com (10.96.214.97)
Cluster node arrays:
  execute: 3 instances, 3 cores, Started
Total nodes: 4
```

terminate_node
The terminate command can be used to shutdown nodes in a running cluster. If the running
node was a "Fixed" node (or part of the cluster template), it will transition to the "off" state. If
the running node was auto-started or added without the "Fixed" attribute, it will be
automatically removed upon termination.

For example to shutdown a node in the demo cluster:

```
cyclecloud terminate_node demo execute-1
Terminating node execute-1 in cluster demo....
----
demo
----
Keypair: cyclecloud
Cluster nodes:
  master: running i-433c413f ec2-75-101-218-237.compute-1.amazonaws.com (10.96.214.97)
Cluster node arrays:
  execute: 1 instances, 1 cores, Termination (AWS.TerminateInstances)
  execute: 2 instances, 2 cores, Started
Total nodes: 4
```

And after a short delay:

```
$ cyclecloud show_cluster demo -l
----
demo
----
Keypair: cyclecloud
cluster nodes:
  execute-1: off
  execute-2: running i-56bc723c ec2-54-226-86-249.compute-1.amazonaws.com (10.171.18.169)
  execute-3: running i-54bc723e ec2-23-20-151-71.compute-1.amazonaws.com (10.233.21.43)
  master: running i-433c413f ec2-75-101-218-237.compute-1.amazonaws.com (10.96.214.97)
Total nodes: 4
```
The terminate node command also supports advanced filtering options to perform bulk removal of nodes. The powerful --filter and --instance-filter options allow you to filter nodes and instances respectively using an arbitrary filter expression written in the classad expression format.

The --creds option also allows you to remove all the nodes started with a given credential name.

For example, to terminate all the nodes with the template 'execute' at once:

```
$ cyclecloud terminate_node sge --filter 'Template=="execute"'
```

The following nodes matched your filter in cluster demo:

<table>
<thead>
<tr>
<th>Name</th>
<th>ClusterName</th>
<th>InstanceId</th>
<th>MachineType</th>
<th>PublicHostname</th>
<th>SecurityGroups</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute-2</td>
<td>demo</td>
<td>i-56bc723c</td>
<td>m1.small</td>
<td>ec2-54-211-138-249.compute-1.amazonaws.com</td>
<td>default</td>
</tr>
<tr>
<td>execute-3</td>
<td>demo</td>
<td>i-54bc723e</td>
<td>m1.small</td>
<td>ec2-75-101-173-88.compute-1.amazonaws.com</td>
<td>default</td>
</tr>
</tbody>
</table>

Do you wish to terminate these instances? [y/N]

Typing 'y' will then terminate all the nodes listed.

**start_node**

The start command is used to re-start individual nodes that are part of a running cluster, but currently in the "off" state. Nodes may be in the "off" state because the underlying instance failed or because the node was previously terminated using the "terminate_node" command.

For example, to restart "execute-1" terminated above:

```
$ cyclecloud start_node demo execute-1
Starting node execute-1 in cluster demo...
```

**remove_node**

The remove command terminates (if running) and removes a node from a cluster. The remove command is used to modify an loaded cluster template by removing nodes without requiring a re-import.

Note that the difference between terminate_node and remove_node, is that remove_node permanently removes the node from the cluster template rather than simply shutting the node down.
For example to remove a node from the demo cluster:

```bash
$ cyclecloud remove_node demo execute-1
Removing node execute-1 in cluster demo....
-----
demo
-----
Keypair: cyclecloud
Cluster nodes:
    master: running i-433c413f ec2-75-101-218-237.compute-1.amazonaws.com (10.96.214.97)
Cluster node arrays:
    execute: 1 instances, 1 cores, Termination (AWS.TerminateInstances)
    execute: 2 instances, 2 cores, Started
Total nodes: 4
```

And after a short delay:

```bash
$ cyclecloud show_cluster demo -1
-----
demo
-----
Keypair: cyclecloud
Cluster nodes:
    execute-2: running i-56bc723c ec2-54-226-86-249.compute-1.amazonaws.com (10.171.18.169)
    execute-3: running i-54bc723e ec2-23-20-151-71.compute-1.amazonaws.com (10.233.21.43)
    master: running i-433c413f ec2-75-101-218-237.compute-1.amazonaws.com (10.96.214.97)
Total nodes: 3
```

The remove node command also supports advanced filtering options to perform bulk removal of nodes. The powerful --filter and --instance-filter options allow you to filter nodes and instances respectively using an arbitrary filter expression written in the classad expression format.

The --creds option also allows you to remove all the nodes started with a given credential name.

For example, to remove all the nodes with the template 'execute' at once:

```bash
$ cyclecloud remove_node sge --filter 'Template==="execute"'
The following nodes matched your filter in cluster demo:

<table>
<thead>
<tr>
<th>Name</th>
<th>ClusterName</th>
<th>InstanceId</th>
<th>MachineType</th>
<th>PublicHostname</th>
<th>SecurityGroups</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute-2</td>
<td>demo</td>
<td>i-56bc723c</td>
<td>m1.small</td>
<td>ec2-54-211-138-249.compute-1.amazonaws.com</td>
<td>default</td>
</tr>
<tr>
<td>execute-3</td>
<td>demo</td>
<td>i-54bc723e</td>
<td>m1.small</td>
<td>ec2-75-101-173-88.compute-1.amazonaws.com</td>
<td>default</td>
</tr>
</tbody>
</table>

Do you wish to remove these instances? [y/N]
```

Typing 'y' will then remove all the nodes listed.

**terminate_cluster**

The terminate command terminates all nodes in the cluster and cancels all spot instance requests.

For example:
$ cyclecloud terminate_cluster demo
Terminating cluster demo....
----
demo
----
Keypair: cyclecloud
Cluster nodes:
   master: Terminating instance i-433c413f
Cluster node arrays:
   execute: 2 instances, 2 cores, Termination (AWS.TerminateInstances)
Total nodes: 3

delete_cluster

The delete command will remove a non-running cluster from CycleCloud. If the cluster is
already running, you will have to execute the terminate_cluster command before
delete_cluster.

For example:

$ cyclecloud delete_cluster demo
Deleting cluster demo....

retry

In the event that an error occurs during any of the operations taking place to start a cluster
such as invalid credentials, bad machine type, security group, etc. the retry command will
re-execute the operations that failed. If you had for instance selected an invalid machine type,
m1.smaller instead of m1.small, you would want to update your template file with the
change, re-import the cluster using the --force option and then run the retry command
to re-issue failed commands.

For example:

$ cyclecloud import_cluster demo -f ~/.cycle/sge_templates.txt --force
$ cyclecloud retry demo
Retrying failed operations for cluster demo....

connect

You can connect to machines running in CycleCloud with the connect command:

cyclecloud connect NODE_NAME
where NODE_NAME is either the name of the node or the instance id to connect to. If the node name is ambiguous because you have two clusters using the same node name, you can specify the cluster with the -c CLUSTER argument.

Connecting to Windows machines is done via RDP (Remote Desktop Protocol). CycleCloud gets the password for the instance and connects to it, by running a command line for your local OS:

- **Windows** Uses the built-in mst.sc client. Note: requires Windows Vista or later.
- **OS X** Uses an rdp:// protocol handler, such as CoRD.
- **Others** Not automatically supported, but you can add your own.

You can specify this command by setting rdp_command inside the cyclecloud section of your config.ini file:

```
[cyclecloud]
# Uses a fictional rdp_client program
rdp_command = rdp_client ${HOSTNAME} ${USERNAME} ${PASSWORD}
```

Connecting to Unix instances uses SSH. If the keypair for the node is known, it is added to the ssh command line. If not, it is assumed to be already added to a running ssh-agent.

Note that the keypair is required for Windows instances as well as Unix instances, because it is used to decrypt the Administrator password. If it was specified with KeyPairLocation on the node, it is used automatically. If not, the -k PATH_TO_KEYPAIR option lets you specify it. Furthermore, when connecting to a Windows instance, the path must be available to CycleCloud.

**show_nodes**

You can query the status of nodes/instances tracked by CycleCloud across multiple clusters using the show_nodes command:

```
cyclecloud show_nodes [NODE_NAME] [FILTER]
```

where the optional NODE_NAME is either the name or instance id of a specific node to show. Instead of a node name, a FILTER may be provided to select a set of nodes to view.

**show_nodes** provides several convenient built-in filters. You can select a node by name or instance id using the node_id. You can also filter nodes by cluster (--cluster), state (--states), and/or credentials used to start the node (--creds).
For complete control, the command also provides a powerful \texttt{--filter} and \texttt{--instance-filter} options which allow you to filter nodes and instances respectively using an arbitrary filter expression written in the classad expression format.

Finally, the command also provides several output formatting options.

You can request a specific list of node attributes using the \texttt{--attrs} option. The \texttt{--summary} option requests a minimal set of data which is useful when viewing potentially large numbers of nodes. The \texttt{--long} option can be used to request the complete set of Node and Instance data. The \texttt{--output} option provides custom formatting of a selection of node and instance attributes. This can be used to format the data for parsing by a user application.

Finally, (when not using the \texttt{--output} mode), you can request output in csv, text, xml, json or tabular format using the \texttt{--format} option.

Here's an example of using a \texttt{--filter} expression to select nodes in the started state, and the \texttt{--attrs} option to limit the amount of data to a few relevant attributes:

```bash
$ cyclecloud show_nodes --filter="State == "Started"" --attrs=Name,MachineType
Name = "winnode-1"
Instance = [MachineType="m1.large"]
MachineType = "m1.large"

Name = "ad_server"
Instance = [MachineType="m1.large"]
MachineType = "m1.large"
```

Note: Both the Node and it's underlying Instance contain a MachineType attribute, so it appears in both the Node data and the contained Instance data.

In the next example, we're requesting the same attributes using the simpler \texttt{--state} option and requesting that the output be formatted as json:

```bash
$ cyclecloud show_nodes --state=Started --format=json --attrs=Name,MachineType
[ {
   "Name" : "winnode-1",
   "Instance" : {
      "MachineType" : "m1.large"
   },
   "MachineType" : "m1.large"
 }, {
   "Name" : "ad_server",
   "Instance" : {
      "MachineType" : "m1.large"
   },
   "MachineType" : "m1.large"
} ]
```
Finally, here's an example of using the `--output` formatter to get the data in an easily parsable "comma + tab" separated format:

```
$ cyclecloud show_nodes --filter='State === "Started"' --output="%(Name)s,	%(MachineType)s"
winnode-1,    m1.large
ad_server,    m1.large
```

Using Custom Machine Images with CycleCloud

CycleCloud ships with a repository of pre-created images that are sufficient and recommended for use by most users. See the CycleCloud Guide for information on using images from the repository in your clusters.

For users who need or want to use custom images, CycleCloud supports building clusters from images created and owned entirely by the user.

Amazon Machine Images

When creating cluster template files you have to specify an Image, ImageName or ImageId for each node you are going to create. Bear in mind that in EC2 each image is specific to both the OS as well as the Amazon Region the machine is in.

JetPack

CycleCloud can launch instances using any Machine Image. However, to build clusters using CycleCloud and to benefit from CycleCloud's orchestration layer, the Cycle Jetpack package must be installed on the Image. If the image does not have Jetpack installed then many of the features of CycleCloud will be lost.

Users have two options for installing Jetpack:

1. Use Images provided by CycleComputing that already have Jetpack installed
2. Customize an existing image with the Jetpack installer to create a new image

CycleCloud currently supports the following operating systems:

- CentOS 5, 6
- Ubuntu 12.04
- SLES 11
- Windows 2008
- Windows 2012

CycleCloud Image Requirements
1. Jetpack must be installed on the Image.
2. Jetpack uses Chef to configure instances launched using the Image. In order for Chef
to function, other infrastructure configuration management tools (such as CloudInit in
AWS) should be disabled.
   a. If your team already uses a configuration management tool such as Chef or
   Puppet, it is generally possible to configure the systems to work together. Please
   ask Cycle Computing support for more help in this case.
3. Port 22 (SSH) and port 3389 (Remote Desktop) for Windows should be open in the
security group (and instance firewall, if enabled) for the instance during the image
baking process.

Building a Custom Image using the AWS Console

For users accustomed to building AMIs via the AWS Console, the easiest way to start out
building custom images for CycleCloud is probably to continue using your current method or
following the method of using the AWS Console described here.

If you intend to build many AMIs or update them regularly, then you may eventually want to
switch to using CycleCloud to build images directly.

Select a Base AMI

The first step in building a new custom image in AWS is to select the base AMI and obtain its
AWS AMI ID to start from. Your organization may have an approved list of base AMIs,
CycleCloud provides a set of base AMIs, or you may select from any AMI available in the
AWS Console.

This guide refers to AMI ID ami-f0b23b98, but this should be replaced with the correct
base AMI ID.

Launch the Instance

1. Go to the **Services -> EC2** dashboard in the AWS Console and select the **AMIs**
   page.
2. Select the AMI you wish to use as your base AMI, and click **Launch** to start an
   instance with the base AMI.
   a. For details on configuring and launching an instance via the AWS Console, see:
b. For most users, using an EBS backed instance with a root volume of at least 8GB in size is a good choice.

c. If the base image is configured to use Cloud-Init, then do not attach an ephemeral drive to the instance at launch, otherwise Cloud-Init will attempt to mount the drive and add it to the fstab. If you do need the ephemeral drive to build the image, then be sure to clear out the `/etc/fstab` as described in the Clean Up section below.

d. Be sure to select a security group with the ports listed above open, and to select a security key-pair for which you have access to the private key (this guide will assume that you have used a keypair named `cyclecloud` as described in the CycleCloud quickstart).

3. After launching the instance, collect the instance id and hostname of the image builder instance.

Custom Configuration:

Once the new instance has started, use the private key to log in to the instance. For the example in this guide, the keypair allows direct access to the instance as root. If your base image uses a different default user, be sure that the user has sudo access.

Next, install any custom software and configuration that your cluster requires.

To simplify rebuilding the image later and to make it easier to switch to using CycleCloud to build images directly later on, we strongly recommend creating a shell script to do the actual software download and installation.

Chef/Cluster-Init vs. Image Baking

The default images provided by CycleCloud are very close to the base images, and all user-level software is installed and configured using Chef and/or Cluster-Init at cluster startup. This makes the images very flexible and usable by nearly any CycleCloud user. But it also shifts most software installation to launch time.

For your own images, you may be able to reduce launch times by pre-installing some software or pre-downloading the packages for software that may only be installed after launch. When making the decision what to burn into the image, here are some questions to think about:

1. Do all users of this image need this software? Or is it required by policy?
a. A great candidate for baking into the image would be software such as anti-virus that is required by policy and may be updated dynamically on each launch.

2. Can this software be updated when instances are launched?

   a. If not, the you may end up baking new images every time a new version is released.

3. Does this software require customization at install time that can only be done after instance launch?

   a. Grid-enabled software often needs to be configured with hostnames and ips of actual, running instances based on cluster search results. (You may still be able to install the software prior to baking the image, and use Chef to re-configure it at instance start-up.)

4. Does this software package belong on the root volume of the instance?

   a. If the software should be installed on EBS or the ephemeral drive, then it cannot be installed directly on the image.

For any of these, even if the software cannot easily be installed prior to image baking, you may consider including a copy of the installer for use by CycleCloud's Thunderball cookbook to avoid the download at instance startup until the installer is updated.

**Installing Jetpack**

Jetpack has no external dependencies, and it includes an easy-to-use installer for all supported platforms.

On linux, Jetpack will be installed at: `/opt/cycle/jetpack`

On Windows, Jetpack will be installed at: `C:\cycle\jetpack`

First, contact Cycle Computing support to request a copy of Jetpack for your platform, then follow the instructions below to install it in your instance.

**Windows**

To install Jetpack on Windows run the following commands from a Powershell session as Administrator:

```
PS> unzip jetpack.zip
PS> cd jetpack
PS> install.cmd
```
If you do not have 7zip or another unzip command available on the command line, you can extract jetpack.zip using Windows Explorer.

Should the installation fail for any reason, error output will be saved in install.log in the same directory as install.cmd.

Linux

To install Jetpack on Linux:

```bash
$ tar xzvf jetpack.tar.gz
$ cd jetpack
$ chmod +x install.sh
$ sudo ./install.sh    # if not running as root
```

Should the installation fail for any reason, error output will be saved in install.log in the same directory as install.sh.

**Note**

The Linux installer requires root privileges.

Clean-Up

An most important part of building a new image is to ensure that only files that you want to exist on every instance launched with the AMI exist on the instance at the time it is baked.

Prior to baking the image it is a good idea to remove the installer and all other temporary files from the instance. Otherwise, these files will be baked into the image and permanently clutter it.

Here are the a few common clean-up steps (in order):

1. Disable password based login for all users::

   ```bash
   $ passwd -l root
   ```

2. (Optional) Configure sshd_config according to your policies.
3. (Optional) Configure the instance level firewall according to your policies.
4. Remove any temporary files and installers for you custom installations.
5. Remove the Jetpack installer and install dir::

   ```bash
   $ cd /tmp
   $ rm -rf jetpack*
   ```
6. Remove any system logs that may contain sensitive data.
7. If you mounted and formatted the ephemeral drive for the instance (or let Cloud-Init do it for you), then be sure to remove the mount configuration from the /etc/fstab file.
8. Clear the bash history for all users, but in particular for root:

```
$ sudo su -
$ history -w
$ history -c
```

9. Remove the authorized key for the key-pair you used to log in: (Do this last: once you perform this step, you won't be able to log back into the instance):

```
$ rm ~/.ssh/authorized_keys
```

**Bake the New Image**

The new image is ready for baking.

Return to the AWS Console and locate the running instance in the EC2 instances list. Select the instance and select **Action -> Create Image**. Give the image an appropriate name and description, and then click the "Create Image" button.

The image creation process will take several minutes, but after that the AMI will be ready for use in CycleCloudStore the new AMI ID for use in your cluster templates.

**The Anatomy of Jetpack**

The Jetpack installer includes:

- Jetpack command line utility
- Configuration scripts to make the instance a member of a cluster
- Chef client 11
- Various other utilities used to interact with CycleCloud

All Jetpack files are installed inside a single directory.

Windows: C:\cycle\jetpack

Linux: /opt/cycle/jetpack

In addition to creating this directory, the jetpack installer creates:
• scripts that run on system startup to configure as an instance as a member of a cluster
• udev rules on Linux
• the 'cyclecloud' user with administrative privileges
• installs the HealthCheck service
• sets the environment variable CYCLECLOUD_HOME

Please note that the Jetpack installer for Windows disables the firewall, which is officially discouraged by Microsoft. We recommend you reenable it but leave open the ports detailed in the section on firewalls.

Important directories within your Jetpack installation:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>useful binaries and scripts</td>
</tr>
<tr>
<td>config</td>
<td>user defined and cluster defined configuration files and scripts</td>
</tr>
<tr>
<td>logs</td>
<td>logs generated by joining a cluster and converging the instance, of particular interest is the chef-client.log which contains the results from converging Chef recipes.</td>
</tr>
<tr>
<td>system</td>
<td>internal files. We don't recommend directly using or accessing any files in this directory as they will change significantly from release to release.</td>
</tr>
</tbody>
</table>

Firewalls and Jetpack

CycleCloud requires that the following ports be open on your instance for incoming connections:

• 22 TCP - SSH for Linux only
• 3389 TCP - RDP for Windows only

We also recommend that you do not block any outgoing connections. These requirements only apply to the most basic requirements to be part of a CycleCloud cluster. Any applications you install may have additional requirements.

The HealthCheck Service

The HealthCheck service executes user-defined scripts to determine the current health. Should any of these scripts exit with an error, the HealthCheck service will terminate the instance and log back a message to the CycleCloud web interface indicating the termination and the reason for the termination. Prior to terminating the instance, the HealthCheck service warns any locally logged on users of the imminent termination. Should you be logged on locally and...
wish that the instance not be terminated, you can delay the termination temporarily (Linux-only) or indefinitely using the `jetpack keepalive` command.

By default, HealthCheck comes with a check that terminates an instance if it is not successfully converged within the first 4 hours of its existence.

The individual checks are stored in `/opt/cycle/jetpack/config/healthcheck.d` on Linux and in `C:\cycle\jetpack\config\healthcheck.d` on Windows.

Jetpack Command Line Tool

The `jetpack` command-line tool, located at `/opt/cycle/jetpack/bin/jetpack` or `C:\cycle\jetpack\bin\jetpack`, provides a useful set of subcommands for manipulating the current instance and interacting with the CycleCloud server.

Jetpack Subcommands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jetpack config</code></td>
<td>retrieve a configuration value</td>
</tr>
<tr>
<td><code>jetpack converge</code></td>
<td>Execute a Chef converge</td>
</tr>
<tr>
<td><code>jetpack keepalive</code></td>
<td>Delay system termination by the HealthCheck Service</td>
</tr>
<tr>
<td><code>jetpack log</code></td>
<td>log a message to CycleCloud cluster UI</td>
</tr>
<tr>
<td><code>jetpack send</code></td>
<td>Send an arbitrary AMQP message to the CycleCloud server</td>
</tr>
</tbody>
</table>

`jetpack config`

This command can be used to fetch information that was passed to the instance through the original cluster template, all the system properties made available by Ohai, a subset of the the instance's own cloud provider metadata, and information about the parent CycleCloud cluster.

`jetpack converge`

This command downloads any Chef cookbooks or cluster-init scripts defined in the cluster template then executes a Chef converge and subsequently executes cluster-init.

`jetpack keepalive`

This command interacts with the HealthCheck service to delay the termination of instance due to a failing HealthCheck. Termination can be delayed for a fixed period or indefinitely. By default, termination is delayed for one hour.

To delay system termination by one hour:
$ jetpack keepalive

To disable the HealthCheck service entirely, i.e. delay termination indefinitely:

$ jetpack keepalive forever

To delay system termination by two days:

$ jetpack keepalive 2d

jetpack log

This command sends a log message back to the CycleCloud that started the node. The message will appear in cycle_server.log, the main event log, and the Cluster UI page.

Each message has two properties level and priority. The level property indicates what kind of message this is. Valid levels are ‘info’, ‘warn’, and ‘error’. The level does not indicate how important the message is. For example, some errors are trivial and some informational messages critical. Priority indicates the importance of the message. Valid priorities are ‘low’, ‘medium’, ‘high’. Only messages with a priority of medium or higher are displayed on the Cluster UI page. This is so that the UI page is not inundated with low priority messages.

To send an informational log message that will appear on the Cluster UI page:

$ jetpack log 'system is now ready'

To send a low priority log message that you do not want to appear on the Cluster UI page:

$ jetpack log 'system is now ready' --priority low

By default, messages with a level of error have a high priority. To send an error message:

$ jetpack log 'the machine cannot process jobs' --level error

To send a trivial error message:

$ jetpack log 'the machine cannot process jobs' --level error --priority low

jetpack send
This is an advanced command that you can use to send an arbitrary AMQP message to CycleCloud. It is most useful if you have created a CycleCloud plugin that can process that information.

You can send arbitrary strings or files with specified AMQP routing keys.

### Common Cookbooks Reference

CycleCloud clusters are built and configured using a combination of a base machine image, CycleCloud Cluster-Init, and the excellent Chef infrastructure automation framework.

Only very advanced CycleCloud users will need to understand how to build Chef cookbooks. However, many users will benefit from a basic knowledge of how CycleCloud uses Chef. In particular, users should understand the concept of a runlist, recipe, and Chef attributes.

### Basic Chef Concepts

Each node in a CycleCloud cluster is initialized by following a Chef runlist. The runlist is an ordered set of features or recipes to be applied to initialize the node. The recipes themselves implement the low-level system operations required to apply the feature. Cookbooks are collections of recipes that make up a feature. Cookbooks and recipes are parameterized by Chef attributes to allow further customization and configuration of the feature.

CycleCloud ships with a set of pre-defined cluster templates which can be used to provision a set of cluster types that is sufficient for many users. And, further customization is easily accomplished using Cluster-Init. So most users will never need to modify runlists or build their own recipes and cookbooks.

However, CycleCloud clusters are provisioned using a set of Common Cookbooks available to all CycleCloud clusters, and those cookbooks have a set of attributes which users may wish to customize. Some of the most commonly used attributes are documented below.

**NOTE**

Prefer Cluster Template features to direct modification of Chef attributes.

Common Cookbook attributes are subject to change. Attribute settings are commonly superceded as the features they control are made available as more general/powerful features of CycleCloud itself. If a customization is available in both the Cluster Template and via a Chef attribute, always prefer the Cluster Template method since it is the more general solution.

Please consider consulting Cycle Support (support@cyclecomputing.com) when using Chef attributes to customize your cluster.
For more information on the Opscode Chef framework itself, see the Opscode website:
http://docs.opscode.com/

Using Chef Attributes

Chef attributes configure the operation of the runlist for an individual Node or Node Array. They should be set in the Node's [[[configuration]]] sub-section. For example, to set the CycleServer Admin Password for a node configured to run CycleServer:

```erb
[[node cycle_server]]

[[[configuration]]]

  run_list = role[monitor], recipe[cyclecloud::searchable], recipe[cfirst], \\ 
             recipe[cuser::admins], recipe[cshared::client], recipe[cycle_server::4-2-x], \\ 
             recipe[cluster_init], recipe[ccallback::start], recipe[ccallback::stop]

  cycle_server.admin.pass=P@ssw0rd
```

Attribute Reference

CycleServer Cookbook

May be applied to any node configured to run a CycleServer instance.

**cycle_server.admin.name**

Set the user name for the CycleServer administrator account. Valid values: any username, default: admin

**cycle_server.admin.pass**

Set the password for the CycleServer administrator account. Valid values: any password, for example: P@ssw0rd

**cycle_server.http_port**

Set the HTTP port for CycleServer. Default: 8080

**cycle_server.https_port**

Set the HTTPS port for CycleServer. Default: 8443

Cluster User

The cluster user is a non-root, non-sudo user that can log into nodes in the cluster and do basic tasks such as creating and submitting jobs. Note: These attributes should be set to the same values for all nodes in a cluster; using node defaults is a good way to accomplish this.

**cyclecloud.shared_user.name**

Set the user name for the shared cluster user. Prior to CycleCloud 1.10 this was the 'Username' attribute on the cluster.
cyclecloud.shared_user.password

Set the password for the shared cluster user. Prior to CycleCloud 1.10 this was the 'Password' attribute on the cluster.

Scheduler Integration

The CycleCloud platform has built-in first-class support for several grid scheduling software solutions allowing for simplified resource and job management in the cloud. CycleCloud can automatically create, manage and scale several well known and widely adopted scheduling technologies including but not limited to: Open Grid Scheduler (Grid Engine), HTCondor, Torque, as well as Cycle's own Jupiter scheduler. Additional scheduler support is always being added and improved so refer to this documentation for updates.

Open Grid Scheduler (Grid Engine)

This document assumes you have a basic understanding of the Grid Engine scheduler. To learn more details on Grid Engine please see the official project website at: http://gridscheduler.sourceforge.net/

Open Grid Scheduler (Grid Engine) can easily be enabled on a CycleCloud cluster by modifying the "run_list" in the configuration section of your cluster definition. The two basic components of a grid engine cluster are the 'master' node which provides a shared filesystem on which the Grid Engine software runs. The second component of a Grid Engine cluster are 'execute' nodes which are the hosts that mount the shared filesystem and execute the jobs submitted. For example a simple Grid Engine cluster template snippet may look like:

```
[cluster grid-engine]

[[node master]]
    ImageId = ami-01d18b68  # CentOS 6.4 (pvm)
    MachineType = m1.xlarge  # 4 cores

    [[[configuration]]]
    run_list = role[sge_master_role]

[[nodearray execute]]
    ImageId = ami-01d18b68  # CentOS 6.4 (pvm)
    MachineType = m1.small  # 1 core

    [[[configuration]]]
    run_list = role[sge_execute_role]
```

Note: The role names contain 'sge' for legacy reasons when Grid Engine was a product of Sun Microsystems.
Importing and starting a cluster with definition in CycleCloud will yield a single 'master' node. Execute nodes can be added to the cluster via the 'cyclecloud add_node' command, for example adding 10 more execute nodes:

```
cyclecloud add_node grid-engine -t execute -c 10
```

**Autoscaling**

CycleCloud supports autoscaling for Grid Engine, which means that the software will monitor the status of your queue and turn on and off nodes as needed to complete the work in an optimal amount of time/cost. You can enable autoscaling for Grid Engine by adding "Autoscale=true" to your cluster definition:

```
[cluster grid-engine]
Autoscale = True
...
```

Note: For autoscaling to work, the nodes in the cloud must have a route back to the CycleCloud machine. The easiest way to accomplish this is to install your CycleCloud instance in the cloud along with the nodes it will be spinning up. Alternatively if you are in a VPC environment you can set up a route back to your machine, port forward the CycleCloud port on your router to your machine, or use the IsReturnProxy feature.

By default, all jobs submitted into the grid engine queue will run on machines of type 'execute', these are machines defined by the nodearray named execute. You are not limited to the name 'execute', nor are you limited to a single type of machine configuration to run jobs and autoscale on.

As an example, a common case may be that you have a cluster with two different node definitions one is for running 'normal' jobs that consume standard CPU while another type of job may use GPU machines. In this case you would want to independently scale your queue by both normal jobs as well as GPU jobs to make sure you have an appropriate amount of each machine to consume the work queue. An example definition would be something like:

```
[cluster grid-engine]
Autoscale = True

[[node master]]
  ImageId = ami-01d18b68  # CentOS 6.4 (pvm)
  MachineType = m1.xlarge # 4 cores

[[[configuration]]]
  run_list = role[sge_master_role]
```
In the above example, notice how there are now two node arrays: One is a 'standard' execute node array, the second is named 'gpu' providing a MachineType that has two Nvidia GPUs (cg1.4xlarge in Amazon EC2). Also notice that there are now two new items in the configuration section aside from just csge::sgeexec recipe. We add 'grid_engine.slot_type=gpu' which tells the software that these nodes should be named 'gpu' nodes and thus should only run 'gpu' jobs. The name 'gpu' is arbitrary, but a name that describes the node is most useful. We also set 'grid_engine.slots = 2' which tells the software to make sure that this type of node can only run two jobs at once (cg1.4xlarge only has 2 GPUs). By default the number of slots per node in Grid Engine will be the number of CPUs on the system which, in this case, would cause too many jobs to concurrently execute on the node.

You can verify the number of slots and slot_type your machines have by running the command:

```
-bash-4.1# qstat -F slot_type
```

Notice that there are one of each 'slot_type' that we specified (execute and gpu) and the number of slots for the 'execute' slot is 4, which is the number of CPUs on the machine. The
number of slots for the 'gpu' slot type is 2, which we specified in our cluster configuration template. The third machine is the master node which does not run jobs.

**Advanced Usage** You can do advanced things using the above configuration settings: for example if you have jobs that require a very specific amount of memory (say 10GB each). You can define an execute nodearray that starts machines with 60GB of memory, and then add in the configuration options 'grid_engine_slots=6' to ensure that only 6 jobs can concurrently run on this type of node (ensuring that each job will have at least 10GB of memory to work with). If you know the average runtime of jobs, you can define `average_runtime` (in seconds) in your job. CycleCloud will use that to start the minimum number of nodes (for example, five 10-minute jobs will only start a single node instead of five when `average_runtime` is set to 600).

**Submitting Jobs**

The most generic way to submit jobs to a Grid Engine scheduler is the command:

```
qsub my_job.sh
```

This command will submit a job that will run on a node of type 'execute', that is a node defined by the nodearray 'execute'. To make a job run on a nodearray of a different type, for example the 'gpu' node type above, we modify our submission a little bit:

```
qsub -l slot_type=gpu my_gpu_job.sh
```

This command will ensure that the job only runs on a 'slot_type' of 'gpu'.

If `slot_type` is omitted, 'execute' will be automatically assigned to the job. This mechanism that automatically assigns `slot_type`s to jobs without them can be modified by the user. A python script located at `/opt/cyclecloud/config/autoscale.py` can be created which should define a single function "`sge_job_handler`". This function receives a dictionary representation of the job, similar to the output of a 'qstat -j <jobID>' command and should return a dictionary of hard resources that need to be updated for the job. As an example, we will describe a script below which will assign a job to the 'gpu' `slot_type` if the job's name contains the letters 'gpu'. This would allow a user to submit their jobs from an automated system without having to modify the job parameters and still have the jobs run on and autoscale the correct nodes:

```
#!/usr/env python
#
# File: /opt/cyclecloud/config/autoscale.py
#
#def sge_job_handler(job):
   # The 'job' parameter is a dictionary containing the data present in a 'qstat -j <jobID>':
```
# Don't modify anything if the job already has a slot type
# You could modify the slot type at runtime by not checking this
if 'hard_resources' in job and 'slot_type' in job['hard_resources']:
    return {}

# If the job's script name contains the string 'gpu' then it's assumed to be a GPU job.
# Return a dictionary containing the new job_slot requirement to be updated.
# For example: 'big_data_gpu.sh' would be run on a 'gpu' node.
if job['job_name'].find('gpu') != -1:
    return {'slot_type': 'gpu'}
else:
    return {'slot_type': 'execute'}

The parameter 'job' passed in is a dictionary that contains the data in a 'qstat -j <jobID>' call:

```
{
    "job_number": 5,
    "job_name": "test.sh",
    "script_file": "test.sh",
    "account": "sge",
    "owner": "cluster.user",
    "uid": 100,
    "group": "cluster.user",
    "gid": 200,
    "submission_time": "2013-10-09T09:09:09",
    "job_args": ['arg1', 'arg2', 'arg3'],
    "hard_resources": {
        'mem_free': '15G',
        'slot_type': 'execute'
    }
}
```

You can use this scripting functionality to automatically assign 'slot_type's based on any parameter defined in the job such as arguments, other resource requirements like memory, submitting user, etc.

Continuing the example, if we were to submit 5 jobs of each 'slot_type':

```
qsub -t 1:5 gpu_job.sh
qsub -t 1:5 normal_job.sh
```

We would now have 10 jobs in the queue. Because of the script we defined above, the five jobs with 'gpu' in the name would be automatically configured to only run on nodes of 'slot_type=gpu'. The CycleCloud autoscale mechanism would detect that we have 5 'gpu' jobs and 5 'execute' jobs. Since the 'gpu' nodearray is defined as having 2 slots per node, CycleCloud would start 3 of these nodes (5/2=2.5 rounded up to 3). There are 5 normal jobs, since the machine type for the 'execute' nodearray has 4 CPU's each, CycleCloud would start 2 of these nodes to handle the jobs (5/4=1.25 rounded up to 2). After a short period of
time for the newly started nodes to boot and configure, all 10 jobs would run to completion and then the 5 nodes would automatically shutdown before you are billed again by the Cloud Provider.

**Configuration Reference**

The following are the Grid Engine specific configuration options you can toggle to customize functionality:

**gridengine.slots**

The number of slots for a given node to report to Grid Engine. The number of slots is the number of concurrent jobs a node can execute, this value defaults to the number of CPUs on a given machine. You can override this value in cases where you don't run jobs based on CPU but on memory, GPUs, etc.

**gridengine.slot_type**

The name of type of 'slot' a node provides. The default is 'execute'. When a job is tagged with the hard resource 'slot_type=<type>', that job will only run on a machine of the same slot type. This allows you to create different software and hardware configurations per node and ensure an appropriate job is always scheduled on the correct type of node.

**gridengine.ignore_fqdn**

Default: true. Set to false if all the nodes in your cluster are not part of a single DNS domain.

**gridengine.version**

Default: '2011.11'. This is the Grid Engine version to install and run. This is currently the default and only option. In the future additional versions of the Grid Engine software may be supported.

**gridengine.root**

Default: '/sched/sge/sge-2011.11' This is where the Grid Engine will be installed and mounted on every node in the system. This value should probably not be changed, but if it is it should be set to the same value on *every* node in the cluster.

**HTCondor**

This document assumes you have a basic understanding of the HTCondor scheduler. For more information on HTCondor, see the HTCondor Manual at: [http://research.cs.wisc.edu/htcondor/manual/latest/](http://research.cs.wisc.edu/htcondor/manual/latest/)

HTCondor can easily be enabled on a CycleCloud cluster by modifying the "run_list" in the configuration section of your cluster definition. There are three basic components of an HTCondor cluster. The first is the "central manager" which provides the scheduling and
management daemons. The second component of an HTCondor cluster is one or more schedulers from which jobs are submitted into the system. The final component is one or more execute nodes which are the hosts perform the computation. An example simple HTCondor template may look like:

```
[cluster htcondor]

[[node manager]]
ImageId = ami-1f57c276
MachineType = m1.large

  [[[configuration]]]
  run_list = role[central_manager]

[[node scheduler]]
ImageId = ami-1f57c276
MachineType = m1.large

  [[[configuration]]]
  run_list = role[condor_scheduler_role],role[filer_role],role[scheduler]

[[nodearray execute]]
ImageId = ami-1f57c276
MachineType = m1.large
Count = 1

  [[[configuration]]]
  run_list = role[usc_execute]
```

Importing and starting a cluster with definition in CycleCloud will yield a "manager" and a "scheduler" node, as well as one "execute" node. Execute nodes can be added to the cluster via the 'cyclecloud add_node' command, for example adding 10 more execute nodes:

```
cyclecloud add_node htcondor -t execute -c 10
```

**Autoscaling**

CycleCloud supports autoscaling for HTCondor, which means that the software will monitor the status of your queue and turn on and off nodes as needed to complete the work in an optimal amount of time/cost. You can enable autoscaling for HTCondor by adding "Autoscale=true" to your cluster definition:

```
[cluster htcondor]
Autoscale = True
```

...
Note: For autoscaling to work, the nodes in the cloud must have a route back to the CycleCloud Server machine. The easiest way to accomplish this is to install your CycleCloud instance in the cloud along with the nodes it will be spinning up. Alternatively if you are in a VPC environment you can set up a route back to your machine or port forward the CycleCloud port on your router to your machine.

Advanced usage If you know the average runtime of jobs, you can define average_runtime (in seconds) in your job. CycleCloud will use that to start the minimum number of nodes (for example, five 10-minute jobs will only start a single node instead of five when average_runtime is set to 600).

Autoscale nodearray By default, HTCondor will request cores from the nodearray called 'execute'. If a job requires a different nodearray (for example if certain jobs within a workflow have a high memory requirement), you can specify a slot_type attribute for the job. For example, adding +slot_type = "highmemory" will cause HTCondor to request a node from the "highmemory" nodearray instead of "execute" (note that this currently requires htcndor.slot_type = "highmemory" to be set in the nodearray's [[configuration]] section). This will not affect how HTCondor schedules the jobs, so you may want to include the slot_type startd attribute in the job's requirements or rank expressions. For example: Requirements = target.slot_type = "highmemory".

Submitting Jobs

The most generic way to submit jobs to an HTCondor scheduler is the command (run from a scheduler node):

```
condor_submit my_job.submit
```

A sample submit file might look like this:

```
Universe = vanilla
Executable = do_science
Arguments = -v --win-prize=true
Output = log/${(Cluster)}.${(Process)}.out
Error = log/${(Cluster)}.${(Process)}.err
Should_transfer_files = if_needed
When_to_transfer_output = On_exit
+average_runtime = 1500
+slot_type = "highmemory"
Queue
```

Configuration Reference
The following are the HTCondor-specific configuration options you can set to customize functionality:

- **htcondor.agent_enabled**
  - If true, use the condor_agent for job submission and polling. Default: false

- **htcondor.agent_version**
  - The version of the condor_agent to use. Default: 1.27

- **htcondor.autostop_stoptime**
  - The amount of uptime remaining (in minutes) in the billing hour for autostop to occur. Default: 5

- **htcondor.autostop_wait**
  - The minimum wait time (in minutes) for a node to be idle before autostop. Default: 20

- **htcondor.classad_lifetime**
  - The default lifetime of classads (in seconds). Default: 700

- **htcondor.condor_owner**
  - The Linux account that owns the HTCondor scaledown scripts. Default: root

- **htcondor.condor_group**
  - The Linux group that owns the HTCondor scaledown scripts. Default: root

- **htcondor.data_dir**
  - The directory for logs, spool directories, execute directories, and local config file. Default: /mnt/condor_data (Linux), C:\condor_local (Windows)

- **htcondor.ignore_hyperthreads**
  - (Windows only) Set the number of CPUs to be half of the detected CPUs as a way to "disable" hyperthreading. If using autoscale, specify the non-hyperthread core count with the Cores configuration setting in the [[node]] or [[nodearray]] section. Default: false

- **htcondor.install_dir**
  - The directory that HTCondor is installed to. Default: /opt/condor (Linux), C:\condor (Windows)

- **htcondor.job_start_count**
  - The number of jobs a schedd will start per cycle. 0 is unlimited. Default: 20

- **htcondor.job_start_delay**
  - The number of seconds between each job start interval. 0 is immediate. Default: 1

- **htcondor.max_history_log**
  - The maximum size of the job history file in bytes. Default: 20971520
htcondor.max_history_rotations
   The maximum number of job history files to keep. Default: 20

htcondor.negotiator_cycle_delay
   The minimum number of seconds before a new negotiator cycle may start. Default: 20

htcondor.negotiator_interval
   How often (in seconds) the condor_negotiator starts a negotiation cycle. Default: 60

htcondor.negotiator_inform_startd
   If true, the negotiator informs the startd when it is matched to a job. Default: true

htcondor.remove_stopped_nodes
   If true, stopped execute nodes are removed from the CycleServer view instead of being marked as "down". Default: true

htcondor.running
   If true, HTCondor collector and negotiator daemons run on the central manager. Otherwise, only the condor_master runs. Default: true

htcondor.scheduler_dual
   If true, schedulers run two schedds. Default: true

htcondor.single_slot
   If true, treats the machine as a single slot (regardless of the number of cores the machine possesses). Default: false

htcondor.slot_type
   Defines the slot_type of a node array for autoscaling. Default: execute

htcondor.update_interval
   The interval (in seconds) for the startd to publish an update to the collector. Default: 240

htcondor.use_cache_config
   If true, use cache_config to have the instance poll CycleServer for configuration. Default: false

htcondor.version
   The version of HTCondor to install. Default: 8.2.6

Auto-generated configuration file
HTCondor offers a large number of configuration settings, including user-defined attributes. CycleCloud offers the ability to create a custom configuration file using attributes defined in the cluster:

htcondor.custom_config.enabled
If true, a configuration file is generated using the specified attributes. Default: false

`htcondor.custom_config.file_name`
The name of the file (placed in `htcondor.data_dir/config`) to write. Default: `ZZZ-custom_config.txt`

`htcondor.custom_config.settings.*`
The attributes to write to the custom config file (e.g. `htcondor.custom_config.settings.max_jobs_running = 5000`)

**Note**
HTCondor configuration attributes containing a . cannot be specified using this method. If such attributes are needed, they should be specified in a cookbook or a file installed with cluster-init.

---

**Cluster Initialization**

Cluster Initialization, otherwise known as cluster-init, is a phase of the node provisioning process in every CycleCloud cluster. This phase allows you to install software and data onto your cluster in a variety of different ways. With cluster-init you can:

- Install operating system packages by name by specifying them in a text file
- Install operating system packages by file (such as an .rpm or .deb package)
- Run an executable script as the root/system user
- Sync data to the shared filesystem
- Sync data to the local scratch space

The following sections will outline how to use the various cluster-init features with examples.

---

**Cluster-Init Setup**

Cluster-init is a hierarchy of scripts, packages, and data that is stored in a remote object store, for example Amazon S3. When a cluster is defined, you can specify where the cluster-init hierarchy is located in the remote object store. While each node boots, the cluster-init files will be downloaded and executed to customize each node in your cluster.

When you configure your Amazon account for CycleCloud via the `initialize` command, you provide a name for your install which will be turned into an Amazon S3 bucket for your cluster-init files. For example, if you specify a name of 'demo' when running the `initialize` command, you would get a bucket named...
com.cyclecloud.demo.locker. Inside this bucket, you can place cluster-init files. An example of how you might structure the contents of the bucket are as follows:

```
com.cyclecloud.demo.locker/
|-- clusterinit/
|---- default/
|-------- executables/
|-------- scratch/
|-------- shared/
|-------- packages/
```

Notice that at the root level of the bucket there is a directory named `clusterinit` and under that another directory named `default`. This `default` subdirectory can be any name you choose. This is the name you will reference in your cluster configuration. Typical examples of cluster-init names will be version numbers like `3.0`, or other meaningful names like `dev` and `prod` to distinguish between different cluster-init configurations for different purposes.

Note: A cluster-init named `default` will be assumed in your cluster template if you do not specify otherwise.

After you give a cluster-init a name, there are several subdirectories under that: executables, scratch, shared, and packages, all of which have specific meanings which will be described below.

To enable cluster-init, the `cluster_init` recipe needs to be included at the end of your Chef run_list. To specify a cluster-init location in your cluster configuration template, use the attribute `ClusterInit` within a node definition. An example:

```
[cluster demo]
...
[<node defaults>]
  ClusterInit = prod  # Use the cluster-init named 'prod' (if omitted, 'default' is assumed)
  run_list = recipe[cyclecloud], recipe[cluster_init]
```

The directory structure represented by the versioned cluster-init locker will be downloaded locally on each instance in the cluster. For Linux, the directory structure is replicated under `/mnt/cluster-init`, and for Windows, the directory structure is replicated under `C:\cluster-init`. The local copy of the cluster-init structure also contains a `run` directory used to track execution of the cluster-init executables and a `log` directory which stores log output for cluster-init execution. For example, the cluster-init directory on a Linux instance will look like this:

```
/mnt/
|-- clusterinit/
|---- executables/
|---- log/
```
Executables

When a CycleCloud node comes up, it will download every file in the executables directory and then run each in lexicographical order as the root/system user. Scripts inside the executables directory can be used to customize each node in your cluster, for example by adding a user, creating directories, copying data, etc. These files are downloaded to /mnt/cluster-init/executables on Linux and C:\cluster-init\executables on Windows. Each script will be run to completion once and only once. Once a file has been executed successfully, a file with a .run extension will be created in the /mnt/cluster-init/run/executables directory on Linux or C:\cluster-init\run\executables on Windows.

You can create subdirectories inside the executables directory but files located in subdirectories will not be run. You can use subdirectories as a place to put library and support files for the scripts intended to be run, or you can put scripts that you will manually run under certain circumstances.

Note: Scripts in the executables directory are run in alphabetical order. To ensure that scripts are run in the proper order, it is common to prefix your scripts with numbers like:

| 01-setup.sh |
| 02-configure.sh |
| ... |
| 99-finalize.sh |

For example, if we wanted to add a SSH key to the authorized_keys file for the root user we could write a cluster-init executable script named 01-add-root-key.sh which contains the following:

```
#!/bin/bash
echo "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAy-INCOMPLETE_KEY" >> /root/.ssh/authorized_keys
```

This file would be located in S3 at a location of com.cyclecloud.demo.locker/cluster-init/default/executables/01-add-root-key.sh. When it is run, it will append the SSH key to the authorized_keys file and then create the file /mnt/cluster-init/run/executables/01-add-root-key.sh.run to indicate that the script has run to completion.
If the script does not run to completion, STDOUT and STDERR are stored for later debugging in the directory /mnt/cluster-init/log/executables or C:\cluster-init\log\executables with filenames similar to the one being run but appended with a timestamp. For example, if the above script had an error we would see log files named something like:

```
01-add-root-key.sh.2013-10-22T14:42:10-04:00.err
01-add-root-key.sh.2013-10-22T14:42:10-04:00.out
```

Failed scripts will **not** prevent the node from successfully converging. (A failed script is defined as one that returns a non-zero exit code.) This is the ideal behavior in almost all cases. Even if a particular executable returns 1 on success, it should be wrapped with a script that returns 0 when the executable returns 1, so that other errors can still be detected. However, if you want to toggle this behavior, set `cluster_init.fail_on_error` to false in the `[[[configuration]]]` section for the node.

**Packages**

Every node in the cluster using the cluster-init will download all the files in the packages directory. If the file is an operating system package (.deb, .rpm etc) that the operating system can install, the package will be installed. If the file is a text file with a .packages extension, the file will be read and every packaged listed per line in this file will be installed. For example we could have a file located at `com.cyclecloud.demo.locker/cluster-init/default/packages/install.packages` which contains:

```
sqlite-devel
this-is-not-a-package
```

In this case the package named `sqlite-devel` will be installed while `this-is-not-a-package` will fail to install. A file named `/mnt/cluster-init/run/packages/sqlite-devel.installed` will be created denoting that this package was sucessfully installed whereas an error will appear in `/mnt/cluster-init/log/packages` for the failed package install.

**Note**

- Installation failures will **not** prevent the cluster from starting.
- The packages feature is currently **not supported** for Windows instances.
Scratch

Files in the cluster-init scratch directory will be replicated on the local ephemeral drive of each instance. The scratch directory provides fast local access to its files, but data stored here will not be persisted when a node is terminated.

In Linux, the cluster-init scratch directory is found at /mnt/cluster-init/scratch and on Windows at C:\cluster-init\scratch for use by cluster-init executables or by applications. Each instance will sync all files in the scratch S3 directory (for example com.cyclecloud.demo.locker/clusterinit/default/scratch/) to the local /mnt/cluster-init/scratch directory.

Note that every instance in the cluster has a general purpose scratch directory on its ephemeral drive. As with the cluster-init scratch directory, data stored here will not be persisted if a node is terminated. This directory is intended to be used for temporary scratch space or local reference files. On Linux, this directory is located at /mnt/scratch and on Windows it is the C:\ drive.

Shared

Many CycleCloud clusters have a shared filesystem (usually shared from the head node in the case of Condor or Grid Engine clusters). For clusters with a shared filesystem, the cluster-init shared directory will be replicated from S3 to the filer at start-up and shared with all instances in the cluster. This directory is intended to be used for data which needs to be shared to all nodes in the cluster.

In Linux, the cluster-init shared directory is found at /mnt/cluster-init/shared and on Windows at C:\cluster-init\shared. The shared directory is synced once from the shared S3 directory onto the filer.

Note that in clusters with a shared drive, every instance has access to the general purpose shared directory as well. On Linux, this directory is located at /shared and on Windows it is the S:\ drive.

Reading Node Configuration

You can read the values sent to each node in its [configuration] section from a cluster-init script using the jetpack config command:

$ jetpack config <name of param> [optional default value]

For example, if you had a node definition where you set the Cycle Server http port to be 8000:
In your cluster-init script you may want to use iptables to route port 80 to port 8000. A simple cluster-init script may look like the following:

```
#!/bin/bash
iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT --to-port 8000
```

Rather than hardcoding the port to 8000, you can use the `jetpack` tool to pull the Cycle Server port out of the configuration so that you can change the port without modifying your cluster-init scripts:

```
#!/bin/bash
# Get the cycle_server.http port value, or 8080 (default) if not specified.
CS_PORT=$(jetpack config cycle_server.http_port 8080)
iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT --to-port $CS_PORT
```

### Tips and Tricks

- Long running scripts will delay node startup time. Try to keep your scripts running as quickly as possible.
- Executables only run once, but you can force them to re-run by deleting their corresponding run file in `/mnt/cluster-init/run/executables` and triggering another converge with `jetpack converge`.
- Many clusters will reconverge every 20 minutes, meaning they will re-run failed executable scripts, re-sync the shared and scratch directories, as well as installing any packages that failed to install previously.
- Executables and packages are run in lexical order (ie, alphabetically), so it can be useful to prefix their names with numbers to easily ensure the right ordering.
- **Windows supports cluster-init with the following notes:**
  - The packages directory is **not** supported on Windows and will be ignored.
  - Executable scripts may be placed inside a `windows` subdirectory to denote that they are Windows-specific executables.
- **The order each of the directories are processed is as follows:**
  1. shared

```
[[node master]]
  [[[[configuration]]]]
cycle_server.http_port = 8000
```
2. scratch
3. packages (Linux only)
4. executables

Amazon EC2
CycleCloud supports launching both on-demand and spot instances, as well as VPCs, EBS volumes, tagging, and placement groups.

Using Spot Instances
You can use spot instances in your cluster to reduce overall cost. In general, you can specify a bid price for each instance, which is the amount you are willing to pay per hour. If there is an instance available in your price range you will be granted that instance until the market price exceeds your bid price, at which time your instance will be terminated. Please see the official Amazon Documentation for details on how the spot market works along with the current prices of various instance types.

You can easily modify the template for a cluster to include bid prices by enabling and modifying the following option on a node-by-node basis in the cluster template:

```

[[nodearray execute]]
BidPrice = 0.10
# Other parameters removed for brevity

```

The above directives state that execute nodes will be a spot instance with a bid price of $.10/instance hour.

When using spot instances, the output from the CLI tools detailing the state of your cluster will differ from on-demand instances based on the state of your spot request, for example:

```
$ cyclecloud start_cluster spot_cluster
Starting cluster spot_cluster....
----------
spot_cluster
----------
Zone:
Keypair: cyclecloud
Cluster nodes:
  execute-1 Waiting for spot instance request to be fulfilled
  master  Waiting for spot instance request to be fulfilled
Total nodes: 2
...

```
$ cyclecloud showcluster spot_cluster
----------
spot_cluster
----------
Zone:
Keypair: cyclecloud
Cluster nodes:
  execute-1 Waiting for instance to start running i-85dd96f9
  master  Waiting for instance to start running i-8bdd96f7
Total nodes: 2

**Warning:** Spot instances will take longer to start than on-demand instances since they have to go through the spot market fulfillment process. If your bid price is lower than the market price, your spot request may not be fulfilled for a long time, if at all.

**Working With EBS Volumes**

CycleCloud supports configuring volumes for specific Elastic Block Store (EBS) features including snapshot and ephemeral volumes. Snapshots can be used to attach common data to multiple instances at once, for example giving every execute node in a cluster the same reference data stored on a snapshot. Ephemeral volumes will automatically be attached to your instances. They exist for the life of the instance and provide a large amount of storage while the instance is running. For more information about EBS volumes, please see the official documentation.

**Customizing EBS Volumes**

The `[[volume]]` section supports several EBS-only settings:

**Type**

Specifies the type of the volume. Current options are `standard` (Magnetic), `gp2` (General Purpose SSD, the default), and `io1` ( Provisioned IOPS)

**IOPS**

Specifies the provisioned IOPS rating.

**DeleteOnTermination**

Attaches this volume with the "delete on termination" setting. This is set by default for new volumes that are not marked `Persistent=true`, but may be specified manually if needed.

**Existing Volumes**
To attach an existing EBS volume, you must know the ID of the volume (which looks like 'vol-xxxxxxxx'). You can specify that it should be attached to an instance by setting VolumeId in the `[[[volume]]]` block within your cluster template:

```
[[node master]]
[[[volume example-vol]]]       # 'example-vol' is the name of the volume within CycleCloud
VolumeId = vol-abcd1234        # This is your volume ID as defined by AWS
```

**Note**

The instance will automatically be placed in the same availability zone as your volume, because AWS will not let you attach volumes to an instance in a different zone. If you specify more than one volume by ID, they must all be in the same zone. If you specify a subnet manually, it must be in the same zone as all volume ids.

**EBS Snapshots**

You can also create a volume from an existing EBS snapshots. This will create a volume on the instance from the snapshot. As with EBS volumes, you will have to create the snapshot from an existing volume in the AWS console. Once you have created a snapshot, you specify SnapshotId in a `[[[volume]]]` section in your template:

```
[[nodearray execute]]
[[[volume reference-data]]]    # 'reference-data' is the name of the volume within CycleCloud
SnapshotId = snap-1234abcd
```

EBS snapshots are available across an entire region, so you can attach a snapshot to an instance in any zone. As with EBS volumes, you can specify a device to attach to or CycleCloud will automatically pick one for you.

**Note**

If you also specify a size, it must be at least as large as the snapshot. If you do not specify a size, the size of the snapshot is used.

**Warning**

You can ensure automatic volumes are kept after instance termination if you specify Persistent=True. However, you may end up with a large number of volumes, increasing your cloud-provider bill. This option is for advanced use only.
Ephemeral/Instance Storage

Please note that this is an advanced topic and should normally not be needed. Each AWS instance may come with built in local disks, or ephemeral/instance storage. This is storage that is available only for the life of the instance, and when the instance is terminated the data is lost. CycleCloud will automatically detect and attach all ephemeral volumes for you so you should not need to specify these. However, if you need to manually specify your ephemeral attachments you can:

```yaml
[[node master]]
MachineType = m1.large   # m1.large has 2 disks

[[[volume ephemeral0]]]
EphemeralId = ephemeral0
Device = /dev/sdj

[[[volume ephemeral1]]]
EphemeralId = ephemeral1
Device = /dev/sdk
```

The above example shows how to attach both ephemeral disks available to an m1.large to devices /dev/sdj and /dev/sdk. If this mapping were not specified, CycleCloud would have automatically attached the ephemeral disks to devices /dev/sdb and /dev/sdc by default. You should almost never have to specify these mappings unless you are doing very custom disk work. For more information on ephemeral disks please see the official EC2 documentation.

**Note**

If you do not want any ephemeral automatically mapped for you, meaning you will either use no ephemeral storage or will rely on the device mapping in your AMI, you can set the DisableAutomaticEphemeral to true:

```yaml
[[node master]]
DisableAutomaticEphemeral = true  # No ephemeral disks will be automatically attached to this instance
```

Custom Security Groups

In the Quickstart guide you created and updated the default security group for use on all of your cluster nodes. You can customize your security groups from AWS and assign them to the nodes in the cluster template files. You may want to do this because the default security group is being used for another purpose, you want to open or close additional ports, etc.
You can create and modify security groups from the AWS console as you did in the Quickstart, and then inside a cluster template you can modify the security groups settings to reference your newly created security group:

```
SECURITY_GROUPS = sg-1234abcd, sg-abcd1234
```

This setting will instruct CycleCloud to start nodes using the two security groups listed. To use the default security group, comment out the line.

**Note**

You can specify different security groups for each class of node. For example your master/head nodes could use one security group that opens ports 22 and 8080 (for CycleServer access), while the worker/execute nodes use a different security group that only opens port 22 for ssh access.

**Remember**: You need to allow all nodes within your cluster to talk to each other, so make sure at least one of your security groups has TCP/UDP ports open to other members of the security group and is used by every class of node!

**Custom Key Pair**

In the quickstart you created a keypair named cyclecloud. To use a different AWS keypair, specify the name and location of the keypair in the cluster template file:

```
[[node defaults]]
# This will set the keypair for all nodes to 'custom'
KeyPair = custom
KeyPairLocation=~/.ssh/custom.pem
```

**Network Interfaces**

You can configure the standard network device or attach additional devices with the `network-interface` element:

```
[[node simple]]
MachineType = m1.small
```

```
[[[network-interface]]]
SecurityGroups = sg-12345678
PublicIp = 198.51.100.1

This configures the first network device to apply rules from security group sg-12345678 to its traffic, and use 198.51.100.1 as its public IP. (Note: this assumes that you have already allocated 198.51.100.1 in your cloud provider, and thus can use that IP.)

If you simply want to use an existing network interface, you can include its id in the section:

```
[[][network-interface]]
InterfaceId = eni-23456789
```

This will use the eni-23456789 network interface, assuming it exists and is not already attached to another instance. If you do not specify an id, then one will be created and set to be automatically deleted when the instance is terminated.

Note that you cannot specify a SubnetId with an existing interface (when InterfaceId is supplied), because the interface already has a subnet.

As an alternative to specifying a public IP manually, you can ask for one to be created for you with the AssociatePublicIpAddress parameter:

```
[[][network-interface]]
AssociatePublicIpAddress = true
```

This is only necessary for VPC, which does not assign public IP addresses by default.

VPC instances may also have a fixed Private IP address assigned to their network interfaces using the PrivateIp parameter:

```
[[][network-interface]]
PrivateIp = 10.0.1.10
```

This attribute is only valid in VPC. To use it, simply select a valid IP within your VPC’s subnet to dedicate to the associated node.

Amazon Virtual Private Cloud

CycleCloud supports launching clusters using Amazon's Virtual Private Cloud. You will have to configure your network through the AWS Console’s VPC section. Once you have configured your network, you can alter cluster templates to use your network by specifying the subnet ID to use when launching instances:
# VPC Settings
SubnetId = subnet-abcd1234

This will instruct CycleCloud to launch instances in this VPC subnet instead of public EC2.

Amazon Placement Groups

CycleCloud supports launching cluster compute class nodes in placement groups for connection intensive computing. A placement group allows the machines to be physically located very close together to take advantage of high speed interconnect between the machines making clusters like this good for MPI type workloads. You will have to create a placement group in the AWS/EC2 console, for example one named 'demo-pg'. Once you have created a placement group you can tell your execute machines to launch in this placement group:

```
[[nodearray execute]]
MachineType = cc2.8xlarge   # Cluster Compute 2 instance
PlacementGroup = demo-pg
Zone = us-east-1a
```

In the above example we are creating execute nodes using the Cluster Compute class instances, telling Amazon to place them all in the demo-pg group in the availability zone us-east-1a.

**Note**

When using placement groups you should always specify the zone you want the machines to be placed in as Amazon will refuse any requests to put machines from different zones in the same placement group.

**Note**

Cluster compute class machines require HVM (Hardware Virtual Machine) images. CycleCloud provides and supports these, but you should make sure to properly specify the image when attempting to launch Cluster Compute instances.

Using Limited Identity Access Management (IAM) Credentials

While we suggest that you use your root AWS Keys which have full access to your account for ease of setup, this level of access is actually not required for CycleCloud to function. CycleCloud requires full EC2 access (to manage compute on your behalf) and read/write
access to a single bucket in S3 (for data storage). Access to other AWS services is not needed at this time. If your AWS account is used by other people at your organization or you use other AWS services you may want to use limited IAM credentials with the CycleCloud software.

While generating IAM credentials is outside the scope of this document, you can find reference documentation online.

Creating a Bucket

If root credentials are not being used, you will have to create a bucket for use with CycleCloud since it cannot create the bucket for you. The naming scheme used by the 'initialize' command when generating buckets is 'com.cyclecloud.name.region.locker' where name is a name of your choice (for instance, it might be your organization name), and region is the AWS region, e.g. us-east-1. You will have to create a bucket following this scheme, tag it appropriately and then grant the IAM user read/write (full) access to this bucket. As an example, if we were to simply use the name 'limited' as name and region 'us-east-1' as region you would create a bucket named 'com.cyclecloud.limited.us-east-1.locker'. Once created, you need to add a single tag to this bucket so that CycleCloud can detect it. The key for the tag should be "CycleCloudAccountId" and the value of the tag should be "limited", or whatever you have chosen as name.

IAM Policy

As mentioned above, the IAM policy can be anything you want as long as it has full EC2 access as well as read/write access to the bucket created and tagged in the step above. An example IAM policy for the example above may look like:

```json
{
  "Statement": [
    {
      "Sid": "Stmt1362611954648",
      "Action": [
        "ec2:*"
      ],
      "Effect": "Allow",
      "Resource": [
        "*"
      ]
    },
    {
      "Sid": "Stmt1362611973688",
      "Action": [
        "s3:*"
      ],
      "Effect": "Allow"
    }
  ]
}
```
Initializing CycleCloud

When you initialize CycleCloud with your IAM credentials you will use the access key and secret key for the IAM account instead of the root credentials. When prompted for a region, you must specify the region identical to that used when naming the bucket. When prompted for a name, you have to specify the exact name used when creating and tagging your bucket (in the above example you would enter 'limited'). CycleCloud will then be configured to use this bucket for all data storage purposes.

Enabling Usage Tracking

AWS provides billing reports that can be programmatically accessed. Cycle uses these reports for billing, auditing and troubleshooting. Many of our customers use these reports to analyze usage data across individual clusters. We recommend that you follow these instructions to enable billing reports so that reports are collected from the beginning. They will provide you with an audit trail for all usage.

Enabling Billing Reports

Create a S3 bucket to hold the usage reports. Update the bucket's usage policy so that AWS has access to write the reports.

Follow the AWS instructions for enabling the usage reports. In short you will be following these steps:

- Select 'Receive Billing Reports'
- Enter your usage reports bucket name (and make sure that AWS has access by attaching the AWS sample policy to it)
• Select the 'Cost allocation report' and the 'Detailed billing report with tags'
• Select the appropriate tags
• Save settings

Adding Cycle Tags to Reports

You will want to add a number of tags to the billing report so that the report can be filtered by cluster and node usage. More information on tagging in AWS.

Select the tags listed on the Tags page and enable these tags:

• Name
• ClusterName
• InstanceName

Granting Cycle Access

Cycle uses these billing reports to audit their internal usage tracking. For customers that are billed based on usage we also use these reports to determine your Cycle bill.

Create a Cycle user on the users page.

• Click ‘Create New User’
• Enter "cyclecomputing" as user name
• Select the 'Generate an access key' checkbox
• Click 'Create'
• Save the access key and the secret key.
• Choose the "cyclecomputing" user
• Under 'Security Credentials' select 'Manage Password'
• Choose 'Assign Password Automatically'
• Click on 'Show Security Credentials' and save the password

Grant access to the usage bucket for the Cycle user

• Select the 'cyclecomputing' user and "Attach User Policy"
• Add the following policy and replace BUCKET_NAME with the name of your usage bucket.

```json
{
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "s3:ListAllMyBuckets",
            "Resource": "arn:aws:s3:::*"
        },
        {
            "Effect": "Allow",
            "Action": "s3:*",
            "Resource": [
                "arn:aws:s3:::BUCKET_NAME",
                "arn:aws:s3:::BUCKET_NAME/*"
            ]
        }
    ]
}
```

**Note**

Please provide the access key, secret key, username, password and sign-in link that can be found here to your Cycle Computing support contact.

**Using the Reports**

The reports are updated throughout the day so you can get a fairly accurate view into your usage. By importing the csv report into Excel and using pivot tables on the ClusterName tags you can get detailed usage information for your individual CycleCloud clusters.

**Configuring AWS VPC for CycleCloud**

This document provides a step-by-step walkthrough for configuring VPC to work with CycleCloud. It assumes you have two security groups set up (the default one, and one called "nat"). In addition, you must have a keypair set up already.

Note that this configuration is set up to allow access to S3 in the us-east-1 region, which currently consists of the following IP ranges:

• 72.21.192.0/19
• 176.32.96.0/19
• 205.251.224.0/19
• 207.171.160.0/19

The following is the IP range for S3 in the us-gov-west-1 region (GovCloud):

• 205.251.192.0/18

If you are creating your VPC in another region, you must currently enable all outgoing HTTPS traffic.

VPC
Create a VPC, if you do not have one already. We recommend using a full /16 block, for example 172.16.0.0/16. (The remainder of this document will assume this block). The "default" tenancy should be selected.

Subnets
Next you must create at least two subnets. The first subnet is the public subnets, and must contain at least one "NAT" instance. This instance will allow traffic directly to the internet.

1. Click "Subnets" on the left menu.
2. Click "Create Subnet".
3. Choose your VPC.
4. Choose "No preference" for the Availability Zone.
5. Enter 172.16.0.0/24 as the CIDR block.

The second subnet (and any additional ones) will contain the hosts that have limited access to the internet and direct access to the VPN. Create a subnet as above using 172.16.16.0/20 as the CIDR block.

Note: CycleCloud will later need to be configured with the subnet id for the private subnet (172.16.16.0/20).

Security Groups
The VPC must be configured with two security groups, one for the NAT instance in the public subnet, and the default one for all instances in the private subnet.

Note: CycleCloud will later need to be configured with the default security group id for the private subnet.

Default Security Group
The default security group used will need to be configured to allow inbound ports 22 (SSH), 80 (HTTP), 8008 (HTTP), and 8443 (HTTPS), as well as port 3389 (RDP) for Windows. All outbound ports should be opened.

**NAT Security Group**

The "nat" security group should be locked down as much as possible since it faces the public Internet. However, it must allow inbound connections from the default security group on port 443 so that instances in the private subnet can make requests to the public subnet which are then proxied to S3. It must also allow outbound connections on HTTPS for the S3 ranges above. It is also highly recommended that you allow outbound UDP connections on port 123 so that instances inside the VPC can coordinate their system clocks via NTP.

Note: If you will be connecting outside your VPN (as described in Connecting to instances in VPC), open port 22 inbound as well for 0.0.0.0/0.

**Launching the NAT**

To run a NAT instance, follow these steps:

1. Click on "VPC Dashboard" in the left menu.
2. Click "Launch EC2 Instances".
3. Click "Launch Instances".
4. Select "Classic Wizard."
5. Select the "Community AMIs" and filter to "Amazon Images."
6. Type in "nat" in the box. Click Select next to the AMI with the "amazon/ami-vpc-nat-1.1.0-beta.x86-64-ebs" manifest.
7. Choose "M1 Medium" (or other instance).
8. Select "EC2 VPC" and pick the public subnet (172.16.0.0/24).
9. Click "Continue".
10. Click "Continue".
11. Click "Continue".
12. Enter "nat" for the Value of the Name tag.
13. Choose the keypair you want to use and click "Continue".
14. Choose the "nat" security group and click "Continue".
15. Click Launch.

When the image starts running, it must have the "Source/Dest Check" disabled.

1. Click "Instances" in the left menu. The NAT instance will show up with a name of "nat".
2. Note the instance ID for later.
3. When it starts running, select it, then click "Actions" and "Change Source/Dest. Check".
Finally, we will need to attach an elastic IP address to the nat instance. Click on "Elastic IPs" in the left menu. Click on "Allocate New Address", select "VPC" from drop-down and click "Allocate". Click on "Associate Address", select the nat instance and then click "Associate"

Internet Gateway
An internet gateway allows your nondefault subnet instances to communicate with the internet.

1. Click on "Internet Gateway" in the left menu.
2. Click on "Create Internet Gateway"
3. Click on "Yes, Create"
4. Select and click on "Attach to VPC"
5. Select your VPC and click on "Yes, Attach"

Route Tables
The route tables will control the networking. You need two route tables: one for the public subnet and one for all private subnets. Do these steps twice:

1. Click "Route Tables" on the left menu.
2. Click "Create Route Table".
3. Select your VPC.
4. Click "Yes, Create".

Now we need to configure the route table for the private subnets. In particular, we are going to limit direct internet access to the S3 IP ranges above. If your region does not have a well-known set of IP ranges, you can make a single route for 0.0.0.0/0.

1. Click the first route table you created (the one marked Main).
2. Select the "Routes" tab.
3. Type the first IP range from the list (for example, 72.21.192.0/19 for us-east) in the text box, and pick the instance ID of the nat instance, then click Add.
4. Repeat for the rest of the ranges.
5. Click Associations and associate it with the private subnet (172.16.16.0/20).

Finally we need the route table for the public subnet.

1. Click the second route table you created.
2. Select the "Routes" tab.
3. Type 0.0.0.0/0 in the text box, and pick the the internet gateway (the id starting with igw-), then click Add.
4. Click Associations and associate it with the public subnet (172.16.0.0/24).

Connecting to instances in VPC
There are three ways to connect to instances running in a VPC: first, your machine is part of your organization's VPN; second, with an elastic IP to the instance, and third, through a "bastion" server.

Inside the VPN
This is the simplest method, and recommended when running in a production scenario in your own network. Instances inside the VPC are directly reachable by your machine.

Outside the VPN via an elastic IP
If you do not have a VPN or cannot extend a VPC to your VPN (for instance, while testing a new VPC), you can attach an elastic IP to the target instance. This will give that instance direct network access. Note that the security groups on that instance may be configured to allow types of traffic that are not authorized, so this is not recommended. It also allows outbound connections to bypass any firewalls you have in place, so it is not an accurate test of how your instance will behave normally (for example, installing packages via the Linux "yum" package manager will succeed because the instance has direct network access).

Outside the VPN via a bastion server
The recommended way to connect to a target instance inside the VPC is to connect through an external-facing server, or "bastion server", which has an elastic IP. This is the NAT/proxy server configured above.

These instructions assume you are using SSH, with public-key authentication. This is typical for Linux instances. For Windows, you can use this method to set up an RDP tunnel (assuming the bastion server is Linux).

First, make the private key accessible to the target instance. The simplest way is to run an SSH agent. This is run on your personal machine, the one which has your private key.

**Warning**
The private key should never leave your personal machine!

From your local machine, start the agent with the `ssh-agent` command:

```
exec ssh-agent bash
```
If the private key is not your default private key ( ~/.ssh/id_rsa or ~/.ssh/identity), add it to the agent:

```bash
ssh-add PATH_TO_KEYPAIR
```

Note: your private key never leaves your machine. The agent uses this to respond to authentication challenges sent by the remote instance.

Those commands only need to be run once (or after you reboot).

Now, when you connect to the server, tell SSH to use agent forwarding.

```bash
ssh -A -t ec2-user@BASTION_SERVER_IP ssh -A root@TARGET_SERVER_IP
```

This connects to the bastion and then immediately runs ssh again, so you get a terminal on the target instance. (The default NAT ami uses ec2-user. You may need to specify a user other than root on the target instance if your cluster is configured differently.) The -A argument forwards the agent connection so your private key on your local machine is used automatically (without ever leaving your local machine). Note that agent forwarding is a chain, so the second ssh command also includes -A so that any subsequent SSH connections initiated from the target instance also use your local private key.

**Connecting to services on the target instance**

You can use the SSH connection to connect to services on the target instance, such as Remote Desktop, a database, etc. For example, if the target instance is Windows, you can create a Remote Desktop tunnel by connecting to the target instance with a similar SSH command from above, using the -L argument:

```bash
ssh -A -t ec2-user@BASTION_SERVER_IP -L 33890:TARGET:3389 ssh -A root@TARGET_SERVER_IP
```

This will tunnel port 3389 on target to 33890 on your local machine. Then if you connect to localhost:33890 you will actually be connected to the target instance.

**Windows Azure**

Currently, CycleCloud supports starting both IaaS and PaaS instances in Windows Azure.

**Azure Endpoints**

Cyclecloud supports launching clusters with pre-defined input endpoints. The endpoints map external ports to local ports for instances within the same cloud service. At it's simplest it will open the defined local port externally:
Alternatively, if you have multiple instances in a service each one can have a defined external port:

```
[[node master]]
  [[[input-endpoint RDP]]]
  LocalPort = 3389
  ExternalPort = 55100
```

Since node arrays don’t allow for customization of individual nodes it is necessary for CycleCloud to determine the external port for each node. A `BaseExternalPort` can be defined and CycleCloud will choose an available external port above the `BaseExternalPort`. In the example below, 4 8-core execute instances would be started each with a unique external port based off `BaseExternalPort`. Assuming no other conflicts in the cloud service, the external ports would be [55100, 55101, 55102, 55103]:

```
[[nodearray execute]]
  InitialCoreCount = 32
  [[[input-endpoint RDP]]]
  LocalPort = 3389
  BaseExternalPort = 55100
```

Endpoints default to TCP protocol but UDP is also supported via `Protocol = UDP`.

**Attribute Reference**

This section describes the attributes that can be assigned to sections within a cluster template.

**Cluster Section**

These attributes are for cluster-wide settings and can be specified within a `[cluster]` block inside the template.

**Autoscale**

If true, nodes in a nodearray will be added or deleted based on demand.

**ParentName**

The name of the parent cluster if you are using a cluster-of-clusters configuration for massive scale. Valid values are any cluster name that is running and set up as a parent cluster.

**Password**
Ignored as of CycleCloud 1.10. For the same behavior, set the node level configuration attribute 'cyclecloud.shared_user.password' instead.

**Username**

Ignored as of CycleCloud 1.10. For the same behavior, set the node level configuration attribute 'cyclecloud.shared_user.name' instead.

**Node / NodeArray Section**

These attributes are for node-specific settings and can be specified within a [node] or [nodearray] block within a template. Note: You can apply these settings to all nodes via node inheritance specifying the attributes inside a [node defaults] section.

**BlackboardLocker**

The name of the blackboard locker you want to use for this node. Valid values are any locker name for your account. This attribute is automatically detected and set for you, so specifying this attribute manually should almost never need to happen.

**ChefRepoLocker**

The name of the Chef repo locker you want to use for this node. Valid values are any locker name for your account. This attribute is automatically detected and set for you, so specifying this attribute manually should almost never need to happen.

**ChefRepoVersion**

The version of the Chef cookbooks to use. When providing your own Chef cookbooks, this attribute can be set to use a version other than the default 'latest'. Any string is valid so long as it defines an actual chef version in one of your lockers.

**ClusterInit**

The name of the cluster-init that you wish to use. If not specified the name 'default' is assumed. Valid values are any string. If the cluster-init is not found, then no cluster-init will be used.

**ClusterInitLocker**

The name of the cluster-init locker you want to use for this node. Valid values are any locker name for your account. This attribute is automatically detected and set for you, so specifying this attribute manually should almost never need to happen.

**CommonChefRepoVersion**

The version of the Cycle Computing-provided chef repository if the default (stable) version is not wanted. If this attribute is not set than the system default will be used. In general, this setting should only be set if you know specific version you want to use or test.

**ConfigureScript**
The path to a script that will be run on the instance on first start.

**CoreCount**

The number of cores that this node is considered to have. Defines the number of cores possessed by this node for the purposes of autoscaling. This value overrides the default value for the cloud resource type.

**Credentials**

The name of the credentials you wish to use to start your node. If not specified (default), the default credentials for your account will be used. This attribute is used if you have more than one set of credentials set up for your CycleCloud account. For example, if you have a 'prod' account and a 'dev' account set up, you can tell all nodes in a cluster to start in 'dev' by specifying `Credentials=dev`.

**DisableAutomaticEphemeral**

Indicates that no local (ephemeral) disks should be attached to this node. If this attribute is set to true, you will have to attach your own ephemeral disks, have them configured on the image, or use no ephemeral disks at all. CycleCloud will **not** automatically generate and attach them.

**Fixed**

If false, these nodes represent only the current intention for instances and may be expanded into fewer large nodes or contracted.

**FixedPublicIp**

A persistent public IP address to assign to this instance. If you want to attach a public-facing IP to give the node a consistent public IP address you can specify the address here. Valid values are a standard IPv4 address, for example: 192.134.5.19. You typically must reserve the specific address from your cloud provider before specifying it on a node.

**Image**

Standard image to start this instance. This specifies the operating system that will run on the machine. For a list of CycleCloud images, please see the Image reference section of the documentation.

**ImageId**

Provider-specific id to use to start this instance. This defines the operating system that will run on the machine. For example, `ami-abcd1234`. This is intended for public non-CycleCloud images or custom user-defined images.

**ImageName**

Name of a standard image to start this instance. Note: this is the name of the image, not the label referenced by ImageName.
**ImageVersion**

The version specification of the image. This matches a specific version (e.g., 1.2.3) or a version pattern (e.g., 1.2.x). This is not typically recommended for standard CycleCloud images, because the correct image for your cluster is chosen automatically. Note that this is **not** the version of the operating system, which is usually included in the image name itself.

**IsReturnProxy**

If true, this node will be used as a proxy to access the controlling CycleCloud instance. The node with this enabled will be configured to allow access back to the controlling CycleServer instance. Additionally, all nodes in the cluster will automatically be configured to direct their CycleServer traffic to this node. Only one node in a cluster may have this setting enabled; otherwise the cluster will not start. If this setting is enabled it requires the KeyPair and KeyPairLocation settings to also be defined. Currently this setting is only supported for Linux nodes.

**IsSpotInstance**

Set automatically if a bid price is set on this node.

**KeyPair**

The name of the keypair to use when starting the node. This keypair will be used as the root login. If you followed the default instructions for CycleCloud this keypair is likely named 'cyclecloud'. The available keypairs are typically available from your cloud provider console.

**KeyPairLocation**

The location of the keypair on your local machine. This is used to log into the remote machine after it has been started. Valid values are any path, for example: `/home/users/test.user/.ssh/cyclecloud.pem`. Note: This attribute is not required to start an instance, but it is required to connect to that instance using the `connect` command.

**MachineType**

Provider-specific name for instance to start (e.g., m1.large). Consult your cloud provider documentation for the available machine types.

**ReturnProxyAddress**

The Return Proxy network address that the cluster should communicate with. If not specified, it will default to '' which indicates that the cluster nodes should communicate with the Return Proxy using the proxies private network address. For situations where the private network is not accessible to the cluster (on separate Cloud Providers or different regions) this should be set to ''. This tells the cluster to use the public network.
TargetState
The target state to move this resource to.

Template
The name of the template that this node’s settings came from.

Username
The username of an administrative user to log into the node.

NodeArray Extensions
In addition, [nodearray] elements support extra attributes.

InitialCoreCount
The number of cores to allocate for this node array when the cluster starts. Valid values are any positive number. If not specified 0 (no nodes) is assumed. This is a one-time operation, and instances that are allocated for this and then shutdown are not replaced. Note: This cannot be specified together with InitialCount.

InitialCount
The number of nodes to allocate for this node array when the cluster starts. Valid values are any positive number. If not specified 0 is assumed. This is a one-time operation, and instances that are allocated for this and then shutdown are not replaced. Note: This cannot be specified together with InitialCoreCount.

MaxCoreCount
The maximum number of cores to allocate for this node array. Valid values are any positive number. To ensure that the cluster never exceed 100 cores of a given node array you would specify a value of 100. Note that MaxCount and MaxCoreCount can be used together, in which case the lower effective constraint will take effect.

MaxCount
The maximum number of instances to allocate for this node array. Valid values are any positive number. To ensure that the cluster never exceed 10 instances of a given node array you would specify a value of 10. Note that MaxCount and MaxCoreCount can be used together, in which case the lower effective constraint will take effect.

TargetCoreCount
How many cores to allocate. CycleCloud will attempt to keep this many cores running in the cloud. Note: This cannot be specified together with TargetCount.

TargetCount
How many instances to allocate. CycleCloud will attempt to keep this many instances running in the cloud. Note: This cannot be specified together with TargetCoreCount.
AWS Extensions

Some settings take advantage of features of a specific cloud provider and are thus specified with a "vendor prefix". Note: These are specified in the template as listed, but actually create a subrecord on the Cloud.Node record called AWS.

AWS.AvailabilityZoneGroup
The user-specified name for a group of spot instances that are in the same availability zone.

AWS.EBSOptimized
Whether the instance is optimized for EBS I/O.

AWS.InstanceProfileArn
The Amazon Resource Name (ARN) of the IAM instance profile to associate with the instances.

AWS.InstanceProfileName
The name of the IAM Instance Profile (IIP) to associate with the instances.

AWS.Kernel
The ID of the kernel. Note: Amazon recommends that you use PV-GRUB instead of kernels and RAM disks.

AWS.LaunchGroup
The user-specified name for a group of spot instances that launch together and terminate together.

AWS.Monitoring
Whether to enable CloudWatch monitoring on the instance.

AWS.RamDisk
The ID of the RAM disk. Note: Amazon recommends that you use PV-GRUB instead of kernels and RAM disks.

AWS.RootDevice
The name of the root device, if different than what is reported on the image.

AWS.RootDeviceIops
The number of IOPS to assign to the root device.

AWS.RootDeviceSize
The size of the root device.

AWS.RootDeviceType
The type of root device to use (’standard’, ’gp2’, ’io1’).
**AWS.SpotRequestType**
The type of spot-instance request ('one-time' or 'persistent').

**AWS.Virtualization**
Whether to use 'hvm' (the default) or 'pvm' virtualization, for instance types that support both. Note: this only applies when selecting images by name, if both HVM and PVM images are available.

Some older attributes are AWS-specific but do not have the AWS. prefix:

**BidDuration**
Indicates the maximum amount of time (in minutes) to keep a spot request open. If defined, this value will be used instead of the default of 20 minutes. If the specified time has elapsed without the spot request being fulfilled the request will be automatically cancelled.

**BidPrice**
Indicates the maximum amount you are willing to pay for spot instances. If defined, this indicates nodes should be created as a bid for spot instances using the Amazon Spot Market. Valid values are any decimal, for example 0.01 means $0.01 (1 cent in USD). If the value is too low, you will not be granted the machine until the value of that machine goes below your bid price. For more information, see Amazon EC2 Spot Instances.

**BidPricePerCore**
Indicates the maximum amount you are willing to pay per-core for spot instances. If defined, this indicates that nodes should be created as a bid for spot instances using the Amazon Spot Market. Like BidPrice, this is the maximum amount of money (in dollars) you are willing to spend, but specified per CPU core hour instead of per instance. Valid values are any decimal, for example 0.01 means $0.01 (1 cent in USD). This attribute allows you to specify bid prices more consistently across instance types like c1.medium and c1.xlarge, which differ only in the number of cores available.

**PlacementGroup**
The placement group to put a node in. All nodes in a placement group must be in the same region. Placement groups ensure that all nodes within the placement group are physically located near each other, making placement groups ideal for latency-sensitive workflows such as MPI jobs. Placement groups are only available for cluster compute instances. Valid values are any placement group name (which have to be created in the AWS console beforehand), for example test-placement-group. For more information on placement groups see: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using_cluster_computing.html.

**SecurityGroups**
A list of AWS EC2 security groups to apply to the node. If not specified the default security group will be used; otherwise, valid values are a comma-separated list of security groups. For example: sg-1234abcd, sg-98761234.

**SubnetId**

The VPC subnet in which to launch this instance, if defined. If your cluster is going to run inside an Amazon VPC, you will need to specify the SubnetId to start the instance in. Valid values are any subnet ID, for example: subnet-1234abcd. You can view available subnets in the AWS VPC console.

**Tenancy**

Specifies the degree to which this instance can be shared with other accounts. For Amazon Web Services, the two accepted values are "default" or "dedicated". If not specified, "default" will be used. There may be additional charges for using this feature.

**TerminateOnShutdown**

Indicates instances should terminate themselves when they are shutdown. If true, when an instance attempts to shut itself down (shutdown -h, poweroff, etc), the instance will go into a terminated state (if false, it goes into a 'stopped' state). This value shouldn't need to be updated unless you have a specific case where terminating an instance is not the correct behavior. Only applies to on-demand AWS nodes.

**TerminationProtection**

Indicates instances should not be allowed to be terminated as normal. If true, termination requests on the instance will fail leaving the instance in a running state. To terminate the instance, Termination Protection will have to be disabled via the AWS console before making a termination request. This attribute should be used to prevent the accidental termination of important instances. Only applies to on-demand AWS nodes.

**Zone**

The AWS availability zone you want to start your instance in. If not specified, one will be chosen for you. Valid values are any valid availability zone, for example: us-east-1d. Specifying an availability zone is required if you are using placement groups. Placing all nodes in the same availability zone will reduce latency between nodes. However, if a zone goes down, all nodes in this zone will be lost. Also, availability of some machine types is constrained by zone. For more information on availability zones see: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

**Status attributes**

These attributes are set automatically by CycleCloud and are available for informational purposes only.
InstallationStatus
   Status of node's software installation.

InstanceId
   The id of the instance started for this node.

Last
   The previous state of various transient attributes. This is set when a node is terminated and retains its value until the node is started and terminated again.

SpotRequestId
   The id of the spot request started for this node, if any.

Node Status Attributes
Nodes report on their status with the following attributes. Note that these cannot be set directly.

ActivePhases
   The cluster initialization steps that are currently being executed for this node.

PhaseFailed
   If true, the initialization process failed.

PhaseMap
   Information on the cluster initialization steps for this node.

State
   The state this node is in currently.

Status
   The overall summary of this's node progress. The status options are Off (no instance is active or being acquired), Acquiring (getting an instance from the cloud provider), Preparing (configuring the instance and installing software), and Ready (instance is up and running). If any phase fails during starting or terminating the node, the status is Failed.

StatusMessage
   The description of this node's current status.

Cloud Resource Reference
This section describes the types of records that are gathered from your cloud-service provider. They allow you to monitor your resource usage and are also used by CycleCloud for its own operations.

Instances
CycleCloud tracks each instance in your account in a Cloud.Instance record, even instances not started by CycleCloud. It collects the data periodically and removes instances that are no longer listed in your cloud provider. Note that Cloud.Instance is an abstract type: records from the cloud provider are actually stored in a cloud-specific type (e.g., AWS.Instance), and Cloud.Instance is a standardized reflection of the records from all the actual types. The terminology used in Cloud.Instance does not always match perfectly with the terminology provided by the cloud-service provider's own tools (e.g., StartTime instead of the AWS-specific LaunchTime).

InstanceId

Uniquely identifies this instance across a cloud provider. This value comes from the cloud provider, and is either their unique id or a composite name that CycleCloud creates which is uniquely identifying.

Provider

The name of the cloud-service provider for this instance. This is a CycleCloud name for the provider being used, e.g., "aws". Taken together with the InstanceId, this uniquely identifies a single instance.

The properties of the instances are available as the following attributes:

CreatedTime

The time that this instance was created.

DeletedTime

The time that this instance was deleted. This is only defined after the instance is no longer listed in the cloud provider.

EndTime

The time this instance stopped running.

EnteredCurrentState

The time that this instance transitioned into its current state. For example, if this instance is Started, this is the time it started.

ImageId

The specific OS image from which this instance was started. This name is supplied by the cloud-service provider.

KeyPair

The name of the keypair used to access this instance. This is supplied by the cloud-service provider.

MachineState

95
Indicates the overall state of this virtual machine. One of Starting, Started, Stopping, Stopped, Terminating, Terminated, or Unknown. Cloud-provider-specific states are translated into this state. Generally speaking, you are billed for the instance while in the Starting and Started states, and not in the Stopping, Stopped, Terminating or Terminated states. You will typically be billed for some period of time after the instance leaves the Started state, according to the billing granularity specified by your cloud provider. If an instance is in the Unknown state, CycleCloud cannot determine if you are being billed or not.

**MachineStateMessage**
Provides more detail about the machine state in human-readable form.

**MachineType**
The cloud-provider-specific name or class of the hardware this instance is running on.

**PrivateHostName**
The internal DNS name used to access this instance.

**PrivateIp**
The internal IP address used to access this instance.

**PublicHostname**
The external DNS name used to access this instance. This is undefined if the instance cannot be reached from outside of its subnet.

**PublicIp**
The external IP address used to access this instance. This is undefined if the instance cannot be reached from outside of its subnet.

**Region**
The name of the geographical locality in which this instance is located. This name is specific to a cloud-service-provider.

**StartTime**
The time this instance was started.

**Status**
An overall summary of the status or health of this instance. If there are no known issues with this instance, the Status is ok. If there are problems that may require attention, the Status attribute is warning. If there are currently known issues with the instance, the Status attribute is error.

**StatusChecks**
A collection of all the status checks that have been run on this instance. This is a nested record containing potentially many entries. Each attribute is the name of the status check.
being run; each value is itself a nested record which consists of Status and Description attributes, as well as additional information that may be provided by the specific check. The Status attribute on each check is ok, warning or error, similar to the overall status. Note: not all cloud providers support the same set of checks.

**TagList**
A list of names used to tag this instance. This attribute depends on the cloud-service provider. Some providers may not offer a way to tag instances. Some may only support a list of strings.

**Tags**
A nested record of custom key-value used to tag this instance. This depends on the cloud-service provider. For providers that only support a list of strings, each name is converted to a key-value pair if it is in the form key:value or key=value.

**Zone**
A data center or availability zone within a region (optional). This name is specific to a cloud-service-provider.

There are also attributes that CycleCloud provides to track the billing on an instance.

**SessionUptime**
How long this instance has been running since it was last started. This is reset every time the instance is stopped and started again.

**SessionBilledTime**
How much billable time this instance has accumulated since it was last started.

**PreviousUptime**
How long this instance has been running in total prior to the current session. 0 if this instance has never been stopped.

**PreviousBilledTime**
How much billable time this instance has accumulated prior to the current session. 0 if this instance has never been stopped.

**TotalUptime**
How long this instance has been running, including previous sessions.

**TotalBilledTime**
How much billable time this instance has accumulated in total, including previous sessions.

**Price**
The hourly cost of this instance at its current rate.
PriceName
The price entry used for this instance (stored in Cloud.Price records). This points to a Cloud.Price record that reflects the per-hour cost of this instance.

PriceUpdatedTime
The time at which the price was last updated.

Release Notes

CycleCloud version 5.3.0

New Features:

• Google Cloud Platform support has been added.
• The `jetpack converge` command now has a `--no-sync` option which prevents automatic updating of configuration information from being downloaded to the node before a converge.
• `recipe[cyclecloud]` and `recipe[cluster_init]` are now automatically added to every node's run_list
• CycleCloud now includes Ganglia monitoring by default
• Ganglia monitoring is now automatically configured when the service registration setting is enabled
• Grid Engine monitoring is now automatically configured when the service registration setting is enabled and the monitoring component is installed
• CycleCloud nodes can now publish their capabilities back to the CycleCloud instance which started them.
• Windows 2008 AWS images can now utilize enhanced networking

Resolved Issues:

• The `show_nodes` command did not work correctly for Azure nodes.
• Proxy setup would not fall back to using the public IP address when a hostname wasn't available.
• An empty or missing run_list in a node's `[[[configuration]]]` section no longer generates an error.

CycleCloud version 3.2.0

Compatibility changes:
Blackboard (cluster search) will now use CycleCloud as a backend instead of S3 by default.
AWS spot-datafeed subscriptions are no longer automatically created on install

New Features:

- Azure accounts can now be created and edited via the cyclecloud command and the browser.
- A storage account and container is now created for new Azure accounts.
- The standard sample cluster templates now support both AWS and Azure.
- You can now specify both the cluster-init and chef repo lockers using the Locker attribute on nodes.
- The AWS g2.8xlarge instance type is now supported
- All cluster parameter types now support Required, DefaultValue, Disabled and Hidden attributes
- Parameters and sections on cluster templates can now be dynamically modified with Conditions.* attributes.
- Return proxies now accept username/password for authentication.
- Azure hostnames are now randomized for both Windows and Linux instances.

Resolved Issues:

- Could not terminate nodes in a cluster if the return proxy was not started
- The Start link on the Clusters dashboard is now inactive while clusters are being terminated or deleted.
- Automatic cloud services in Azure were not being deleted reliably.
- Fixed the "Invalid Storage Credentials" error that Azure Paas nodes were experiencing

CycleCloud version 3.1.0

New Features:

- CycleCloud now supports creating volumes automatically, including volumes that persist across cluster restarts.
- DataMan endpoints are now created when new lockers are added for a provider account
- A DataMan endpoint to CycleCloud "example" directory is now automatically created on installation
The clusters page now includes a "Connect" button that displays information on how to connect to any node.

- Added monitoring of all EBS volumes in an AWS account
- Azure templates can now be started with reference to the standard Windows 2012 and Ubuntu image names
- Grid engine jobs can now gain exclusive access to a execute node using the exclusive flag: qsub -l exclusive job.sh
- Basic cluster init "pi" demo is show installed to $CS_HOME/examples on initial installation.
- Cluster forms may now be split into sections with a section list on the left-hand side.
- Templates with the same "Category" attribute are now grouped together when selecting a cluster in the UI
- Azure cloud services are now created and deleted on demand for both IaaS and PaaS nodes.
- The HealthCheck service now includes a "scheduled shutdown" check. Execute nodes will schedule a shutdown before making an autostop request which allows the node to shut down if CycleCloud is unreachable.
- Autoscaling HTCondor clusters now supports the use of the slot_type job attribute to select which nodearray to request.

Resolved Issues:

- Fixed a regression in the 5.0.0 release where double-clicking a node would pop up two dialog boxes
- Windows images did set the Virtual Disk Service to start automatically, which led to an occasional issue where AWS instances did not correctly attach their ephemeral volumes.
- Cluster init executables are now restricted to certain file extensions to prevent unintended files from trying to be executed. (Linux: .sh Windows: .bat, .cmd, .exe)
- GridEngine user and group are now configurable via the gridengine.user.* and gridengine.group.* configuration settings.

CycleCloud version 3.0.0

New Features:
• CycleCloud now supports HTCondor clusters using IaaS and PaaS nodes in Windows Azure.
• All instance attributes (ip address, hostname, etc) can now be accessed via cyclecloud.instance.* (cyclecloud.instance.ipv4, cyclecloud.instance.hostname, etc).
• Pogo now supports accessing blobs in Azure Storage Accounts.
• Added "cp" command to pogo to allow copies within a single account. Copies between endpoints or accounts not yet supported.
• The pogo list, copy, and delete commands now accept the "--recursive" option to operate on directories.
• Running "pogo ls" with no url is now equivalent to running "pogo urls" with no arguments.
• The pogo 'del' command is now capable of deleting s3 files that end in a delimiter (ie "foo/").

Resolved Issues:

• The value of IsServer was not correctly set on HTCondor execute nodes, which caused terminated nodes to show as "Down" in CycleServer.