Cluster in the cloud Documentation

Matt Williams

Jul 07, 2023

Tutorial

Welcome to the documentation for Cluster in the Cloud. By the end of this you will have a fully-operational, elastically-scaling, heterogeneous Slurm cluster running on cloud resources.

In the future, the intention is that this tutorial will cover installing on all major cloud providers but for now AWS, Google Cloud Platform and Oracle Cloud Infrastructure are covered.

This tutorial and the Cluster in the Cloud software was created by Matt Williams at the ACRC in Bristol. Contributions to this tutorial document are welcome at GitHub.

If you need help

If you have any questions or issues with the system as a whole, please raise them as a GitHub issue at clusterinthecloud/support.

Note: If you use Cluster in the Cloud for any published research, please mention us in your acknowledgements.

CHAPTER 1

Set up the cloud infrastructure

If you are using AWS or Google and want to use the 1-click installer, you do not need to install Terraform. You can skip this chapter and go straight to their page.

1.1 Getting Terraform

The first step is to get a bunch of servers powered on in your cloud. We do this using a tool called Terraform.

Make sure that Terraform is installed by running the following in the command-line:

```
$ terraform version
```

you should get output like:

```
Terraform v1.0.3
```

1.2 Getting the Terraform config

Terraform is a tool for creating infrastructure but we need to provide it with some configuration.

Start by making a new directory which will hold all our configuration. Change to that directory in your terminal.

Grab the Terraform config from Git using:

```
$ git clone https://github.com/clusterinthecloud/terraform.git
$ cd terraform
```

We're now ready to start configuring our infrastructure on either:

- *AWS* or
- *Google Cloud Platform* or
- *Oracle Cloud Infrastructure*.

CHAPTER 2

Creating the infrastructure on AWS

The first thing to do is to set up your local credentials so that you can communicate with the AWS APIs.

Install the AWS command-line tools by following their instructions.

Once installed, configure the tool by running:

```
$ aws configure
```

All later steps will use these credentials.

There are two ways of creating Cluster in the Cloud on AWS. The older way was to download the code to your computer and run it from there. This requires installing Terraform, Git and setting up SSH locally.

There is now also a "1-click" installer available which is covered in the first section on this page. The older method is also documented here for posterity.

2.1 1-click installer

Download the installer (`install-citc.py`) and uninstaller (`destroy-citc.py`) scripts from clusterinthecloud/installer. You can do this either by cloning the Git repo (`git clone https://github.com/clusterinthecloud/installer.git`), by downloading the zip (master.zip) or by downloading the two files individually.

The only requirements for the script are Python >= 2.7, `ssh`, `ssh-keygen` and `scp`.

Run the install script as:

```
$ ./install-citc.sh aws
```

If you need to specify the region for the cluster or an AWS credential profile, you can pass them in with `--region` or `--profile`.

This should download all it needs automatically, start the cluster and provide you with a private key (in a subdirectory named after the cluster ID, and called `citc-key`) and an IP address. Something like:

```
The file 'citc-terraform-useful-gnu/citc-key' will allow you to log into the new␣
↪cluster
Make sure you save this key as it is needed to destroy the cluster later.
The IP address of the cluster is 130.61.43.69
Connect with:
  ssh -i citc-terraform-useful-gnu/citc-key citc@130.61.43.69
```

You can now move on to the next page of the tutorial, *finalising the setup on the cluster*.

Otherwise, if you wish to do the steps manually, then read on from here...

2.2 Setting the config

To initialise the local Terraform repo, start by running the following:

```
$ terraform -chdir=aws init
```

Now, when you check the Terraform version, you should see the AWS provider showing up:

```
$ terraform -chdir=aws version
Terraform v1.0.3
on linux_amd64
+ provider registry.terraform.io/hashicorp/aws v3.51.0
+ provider registry.terraform.io/hashicorp/external v2.1.0
+ provider registry.terraform.io/hashicorp/local v2.1.0
+ provider registry.terraform.io/hashicorp/null v3.1.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
+ provider registry.terraform.io/hashicorp/template v2.2.0
+ provider registry.terraform.io/hashicorp/tls v3.1.0
```

Rename the example config file `aws/terraform.tfvars.example` to `aws/terraform.tfvars` and open it in a text editor:

```
$ mv aws/terraform.tfvars.example aws/terraform.tfvars
$ vim aws/terraform.tfvars
```

There's a few variables which can be set in here. The only variable which you must set is `admin_public_keys` which must contain the public key you wish to use to log in to the admin account on the cluster. You can set multiple public keys here if you wish.

To see what other possible configuration opions there are, look at `aws/variables.tf`.

2.3 Running Terraform

At this point, we are ready to provision our infrastructure. Check that there's no immediate errors with

```
$ terraform -chdir=aws validate
```

It should return with no errors. If there are any problems, fix them before continuing.

Next, check that Terraform is ready to run with

```
$ terraform -chdir=aws plan
```

which should have, near the end, something like `Plan:  25 to add, 0 to change, 0 to destroy.`.

We're now ready to go. Run

```
$ terraform -chdir=aws apply
```

and, when prompted, tell it that "yes", you do want to apply.

It will take some time but should return without any errors with something green that looks like:

```
Apply complete! Resources: 25 added, 0 changed, 0 destroyed.

Outputs:

ManagementPublicIP = "130.61.43.69"
cluster_id = "cheerful-macaw"
```

You are now ready to move on to *finalising the setup on the cluster*.

CHAPTER 3

Creating the infrastructure on Google

There are two ways of creating Cluster in the Cloud on Google Cloud. The older way was to download the code to your computer and run it from there. This requires installing Terraform, Git and setting up SSH locally.

There is now also a "1-click" installer available which is covered in the first section on this page. The older method is also documented here for posterity.

3.1 1-click installer

Go to the Google Cloud Console and open a Cloud Shell. In the cloud shell type:

```
$ docker run -it -e CLOUDSDK_CONFIG=/config/gcloud \
                 -v $CLOUDSDK_CONFIG:/config/gcloud \
                 clusterinthecloud/google-install
```

It will download and start running the setup program and will ask some questions. If a question has a default it will show you it in `[square brackets]`, to accept a default, just press enter.

When it asks for your SSH keys, the simplest approch is to point it to a URL containing your public keys, such as on GitHub as a URL like `https://github.com/milliams.keys`.

It may ask if you want to continue with the authorisation method used, press `Y` and enter..

It will then authenticate with Google and give you a long URL to click on. Click it, sign-in to Google and allow access. Take the authentication token it gives you and paste it into the Cloud Shell (ctrl-v and press enter).

It will then go through the process of setting up Cluster in the Cloud and will give you the IP address of the login node to SSH into:

```
Your Cluster-in-the-Cloud has now been created :-)
Proceed to the next stage. Connect to the cluster
by running 'ssh citc@130.61.43.69'

{"status":"0", "cluster_ip":"130.61.43.69"}
```

You can now move on to the next page of the tutorial, *finalising the setup on the cluster*.

Otherwise, if you wish to do the steps manually, then read on from here...

3.2 Setting up the environment

Before we can install the cluster onto your cloud environment, we need to do some initial one-off setup. Firstly, we need to install the command-line tool `gcloud` which allows you to configure Google Cloud. Download and setup `gcloud` based on the instructions from Google.

Once you have `gcloud` installed, start by associating it with your Google account:

```
$ gcloud config set account <googleaccount@example.com>
```

where you should replace `<googleaccount@example.com>` with your email address.

Now that it knows who you are, you should set a default project. You can find the ID of your project with `gcloud projects list`.

```
$ gcloud config set project <citc-123456>
```

Once the project has been set, we can enable the required APIs to build Cluster in the Cloud. This step will likely take a few minutes so be patient:

```
$ gcloud services enable compute.googleapis.com \
                         iam.googleapis.com \
                         cloudresourcemanager.googleapis.com \
                         file.googleapis.com
```

That's all the structural setup for the account needed. The last `gcloud` thing we need to do is create a service account which Terraform uses to communicate with GCP. Make sure to replace every instance of `<citc-123456>` with your project ID:

```
$ gcloud iam service-accounts create citc-terraform --display-name "CitC Terraform"
$ gcloud projects add-iam-policy-binding <citc-123456> --member serviceAccount:citc-
↪terraform@<citc-123456>.iam.gserviceaccount.com --role='roles/editor'
$ gcloud projects add-iam-policy-binding <citc-123456> --member serviceAccount:citc-
↪terraform@<citc-123456>.iam.gserviceaccount.com --role='roles/iam.securityAdmin'
$ gcloud iam service-accounts keys create citc-terraform-credentials.json --iam-
↪account=citc-terraform@<citc-123456>.iam.gserviceaccount.com
```

This will create a local JSON file which contains the credentials for this user.

3.3 Setting the config

To initialise the local Terraform repo, start by running the following:

```
$ terraform -chdir=google init
```

Now, when you check the Terraform version, you should see the Google provider showing up:

```
$ terraform -chdir=google version
Terraform v1.0.3
on linux_amd64
```

(continues on next page)

(continued from previous page)

```
+ provider registry.terraform.io/hashicorp/external v2.1.0
+ provider registry.terraform.io/hashicorp/google v3.76.0
+ provider registry.terraform.io/hashicorp/local v2.1.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
+ provider registry.terraform.io/hashicorp/template v2.2.0
+ provider registry.terraform.io/hashicorp/tls v3.1.0
```

Rename the example config file `google/terraform.tfvars.example` to `google/terraform.tfvars` and open it in a text editor:

```
$ mv google/terraform.tfvars.example google/terraform.tfvars
$ vim google/terraform.tfvars
```

There's a few variables which we need to change in here. First you must set the `region` and `zone` variables to the correct values for your account. This will depend on what regions you have access to and where you want to build your cluster.

Then the `project` variable must be set to the project ID as we used above when running `gcloud`.

You must set `admin_public_keys` to contain the public key you wish to use to log in to the admin account on the cluster. You can set multiple public keys here if you wish.

Finally, if you wish you can change the node type used for the management node. By default it's a lightweight single-core VM which should be sufficient for most uses but you can change it if you wish.

The rest of the variables should usually be left as they are.

3.4 Running Terraform

At this point, we are ready to provision our infrastructure. Check that there's no immediate errors with

```
$ terraform -chdir=google validate
```

It should return with no errors. If there are any problems, fix them before continuing.

Next, check that Terraform is ready to run with

```
$ terraform -chdir=google plan
```

which should have, near the end, something like `Plan: 11 to add, 0 to change, 0 to destroy.`.

We're now ready to go. Run

```
$ terraform -chdir=google apply
```

and, when prompted, tell it that "yes", you do want to apply.

It will take some time but should return without any errors with something green that looks like:

```
Apply complete! Resources: 11 added, 0 changed, 0 destroyed.

Outputs:

ManagementPublicIP = "130.61.43.69"
cluster_id = "cheerful-macaw"
```

You are now ready to move on to *finalising the setup on the cluster*.

CHAPTER 4

Creating the infrastructure on Oracle

4.1 Setting the config

```
$ terraform -chdir=oracle init
```

Now, when you check the Terraform version, you should see the OCI provider showing up:

```
$ terraform -chdir=oracle version
Terraform v1.0.3
on linux_amd64
+ provider registry.terraform.io/hashicorp/oci v4.36.0
+ provider registry.terraform.io/hashicorp/random v3.1.0
+ provider registry.terraform.io/hashicorp/template v2.2.0
+ provider registry.terraform.io/hashicorp/tls v3.1.0
```

Rename the example config file `oracle/terraform.tfvars.example` to `oracle/terraform.tfvars` and open it in a text editor:

```
$ mv oracle/terraform.tfvars.example oracle/terraform.tfvars
$ vim oracle/terraform.tfvars
```

Following the instructions at the Oracle Terraform plugin docs, set the values of `tenancy_ocid`, `user_ocid`, `private_key_path`, `fingerprint` and `region`. Make sure that the user account you use for `user_ocid` has admin access in your tenancy to create infrastructure.

You will also need to set the compartment OCID of the compartment that you are using. If you are using the default root compartment, this will be the same as your tenancy OCID.

The next thing to set is an SSH key that you will use to connect to the server once it is built. See GitHub's documentation on information on how to do this and then paste the contents of the public key into the `ssh_public_key` config variable between the two `EOF`s.

You will want a simple, lightweight VM for the management node so for this tutorial, we will use `VM.Standard2.1 6` for the management node.

Set the `ManagementShape` config variable to the shape you want for the management node:

```
ManagementShape = "VM.Standard2.1"
```

The second thing we need to do for the management node is decide which AD it should reside in. Set the variable `ManagementAD` to whichever AD you'd like to use:

```
ManagementAD = "1"
```

4.2 Running Terraform

At this point, we are ready to provision our infrastructure. Check that there's no immediate errors with

```
$ terraform -chdir=oracle validate
```

It should return with no errors. If there are any problems, fix them before continuing.

Next, check that Terraform is ready to run with

```
$ terraform -chdir=oracle plan
```

which should have, near the end, something like `Plan: 11 to add, 0 to change, 0 to destroy.`.

We're now ready to go. Run

```
$ terraform -chdir=oracle apply
```

and, when prompted, tell it that "yes", you do want to apply.

It will take some time but should return without any errors with something green that looks like:

```
Apply complete! Resources: 9 added, 0 changed, 0 destroyed.

Outputs:

ManagementPublicIP = "130.61.43.69"
cluster_id = "cheerful-macaw"
```

You are now ready to move on to *finalising the setup on the cluster*.

CHAPTER 5

Finalising the setup

Terraform will have automatically started the cluster software configuration step. It will run in the background and will take some time to complete. In the meantime, you can connect to the custer and follow its progress.

5.1 Setting service limits

You can log into the management node at `citc@mgmtipaddress`, using the IP address that terraform printed at the end of its run.

For example:

```
$ ssh citc@130.61.43.69
```

Once logged in, you can run the `finish` script:

```
[citc@mgmt ~]$ finish
```

It will most likely tell you that the system has not finished configuring. If the `finish` script is not there, wait a minute or two and it should appear.

To follow the progress, you can look at the file `ansible-pull.log` in `root`'s home directory:

```
[citc@mgmt ~]$ sudo tail -f /root/ansible-pull.log
```

Use `ctrl-c` to stop following the log.

You can keep on running trying to run `finish` until the node has finished configuring. Once it has, you need to tell the system what limits you want to place on the scaling of the cloud. Edit the file `/home/citc/limits.yaml` with:

```
[citc@mgmt ~]$ vi limits.yaml
```

5.1.1 Oracle

On an Oracle-based system, you should set its contents to something like:

```
VM.Standard2.1:
  1: 1
  2: 1
  3: 1
VM.Standard2.2:
  1: 1
  2: 1
  3: 1
```

which specifies for each shape, what the service limit is for each AD in the region your cluster lives in. In this case each of the shapes `VM.Standard2.1` and `VM.Standard2.2` have a service limit of 1 in each AD. The system will automatically adjust for the shape used by the management node.

To decide what to put here, you should refer to your service limits in the OCI website. In the future, we hope to be able to extract these automatically but for now you need to replicate it manually.

5.1.2 Google

On a Google-based system, you should set it to something like:

```
n1-standard-1: 3
n1-standard-2: 3
```

which restricts the cluster to having at most 3 `n1-standard-1` and 3 `n1-standard-1` nodes. Since Google does not have per-node-type limits, you can make these numbers as large as you like in principle.

5.1.3 AWS

On a AWS-based system, you should set it to something like:

```
t3a.small: 3
t3a.medium: 3
```

which restricts the cluster to having at most 3 `t3a.small` and 3 `t3a.medium` nodes. Since AWS does not have per-node-type limits, you can make these numbers as large as you like in principle.

5.1.4 Finalise configuration

Run `finish` again and it should configure and start the Slurm server:

```
[citc@mgmt ~]$ finish
```

If your service limits change, you can update the file and run the script again.

5.2 Adding users

To add users to the system, you run the command `/usr/local/sbin/add_user_ldap` passing it the username of the user you want to add, the user's first and surnames and the URL of a file containing their SSH public keys.

```
[citc@mgmt ~]$ sudo /usr/local/sbin/add_user_ldap matt Matt Williams https://github.
↪com/milliams.keys
```

You can run this command again to add another user.

If the user does not have an online list of their keys, you can copy the public key to the `mgmt` node with `scp` and then use the `file` protocol:

```
[citc@mgmt ~]$ sudo /usr/local/sbin/add_user_ldap matt Matt Williams file:///home/
↪citc/users_key.pub
```

Once it has succeeded, log out and try logging as one of those users.

5.3 Check Slurm is running

```
$ ssh -A matt@130.61.43.69
```

Once logged in, try running the `sinfo` command to check that Slurm is running:

```
[matt@mgmt ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
compute*     up   infinite      0    n/a
```

It shows the number of nodes being zero as the nodes will be automatically created as they are required, up to the limit specified in the earlier step. This is all looking good so let's start submitting jobs.

Check out the information on *running the cluster*.

CHAPTER 6

Running the cluster

Now that you have a cluster which is up and running, it's worth knowing what you can do with it.

6.1 Slurm jobs

A full Slurm tutorial is outside of the scope of this document but it's configured in a fairly standard way. By default there's one single partition called `compute` which contains all the compute nodes.

A simple first Slurm script, `test.slm`, could look like:

```
#! /bin/bash

hostname
```

which you could run with:

```
[matt@mgmt ~]$ sbatch test.slm
```

To check that Slurm has started the node you need, you can run `sinfo`:

```
[matt@mgmt ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
compute*     up   infinite      1 alloc# vm-standard2-1-ad1-0001
```

where the `state` being `alloc#` means that the node has been allocated to a job and the `#` means that it is currently in the process of being turned on.

Eventually, once the node has started, the state will change to `alloc`:

```
[matt@mgmt ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
compute*     up   infinite      1  alloc vm-standard2-1-ad1-0001
```

and then once the job has finished the state will move to `idle`:

```
[matt@mgmt ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
compute*     up   infinite      1   idle vm-standard2-1-ad1-0001
```

If you want more control over the size of your job etc., then you can set those flags in the job script:

```
#! /bin/bash

#SBATCH --job-name=test
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=1
#SBATCH --time=10:00

hostname
```

6.2 Slurm elastic scaling

Slurm is configured to use its elastic computing mode. This allows Slurm to automatically terminate any nodes which are not currently being used for running jobs and create any nodes which are needed for running jobs. This is particularly useful in the cloud as a node which has been terminated will not be charged for.

Slurm does this by calling a script `/usr/local/bin/startnode` as the `slurm` user. If necessary, you can call this yourself from the `citc` user like:

```
[citc@mgmt ~]$ sudo scontrol update NodeName=vm-standard2-1-ad1-0001 State=POWER_UP
```

to turn on the node `vm-standard2-1-ad1-0001`.

You should never have to do anything to explicitly shut down the cluster, it will automatically terminate all nodes which are not in use after a timeout. The management node will always stay running which is why it's worth only using a relatively cheap VM for it.

The rate at which Slurm shuts down is managed in `/mnt/shared/etc/slurm/slurm.conf` by the `SuspendTime` parameter. See the slurm.conf documentation for more details.

You can see the state of all the Slurm nodes (on or off) by running

```
[citc@mgmt ~]$ list_nodes
```

6.3 Cluster shell

A common task is to want to run commands across all nodes in a cluster. By default you have access to clustershell. Read the documentation there to get details of how to use the tool.

The gist is that you give it a hostname or a group and a command to run. You can see a list of the available groups with `cluset`:

```
[citc@mgmt ~]$ cluset --list-all
@compute
@state:idle
@role:mgmt
```

You can then run a command with `clush`:

```
[citc@mgmt ~]$ clush -w @compute uname -r
vm-standard2-1-ad1-0001: 4.14.35-1844.2.5.el7uek.x86_64
vm-standard2-1-ad3-0001: 4.14.35-1844.2.5.el7uek.x86_64
vm-standard2-2-ad3-0001: 4.14.35-1844.2.5.el7uek.x86_64
vm-standard2-2-ad2-0001: 4.14.35-1844.2.5.el7uek.x86_64
vm-standard2-1-ad2-0001: 4.14.35-1844.2.5.el7uek.x86_64
```

You can combine the output from different nodes using the `-b` flag:

```
[citc@mgmt ~]$ clush -w @compute -b uname -r
---------------
vm-standard2-[1-2]-ad3-0001,vm-standard2-1-ad[1-2]-0001,vm-standard2-2-ad2-0001 (5)
---------------
4.14.35-1844.2.5.el7uek.x86_64
```

Bear in mind that since the nodes are created afresh each time they are started, any changes you make to a running node will not be persisted. It will also not be able to run on any nodes that are not currently running.

6.4 Installing software on your cluster

To make software available across your cluster, the best way is to install it onto the shared filesystem at `/mnt/shared`. Make sure that all the dependencies for it are available either on the shared filesystem or in the base image you're using. i.e. don't use `yum install` to provide dependencies.

Consider using a tool like EasyBuild or Spack to manage your software stack.

6.5 Monitoring

The cluster automatically collects data from all the nodes and makes them available in a web dashboard.

It is available at the IP address of your management node on port 3000. Point your browser at `http://your.mgmt.ip.address:3000` and log in with the username *admin*. The password for the dashboard can be found by running:

```
[citc@mgmt ~]$ sudo get_secrets
```

Once you are logged in, you can find a dashboard showing the state of nodes by clicking on "Home" in the top-left and selecting "Slurm".

6.6 Reporting issues

Please report questions and problems to Cluster in the Cloud GitHub Issues.

You can gather diagnostic information from your cluster which might help pinpoint problems by running

```
[citc@mgmt ~]$ sudo sosreport --only-plugins citc && sudo chown $USER /var/tmp/
↪sosreport*
```

For `case id` put your GitHub username (if you have one). This will generate a `tar.xz` file that can be downloaded and then attached to, for example, a GitHub issue.

6.7 Destroying the whole cluster

Warning: Please bear in mind that this will also destroy your file system which contains your user's home area and any data stored on the cluster.

When you've completely finished with the cluster, you can destroy it.

First you must make sure that you do not have any running jobs or compute nodes. You can kill all running nodes by running the following on the management node:

```
[citc@mgmt ~]$ /usr/local/bin/kill_all_nodes
```

Then, you can log out and locally run:

```
$ terraform -chdir=aws apply -destroy
```

or

```
$ terraform -chdir=google apply -destroy
```

or

```
$ terraform -chdir=oracle apply -destroy
```

This command *will* ask for confirmation before destroying anything but be sure to read the list of things it's going to terminate to check that it's doing the right thing. It will also attempt to terminate any running compute nodes you still have but make sure to check the web interface afterwards.

6.7.1 AWS 1-click uninstall

If you installed the cluster with the 1-click installer on AWS then you can uninstall it in a similar way by running following:

```
$ ./destroy-citc.py aws 130.61.43.69 citc-terraform-useful-gnu/citc-key
```

but passing in the IP address of the management node that you we given when the cluster was created, as well as the SSH key to connect with.

6.7.2 Google 1-click uninstall

If you installed the cluster with the 1-click installer on Google Cloud then you can uninstall it in a similar way by running the following in a Google Cloud Shell:

```
$ docker run -it -e CLOUDSDK_CONFIG=/config/gcloud \
                 -v $CLOUDSDK_CONFIG:/config/gcloud \
                 clusterinthecloud/google-destroy
```

CHAPTER 7

Prerequisites

To complete this tutorial you will need:

- access to a command-line (i.e. Linux, MacOS Terminal or WSL)
- an SSH key pair
- an account with credit on AWS, Google or Oracle cloud
  - the account must have admin permissions to create infrastructure
- local software installed
  - Terraform 1.0 or newer
  - SSH
  - Git

Alternatively, for some cloud providers (only AWS and Google so far) we have a simpler "one-click" installer which has fewer or no up-front requirements at all. That is covered in the AWS and Google chapters.

Start the tutorial here by *creating the infrastructure*.

CHAPTER 8

More info

Once you've got your cluster up and running, see these topics for more information:

8.1 Configuring node images

What is this page about?

This page explains how you can configure the image that is used on the compute nodes.

The images that are used by the compute nodes are built using Packer. The initial image is build automatically when the cluster is first created with the bare essentials needed to run jobs.

If you want to change the image in any way, you can edit the script `/home/citc/compute_image_extra.sh`. This script is run automatically at the end of the Packer inside the new image so you can fill it with things like:

```
#! /bin/bash
sudo yum -y install opencl-headers clinfo
```

Note the use of `sudo` as this script does not run as `root`.

Once the script has been edited to your liking, re-run Packer with:

```
[citc@mgmt ~]$ sudo /usr/local/bin/run-packer
```

This will start a VM inside your cloud account, build the image and then shut down the VM. From that point on, any newly-started nodes will use the new image.

8.1.1 GPU nodes

The default image that is built by CitC does not include any GPU drivers. If you want to use the GPUs on a given node, you will need to compile the drivers into the node image.

You can follow the NVIDIA documentation for this, which at its simplest means adding:

```
if [[ $(arch) == "x86_64" ]]; then
  sudo dnf config-manager --add-repo https://developer.download.nvidia.com/compute/
↪cuda/repos/rhel8/x86_64/cuda-rhel8.repo
  sudo dnf module install -y nvidia-driver:latest-dkms
fi
```

to `/home/citc/compute_image_extra.sh` and re-running Packer as shown above.

This will just install the base drivers, but if you need CUDA too, then add `sudo dnf install -y cuda` to the commands in that file too.

8.1.2 AWS ARM nodes

AWS provides instance types with Graviton processors which are based on the aarch64 architecture. This requires a special image to be built which is not created by default. To build this image, run the following:

```
[citc@mgmt ~]$ sudo /usr/local/bin/run-packer aarch64
```

which will build the aarch64 image.

8.2 Running Ansible

Both the management node and the compute nodes are configured using Ansible. This is a configuration tool which does things like putting config files in the right place, installing software and creating user accounts. The configuration for it is stored on GitHub at github.com/clusterinthecloud/ansible

A local clone of the Ansible configuration is made early in the setup of CitC in the folder `/root/citc-ansible`. All further configuration of the system is made in reference to that folder.

Under normal circumstances, you should not have to deal with Ansible manually, but if something goes wrong or you are adding a new feature then you will need this.

8.2.1 Updating Ansible

If any bug fixes or new features are added to the version of Ansible on GitHub, they will not automatically become available on your cluster. In order to retrieve the changes, you must run:

```
$ sudo /root/update_ansible_repo
```

8.2.2 Configuring the management node

The management node is configured automatically by a run of Ansible which is started when the node is first created.

To re-run this manually (for example to retry a failure, or if you've updated the config) you can run:

```
$ sudo /root/run_ansible
```

This will run Ansible in the foreground and also send its output to the usual `/root/ansible-pull.log` file.

This will, by default, also re-build the compute node image which a, may not be what you want and b, will take longer. You can disable the running of the image rebuild step with:

```
$ sudo /root/run_ansible --skip-tags=packer
```

8.2.3 Compute nodes

The same Ansible configuration is used by Packer when creating compute node images. The process described under *Configuring node images* uses this same configuration folder.

8.3 Overview

What is this page about?

This document is intended for people who are interested in contributing to Cluster in the Cloud or who are facing a bug or issue and want to understand how things work beneath the surface.

Cluster in the Cloud can be though of as working in three main steps:

- *Creating the infrastructure*
- *Configuring the software*
- *Dynamically managing nodes*

The first two are relatively static and only happens on cluster creation (and are destroyed on cluster destruction) while the third will be constantly happening while submitting jobs.

8.3.1 Creating the infrastructure

The first step is to create the base infrastructure for the cluster. Exactly what gets created here varies from one cloud provider to another but in general it is:

- Virtual network
- Management/login node
- Shared storage

The resources are created using Terraform from the configuration stored in clusterinthecloud/terraform.

Terraform only manages these resources, and not the compute nodes themselves. This means that when shutting down the cluster, the compute nodes must be destroyed by some other channel.

8.3.2 Configuring the software

Once the management node has been created it will kick off a configuration step using Ansible.

The Ansible configuration for the management node is all contained at clusterinthecloud/ansible and it's in this repo that the majority of the logic of the system is codified.

The Ansible playbook for the management node does a lot of steps, but importantly includes:

- Setting up the Slurm controller daemon

- Setting up the LDAP server for managing users
- Configuring the monitoring system
- Kicking off the creation of the compute node image

This last step uses Packer to start a new VM instance and run a related Ansible playbook on it and then save the resulting machine as the template node image for all compute nodes. This image creation step *can be run manually at any time* by the admin to update the image.

8.3.3 Dynamically managing nodes

The final step is how the system provisions and destroys compute nodes.

The core logic is based on Slurm's power management and elastic computing support. When a job is submitted, Slurm assigns a node from the list of potential nodes. It then runs a Python script, provided by CitC which talks to the cloud provider's API to create the instance with the correct settings (including choosing the node image). Once Slurm sees that a node has been idle for a while, it will run another CitC Python script which again talks to the API to terminate the instance.

8.4 Versioning

8.4.1 Historical versioning

In the past, when nodes booted they would run `ansible-pull` every time to set up their configuration. The Ansible playbook backing them was accessed directly as the `master` branch on GitHub. Of course, if the master branch had incompatible changes (either things which could not just be Ansible-updated to or didn't match the Terraform config) then it would not work.

To help resolve this, versioning was introduced where, on the initial creation of the cluster, a version number would be baked in. The Ansible config would then have a long-lived branch matching that version which nodes would boot from. The promise of the version number was then that any cluster which was created at version X would always be safe having its compute nodes boot from Ansible version X.

In early 2020, we went from configuring the nodes one boot to pre-creating node images. This means that unless the admin actively does something (e.g. update and rerun the management Ansible or recreate the node images) things will stay in sync.

Then, in mid 2021 we switched away from using `ansible-pull` and instead run everything from a copy of the Ansible config stored on the management node. Now, even re-creating the node image is safe as it will always read from the same Ansible configuration. This means that the versioning was now much less important as the admin would have to explicitly update the repo for things to break.