cloud-init

Release 24.2

unknown

Jul 26, 2024

CONTENTS

`Cloud-init` is the *industry standard* multi-distribution method for cross-platform cloud instance initialisation. It is supported across all major public cloud providers, provisioning systems for private cloud infrastructure, and bare-metal installations.

During boot, `cloud-init` identifies the cloud it is running on and initialises the system accordingly. Cloud instances will automatically be provisioned during first boot with networking, storage, SSH keys, packages and various other system aspects already configured.

`Cloud-init` provides the necessary glue between launching a cloud instance and connecting to it so that it works as expected.

For cloud users, `cloud-init` provides no-install first-boot configuration management of a cloud instance. For cloud providers, it provides instance setup that can be integrated with your cloud.

If you would like to read more about what cloud-init is, what it does and how it works, check out our *high-level introduction* to the tool.

Tutorials Get started - a hands-on introduction to `cloud-init` for new users

Tutorials **How-to guides** Step-by-step guides covering key operations and common tasks

How-to guides **Reference** Technical information - specifications, APIs, architecture

Reference **Explanation** Discussion and clarification of key topics

Explanation

CHAPTER

ONE

HAVING TROUBLE? WE WOULD LIKE TO HELP!

- *Check out our tutorials* if you're new to `cloud-init`
- *Try the FAQ* for answers to some common questions
- You can also search the `cloud-init` mailing list archive
- Find a bug? Report bugs on GitHub Issues

CHAPTER

TWO

PROJECT AND COMMUNITY

`Cloud-init` is an open source project that warmly welcomes community projects, contributions, suggestions, fixes and constructive feedback.

- Read our Code of Conduct
- Ask questions in the `#cloud-init` IRC channel on Libera
- Follow announcements or ask a question on the cloud-init Discourse forum
- Join the cloud-init mailing list
- *Contribute on GitHub*
- Release schedule

2.1 Tutorials

This section contains step-by-step tutorials to help you get started with `cloud-init`. We hope our tutorials make as few assumptions as possible and are accessible to anyone with an interest in `cloud-init`. They should be a great place to start learning about `cloud-init`, how it works, and what it's capable of.

2.1.1 Core tutorial

This tutorial, which we recommend if you are completely new to `cloud-init`, uses the QEMU emulator to introduce you to all of the key concepts, tools, processes and operations that you will need to get started.

Core tutorial with QEMU

QEMU tutorial debugging

You may wish to test out the commands in this tutorial as a `script` to check for copy-paste mistakes.

If you successfully launched the virtual machine, but couldn't log in, there are a few places to check to debug your setup.

To debug, answer the following questions:

Did `cloud-init` discover the IMDS webserver?

The webserver should print a message in the terminal for each request it receives. If it didn't print out any messages when the virtual machine booted, then `cloud-init` was unable to obtain the config. Make sure that the webserver can be locally accessed using **`curl`** or **`wget`**.

```
$ curl 0.0.0.0:8000/user-data
$ curl 0.0.0.0:8000/meta-data
$ curl 0.0.0.0:8000/vendor-data
```

Did the IMDS webserver serve the expected files?

If the webserver prints out `404 errors` when launching QEMU, then check that you started the server in the temp directory.

Were the configurations inside the file correct?

When launching QEMU, if the webserver shows that it succeeded in serving `user-data`, `meta-data` and `vendor-data`, but you cannot log in, then you may have provided incorrect cloud-config files. If you can mount a copy of the virtual machine's filesystem locally to inspect the logs, it should be possible to get clues about what went wrong.

In this tutorial, we will launch an Ubuntu cloud image in a virtual machine that uses `cloud-init` to pre-configure the system during boot.

The goal of this tutorial is to provide a minimal demonstration of `cloud-init`, which you can then use as a development environment to test your `cloud-init` configurations locally before launching to the cloud.

Why QEMU?

QEMU is a cross-platform emulator capable of running performant virtual machines. QEMU is used at the core of a broad range of production operating system deployments and open source software projects (including libvirt, LXD, and vagrant) and is capable of running Windows, Linux, and Unix guest operating systems. While QEMU is flexibile and feature-rich, we are using it because of the broad support it has due to its broad adoption and ability to run on *nix-derived operating systems.

How to use this tutorial

In this tutorial, the commands in each code block can be copied and pasted directly into the terminal. Omit the prompt (`$`) before each command, or use the "copy code" button on the right-hand side of the block, which will copy the command for you without the prompt.

Each code block is preceded by a description of what the command does, and followed by an example of the type of output you should expect to see.

Install QEMU

```
$ sudo apt install qemu-system-x86
```

If you are not using Ubuntu, you can visit QEMU's install instructions for additional information.

Create a temporary directory

This directory will store our cloud image and configuration files for *user data*, *metadata*, and *vendor data*.

You should run all commands from this temporary directory. If you run the commands from anywhere else, your virtual machine will not be configured.

Let's create a temporary directory and make it our current working directory with **cd**:

```
$ mkdir temp
$ cd temp
```

Download a cloud image

Cloud images typically come with `cloud-init` pre-installed and configured to run on first boot. You will not need to worry about installing `cloud-init` for now, since we are not manually creating our own image in this tutorial.

In our case, we want to select the latest Ubuntu LTS. Let's download the server image using **wget**:

```
$ wget https://cloud-images.ubuntu.com/jammy/current/jammy-server-cloudimg-amd64.img
```

Note: This example uses emulated CPU instructions on non-x86 hosts, so it may be slow. To make it faster on non-x86 architectures, one can change the image type and qemu-system-<arch> command name to match the architecture of your host machine.

Define our user data

Now we need to create our `user-data` file. This user data cloud-config sets the password of the default user, and sets that password to never expire. For more details you can refer to the *Set Passwords module page*.

Run the following command, which creates a file named `user-data` containing our configuration data.

```
$ cat << EOF > user-data
#cloud-config
password: password
chpasswd:
  expire: False

EOF
```

What is user data?

Before moving forward, let's inspect our `user-data` file.

```
$ cat user-data
```

You should see the following contents:

```
#cloud-config
password: password
chpasswd:
  expire: False
```

The first line starts with `#cloud-config`, which tells `cloud-init` what type of user data is in the config. Cloud-config is a YAML-based configuration type that tells `cloud-init` how to configure the virtual machine instance. Multiple different format types are supported by `cloud-init`. For more information, see the *documentation describing different formats*.

The second line, `password: password`, as per *the Users and Groups module docs*, sets the default user's password to `password`.

The third and fourth lines direct `cloud-init` to not require a password reset on first login.

Define our metadata

Now let's run the following command, which creates a file named `meta-data` containing configuration data.

```
$ cat << EOF > meta-data
instance-id: someid/somehostname

EOF
```

Define our vendor data

Now we will create the empty file `vendor-data` in our temporary directory. This will speed up the retry wait time.

```
$ touch vendor-data
```

Start an ad hoc IMDS webserver

Open up a second terminal window, change to your temporary directory and then start the built-in Python webserver:

```
$ cd temp
$ python3 -m http.server --directory .
```

What is an IMDS?

Instance Metadata Service (IMDS) is a service provided by most cloud providers as a means of providing information to virtual machine instances. This service is used by cloud providers to expose information to a virtual machine. This service is used for many different things, and is the primary mechanism for some clouds to expose `cloud-init` configuration data to the instance.

How does `cloud-init` use the IMDS?

The IMDS uses a private http webserver to provide metadata to each operating system instance. During early boot, `cloud-init` sets up network access and queries this webserver to gather configuration data. This allows `cloud-init` to configure your operating system while it boots.

In this tutorial we are emulating this workflow using QEMU and a simple Python webserver. This workflow is suitable for developing and testing `cloud-init` configurations prior to cloud deployments.

Launch a virtual machine with our user data

Switch back to your original terminal, and run the following command so we can launch our virtual machine. By default, QEMU will print the kernel logs and `systemd` logs to the terminal while the operating system boots. This may take a few moments to complete.

```
$ qemu-system-x86_64                                                    \
    -net nic                                                            \
    -net user                                                           \
    -machine accel=kvm:tcg                                              \
    -m 512                                                              \
    -nographic                                                          \
    -hda jammy-server-cloudimg-amd64.img                                \
    -smbios type=1,serial=ds='nocloud;s=http://10.0.2.2:8000/'
```

Note: If the output stopped scrolling but you don't see a prompt yet, press `Enter` to get to the login prompt.

How is QEMU configured for `cloud-init`?

When launching QEMU, our machine configuration is specified on the command line. Many things may be configured: memory size, graphical output, networking information, hard drives and more.

Let us examine the final two lines of our previous command. The first of them, **`-hda jammy-server-cloudimg-amd64.img`**, tells QEMU to use the cloud image as a virtual hard drive. This will cause the virtual machine to boot Ubuntu, which already has `cloud-init` installed.

The second line tells `cloud-init` where it can find user data, using the *NoCloud datasource*. During boot, `cloud-init` checks the `SMBIOS` serial number for `ds=nocloud`. If found, `cloud-init` will use the specified URL to source its user data config files.

In this case, we use the default gateway of the virtual machine (`10.0.2.2`) and default port number of the Python webserver (`8000`), so that `cloud-init` will, inside the virtual machine, query the server running on host.

Verify that `cloud-init` ran successfully

After launching the virtual machine, we should be able to connect to our instance using the default distro username.

In this case the default username is `ubuntu` and the password we configured is `password`.

If you can log in using the configured password, it worked!

If you couldn't log in, see *this page for debug information*.

Check `cloud-init` status

Run the following command, which will allow us to check if `cloud-init` has finished running:

```
$ cloud-init status --wait
```

If you see `status: done` in the output, it succeeded!

If you see a failed status, you'll want to check `/var/log/cloud-init.log` for warning/error messages.

Tear down

In our main terminal, let's exit the QEMU shell using `ctrl-a x` (that's `ctrl` and `a` simultaneously, followed by `x`).

In the second terminal, where the Python webserver is running, we can stop the server using (`ctrl-c`).

What's next?

In this tutorial, we configured the default user's password and ran `cloud-init` inside our QEMU virtual machine.

The full list of modules available can be found in *our modules documentation*. The documentation for each module contains examples of how to use it.

You can also head over to the *examples page* for examples of more common use cases.

2.1.2 Quick-start tutorial

This tutorial is recommended if you have some familiarity with `cloud-init` or the concepts around it, and are looking to get started as quickly as possible. Here, you will use an LXD container to deploy a `cloud-init` user data script.

Quick-start tutorial with LXD

In this tutorial, we will create our first `cloud-init` user data script and deploy it into an LXD container.

Why LXD?

We'll be using LXD for this tutorial because it provides first class support for `cloud-init` user data, as well as `systemd` support. Because it is container based, it allows us to quickly test and iterate upon our user data definition.

How to use this tutorial

In this tutorial, the commands in each code block can be copied and pasted directly into the terminal. Omit the prompt (`$`) before each command, or use the "copy code" button on the right-hand side of the block, which will copy the command for you without the prompt.

Each code block is preceded by a description of what the command does, and followed by an example of the type of output you should expect to see.

Install and initialise LXD

If you already have LXD set up, you can skip this section. Otherwise, let's install LXD:

```
$ sudo snap install lxd
```

If you don't have snap, you can install LXD using one of the other installation options.

Now we need to initialise LXD. The minimal configuration will be enough for the purposes of this tutorial. If you need to, you can always change the configuration at a later time.

```
$ lxd init --minimal
```

Define our user data

Now that LXD is set up, we can define our user data. Create the following file on your local filesystem at `/tmp/my-user-data`:

```
#cloud-config
runcmd:
  - echo 'Hello, World!' > /var/tmp/hello-world.txt
```

Here, we are defining our `cloud-init` user data in the *#cloud-config* format, using the *runcmd module* to define a command to run. When applied, it will write `Hello, World!` to `/var/tmp/hello-world.txt` (as we shall see later!).

Launch a LXD container with our user data

Now that we have LXD set up and our user data defined, we can launch an instance with our user data:

```
$ lxc launch ubuntu:focal my-test --config=user.user-data="$(cat /tmp/my-user-data)"
```

Verify that `cloud-init` ran successfully

After launching the container, we should be able to connect to our instance using:

```
$ lxc shell my-test
```

You should now be in a shell inside the LXD instance.

Before validating the user data, let's wait for `cloud-init` to complete successfully:

```
$ cloud-init status --wait
```

Which provides the following output:

```
status: done
```

Verify our user data

Now we know that `cloud-init` has been successfully run, we can verify that it received the expected user data we provided earlier:

```
$ cloud-init query userdata
```

Which should print the following to the terminal window:

```
#cloud-config
runcmd:
  - echo 'Hello, World!' > /var/tmp/hello-world.txt
```

We can also assert the user data we provided is a valid cloud-config:

```
$ cloud-init schema --system --annotate
```

Which should print the following:

```
Valid schema user-data
```

Finally, let us verify that our user data was applied successfully:

```
$ cat /var/tmp/hello-world.txt
```

Which should then print:

```
Hello, World!
```

We can see that `cloud-init` has received and consumed our user data successfully!

Tear down

Exit the container shell (by typing **exit** or pressing ctrl-d). Once we have exited the container, we can stop the container using:

```
$ lxc stop my-test
```

We can then remove the container completely using:

```
$ lxc rm my-test
```

What's next?

In this tutorial, we used the *runcmd module* to execute a shell command. The full list of modules available can be found in our *modules documentation*. Each module contains examples of how to use it.

You can also head over to the *examples page* for examples of more common use cases.

2.1.3 WSL tutorial

This tutorial is for learning to use cloud-init within a WSL environment. You will use a cloud-init user data script to customize a WSL instance.

WSL Tutorial

In this tutorial, we will customize a Windows Subsystem for Linux (WSL) instance using cloud-init on Ubuntu.

How to use this tutorial

In this tutorial, the commands in each code block can be copied and pasted directly into a PowerShell Window . Omit the prompt before each command, or use the "copy code" button on the right-hand side of the block, which will copy the command for you without the prompt.

Prerequisites

This tutorial assumes you are running within a Windows 11 or Windows Server 2022 environment. If wsl is already installed, you must be running version 2. You can check your version of wsl by running the following command:

```
PS> wsl --version
```

Example output:

```
WSL version: 2.1.5.0
Kernel version: 5.15.146.1
WSLg version: 1.0.60
MSRDC version: 1.2.5105
Direct3D version: 1.611.1-81528511
DXCore version: 10.0.25131.1002-220531-1700.rs-onecore-base2-hyp
Windows version: 10.0.20348.2402
```

If running this tutorial within a virtualized environment (including in the cloud), ensure that nested virtualization is enabled.

Install WSL

Note: If you have already installed WSL, you can skip this section.

```
PS> wsl --install
```

Example output:

```
Installing: Virtual Machine Platform
Virtual Machine Platform has been installed.
Installing: Windows Subsystem for Linux
Windows Subsystem for Linux has been installed.
Installing: Ubuntu
Ubuntu has been installed.
The requested operation is successful. Changes will not be effective until the system is␣
↪rebooted.
```

Reboot the system when prompted.

Obtain the Ubuntu WSL image

Ubuntu 24.04 is the first Ubuntu version to support cloud-init in WSL, so that is the image that we'll use.

We have two options to obtain the Ubuntu 24.04 WSL image: the Microsoft Store and the Ubuntu image server.

Option #1: The Microsoft Store

If you have access to the Microsoft Store, you can download the Ubuntu 24.04 WSL image from within the app.

Click on the "Get" button to download the image.

Once the image has downloaded, do **NOT** click open as that will start the instance before we have defined our cloud-init user data used to customize the instance.

Once the image has downloaded, you can verify that it is available by running the following command:

```
PS> wsl --list
```

Example output:

```
Windows Subsystem for Linux Distributions:
Ubuntu (Default)
Ubuntu-24.04
```

It should show `Ubuntu-24.04` in the list of available WSL instances.

Option #2: The Ubuntu image server

If the Microsoft Store is not an option, we can instead download the Ubuntu 24.04 WSL image from the Ubuntu image server.

Create a directory under the user's home directory to store the WSL image and install data.

```
PS> mkdir ~\wsl-images
```

Download the Ubuntu 24.04 WSL image.

```
PS> Invoke-WebRequest -Uri https://cloud-images.ubuntu.com/wsl/noble/current/ubuntu-
↪noble-wsl-amd64-wsl.rootfs.tar.gz -OutFile wsl-images\ubuntu-noble-wsl-amd64-wsl.
↪rootfs.tar.gz
```

Import the image into WSL storing it in the `wsl-images` directory.

```
PS> wsl --import Ubuntu-24.04 wsl-images .\wsl-images\ubuntu-noble-wsl-amd64-wsl.rootfs.
↪tar.gz
```

Example output:

```
Import in progress, this may take a few minutes.
The operation completed successfully.
```

Create our user data

User data is the primary way for a user to customize a cloud-init instance. Open Notepad and paste the following:

```
#cloud-config
write_files:
- content: |
    Hello from cloud-init
  path: /var/tmp/hello-world.txt
  permissions: '0777'
```

Save the file to `%USERPROFILE%\.cloud-init\Ubuntu-24.04.user-data`.

For example, if your username is `me`, the path would be `C:\Users\me\.cloud-init\Ubuntu-24.04.user-data`. Ensure that the file is saved with the `.user-data` extension and not as a `.txt` file.

Note: We are creating user data that is tied to the instance we just created, but by changing the filename, we can create user data that applies to multiple or all WSL instances. See *WSL Datasource reference page* for more information.

What is user data?

Before moving forward, let's inspect our `user-data` file.

We created the following contents:

```
#cloud-config
write_files:
- content: |
    Hello from cloud-init
  path: /var/tmp/hello-world.txt
  permissions: '0770'
```

The first line starts with `#cloud-config`, which tells cloud-init what type of user data is in the config. Cloud-config is a YAML-based configuration type that tells cloud-init how to configure the instance being created. Multiple different format types are supported by cloud-init. For more information, see the *documentation describing different formats*.

The remaining lines, as per *the Write Files module docs*, creates a file `/var/tmp/hello-world.txt` with the content `Hello from cloud-init` and permissions allowing anybody on the system to read or write the file.

Start the Ubuntu WSL instance

```
PS> wsl --distribution Ubuntu-24.04
```

The Ubuntu WSL instance will start, and you may be prompted for a username and password.

```
Installing, this may take a few minutes...
Please create a default UNIX user account. The username does not need to match your␣
↪Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username:
New password:
Retype new password:
```

Once the credentials have been entered, you should see a welcome screen similar to the following:

```
Welcome to Ubuntu Noble Numbat (GNU/Linux 5.15.146.1-microsoft-standard-WSL2 x86_64)

* Documentation:  https://help.ubuntu.com
* Management:     https://landscape.canonical.com
* Support:        https://ubuntu.com/pro

System information as of Mon Apr 22 21:06:49 UTC 2024

System load:  0.08               Processes:             51
Usage of /:   0.1% of 1006.85GB  Users logged in:       0
Memory usage: 4%                 IPv4 address for eth0: 172.29.240.255
Swap usage:   0%


This message is shown once a day. To disable it please create the
/root/.hushlogin file.
root@machine:/mnt/c/Users/me#
```

You should now be in a shell inside the WSL instance.

Verify that `cloud-init` ran successfully

Before validating the user data, let's wait for `cloud-init` to complete successfully:

```
$ cloud-init status --wait
```

Which provides the following output:

```
status: done
```

Now we can now see that cloud-init has detected that we running in WSL:

```
$ cloud-id
```

Which provides the following output:

```
wsl
```

Verify our user data

Now we know that `cloud-init` has been successfully run, we can verify that it received the expected user data we provided earlier:

```
$ cloud-init query userdata
```

Which should print the following to the terminal window:

```
#cloud-config
write_files:
- content: |
    Hello from cloud-init
path: /var/tmp/hello-world.txt
permissions: '0770'
```

We can also assert the user data we provided is a valid cloud-config:

```
$ cloud-init schema --system --annotate
```

Which should print the following:

```
Valid schema user-data
```

Finally, let us verify that our user data was applied successfully:

```
$ cat /var/tmp/hello-world.txt
```

Which should then print:

```
Hello from cloud-init
```

We can see that `cloud-init` has received and consumed our user data successfully!

What's next?

In this tutorial, we used the *Write Files module* to write a file to our WSL instance. The full list of modules available can be found in our *modules documentation*. Each module contains examples of how to use it.

You can also head over to the *examples page* for examples of more common use cases.

Cloud-init's WSL reference documentation can be found on the *WSL Datasource reference page*.

2.2 How-to guides

If you have a specific goal in mind and are already familiar with the basics of `cloud-init`, our how-to guides cover some of the more common operations and tasks that you may need to complete.

They will help you to achieve a particular end result, but may require you to understand and adapt the steps to fit your specific requirements.

2.2.1 How do I...?

How to run `cloud-init` locally

It's very likely that you will want to test `cloud-init` locally before deploying it to the cloud. Fortunately, there are several different virtual machine (VM) and container tools that are ideal for this sort of local testing.

- *boot cloud-init with QEMU*
- *boot cloud-init with LXD*
- *boot cloud-init with Libvirt*
- *boot cloud-init with Multipass*

QEMU

QEMU is a general purpose computer hardware emulator that is capable of running virtual machines with hardware acceleration as well as emulating the instruction sets of different architectures than the host that you are running on.

The `NoCloud` datasource allows users to provide their own user data, metadata, or network configuration directly to an instance without running a network service. This is helpful for launching local cloud images with QEMU.

Create your configuration

We will leave the `network-config` and `meta-data` files empty, but populate `user-data` with a cloud-init configuration. You may edit the `network-config` and `meta-data` files if you have a config to provide.

```
$ touch network-config
$ touch meta-data
$ cat >user-data <<EOF
#cloud-config
password: password
chpasswd:
```

(continues on next page)

(continued from previous page)

```
  expire: False
ssh_pwauth: True
EOF
```

Create an ISO disk

This disk is used to pass configuration to cloud-init. Create it with the **genisoimage** command:

```
genisoimage \
    -output seed.img \
    -volid cidata -rational-rock -joliet \
    user-data meta-data network-config
```

Download a cloud image

Download an Ubuntu image to run:

```
wget https://cloud-images.ubuntu.com/jammy/current/jammy-server-cloudimg-amd64.img
```

Note: This example uses emulated CPU instructions on non-x86 hosts, so it may be slow. To make it faster on non-x86 architectures, one can change the image type and qemu-system-<arch> command name to match the architecture of your host machine.

Boot the image with the ISO attached

Boot the cloud image with our configuration, `seed.img`, to QEMU:

```
$ qemu-system-x86_64 -m 1024 -net nic -net user \
    -drive file=jammy-server-cloudimg-amd64.img,index=0,format=qcow2,media=disk \
    -drive file=seed.img,index=1,media=cdrom \
    -machine accel=kvm:tcg
```

The now-booted image will allow for login using the password provided above.

For additional configuration, users can provide much more detailed configuration in the empty `network-config` and `meta-data` files.

Note: See the *Networking config Version 2* page for details on the format and config of network configuration. To learn more about the possible values for metadata, check out the *NoCloud* page.

LXD

LXD offers a streamlined user experience for using Linux system containers. With LXD, the following command initialises a container with user data:

```
$ lxc init ubuntu-daily:jammy test-container
$ lxc config set test-container user.user-data - < userdata.yaml
$ lxc start test-container
```

To avoid the extra commands this can also be done at launch:

```
$ lxc launch ubuntu-daily:jammy test-container --config=user.user-data="$(cat userdata.
↪yaml)"
```

Finally, a profile can be set up with the specific data if you need to launch this multiple times:

```
$ lxc profile create dev-user-data
$ lxc profile set dev-user-data user.user-data - < cloud-init-config.yaml
$ lxc launch ubuntu-daily:jammy test-container -p default -p dev-user-data
```

LXD configuration types

The above examples all show how to pass user data. To pass other types of configuration data use the configuration options specified below:

Data	Configuration option
user data	cloud-init.user-data
vendor data	cloud-init.vendor-data
network config	cloud-init.network-config

See the LXD Instance Configuration docs for more info about configuration values or the LXD Custom Network Configuration document for more about custom network config.

Libvirt

Libvirt is a tool for managing virtual machines and containers.

Create your configuration

We will leave the `network-config` and `meta-data` files empty, but populate user-data with a cloud-init configuration. You may edit the `network-config` and `meta-data` files if you have a config to provide.

```
$ touch network-config
$ touch meta-data
$ cat >user-data <<EOF
#cloud-config
password: password
chpasswd:
  expire: False
```

(continues on next page)

(continued from previous page)

```
ssh_pwauth: True
EOF
```

Download a cloud image

Download an Ubuntu image to run:

```
wget https://cloud-images.ubuntu.com/jammy/current/jammy-server-cloudimg-amd64.img
```

Create an instance

```
virt-install --name cloud-init-001 --memory 4000 --noreboot \
    --os-variant detect=on,name=ubuntujammy \
    --disk=size=10,backing_store="$(pwd)/jammy-server-cloudimg-amd64.img" \
    --cloud-init user-data="$(pwd)/user-data,meta-data=$(pwd)/meta-data,network-config=
↪$(pwd)/network-config"
```

Multipass

Multipass is a cross-platform tool for launching Ubuntu VMs across Linux, Windows, and macOS.

When a user launches a Multipass VM, user data can be passed by adding the `--cloud-init` flag and the appropriate YAML file containing the user data:

```
$ multipass launch bionic --name test-vm --cloud-init userdata.yaml
```

Multipass will validate the user-data cloud-config file before attempting to start the VM. This breaks all cloud-init configuration formats except user data cloud-config.

How to re-run `cloud-init`

How to fully re-run cloud-init

Most cloud-init configuration is only applied to the system once. This means that simply rebooting the system will only re-run a subset of cloud-init. Cloud-init provides two different options for re-running cloud-init for debugging purposes.

> **Warning:** Making cloud-init run again may be destructive and must never be done on a production system. Artefacts such as ssh keys or passwords may be overwritten.

Remove the logs and cache, then reboot

This method will reboot the system as if cloud-init never ran. This command does not remove all cloud-init artifacts from previous runs of cloud-init, but it will clean enough artifacts to allow cloud-init to think that it hasn't run yet. It will then re-run after a reboot.

```
cloud-init clean --logs --reboot
```

Run a single cloud-init module

If you are using *user data cloud-config* format, you might wish to re-run just a single configuration module. Cloud-init provides the ability to run a single module in isolation and separately from boot. This command is:

```
$ sudo cloud-init single --name cc_ssh --frequency always
```

Example output:

```
...
Generating public/private ed25519 key pair
...
```

This subcommand is not called by the init system. It can be called manually to load the configured datasource and run a single cloud-config module once, using the cached user data and metadata after the instance has booted.

Note: Each cloud-config module has a module `FREQUENCY` configured: `PER_INSTANCE`, `PER_BOOT`, `PER_ONCE` or `PER_ALWAYS`. When a module is run by cloud-init, it stores a semaphore file in `/var/lib/cloud/instance/sem/config_<module_name>.<frequency>` which marks when the module last successfully ran. Presence of this semaphore file prevents a module from running again if it has already been run.

Inspect `cloud-init.log` for output of what operations were performed as a result.

How to partially re-run cloud-init

If the behavior you are testing runs on every boot, there are a couple of ways to test this behavior.

Manually run cloud-init stages

Note that during normal boot of cloud-init, the init system runs these stages at specific points during boot. This means that running the code manually after booting the system may cause the code to interact with the system in a different way than it does while it boots.

```
cloud-init init --local
cloud-init init
cloud-init modules --mode=config
cloud-init modules --mode=final
```

Reboot the instance

Rebooting the instance will take a little bit longer, however it will make cloud-init stages run at the correct times during boot, so it will behave more correctly.

```
reboot -h now
```

How to change a module's run frequency

You may want to change the default frequency at which a module runs, for example, to make the module run on every boot.

To override the default frequency, you will need to modify the module list in `/etc/cloud/cloud.cfg`:

1. Change the module from a string (default) to a list.
2. Set the first list item to the module name and the second item to the frequency.

Example

The following example demonstrates how to log boot times to a file every boot.

Update `/etc/cloud/cloud.cfg`:

```
    cloud_final_modules:
    # list shortened for brevity
     - [phone_home, always]
     - final_message
     - power_state_change
```

Then your user data could then be:

```
#cloud-config
phone_home:
    url: http://example.com/$INSTANCE_ID/
    post: all
```

How to validate user data cloud config

The two most common issues with cloud config user data are:

1. Incorrectly formatted YAML
2. The first line does not start with `#cloud-config`

Static user data validation

Cloud-init is capable of validating cloud config user data directly from its datasource (i.e. on a running cloud instance). To do this, you can run:

```
sudo cloud-init schema --system --annotate
```

Or, to test YAML in a specific file:

```
cloud-init schema -c test.yml --annotate
```

Example output:

```
$ cloud-init schema --config-file=test.yaml --annotate
#cloud-config
users:
  - name: holmanb        # E1,E2,E3
    gecos: Brett Holman
    primary_group: holmanb
    lock_passwd: false
    invalid_key: true

# Errors: -------------
# E1: Additional properties are not allowed ('invalid_key' was unexpected)
# E2: {'name': 'holmanb', 'gecos': 'Brett Holman', 'primary_group': 'holmanb', 'lock_
↪passwd': False, 'invalid_key': True} is not of type 'array'
# E3: {'name': 'holmanb', 'gecos': 'Brett Holman', 'primary_group': 'holmanb', 'lock_
↪passwd': False, 'invalid_key': True} is not of type 'string'
```

Debugging

If your user-data cloud config is correct according to the *cloud-init schema* command, but you are still having issues, then please refer to our *debugging guide*.

To report any bugs you find, *refer to this guide*.

How to debug cloud-init

There are several cloud-init *failure modes* that one may need to debug. Debugging is specific to the scenario, but the starting points are often similar:

- *I cannot log in*
- *Cloud-init did not run*
- *Cloud-init did the unexpected*
- *Cloud-init never finished running*

I can't log in to my instance

One of the more challenging scenarios to debug is when you don't have shell access to your instance. You have a few options:

1. Acquire log messages from the serial console and check for any errors.
2. To access instances without SSH available, create a user with password access (using the user-data) and log in via the cloud serial port console. This only works if `cc_users_groups` successfully ran.
3. Try running the same user-data locally, such as in one of the *tutorials*. Use LXD or QEMU locally to get a shell or logs then debug with *these steps*.
4. Try copying the image to your local system, mount the filesystem locally and inspect the image logs for clues.

Cloud-init did not run

1. Check the output of `cloud-init status --long`
   - what is the value of the `'extended_status'` key?
   - what is the value of the `'boot_status_code'` key?

   See *our reported status explanation* for more information on the status.
2. Check the contents of `/run/cloud-init/ds-identify.log`

   This log file is used when the platform that cloud-init is running on *is detected*. This stage enables or disables cloud-init.
3. Check the status of the services

   ```
   systemctl status cloud-init-local.service cloud-init.service\
      cloud-config.service cloud-final.service
   ```

   Cloud-init may have started to run, but not completed. This shows how many, and which, cloud-init stages completed.

Cloud-init ran, but didn't do what I want it to

1. If you are using cloud-init's user data *cloud config*, make sure to *validate your user data cloud config*
2. Check for errors in `cloud-init status --long`
   - what is the value of the `'errors'` key?
   - what is the value of the `'recoverable_errors'` key?

   See *our guide on exported errors* for more information on these exported errors.
3. For more context on errors, check the logs files:
   - `/var/log/cloud-init.log`
   - `/var/log/cloud-init-output.log`

   Identify errors in the logs and the lines preceding these errors.

   Ask yourself:
   - According to the log files, what went wrong?

- How does the cloud-init error relate to the configuration provided to this instance?
- What does the documentation say about the parts of the configuration that relate to this error? Did a configuration module fail?
- What *failure state* is cloud-init in?

Cloud-init never finished running

There are many reasons why cloud-init may fail to complete. Some reasons are internal to cloud-init, but in other cases, cloud-init failure to complete may be a symptom of failure in other components of the system, or the result of a user configuration.

External reasons

- Other services failed or are stuck.
- Bugs in the kernel or drivers.
- Bugs in external userspace tools that are called by `cloud-init`.

Internal reasons

- A command in `bootcmd` or `runcmd` that never completes (e.g., running **`cloud-init status --wait`** will deadlock).
- Configurations that disable timeouts or set extremely high timeout values.

To start debugging

1. Check `dmesg` for errors:

   ```
   dmesg -T | grep -i -e warning -e error -e fatal -e exception
   ```

2. Investigate other systemd services that failed

   ```
   systemctl --failed
   ```

3. Check the output of `cloud-init status --long`

   - what is the value of the `'extended_status'` key?
   - what is the value of the `'boot_status_code'` key?

   See *our guide on exported errors* for more information on these exported errors.

4. Inspect running services *boot stage*:

   ```
   $ systemctl list-jobs --after
   JOB UNIT                                          TYPE  STATE
   150 cloud-final.service                           start waiting
   └─      waiting for job 147 (cloud-init.target/start)   -     -
   155 blocking-daemon.service                         start running
   └─      waiting for job 150 (cloud-final.service/start) -     -
   ```

(continues on next page)

(continued from previous page)

```
147 cloud-init.target                                  start waiting

3 jobs listed.
```

In the above example we can see that `cloud-final.service` is waiting and is ordered before `cloud-init.target`, and that `blocking-daemon.service` is currently running and is ordered before `cloud-final.service`. From this output, we deduce that cloud-init is not complete because the service named `blocking-daemon.service` hasn't yet completed, and that we should investigate `blocking-daemon.service` to understand why it is still running.

5. Use the PID of the running service to find all running subprocesses. Any running process that was spawned by cloud-init may be blocking cloud-init from continuing.

```
pstree <PID>
```

Ask yourself:

- Which process is still running?
- Why is this process still running?
- How does this process relate to the configuration that I provided?

6. For more context on errors, check the logs files:
   - `/var/log/cloud-init.log`
   - `/var/log/cloud-init-output.log`

   Identify errors in the logs and the lines preceding these errors.

   Ask yourself:

   - According to the log files, what went wrong?
   - How does the cloud-init error relate to the configuration provided to this instance?
   - What does the documentation say about the parts of the configuration that relate to this error?

Reported status

When interacting with cloud-init, it may be useful to know whether cloud-init has run, or is currently running. Since cloud-init consists of several different stages, interacting directly with your init system might yield different reported results than one might expect, unless one has intimate knowledge of cloud-init's *boot stages*.

Cloud-init status

To simplify this, cloud-init provides a tool, `cloud-init status` to report the current status of cloud-init.

```
$ cloud-init status
"done"
```

Cloud-init's extended status

Cloud-init is also capable of reporting when cloud-init has not been able to complete the tasks described in a user configuration. If cloud-init has experienced issues while running, the extended status will include the word "degraded" in its status.

Cloud-init can report its internal state via the `status --format json` subcommand under the `extended_status` key.

```
$ cloud-init status --format json
{
    "boot_status_code": "enabled-by-generator",
    "datasource": "lxd",
    "detail": "DataSourceLXD",
    "errors": [],
    "extended_status": "degraded done",
    "init": {
        "errors": [],
        "finished": 1708550839.1837437,
        "recoverable_errors": {},
        "start": 1708550838.6881146
    },
    "init-local": {
        "errors": [],
        "finished": 1708550838.0196638,
        "recoverable_errors": {},
        "start": 1708550837.7719762
    },
    "last_update": "Wed, 21 Feb 2024 21:27:24 +0000",
    "modules-config": {
        "errors": [],
        "finished": 1708550843.8297973,
        "recoverable_errors": {
        "WARNING": [
            "Removing /etc/apt/sources.list to favor deb822 source format"
        ]
        },
        "start": 1708550843.7163966
    },
    "modules-final": {
        "errors": [],
        "finished": 1708550844.0884337,
        "recoverable_errors": {},
        "start": 1708550844.029698
    },
    "recoverable_errors": {
        "WARNING": [
        "Removing /etc/apt/sources.list to favor deb822 source format"
        ]
    },
    "stage": null,
    "status": "done"
}
```

See the list of all possible reported statuses:

```
"not started"
"running"
"done"
"error - done"
"error - running"
"degraded done"
"degraded running"
"disabled"
```

Cloud-init enablement status

Separately from the current running status described above, cloud-init can also report how it was disabled or enabled. This can be viewed by checking the *boot_status_code* in `cloud-init status --long`, which may contain any of the following states:

- `'unknown'`: `ds-identify` has not run yet to determine if cloud-init should be run during this boot
- `'disabled-by-marker-file'`: `/etc/cloud/cloud-init.disabled` exists which prevents cloud-init from ever running
- `'disabled-by-generator'`: `ds-identify` determined no applicable cloud-init datasources
- `'disabled-by-kernel-command-line'`: kernel command line contained cloud-init=disabled
- `'disabled-by-environment-variable'`: environment variable `KERNEL_CMDLINE` contained `cloud-init=disabled`
- `'enabled-by-kernel-command-line'`: kernel command line contained cloud-init=enabled
- `'enabled-by-generator'`: `ds-identify` detected possible cloud-init datasources
- `'enabled-by-sysvinit'`: enabled by default in SysV init environment

See *our explanation of failure states* for more information.

Reporting bugs

In this guide, we will show you how to:

1) Collect logs to support your bug report.
2) File bugs to the upstream `cloud-init` project via GitHub Issues.
3) Report issues for distro-specific packages.

Collect logs

To aid in debugging, please collect the necessary logs. To do so, run the **`collect-logs`** subcommand to produce a tarfile that you can easily upload:

```
$ sudo cloud-init collect-logs
```

Example output:

```
Wrote /home/ubuntu/cloud-init.tar.gz
```

If your version of `cloud-init` does not have the **`collect-logs`** subcommand, then please manually collect the base log files by running the following:

```
$ sudo dmesg > dmesg.txt
$ sudo journalctl -o short-precise > journal.txt
$ sudo tar -cvf cloud-init.tar dmesg.txt journal.txt /run/cloud-init \
   /var/log/cloud-init.log /var/log/cloud-init-output.log
```

Report upstream bugs

Bugs for upstream `cloud-init` are tracked using GitHub Issues. To file a bug:

1. Collect the necessary debug logs as described above.
2. Report an upstream cloud-init bug on GitHub.

If debug logs are not provided, you will be asked for them before any further time is spent debugging. If you are unable to obtain the required logs please explain why in the bug.

If your bug is for a specific distro using `cloud-init`, please first consider reporting it with the downstream distro or confirm that it still occurs with the latest upstream `cloud-init` code. See the following section for details on specific distro reporting.

Distro-specific issues

For issues specific to your distro please use one of the following distro-specific reporting mechanisms:

Ubuntu

To report a bug on Ubuntu use the **`ubuntu-bug`** command on the affected system to automatically collect the necessary logs and file a bug on Launchpad:

```
$ ubuntu-bug cloud-init
```

If that does not work or is not an option, please collect the logs using the commands in the above Collect Logs section and then report the bug on the Ubuntu bug tracker. Make sure to attach your collected logs!

Debian

To file a bug against the Debian package of `cloud-init` please use the Debian bug tracker to file against 'Package: cloud-init'. See the Debian bug reporting wiki page for more details.

Red Hat, CentOS and Fedora

To file a bug against the Red Hat or Fedora packages of `cloud-init` please use the Red Hat bugzilla.

SUSE and openSUSE

To file a bug against the SUSE packages of `cloud-init` please use the SUSE bugzilla.

Arch Linux

To file a bug against the Arch package of `cloud-init` please use the Arch Linux Bugtracker. See the Arch Linux bug reporting wiki for more details.

How to identify the datasource I'm using

To correctly set up an instance, `cloud-init` must correctly identify the cloud it is on. Therefore, knowing which datasource is being used on an instance launch can aid in debugging.

To find out which datasource is being used run the **`cloud-id`** command:

```
cloud-id
```

This will tell you which datasource is being used – for example:

```
nocloud
```

If the `cloud-id` is not what is expected, then running the `ds-identify` script in debug mode and providing that in a bug report can aid in resolving any issues:

```
sudo DEBUG_LEVEL=2 DI_LOG=stderr /usr/lib/cloud-init/ds-identify --force
```

The `force` parameter allows the command to be run again since the instance has already launched. The other options increase the verbosity of logging and outputs the logs to `STDERR`.

How to disable cloud-init

One may wish to disable cloud-init to ensure that it doesn't do anything on subsequent boots. Some parts of cloud-init may run once per boot otherwise.

There are three cross-platform methods of disabling `cloud-init`.

Method 1: text file

To disable cloud-init, create the empty file `/etc/cloud/cloud-init.disabled`. During boot the operating system's init system will check for the existence of this file. If it exists, cloud-init will not be started.

Example:

```
$ touch /etc/cloud/cloud-init.disabled
```

Method 2: kernel command line

To disable cloud-init, add `cloud-init=disabled` to the kernel command line.

Example (using GRUB2 with Ubuntu):

```
$ echo 'GRUB_CMDLINE_LINUX="cloud-init=disabled"' >> /etc/default/grub
$ grub-mkconfig -o /boot/efi/EFI/ubuntu/grub.cfg
```

Method 3: environment variable

To disable cloud-init, pass the environment variable `KERNEL_CMDLINE=cloud-init=disabled` into each of cloud-init's processes.

Example (using systemd):

```
$ echo "DefaultEnvironment=KERNEL_CMDLINE=cloud-init=disabled" >> /etc/systemd/system.
↪conf
```

Test pre-release cloud-init

After the cloud-init team creates an upstream release, cloud-init will be released in the -proposed APT repository for a *period of testing*. Users are encouraged to test their workloads on this pending release so that bugs can be caught and fixed prior to becoming more broadly available via the -updates repository. This guide describes how to test the pre-release package on Ubuntu.

Add the -proposed repository pocket

The -proposed repository pocket will contain the cloud-init package to be tested prior to release in the -updates pocket.

```
echo "deb http://archive.ubuntu.com/ubuntu $(lsb_release -sc)-proposed main" >> /etc/apt/
↪sources.list.d/proposed.list
apt update
```

Install the pre-release cloud-init package

```
apt install cloud-init
```

Test the package

Whatever workload you use cloud-init for in production is the best one to test. This ensures that you can discover and report any bugs that the cloud-init developers missed during testing before cloud-init gets released more broadly.

If issues are found during testing, please file a new cloud-init bug and leave a message in the #cloud-init IRC channel.

Remove the proposed repository

Do this to avoid unintentionally installing other unreleased packages.

```
rm -f /etc/apt/sources.list.d/proposed.list
apt update
```

Remove artifacts and reboot

This will cause cloud-init to rerun as if it is a first boot.

```
sudo cloud-init clean --logs --reboot
```

2.3 Explanation

Our explanatory and conceptual guides are written to provide a better understanding of how `cloud-init` works. They enable you to expand your knowledge and become better at using and configuring `cloud-init`.

2.3.1 Introduction to cloud-init

Managing and configuring cloud instances and servers can be a complex and time-consuming task. Cloud-init is an open source initialisation tool that was designed to make it easier to get your systems up and running with a minimum of effort, already configured according to your needs.

It's most often used by developers, system administrators and other IT professionals to automate configuration of VMs, cloud instances, or machines on a network. For brevity, we'll refer only to instances for the rest of this page, but assume we're also including bare metal and VMs in the discussion as well. By automating routine setup tasks, cloud-init ensures repeatability and efficiency in your system provisioning.

What is the benefit of cloud-init?

When you deploy a new cloud instance, cloud-init takes an initial configuration that you supply, and it automatically applies those settings when the instance is created. It's rather like writing a to-do list, and then letting cloud-init deal with that list for you.

The real power of cloud-init comes from the fact that you can re-use your configuration instructions as often as you want, and always get consistent, reliable results. If you're a system administrator and you want to deploy a whole fleet of machines, you can do so with a fraction of the time and effort it would take to manually provision them.

What does cloud-init do?

Cloud-init can handle a range of tasks that normally happen when a new instance is created. It's responsible for activities like setting the hostname, configuring network interfaces, creating user accounts, and even running scripts. This streamlines the deployment process; your cloud instances will all be automatically configured in the same way, which reduces the chance to introduce human error.

How does cloud-init work?

The operation of cloud-init broadly takes place in two separate phases during the boot process. The first phase is during the early (local) boot stage, before networking has been enabled. The second is during the late boot stages, after cloud-init has applied the networking configuration.

During early boot

In this pre-networking stage, cloud-init discovers the datasource, obtains all the configuration data from it, and configures networking. In this phase, it will:

- **Identify the datasource**: The hardware is checked for built-in values that will identify the datasource your instance is running on. The datasource is the source of all configuration data.
- **Fetch the configuration**: Once the datasource is identified, cloud-init fetches the configuration data from it. This data tells cloud-init what actions to take. This can be in the form of:
  - **Metadata** about the instance, such as the machine ID, hostname and network config, or
  - **Vendor data** and/or **user data**. These take the same form, although Vendor data is provided by the cloud vendor, and user data is provided by the user. These data are usually applied in the post-networking phase, and might include:
    - Hardware optimisations
    - Integration with the specific cloud platform
    - SSH keys
    - Custom scripts
- **Write network configuration**: Cloud-init writes the network configuration and configures DNS, ready to be applied by the networking services when they come up.

During late boot

In the boot stages that come after the network has been configured, cloud-init runs through the tasks that were not critical for provisioning. This is where it configures the running instance according to your needs, as specified in the vendor data and/or user data. It will take care of:

- **Configuration management**: Cloud-init can interact with tools like Puppet, Ansible, or Chef to apply more complex configuration - and ensure the system is up-to-date.
- **Installing software**: Cloud-init can install software at this stage, and run software updates to make sure the system is fully up-to-date and ready to use.
- **User accounts**: Cloud-init is able to create and modify user accounts, set default passwords, and configure permissions.

- **Execute user scripts**: If any custom scripts were provided in the user data, cloud-init can run them. This allows additional specified software to be installed, security settings to be applied, etc. It can also inject SSH keys into the instance's `authorized_keys` file, which allows secure remote access to the machine.

After this stage is complete, your instance is fully configured!

What's next?

Now that you have an overview of the basics of what cloud-init is, what it does and how it works, you will probably want to *try it out for yourself*.

You can also read in more detail about what cloud-init does *during the different boot stages*, and the *types of configuration* you can pass to cloud-init and how they're used.

2.3.2 Configuration sources

Internally, `cloud-init` builds a single configuration that is then referenced throughout the life of `cloud-init`. The configuration is built from multiple sources such that if a key is defined in multiple sources, the higher priority source overwrites the lower priority source.

Base configuration

The base configuration format uses YAML version 1.1, but may be declared as jinja templates which cloud-init will render at runtime with *instance data* variables.

From lowest priority to highest, configuration sources are:

- **Hardcoded config** Config that lives within the source of `cloud-init` and cannot be changed.
- **Configuration directory**: Anything defined in `/etc/cloud/cloud.cfg` and `/etc/cloud/cloud.cfg.d/*.cfg`.
- **Runtime config**: Anything defined in `/run/cloud-init/cloud.cfg`.
- **Kernel command line**: On the kernel command line, anything found between `cc:` and `end_cc` will be interpreted as cloud-config user data.

These four sources make up the base configuration. The contents of this configuration are defined in the *base configuration reference page*.

Note: Base configuration may contain *cloud-config* which may be overridden by vendor data and user data.

Vendor and user data

Added to the base configuration are *vendor data* and *user data* which are both provided by the datasource.

These get fetched from the datasource and are defined at instance launch.

Network configuration

Network configuration happens independently from other `cloud-init` configuration. See *network configuration documentation* for more information.

Specifying configuration

End users

Pass *user data* to the cloud provider. Every platform supporting `cloud-init` will provide a method of supplying user data. If you're unsure how to do this, reference the documentation provided by the cloud platform you're on. Additionally, there may be related `cloud-init` documentation in the *datasource* section.

Once an instance has been initialised, the user data may not be edited. It is sourced directly from the cloud, so even if you find a local file that contains user data, it will likely be overwritten in the next boot.

Distro providers

Modify the base config. This often involves submitting a PR to modify the base cloud.cfg template, which is used to customise `/etc/cloud/cloud.cfg` per distro. Additionally, a file can be added to `/etc/cloud/cloud.cfg.d` to override a piece of the base configuration.

Cloud providers

Pass vendor data. This is the preferred method for clouds to provide their own customisation. In some cases, it may make sense to modify the base config in the same manner as distro providers on cloud-supported images.

2.3.3 Boot stages

There are five stages to boot which are run seqentially: `Detect`, `Local`, `Network`, `Config` and `Final`

Visual representation of cloud-init boot stages with respect to network config and system accessibility:

Detect

A platform identification tool called `ds-identify` runs in the first stage. This tool detects which platform the instance is running on. This tool is integrated into the init system to disable cloud-init when no platform is found, and enable cloud-init when a valid platform is detected. This stage might not be present for every installation of cloud-init.

Local

systemd service	`cloud-init-local.service`
runs	as soon as possible with `/` mounted read-write
blocks	as much of boot as possible, *must* block network
modules	none

The purpose of the local stage is to:

- Locate "local" data sources, and

- Apply networking configuration to the system (including "fallback").

In most cases, this stage does not do much more than that. It finds the datasource and determines the network configuration to be used. That network configuration can come from:

- **datasource**: Cloud-provided network configuration via metadata.
- **fallback**: `Cloud-init`'s fallback networking consists of rendering the equivalent to `dhcp on eth0`, which was historically the most popular mechanism for network configuration of a guest.
- **none**: Network configuration can be disabled by writing the file `/etc/cloud/cloud.cfg` with the content: `network: {config: disabled}`.

If this is an instance's first boot, then the selected network configuration is rendered. This includes clearing of all previous (stale) configuration including persistent device naming with old MAC addresses.

This stage must block network bring-up or any stale configuration that might have already been applied. Otherwise, that could have negative effects such as DHCP hooks or broadcast of an old hostname. It would also put the system in an odd state to recover from, as it may then have to restart network devices.

`Cloud-init` then exits and expects for the continued boot of the operating system to bring network configuration up as configured.

Note: In the past, local datasources have been only those that were available without network (such as 'ConfigDrive'). However, as seen in the recent additions to the *DigitalOcean datasource*, even data sources that require a network can operate at this stage.

Network

systemd service	`cloud-init.service`
runs	after local stage and configured networking is up
blocks	majority of remaining boot (e.g. SSH and console login)
modules	*cloud_init_modules* in `/etc/cloud/cloud.cfg`

This stage requires all configured networking to be online, as it will fully process any user data that is found. Here, processing means it will:

- retrieve any `#include` or `#include-once` (recursively) including http,
- decompress any compressed content, and
- run any part-handler found.

This stage runs the `disk_setup` and `mounts` modules which may partition and format disks and configure mount points (such as in `/etc/fstab`). Those modules cannot run earlier as they may receive configuration input from sources only available via the network. For example, a user may have provided user data in a network resource that describes how local mounts should be done.

On some clouds, such as Azure, this stage will create filesystems to be mounted, including ones that have stale (previous instance) references in `/etc/fstab`. As such, entries in `/etc/fstab` other than those necessary for cloud-init to run should not be done until after this stage.

A part-handler and *boothooks* will run at this stage.

After this stage completes, expect to be able to access the system via serial console login or SSH.

Config

systemd service	`cloud-config.service`
runs	after network
blocks	nothing
modules	*cloud_config_modules* in `/etc/cloud/cloud.cfg`

This stage runs config modules only. Modules that do not really have an effect on other stages of boot are run here, including `runcmd`.

Final

systemd service	`cloud-final.service`
runs	as final part of boot (traditional "rc.local")
blocks	nothing
modules	*cloud_final_modules* in `/etc/cloud/cloud.cfg`

This stage runs as late in boot as possible. Any scripts that a user is accustomed to running after logging into a system should run correctly here. Things that run here include:

- package installations,
- configuration management plugins (Ansible, Puppet, Chef, salt-minion), and
- user-defined scripts (i.e., shell scripts passed as user data).

For scripts external to `cloud-init` looking to wait until `cloud-init` is finished, the **`cloud-init status --wait`** subcommand can help block external scripts until `cloud-init` is done without having to write your own `systemd` units dependency chains. See *status* for more info.

First boot determination

`Cloud-init` has to determine whether or not the current boot is the first boot of a new instance, so that it applies the appropriate configuration. On an instance's first boot, it should run all "per-instance" configuration, whereas on a subsequent boot it should run only "per-boot" configuration. This section describes how `cloud-init` performs this determination, as well as why it is necessary.

When it runs, `cloud-init` stores a cache of its internal state for use across stages and boots.

If this cache is present, then `cloud-init` has run on this system before[1]. There are two cases where this could occur. Most commonly, the instance has been rebooted, and this is a second/subsequent boot. Alternatively, the filesystem has been attached to a *new* instance, and this is the instance's first boot. The most obvious case where this happens is when an instance is launched from an image captured from a launched instance.

By default, `cloud-init` attempts to determine which case it is running in by checking the instance ID in the cache against the instance ID it determines at runtime. If they do not match, then this is an instance's first boot; otherwise, it's a subsequent boot. Internally, `cloud-init` refers to this behaviour as `check`.

[1] It follows that if this cache is not present, `cloud-init` has not run on this system before, so this is unambiguously this instance's first boot.

This behaviour is required for images captured from launched instances to behave correctly, and so is the default that generic cloud images ship with. However, there are cases where it can cause problems[2]. For these cases, `cloud-init` has support for modifying its behaviour to trust the instance ID that is present in the system unconditionally. This means that `cloud-init` will never detect a new instance when the cache is present, and it follows that the only way to cause `cloud-init` to detect a new instance (and therefore its first boot) is to manually remove `cloud-init`'s cache. Internally, this behaviour is referred to as `trust`.

To configure which of these behaviours to use, `cloud-init` exposes the `manual_cache_clean` configuration option. When `false` (the default), `cloud-init` will `check` and clean the cache if the instance IDs do not match (this is the default, as discussed above). When `true`, `cloud-init` will `trust` the existing cache (and therefore not clean it).

Manual cache cleaning

`Cloud-init` ships a command for manually cleaning the cache: **`cloud-init clean`**. See *clean*'s documentation for further details.

Reverting `manual_cache_clean` setting

Currently there is no support for switching an instance that is launched with `manual_cache_clean: true` from `trust` behaviour to `check` behaviour, other than manually cleaning the cache.

Warning: If you want to capture an instance that is currently in `trust` mode as an image for launching other instances, you **must** manually clean the cache. If you do not do so, then instances launched from the captured image will all detect their first boot as a subsequent boot of the captured instance, and will not apply any per-instance configuration.

This is a functional issue, but also a potential security one: `cloud-init` is responsible for rotating SSH host keys on first boot, and this will not happen on these instances.

2.3.4 User data formats

User data is opaque configuration data provided by a platform to an instance at launch configure the instance. User data can be one of the following types.

[2] A couple of ways in which this strict reliance on the presence of a datasource has been observed to cause problems:

- If a cloud's metadata service is flaky and `cloud-init` cannot obtain the instance ID locally on that platform, `cloud-init`'s instance ID determination will sometimes fail to determine the current instance ID, which makes it impossible to determine if this is an instance's first or subsequent boot (#1885527).
- If `cloud-init` is used to provision a physical appliance or device and an attacker can present a datasource to the device with a different instance ID, then `cloud-init`'s default behaviour will detect this as an instance's first boot and reset the device using the attacker's configuration (this has been observed with the *NoCloud datasource* in #1879530).

Cloud config data

Cloud-config is the preferred user data format. The cloud config format is a declarative syntax which uses YAML version 1.1 with keys which describe desired instance state. Cloud-config can be used to define how an instance should be configured in a human-friendly format.

These things may include:

- performing package upgrades on first boot
- configuration of different package mirrors or sources
- initial user or group setup
- importing certain SSH keys or host keys
- *and many more...*

See the *Cloud config examples* section for a commented set of examples of supported cloud config formats.

Begins with: `#cloud-config` or `Content-Type: text/cloud-config` when using a MIME archive.

Note: Cloud config data can also render cloud instance metadata variables using *jinja templates*.

User data script

Typically used by those who just want to execute a shell script.

Begins with: `#!` or `Content-Type: text/x-shellscript` when using a MIME archive.

User data scripts can optionally render cloud instance metadata variables using *jinja templates*.

Example script

Create a script file `myscript.sh` that contains the following:

```
#!/bin/sh
echo "Hello World.  The time is now $(date -R)!" | tee /root/output.txt
```

Now run:

```
$ euca-run-instances --key mykey --user-data-file myscript.sh ami-a07d95c9
```

Kernel command line

When using the NoCloud datasource, users can pass user data via the kernel command line parameters. See the *NoCloud datasource* and *Kernel command line* documentation for more details.

Gzip compressed content

Content found to be gzip compressed will be uncompressed. The uncompressed data will then be used as if it were not compressed. This is typically useful because user data is limited to ~16384[1] bytes.

MIME multi-part archive

This list of rules is applied to each part of this multi-part file. Using a MIME multi-part file, the user can specify more than one type of data.

For example, both a user data script and a cloud-config type could be specified.

Supported content-types are listed from the `cloud-init` subcommand **`make-mime`**:

```
$ cloud-init devel make-mime --list-types
```

Example output:

```
cloud-boothook
cloud-config
cloud-config-archive
cloud-config-jsonp
jinja2
part-handler
x-include-once-url
x-include-url
x-shellscript
x-shellscript-per-boot
x-shellscript-per-instance
x-shellscript-per-once
```

Helper subcommand to generate MIME messages

The `cloud-init` make-mime subcommand can also generate MIME multi-part files.

The **`make-mime`** subcommand takes pairs of (filename, "text/" mime subtype) separated by a colon (e.g., `config.yaml:cloud-config`) and emits a MIME multipart message to `stdout`.

Examples

Create user data containing both a cloud-config (`config.yaml`) and a shell script (`script.sh`)

```
$ cloud-init devel make-mime -a config.yaml:cloud-config -a script.sh:x-shellscript >␣
↪userdata
```

Create user data containing 3 shell scripts:

- `always.sh` - run every boot
- `instance.sh` - run once per instance
- `once.sh` - run once

[1] See your cloud provider for applicable user-data size limitations...

```
$ cloud-init devel make-mime -a always.sh:x-shellscript-per-boot -a instance.sh:x-
↪shellscript-per-instance -a once.sh:x-shellscript-per-once
```

include file

This content is an `include` file.

The file contains a list of URLs, one per line. Each of the URLs will be read and their content will be passed through this same set of rules, i.e., the content read from the URL can be gzipped, MIME multi-part, or plain text. If an error occurs reading a file the remaining files will not be read.

Begins with: `#include` or `Content-Type: text/x-include-url` when using a MIME archive.

cloud-boothook

One line `#cloud-boothook` header and then executable payload.

This is run very early on the boot process, during the *Network boot stage*, even before `cc_bootcmd`.

This can be used when something has to be configured very early on boot, potentially on every boot, with less convenience as `cc_bootcmd` but more flexibility.

Note: Boothooks are executed on every boot. The environment variable `INSTANCE_ID` will be set to the current instance ID. `INSTANCE_ID` can be used to implement a *once-per-instance* type of functionality.

Begins with: `#cloud-boothook`.

Example with simple script

```
#cloud-boothook
#!/bin/sh
echo 192.168.1.130 us.archive.ubuntu.com > /etc/hosts
```

Example of once-per-instance script

```
#cloud-boothook
#!/bin/sh

PERSIST_ID=/var/lib/cloud/first-instance-id
_id=""
if [ -r $PERSIST_ID ]; then
  _id=$(cat /var/lib/cloud/first-instance-id)
fi

if [ -z $_id ]  || [ $INSTANCE_ID != $_id ]; then
  echo 192.168.1.130 us.archive.ubuntu.com >> /etc/hosts
fi
sudo echo $INSTANCE_ID > $PERSIST_ID
```

Part-handler

This is a *part-handler*: It contains custom code for either supporting new mime-types in multi-part user data, or overriding the existing handlers for supported mime-types. It will be written to a file in `/var/lib/cloud/data` based on its filename (which is generated).

This must be Python code that contains a `list_types` function and a `handle_part` function. Once the section is read the `list_types` method will be called. It must return a list of mime-types that this *part-handler* handles. Since MIME parts are processed in order, a *part-handler* part must precede any parts with mime-types it is expected to handle in the same user data.

The `handle_part` function must be defined like:

```
def handle_part(data, ctype, filename, payload):
  # data = the cloudinit object
  # ctype = "__begin__", "__end__", or the mime-type of the part that is being handled.
  # filename = the filename of the part (or a generated filename if none is present in␣
↪mime data)
  # payload = the parts' content
```

`Cloud-init` will then call the `handle_part` function once before it handles any parts, once per part received, and once after all parts have been handled. The `'__begin__'` and `'__end__'` sentinels allow the part handler to do initialisation or teardown before or after receiving any parts.

Begins with: `#part-handler` or `Content-Type: text/part-handler` when using a MIME archive.

Example

```
#part-handler

def list_types():
    # return a list of mime-types that are handled by this module
    return(["text/plain", "text/go-cubs-go"])

def handle_part(data, ctype, filename, payload):
    # data: the cloudinit object
    # ctype: '__begin__', '__end__', or the specific mime-type of the part
    # filename: the filename for the part, or dynamically generated part if
    #           no filename is given attribute is present
    # payload: the content of the part (empty for begin or end)
    if ctype == "__begin__":
       print("my handler is beginning")
       return
    if ctype == "__end__":
       print("my handler is ending")
       return

    print(f"==== received ctype={ctype} filename={filename} ====")
    print(payload)
    print(f"==== end ctype={ctype} filename={filename}")
```

Also, this blog post offers another example for more advanced usage.

Disabling user data

`Cloud-init` can be configured to ignore any user data provided to instance. This allows custom images to prevent users from accidentally breaking closed appliances. Setting `allow_userdata: false` in the configuration will disable `cloud-init` from processing user data.

2.3.5 Events and updates

Events

`Cloud-init` will fetch and apply cloud and user data configuration upon several event types. The two most common events for `cloud-init` are when an instance first boots and any subsequent boot thereafter (reboot). In addition to boot events, `cloud-init` users and vendors are interested in when devices are added. `Cloud-init` currently supports the following event types:

- `BOOT_NEW_INSTANCE`: New instance first boot.
- `BOOT`: Any system boot other than `BOOT_NEW_INSTANCE`.
- `BOOT_LEGACY`: Similar to `BOOT`, but applies networking config twice each boot: once during the *Local stage*, then again in the *Network stage*. As this behaviour was previously the default behaviour, this option exists to prevent regressing such behaviour.
- `HOTPLUG`: Dynamic add of a system device.

Future work will likely include infrastructure and support for the following events:

- `METADATA_CHANGE`: An instance's metadata has changed.
- `USER_REQUEST`: Directed request to update.

Datasource event support

All *datasources* support the `BOOT_NEW_INSTANCE` event by default. Each datasource will declare a set of these events that it is capable of handling. Datasources may not support all event types. In some cases a system may be configured to allow a particular event but may be running on a platform whose datasource cannot support the event.

Configuring event updates

Update configuration may be specified via user data, which can be used to enable or disable handling of specific events. This configuration will be honored as long as the events are supported by the datasource. However, configuration will always be applied at first boot, regardless of the user data specified.

Updates

Update policy configuration defines which events are allowed to be handled. This is separate from whether a particular platform or datasource has the capability for such events.

`scope:` <*name of the* `scope` *for event policy*>
: The `scope` value is a string which defines which domain the event occurs under. Currently, the only known `scope` is `network`, though more `scopes` may be added in the future. `Scopes` are defined by convention but arbitrary values can be used.

`when:` <*list of events to handle for a particular* `scope` >
: Each `scope` requires a `when` element to specify which events are to allowed to be handled.

Hotplug

When the `hotplug` event is supported by the datasource and configured in user data, `cloud-init` will respond to the addition or removal of network interfaces to the system. In addition to fetching and updating the system metadata, `cloud-init` will also bring up/down the newly added interface.

> **Warning:** Due to its use of `systemd` sockets, `hotplug` functionality is currently incompatible with SELinux on Linux distributions using systemd. This issue is being tracked in GitHub #3890. Additionally, `hotplug` support is considered experimental for non-Alpine and non-Debian-based systems.

Example

Apply network config every boot

On every boot, apply network configuration found in the datasource.

```
# apply network config on every boot
updates:
  network:
    when: ['boot']
```

2.3.6 Instance metadata

Kernel command line

Providing configuration data via the kernel command line is somewhat of a last resort, since this method only supports *cloud config* starting with *#cloud-config*, and many datasources do not support injecting kernel command line arguments without modifying the bootloader.

Despite the limitations of using the kernel command line, cloud-init supports some use-cases.

Note that this page describes kernel command line behavior that applies to all clouds. To provide a local configuration with an image using kernel command line, see *datasource NoCloud* which provides more configuration options.

Datasource discovery override

During boot, cloud-init must identify which datasource it is running on (OpenStack, AWS, Azure, GCP, etc). This discovery step can be optionally overridden by specifying the datasource name, such as:

```
root=/dev/sda ro ds=openstack
```

Kernel cloud-config-url configuration

In order to allow an ephemeral, or otherwise pristine image to receive some configuration, `cloud-init` can read a URL directed by the kernel command line and proceed as if its data had previously existed.

This allows for configuring a metadata service, or some other data.

When *the local stage* runs, it will check to see if `cloud-config-url` appears in key/value fashion in the kernel command line, such as:

```
root=/dev/sda ro cloud-config-url=http://foo.bar.zee/abcde
```

`Cloud-init` will then read the contents of the given URL. If the content starts with `#cloud-config`, it will store that data to the local filesystem in a static filename `/etc/cloud/cloud.cfg.d/91_kernel_cmdline_url.cfg`, and consider it as part of the config from that point forward.

Note: If `/etc/cloud/cloud.cfg.d/91_kernel_cmdline_url.cfg` already exists, cloud-init will not overwrite the file, and the `cloud-config-url` parameter is completely ignored.

This is useful, for example, to be able to configure the MAAS datasource by controlling the kernel command line from outside the image, you can append:

```
cloud-config-url=http://your.url.here/abcdefg
```

Then, have the following content at that url:

```
#cloud-config
datasource:
  MAAS:
    metadata_url: http://mass-host.localdomain/source
    consumer_key: Xh234sdkljf
    token_key: kjfhgb3n
    token_secret: 24uysdfx1w4
```

Warning: `url` kernel command line key is deprecated. Please use `cloud-config-url` parameter instead.

Note: Since `cloud-config-url=` is so generic, in order to avoid false positives, only *cloud config* user data starting with `#cloud-config` is supported.

Note: The `cloud-config-url=` is unencrypted http GET, and may contain credentials. Care must be taken to ensure this data is only transferred via trusted channels (i.e., within a closed system).

What is `instance-data`?

Each cloud provider presents unique configuration metadata to a launched cloud instance. `Cloud-init` crawls this metadata and then caches and exposes this information as a standardised and versioned JSON object known as `instance-data`. This `instance-data` may then be queried or later used by `cloud-init` in templated configuration and scripts.

An example of a small subset of instance-data on a launched EC2 instance:

```
{
  "v1": {
    "cloud_name": "aws",
    "distro": "ubuntu",
    "distro_release": "jammy",
    "distro_version": "22.04",
    "instance_id": "i-06b5687b4d7b8595d",
    "machine": "x86_64",
    "platform": "ec2",
    "python_version": "3.10.4",
    "region": "us-east-2",
    "variant": "ubuntu"
  }
}
```

Discovery

One way to easily explore which `instance-data` variables are available on your machine is to use the *cloud-init query* tool. Warnings or exceptions will be raised on invalid `instance-data` keys, paths or invalid syntax.

The **query** command also publishes `userdata` and `vendordata` keys to the root user which will contain the decoded user and vendor data provided to this instance. Non-root users referencing `userdata` or `vendordata` keys will see only redacted values.

Note: To save time designing a user data template for a specific cloud's `instance-data.json`, use the **render** command on an instance booted on your favorite cloud. See *devel* for more information.

Using `instance-data`

`instance-data` can be used in:

- *User data scripts*.
- *Cloud-config data*.
- *Base configuration*.
- Command line interface via **`cloud-init query`** or **`cloud-init devel render`**.

The aforementioned configuration sources support jinja template rendering. When the first line of the provided configuration begins with **## template: jinja**, `cloud-init` will use jinja to render that file. Any `instance-data` variables are surfaced as jinja template variables.

Note: Trying to reference jinja variables that don't exist in `instance-data` will result in warnings in `/var/log/cloud-init.log` and the following string in your rendered `user-data`: `CI_MISSING_JINJA_VAR/<your_varname>`.

Sensitive data such as user passwords may be contained in `instance-data`. `Cloud-init` separates this sensitive data such that is it only readable by root. In the case that a non-root user attempts to read sensitive `instance-data`, they will receive redacted data or the same warnings and text that occur if a variable does not exist.

Example: Cloud config with `instance-data`

```
## template: jinja
#cloud-config
runcmd:
    - echo 'EC2 public hostname allocated to instance: {{
      ds.meta_data.public_hostname }}' > /tmp/instance_metadata
    - echo 'EC2 availability zone: {{ v1.availability_zone }}' >>
      /tmp/instance_metadata
    - curl -X POST -d '{"hostname": "{{ds.meta_data.public_hostname }}",
      "availability-zone": "{{ v1.availability_zone }}"}'
      https://example.com
```

Example: User data script with `instance-data`

```
## template: jinja
#!/bin/bash
{% if v1.region == 'us-east-2' -%}
echo 'Installing custom proxies for {{ v1.region }}'
sudo apt-get install my-xtra-fast-stack
{%- endif %}
...
```

Example: CLI discovery of `instance-data`

```
# List all instance-data keys and values as root user
$ sudo cloud-init query --all
{...}

# List all top-level instance-data keys available
$ cloud-init query --list-keys

# Introspect nested keys on an object
$ cloud-init query -f "{{ds.keys()}}"
dict_keys(['meta_data', '_doc'])

# Failure to reference valid dot-delimited key path on a known top-level key
$ cloud-init query v1.not_here
ERROR: instance-data 'v1' has no 'not_here'
```

(continues on next page)

(continued from previous page)

```
# Test expected value using valid instance-data key path
$ cloud-init query -f "My AMI: {{ds.meta_data.ami_id}}"
My AMI: ami-0fecc35d3c8ba8d60

# The --format command renders jinja templates, this can also be used
# to develop and test jinja template constructs
$ cat > test-templating.yaml <<EOF
  {% for val in ds.meta_data.keys() %}
  - {{ val }}
  {% endfor %}
  EOF
$ cloud-init query --format="$( cat test-templating.yaml )"
- instance_id
- dsmode
- local_hostname
```

Reference

Storage locations

- `/run/cloud-init/instance-data.json`: world-readable JSON containing standardised keys, sensitive keys redacted.
- `/run/cloud-init/instance-data-sensitive.json`: root-readable unredacted JSON blob.
- `/run/cloud-init/combined-cloud-config.json`: root-readable unredacted JSON blob. Any meta-data, vendor-data and user-data overrides are applied to the `/run/cloud-init/combined-cloud-config.json` config values.

`instance-data.json` top level keys

`base64_encoded_keys`

A list of forward-slash delimited key paths into the `instance-data.json` object whose value is base64encoded for JSON compatibility. Values at these paths should be decoded to get the original value.

`features`

A dictionary of feature name and boolean value pairs. A value of `True` means the feature is enabled.

sensitive_keys

A list of forward-slash delimited key paths into the `instance-data.json` object whose value is considered by the datasource as 'security sensitive'. Only the keys listed here will be redacted from `instance-data.json` for non-root users.

merged_cfg

Deprecated use `merged_system_cfg` instead.

merged_system_cfg

Merged `cloud-init` *Base configuration* from `/etc/cloud/cloud.cfg` and `/etc/cloud/cloud-cfg.d`. Values under this key could contain sensitive information such as passwords, so it is included in the `sensitive-keys` list which is only readable by root.

Note: `merged_system_cfg` represents only the merged config from the underlying filesystem. These values can be overridden by meta-data, vendor-data or user-data. The fully merged cloud-config provided to a machine which accounts for any supplemental overrides is the file `/run/cloud-init/combined-cloud-config.json`.

---

ds

Datasource-specific metadata crawled for the specific cloud platform. It should closely represent the structure of the cloud metadata crawled. The structure of content and details provided are entirely cloud-dependent. Mileage will vary depending on what the cloud exposes. The content exposed under the `ds` key is currently **experimental** and expected to change slightly in the upcoming `cloud-init` release.

sys_info

Information about the underlying OS, Python, architecture and kernel. This represents the data collected by `cloudinit.util.system_info`.

system_info

This is a cloud-init configuration key present in `/etc/cloud/cloud.cfg` which describes cloud-init's configured *default_user*, *distro*, *network* renderers, and *paths* that cloud-init will use. Not to be confused with the underlying host `sys_info` key above.

v1

Standardised `cloud-init` metadata keys, these keys are guaranteed to exist on all cloud platforms. They will also retain their current behaviour and format, and will be carried forward even if `cloud-init` introduces a new version of standardised keys with `v2`.

To cut down on keystrokes on the command line, `cloud-init` also provides top-level key aliases for any standardised `v#` keys present. The preceding `v1` is not required of `v1.var_name` These aliases will represent the value of the highest versioned standard key. For example, `cloud_name` value will be `v2.cloud_name` if both `v1` and `v2` keys are present in `instance-data.json`.

`Cloud-init` also provides jinja-safe key aliases for any `instance-data` keys which contain jinja operator characters such as `+`, `-`, `.`, `/`, etc. Any jinja operator will be replaced with underscores in the jinja-safe key alias. This allows for `cloud-init` templates to use aliased variable references which allow for jinja's dot-notation reference such as `{{ ds.v1_0.my_safe_key }}` instead of `{{ ds["v1.0"]["my/safe-key"] }}`.

Standardised `instance-data.json` v1 keys

v1._beta_keys

List of standardised keys still in 'beta'. The format, intent or presence of these keys can change. Do not consider them production-ready.

Example output:

- [subplatform]

v1.cloud_name

Where possible this will indicate the 'name' of the cloud the system is running on. This is different than the 'platform' item. For example, the cloud name of Amazon Web Services is 'aws', while the platform is 'ec2'.

If determining a specific name is not possible or provided in `meta-data`, then this filed may contain the same content as 'platform'.

Example output:

- aws
- openstack
- azure
- configdrive
- nocloud
- ovf

v1.distro, v1.distro_version, v1.distro_release

This shall be the distro name, version and release as determined by `cloudinit.util.get_linux_distro`.

Example output:

- alpine, 3.12.0, 'Alpine Linux v3.12'
- centos, 7.5, core
- debian, 9, stretch
- freebsd, 12.0-release-p10,
- opensuse, 42.3, x86_64
- opensuse-tumbleweed, 20180920, x86_64
- redhat, 7.5, 'maipo'
- sles, 12.3, x86_64
- ubuntu, 20.04, focal

v1.instance_id

Unique `instance_id` allocated by the cloud.

Example output:

- i-<hash>

v1.kernel_release

This shall be the running kernel `uname -r`.

Example output:

- 5.3.0-1010-aws

v1.local_hostname

The internal or local hostname of the system.

Example output:

- `ip-10-41-41-70`
- `<user-provided-hostname>`

v1.machine

This shall be the running cpu machine architecture `uname -m`.

Example output:

- x86_64
- i686
- ppc64le
- s390x

v1.platform

An attempt to identify the cloud platform instance that the system is running on.

Example output:

- ec2
- openstack
- lxd
- gce
- nocloud
- ovf

v1.subplatform

Additional platform details describing the specific source or type of metadata used. The format of subplatform will be: `<subplatform_type> (<url_file_or_dev_path>)`

Example output:

- metadata (http://169.254.169.254)
- seed-dir (/path/to/seed-dir/)
- config-disk (/dev/cd0)
- configdrive (/dev/sr0)

v1.public_ssh_keys

A list of SSH keys provided to the instance by the datasource metadata.

Example output:

- ['ssh-rsa AA...', ...]

v1.python_version

The version of Python that is running `cloud-init` as determined by `cloudinit.util.system_info`.

Example output:

- 3.7.6

v1.region

The physical region/data centre in which the instance is deployed.

Example output:

- us-east-2

v1.availability_zone

The physical availability zone in which the instance is deployed.

Example output:

- us-east-2b
- nova
- null

Example Output

Below is an example of `/run/cloud-init/instance-data-sensitive.json` on an EC2 instance:


```
{
 "_beta_keys": [
  "subplatform"
 ],
 "availability_zone": "us-east-1b",
 "base64_encoded_keys": [],
 "merged_cfg": {
  "_doc": "Merged cloud-init base config from /etc/cloud/cloud.cfg and /etc/cloud/cloud.
↪cfg.d/",
  "_log": [
   "[loggers]\nkeys=root,cloudinit\n\n[handlers]\nkeys=consoleHandler,cloudLogHandler\n\
↪n[formatters]\nkeys=simpleFormatter,arg0Formatter\n\n[logger_root]\nlevel=DEBUG\
↪nhandlers=consoleHandler,cloudLogHandler\n\n[logger_cloudinit]\nlevel=DEBUG\
↪nqualname=cloudinit\nhandlers=\npropagate=1\n\n[handler_consoleHandler]\
↪nclass=StreamHandler\nlevel=WARNING\nformatter=arg0Formatter\nargs=(sys.stderr,)\n\
↪n[formatter_arg0Formatter]\nformat=%(asctime)s - %(filename)s[%(levelname)s]:
↪%(message)s\n\n[formatter_simpleFormatter]\nformat=[CLOUDINIT] %(filename)s[
↪%(levelname)s]: %(message)s\n",
   "[handler_cloudLogHandler]\nclass=FileHandler\nlevel=DEBUG\nformatter=arg0Formatter\
↪nargs=('/var/log/cloud-init.log',)\n",
   "[handler_cloudLogHandler]\nclass=handlers.SysLogHandler\nlevel=DEBUG\
↪nformatter=simpleFormatter\nargs=(\"/dev/log\", handlers.SysLogHandler.LOG_USER)\n"
```


(continues on next page)

(continued from previous page)

```
],
"cloud_config_modules": [
 "snap",
 "ssh_import_id",
 "locale",
 "set_passwords",
 "grub_dpkg",
 "apt_pipelining",
 "apt_configure",
 "ubuntu_pro",
 "ntp",
 "timezone",
 "disable_ec2_metadata",
 "runcmd",
 "byobu"
],
"cloud_final_modules": [
 "package_update_upgrade_install",
 "fan",
 "landscape",
 "lxd",
 "ubuntu_drivers",
 "puppet",
 "chef",
 "mcollective",
 "salt_minion",
 "scripts_vendor",
 "scripts_per_once",
 "scripts_per_boot",
 "scripts_per_instance",
 "scripts_user",
 "ssh_authkey_fingerprints",
 "keys_to_console",
 "phone_home",
 "final_message",
 "power_state_change"
],
"cloud_init_modules": [
 "seed_random",
 "bootcmd",
 "write_files",
 "growpart",
 "resizefs",
 "disk_setup",
 "mounts",
 "set_hostname",
 "update_hostname",
 "update_etc_hosts",
 "ca_certs",
 "rsyslog",
 "users_groups",
 "ssh"
```

(continues on next page)

(continued from previous page)


```
  ],
  "datasource_list": [
   "Ec2",
   "None"
  ],
  "def_log_file": "/var/log/cloud-init.log",
  "disable_root": true,
  "log_cfgs": [
   [
    "[loggers]\nkeys=root,cloudinit\n\n[handlers]\nkeys=consoleHandler,cloudLogHandler\n\
↪n[formatters]\nkeys=simpleFormatter,arg0Formatter\n\n[logger_root]\nlevel=DEBUG\
↪nhandlers=consoleHandler,cloudLogHandler\n\n[logger_cloudinit]\nlevel=DEBUG\
↪nqualname=cloudinit\nhandlers=\npropagate=1\n\n[handler_consoleHandler]\
↪nclass=StreamHandler\nlevel=WARNING\nformatter=arg0Formatter\nargs=(sys.stderr,)\n\
↪n[formatter_arg0Formatter]\nformat=%(asctime)s - %(filename)s[%(levelname)s]:
↪%(message)s\n\n[formatter_simpleFormatter]\nformat=[CLOUDINIT] %(filename)s[
↪%(levelname)s]: %(message)s\n",
    "[handler_cloudLogHandler]\nclass=FileHandler\nlevel=DEBUG\nformatter=arg0Formatter\
↪nargs=('/var/log/cloud-init.log',)\n"
   ]
  ],
  "output": {
   "all": "| tee -a /var/log/cloud-init-output.log"
  },
  "preserve_hostname": false,
  "syslog_fix_perms": [
   "syslog:adm",
   "root:adm",
   "root:wheel",
   "root:root"
  ],
  "users": [
   "default"
  ],
  "vendor_data": {
   "enabled": true,
   "prefix": []
  }
 },
 "cloud_name": "aws",
 "distro": "ubuntu",
 "distro_release": "focal",
 "distro_version": "20.04",
 "ds": {
  "_doc": "EXPERIMENTAL: The structure and format of content scoped under the 'ds' key␣
↪may change in subsequent releases of cloud-init.",
  "_metadata_api_version": "2016-09-02",
  "dynamic": {
   "instance_identity": {
    "document": {
     "accountId": "329910648901",
     "architecture": "x86_64",
```


(continues on next page)

(continued from previous page)


```
    "availabilityZone": "us-east-1b",
    "billingProducts": null,
    "devpayProductCodes": null,
    "imageId": "ami-02e8aa396f8be3b6d",
    "instanceId": "i-0929128ff2f73a2f1",
    "instanceType": "t2.micro",
    "kernelId": null,
    "marketplaceProductCodes": null,
    "pendingTime": "2020-02-27T20:46:18Z",
    "privateIp": "172.31.81.43",
    "ramdiskId": null,
    "region": "us-east-1",
    "version": "2017-09-30"
   },
   "pkcs7": [
    "MIAGCSqGSIb3DQ...",
    "REDACTED",
    "AhQUgq0iPWqPTVnT96tZE6L1XjjLHQAAAAAAAA=="
   ],
   "rsa2048": [
    "MIAGCSqGSIb...",
    "REDACTED",
    "clYQvuE45xXm7Yreg3QtQbrP//owl1eZHj6s350AAAAAAAA="
   ],
   "signature": [
    "dA+QV+LLCWCRNddnrKleYmh2GvYo+t8urDkdgmDSsPi",
    "REDACTED",
    "kDT4ygyJLFkd3b4qjAs="
   ]
  }
 },
 "meta_data": {
  "ami_id": "ami-02e8aa396f8be3b6d",
  "ami_launch_index": "0",
  "ami_manifest_path": "(unknown)",
  "block_device_mapping": {
   "ami": "/dev/sda1",
   "root": "/dev/sda1"
  },
  "hostname": "ip-172-31-81-43.ec2.internal",
  "instance_action": "none",
  "instance_id": "i-0929128ff2f73a2f1",
  "instance_type": "t2.micro",
  "local_hostname": "ip-172-31-81-43.ec2.internal",
  "local_ipv4": "172.31.81.43",
  "mac": "12:7e:c9:93:29:af",
  "metrics": {
   "vhostmd": "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
  },
  "network": {
   "interfaces": {
    "macs": {
```


(continues on next page)

(continued from previous page)


```
      "12:7e:c9:93:29:af": {
       "device_number": "0",
       "interface_id": "eni-0c07a0474339b801d",
       "ipv4_associations": {
        "3.89.187.177": "172.31.81.43"
       },
       "local_hostname": "ip-172-31-81-43.ec2.internal",
       "local_ipv4s": "172.31.81.43",
       "mac": "12:7e:c9:93:29:af",
       "owner_id": "329910648901",
       "public_hostname": "ec2-3-89-187-177.compute-1.amazonaws.com",
       "public_ipv4s": "3.89.187.177",
       "security_group_ids": "sg-0100038b68aa79986",
       "security_groups": "launch-wizard-3",
       "subnet_id": "subnet-04e2d12a",
       "subnet_ipv4_cidr_block": "172.31.80.0/20",
       "vpc_id": "vpc-210b4b5b",
       "vpc_ipv4_cidr_block": "172.31.0.0/16",
       "vpc_ipv4_cidr_blocks": "172.31.0.0/16"
      }
     }
    }
   },
   "placement": {
    "availability_zone": "us-east-1b"
   },
   "profile": "default-hvm",
   "public_hostname": "ec2-3-89-187-177.compute-1.amazonaws.com",
   "public_ipv4": "3.89.187.177",
   "reservation_id": "r-0c481643d15766a02",
   "security_groups": "launch-wizard-3",
   "services": {
    "domain": "amazonaws.com",
    "partition": "aws"
   }
  }
 },
 "instance_id": "i-0929128ff2f73a2f1",
 "kernel_release": "5.3.0-1010-aws",
 "local_hostname": "ip-172-31-81-43",
 "machine": "x86_64",
 "platform": "ec2",
 "public_ssh_keys": [],
 "python_version": "3.7.6",
 "region": "us-east-1",
 "sensitive_keys": [],
 "subplatform": "metadata (http://169.254.169.254)",
 "sys_info": {
  "dist": [
   "ubuntu",
   "20.04",
   "focal"
```


(continues on next page)

(continued from previous page)


```
  ],
  "platform": "Linux-5.3.0-1010-aws-x86_64-with-Ubuntu-20.04-focal",
  "python": "3.7.6",
  "release": "5.3.0-1010-aws",
  "system": "Linux",
  "uname": [
   "Linux",
   "ip-172-31-81-43",
   "5.3.0-1010-aws",
   "#11-Ubuntu SMP Thu Jan 16 07:59:32 UTC 2020",
   "x86_64",
   "x86_64"
  ],
  "variant": "ubuntu"
 },
 "system_platform": "Linux-5.3.0-1010-aws-x86_64-with-Ubuntu-20.04-focal",
 "userdata": "#cloud-config\nssh_import_id: [<my-launchpad-id>]\n...",
 "v1": {
  "_beta_keys": [
   "subplatform"
  ],
  "availability_zone": "us-east-1b",
  "cloud_name": "aws",
  "distro": "ubuntu",
  "distro_release": "focal",
  "distro_version": "20.04",
  "instance_id": "i-0929128ff2f73a2f1",
  "kernel": "5.3.0-1010-aws",
  "local_hostname": "ip-172-31-81-43",
  "machine": "x86_64",
  "platform": "ec2",
  "public_ssh_keys": [],
  "python": "3.7.6",
  "region": "us-east-1",
  "subplatform": "metadata (http://169.254.169.254)",
  "system_platform": "Linux-5.3.0-1010-aws-x86_64-with-Ubuntu-20.04-focal",
  "variant": "ubuntu"
 },
 "variant": "ubuntu",
 "vendordata": ""
}
```

2.3.7 Vendor data

Overview

Vendor data is data provided by the entity that launches an instance (e.g., the cloud provider). This data can be used to customise the image to fit into the particular environment it is being run in.

Vendor data follows the same rules as user data, with the following caveats:

1. Users have ultimate control over vendor data. They can disable its execution or disable handling of specific parts of multi-part input.
2. By default it only runs on first boot.
3. Vendor data can be disabled by the user. If the use of vendor data is required for the instance to run, then vendor data should not be used.
4. User-supplied cloud-config is merged over cloud-config from vendor data.

Users providing cloud-config data can use the `#cloud-config-jsonp` method to more finely control their modifications to the vendor-supplied cloud-config. For example, if both vendor and user have provided `runcmd` then the default merge handler will cause the user's `runcmd` to override the one provided by the vendor. To append to `runcmd`, the user could better provide multi-part input with a `cloud-config-jsonp` part like:

```
#cloud-config-jsonp
[{ "op": "add", "path": "/runcmd", "value": ["my", "command", "here"]}]
```

Further, we strongly advise vendors to not "be evil". By evil, we mean any action that could compromise a system. Since users trust you, please take care to make sure that any vendor data is safe, atomic, idempotent and does not put your users at risk.

Input formats

`Cloud-init` will download and cache to filesystem any vendor data that it finds. Vendor data is handled exactly like user data. This means that the vendor can supply multi-part input and have those parts acted on in the same way as with user data.

The only differences are:

- Vendor-data-defined scripts are stored in a different location than user-data-defined scripts (to avoid namespace collision).
- The user can disable part handlers via the cloud-config settings. For example, to disable handling of 'part-handlers' in vendor data, the user could provide user data like this:

```
#cloud-config
vendordata: {excluded: 'text/part-handler'}
```

Examples

You can find examples in the examples subdirectory.

Additionally, the `tools` directory contains `write-mime-multipart`, which can be used to easily generate MIME multi-part files from a list of input files. That data can then be given to an instance.

See **`write-mime-multipart --help`** for usage.

2.3.8 Security

Security policy

The following documents the upstream cloud-init security policy.

Reporting

If a security bug is found, please send an email to cloud-init-security@lists.canonical.com . After the bug is received, the issue is triaged within 2 working days of being reported and a response is sent to the reporter.

cloud-init-security

The cloud-init-security Launchpad team is a private, invite-only team used to discuss and coordinate security issues with the project.

Any issues disclosed to the cloud-init-security mailing list are considered embargoed and should only be discussed with other members of the cloud-init-security mailing list before the coordinated release date, unless specific exception is granted by the administrators of the mailing list. This includes disclosure of any details related to the vulnerability or the presence of a vulnerability itself. Violation of this policy may result in removal from the list for the company or individual involved.

Evaluation

If the reported bug is deemed a real security issue a CVE is assigned by the Canonical Security Team as CVE Numbering Authority (CNA).

If it is deemed a regular, non-security issue, the reporter will be asked to follow typical bug reporting procedures.

In addition to the disclosure timeline, the core Canonical cloud-init team will enlist the expertise of the Ubuntu Security team for guidance on industry-standard disclosure practices as necessary.

If an issue specifically involves another distro or cloud vendor, additional individuals will be informed of the issue to help in evaluation.

Disclosure

Disclosure of security issues will be made with a public statement. Once the determined time for disclosure has arrived the following will occur:

- A public bug is filed/made public with vulnerability details, CVE, mitigations and where to obtain the fix
- An email is sent to the public cloud-init mailing list

The disclosure timeframe is coordinated with the reporter and members of the cloud-init-security list. This depends on a number of factors:

- The reporter might have their own disclosure timeline (e.g. Google Project Zero and many others use a 90-days after initial report OR when a fix becomes public)
- It might take time to decide upon and develop an appropriate fix
- A distros might want extra time to backport any possible fixes before the fix becomes public
- A cloud may need additional time to prepare to help customers or implement a fix
- The issue might be deemed low priority
- May wish to align with an upcoming planned release

2.3.9 Performance

The **`analyze`** subcommand was added to `cloud-init` to help analyze `cloud-init` boot time performance. It is loosely based on `systemd-analyze`, where there are four subcommands:

- **`blame`**
- **`show`**
- **`dump`**
- **`boot`**

Usage

The **`analyze`** command requires one of the four subcommands:

```
$ cloud-init analyze blame
$ cloud-init analyze show
$ cloud-init analyze dump
$ cloud-init analyze boot
```

Availability

The **`analyze boot`** subcommand only works on operating systems that use *systemd*.

Subcommands

Blame

The **blame** subcommand matches **systemd-analyze blame** where it prints, in descending order, the units that took the longest time to run. This output is highly useful for examining where `cloud-init` is spending its time.

```
$ cloud-init analyze blame
```

Example output:

```
-- Boot Record 01 --
    00.80300s (init-network/config-growpart)
    00.64300s (init-network/config-resizefs)
    00.62100s (init-network/config-ssh)
    00.57300s (modules-config/config-grub_dpkg)
    00.40300s (init-local/search-NoCloud)
    00.38200s (init-network/config-users_groups)
    00.19800s (modules-config/config-apt_configure)
    00.03700s (modules-final/config-keys_to_console)
    00.02100s (init-network/config-update_etc_hosts)
    00.02100s (init-network/check-cache)
    00.00800s (modules-final/config-ssh_authkey_fingerprints)
    00.00800s (init-network/consume-vendor-data)
    00.00600s (modules-config/config-timezone)
    00.00500s (modules-final/config-final_message)
    00.00400s (init-network/consume-user-data)
    00.00400s (init-network/config-mounts)
    00.00400s (init-network/config-disk_setup)
    00.00400s (init-network/config-bootcmd)
    00.00400s (init-network/activate-datasource)
    00.00300s (init-network/config-update_hostname)
    00.00300s (init-network/config-set_hostname)
    00.00200s (modules-final/config-snappy)
    00.00200s (init-network/config-rsyslog)
    00.00200s (init-network/config-ca_certs)
    00.00200s (init-local/check-cache)
    00.00100s (modules-final/config-scripts_vendor)
    00.00100s (modules-final/config-scripts_per_once)
    00.00100s (modules-final/config-salt_minion)
    00.00100s (modules-final/config-phone_home)
    00.00100s (modules-final/config-package_update_upgrade_install)
    00.00100s (modules-final/config-fan)
    00.00100s (modules-config/config-ubuntu_pro)
    00.00100s (modules-config/config-ssh_import_id)
    00.00100s (modules-config/config-snap)
    00.00100s (modules-config/config-set_passwords)
    00.00100s (modules-config/config-runcmd)
    00.00100s (modules-config/config-locale)
    00.00100s (modules-config/config-byobu)
    00.00100s (modules-config/config-apt_pipelining)
    00.00100s (init-network/config-write_files)
    00.00100s (init-network/config-seed_random)
```

(continues on next page)

(continued from previous page)

```
    00.00000s (modules-final/config-ubuntu_drivers)
    00.00000s (modules-final/config-scripts_user)
    00.00000s (modules-final/config-scripts_per_instance)
    00.00000s (modules-final/config-scripts_per_boot)
    00.00000s (modules-final/config-puppet)
    00.00000s (modules-final/config-power_state_change)
    00.00000s (modules-final/config-mcollective)
    00.00000s (modules-final/config-lxd)
    00.00000s (modules-final/config-landscape)
    00.00000s (modules-final/config-chef)
    00.00000s (modules-config/config-snap_config)
    00.00000s (modules-config/config-ntp)
    00.00000s (modules-config/config-disable_ec2_metadata)
    00.00000s (init-network/setup-datasource)

1 boot records analyzed
```

Show

The **show** subcommand is similar to **systemd-analyze critical-chain** which prints a list of units, the time they started and how long they took. `Cloud-init` has five *boot stages*, and within each stage a number of modules may run depending on configuration. **cloudinit-analyze show** will, for each boot, print this information and a summary of the total time.

The following is an abbreviated example of the **show** subcommand:

```
$ cloud-init analyze show
```

Example output:

```
-- Boot Record 01 --
The total time elapsed since completing an event is printed after the "@" character.
The time the event takes is printed after the "+" character.

Starting stage: init-local
|``->no cache found @00.01700s +00.00200s
|`->found local data from DataSourceNoCloud @00.11000s +00.40300s
Finished stage: (init-local) 00.94200 seconds

Starting stage: init-network
|`->restored from cache with run check: DataSourceNoCloud [seed=/dev/sr0][dsmode=net]␣
↪@04.79500s +00.02100s
|`->setting up datasource @04.88900s +00.00000s
|`->reading and applying user-data @04.90100s +00.00400s
|`->reading and applying vendor-data @04.90500s +00.00800s
|`->activating datasource @04.95200s +00.00400s
Finished stage: (init-network) 02.72100 seconds

Starting stage: modules-config
|`->config-snap ran successfully @15.43100s +00.00100s
...
```

(continues on next page)

(continued from previous page)

```
|`->config-runcmd ran successfully @16.22300s +00.00100s
|`->config-byobu ran successfully @16.23400s +00.00100s
Finished stage: (modules-config) 00.83500 seconds

Starting stage: modules-final
|`->config-snappy ran successfully @16.87400s +00.00200s
|`->config-package_update_upgrade_install ran successfully @16.87600s +00.00100s
...
|`->config-final_message ran successfully @16.93700s +00.00500s
|`->config-power_state_change ran successfully @16.94300s +00.00000s
Finished stage: (modules-final) 00.10300 seconds

Total Time: 4.60100 seconds

1 boot records analyzed
```

If additional boot records are detected then they are printed out from oldest to newest.

Dump

The **dump** subcommand simply dumps the `cloud-init` logs that the **analyze** module is performing its analysis on, and returns a list of dictionaries that can be consumed for other reporting needs. Each element in the list is a boot entry.

```
$ cloud-init analyze dump
```

Example output:

```
[
{
  "description": "starting search for local datasources",
  "event_type": "start",
  "name": "init-local",
  "origin": "cloudinit",
  "timestamp": 1567057578.037
},
{
  "description": "attempting to read from cache [check]",
  "event_type": "start",
  "name": "init-local/check-cache",
  "origin": "cloudinit",
  "timestamp": 1567057578.054
},
{
  "description": "no cache found",
  "event_type": "finish",
  "name": "init-local/check-cache",
  "origin": "cloudinit",
  "result": "SUCCESS",
  "timestamp": 1567057578.056
},
{
```

(continues on next page)

(continued from previous page)


```
  "description": "searching for local data from DataSourceNoCloud",
  "event_type": "start",
  "name": "init-local/search-NoCloud",
  "origin": "cloudinit",
  "timestamp": 1567057578.147
},
{
  "description": "found local data from DataSourceNoCloud",
  "event_type": "finish",
  "name": "init-local/search-NoCloud",
  "origin": "cloudinit",
  "result": "SUCCESS",
  "timestamp": 1567057578.55
},
{
  "description": "searching for local datasources",
  "event_type": "finish",
  "name": "init-local",
  "origin": "cloudinit",
  "result": "SUCCESS",
  "timestamp": 1567057578.979
},
{
  "description": "searching for network datasources",
  "event_type": "start",
  "name": "init-network",
  "origin": "cloudinit",
  "timestamp": 1567057582.814
},
{
  "description": "attempting to read from cache [trust]",
  "event_type": "start",
  "name": "init-network/check-cache",
  "origin": "cloudinit",
  "timestamp": 1567057582.832
},
...
{
  "description": "config-power_state_change ran successfully",
  "event_type": "finish",
  "name": "modules-final/config-power_state_change",
  "origin": "cloudinit",
  "result": "SUCCESS",
  "timestamp": 1567057594.98
},
{
  "description": "running modules for final",
  "event_type": "finish",
  "name": "modules-final",
  "origin": "cloudinit",
  "result": "SUCCESS",
  "timestamp": 1567057594.982
```


(continues on next page)

(continued from previous page)

```
}
]
```

Boot

The **boot** subcommand prints out kernel-related timestamps that are not included in any of the `cloud-init` logs. There are three different timestamps that are presented to the user:

- `kernel start`
- `kernel finish boot`
- `cloud-init start`

This was added for additional clarity into the boot process that `cloud-init` does not have control over, to aid in debugging performance issues related to `cloud-init` startup, and tracking regression.

```
$ cloud-init analyze boot
```

Example output:

```
-- Most Recent Boot Record --
    Kernel Started at: 2019-08-29 01:35:37.753790
    Kernel ended boot at: 2019-08-29 01:35:38.807407
    Kernel time to boot (seconds): 1.053617000579834
    Cloud-init activated by systemd at: 2019-08-29 01:35:43.992460
    Time between Kernel end boot and Cloud-init activation (seconds): 5.185053110122681
    Cloud-init start: 2019-08-29 08:35:45.867000
successful
```

Timestamp gathering

The following boot-related timestamps are gathered on demand when **`cloud-init analyze boot`** runs:

- Kernel startup gathered from system uptime
- Kernel finishes initialization from systemd `UserSpaceMonotonicTimestamp` property
- `Cloud-init` activation from the property `InactiveExitTimestamp` of the `cloud-init` local systemd unit

In order to gather the necessary timestamps using systemd, running the following command will gather the `UserspaceTimestamp`:

```
$ systemctl show -p UserspaceTimestampMonotonic
```

Example output:

```
UserspaceTimestampMonotonic=989279
```

The `UserspaceTimestamp` tracks when the init system starts, which is used as an indicator of the kernel finishing initialisation.

Running the following command will gather the `InactiveExitTimestamp`:

```
$ systemctl show cloud-init-local -p InactiveExitTimestampMonotonic
```

Example output:

```
InactiveExitTimestampMonotonic=4493126
```

The `InactiveExitTimestamp` tracks when a particular systemd unit transitions from the *Inactive* to *Active* state, which can be used to mark the beginning of systemd's activation of `cloud-init`.

Currently this only works for distros that use systemd as the init process. We will be expanding support for other distros in the future and this document will be updated accordingly.

If systemd is not present on the system, `dmesg` is used to attempt to find an event that logs the beginning of the init system. However, with this method only the first two timestamps are able to be found; `dmesg` does not monitor userspace processes, so no `cloud-init` start timestamps are emitted – unlike when using systemd.

2.3.10 Failure states

Cloud-init has multiple modes of failure. This page describes these modes and how to gather information about failures.

Critical failure

Critical failures happens when cloud-init experiences a condition that it cannot safely handle. When this happens, cloud-init may be unable to complete, and the instance is likely to be in an unknown broken state.

Cloud-init experiences critical failure when:

- there is a major problem with the cloud image that is running cloud-init
- there is a severe bug in cloud-init

When this happens, error messages will be visible in output of `cloud-init status --long` within the `'error'`.

The same errors will also be located under the key nested under the module-level keys that store information related to each *stage of cloud-init*: `init-local`, `init`, `modules-config`, `modules-final`.

Recoverable failure

In the case that cloud-init is able to complete yet something went wrong, cloud-init has experienced a "recoverable failure". When this happens, the service will return with exit code 2, and error messages will be visible in the output of `cloud-init status --long` under the top level `recoverable_errors` and `error` keys.

To identify which stage an error came from, one can check under the module-level keys: `init-local`, `init`, `modules-config`, `modules-final` for the same error keys.

See *this more detailed explanation* for to learn how to use cloud-init's exported errors.

Cloud-init error codes

Cloud-init's `status` subcommand is useful for understanding which type of error cloud-init experienced while running. The return code will be one of the following:

```
0 - success
1 - unrecoverable error
2 - recoverable error
```

If `cloud-init status` exits with exit code 1, cloud-init experienced critical failure and was unable to recover. In this case, something is likely seriously wrong with the system, or cloud-init has experienced a serious bug. If you believe that you have experienced a serious bug, please file a *bug report*.

If cloud-init exits with exit code 2, cloud-init was able to complete gracefully, however something went wrong and the user should investigate.

See *this more detailed explanation* for more information on cloud-init's status.

Where to next?

See *our more detailed guide* for a detailed guide to debugging cloud-init.

2.3.11 Exported errors

Cloud-init makes internal errors available to users for debugging. These errors map to logged errors and may be useful for understanding what happens when cloud-init doesn't do what you expect.

Aggregated errors

When a *recoverable error* occurs, the internal cloud-init state information is made visible under a top level aggregate key `recoverable_errors` with errors sorted by error level:

```
$ cloud-init status --format json
{
  "boot_status_code": "enabled-by-generator",
  "config": {...},
  "datasource": "",
  "detail": "Cloud-init enabled by systemd cloud-init-generator",
  "errors": [],
  "extended_status": "degraded done",
  "init": {...},
  "last_update": "",
  "recoverable_errors":
  {
    "WARNING": [
      "Failed at merging in cloud config part from p-01: empty cloud config",
      "No template found in /etc/cloud/templates for template source.deb822",
      "No template found in /etc/cloud/templates for template sources.list",
      "No template found, not rendering /etc/apt/soures.list.d/ubuntu.source"
    ]
  },
  "status": "done"
}
```

Reported recoverable error messages are grouped by the level at which they are logged. Complete list of levels in order of increasing criticality:

```
WARNING
DEPRECATED
ERROR
CRITICAL
```

Each message has a single level. In cloud-init's *log files*, the level at which logs are reported is configurable. These messages are exported via the `'recoverable_errors'` key regardless of which level of logging is configured.

Per-stage errors

The keys `errors` and `recoverable_errors` are also exported for each stage to allow identifying when recoverable and non-recoverable errors occurred.

```
$ cloud-init status --format json
{
  "boot_status_code": "enabled-by-generator",
  "config":
  {
    "WARNING": [
      "No template found in /etc/cloud/templates for template source.deb822",
      "No template found in /etc/cloud/templates for template sources.list",
      "No template found, not rendering /etc/apt/soures.list.d/ubuntu.source"
    ]
  },
  "datasource": "",
  "detail": "Cloud-init enabled by systemd cloud-init-generator",
  "errors": [],
  "extended_status": "degraded done",
  "init":
  {
    "WARNING": [
      "Failed at merging in cloud config part from p-01: empty cloud config",
    ]
  },
  "last_update": "",
  "recoverable_errors":
  {
    "WARNING": [
      "Failed at merging in cloud config part from p-01: empty cloud config",
      "No template found in /etc/cloud/templates for template source.deb822",
      "No template found in /etc/cloud/templates for template sources.list",
      "No template found, not rendering /etc/apt/soures.list.d/ubuntu.source"
    ]
  },
  "status": "done"
}
```

Note: Only completed cloud-init stages are listed in the output of `cloud-init status --format json`.

The JSON representation of cloud-init *boot stages* (in run order) is:

```
"init-local"
"init"
"modules-config"
"modules-final"
```

Limitations of exported errors

- Exported recoverable errors represent logged messages, which are not guaranteed to be stable between releases. The contents of the `'errors'` and `'recoverable_errors'` keys are not guaranteed to have stable output.
- Exported errors and recoverable errors may occur at different stages since users may reorder configuration modules to run at different stages via `cloud.cfg`.

Where to next?

See *here* for a detailed guide to debugging cloud-init.

2.3.12 Why did *cloud-init status* start returning exit code 2?

Cloud-init introduced *a new error code* in 23.4. This page describes the purpose of this change and gives some context for why this change was made.

Background

Since cloud-init provides access to cloud instances, the paradigm for handling errors was "log errors, but proceed". Exiting on failure conditions doesn't make sense when that may prevent one from accessing the system to debug it.

Since cloud-init's behavior is heavily tied to specific cloud platforms, reproducing cloud-init bugs without exactly reproducing a specific cloud environment is often impossible, and often requires guesswork. To make debugging cloud-init possible without reproducing exactly, cloud-init logs are quite verbose.

Pain points

1) Invalid configurations were historically ignored.
2) Log verbosity is unfriendly to end users that may not know what to look for. Verbose logs means users often ignore real errors.
3) Cloud-init's reported status was only capable of telling the user whether cloud-init crashed. Cloud-init would report a status of "done" in the following cases:
    - a user's configuration was invalid
    - if the operating system or cloud environment experienced some error that prevented cloud-init from configuring the instance
    - if cloud-init internally experienced an error - all of these previously reported a status of "done".

Efforts to improve cloud-init

Several changes have been introduced to cloud-init to address the pain points described above.

JSON schema

Cloud-init has defined a JSON schema which fully documents the user-data cloud-config. This JSON schema may be used in several different ways:

Text editor integration

Thanks to yaml-language-server, cloud-init's JSON schema may be used for YAML syntax checking, warnings when invalid keys are used, and autocompletion. Several different text editors are capable of this. See this blog post on configuring this for neovim, or for VScode one can install the extension and then a file named `cloud-config.yaml` will automatically use cloud-init's JSON schema.

Cloud-init schema subcommand

The cloud-init package includes a cloud-init subcommand, *cloud-init schema* which uses the schema to validate either the configuration passed to the instance that you are running the command on, or to validate an arbitrary text file containing a configuration.

Return codes

Cloud-init historically used two return codes from the `cloud-init status` subcommand: 0 to indicate success and 1 to indicate failure. These return codes lacked nuance. Return code 0 (success) included the in-between when something went wrong, but cloud-init was able to finish.

Many users of cloud-init run `cloud-init status --wait` and expect that when complete, cloud-init has finished. Since cloud-init is not guaranteed to succeed, users should also be check the return code of this command.

As of 23.4, errors that do not crash cloud-init will have an exit code of 2. Exit code of 1 means that cloud-init crashed, and an exit code 0 more correctly means that cloud-init succeeded. Anyone that previously checked for exit code 0 should probably update their assumptions in one of the following two ways:

Users that wish to take advantage of cloud-init's error reporting capabilities should check for exit code of 2 from `cloud-init status`. An example of this:

```
from logging import getLogger
from json import loads
from subprocess import run
from sys import exit

logger = getLogger(__name__)
completed = run("cloud-init status --format json")
output = loads(completed.stdout)

if 2 == completed.return_code:
    # something bad might have happened - we should check it out
    logger.warning("cloud-init experienced a recoverable error")
    logger.warning("status: %s", output.get("extended_status"))
```

(continues on next page)

(continued from previous page)

```
    logger.warning("recoverable error: %s", output.get("recoverable_errors"))

elif 1 == completed.return_code:
    # cloud-init completely failed
    logger.error("cloud-init crashed, all bets are off!")
    exit(1)
```

Users that wish to use ignore cloud-init's errors and check the return code in a backwards-compatible way should check that the return code is not equal to 1. This will provide the same behavior before and after the changed exit code. See an example of this:

```
from logging import getLogger
from subprocess import run
from sys import exit

logger = getLogger(__name__)
completed = run("cloud-init status --format json")

if 1 == completed.return_code:
    # cloud-init completely failed
    logger.error("cloud-init crashed, all bets are off!")
    exit(1)

# cloud-init might have failed, but this code ignores that possibility
# in preference of backwards compatibility
```

See *our explanation of failure states* for more information.

2.4 Reference

Our reference section contains support information for `cloud-init`. This includes details on the network requirements, API definitions, support matrices and so on.

2.4.1 Module reference

Deprecation schedule and versions

Keys can be documented as `deprecated`, `new`, or `changed`. This allows cloud-init to evolve as requirements change, and to adopt better practices without maintaining design decisions indefinitely.

Keys marked as `deprecated` or `changed` may be removed or changed 5 years from the deprecation date. For example, if a key is deprecated in version `22.1` (the first release in 2022) it is scheduled to be removed in `27.1` (first release in 2027). Use of deprecated keys may cause warnings in the logs. If a key's expected value changes, the key will be marked `changed` with a date. A 5 year timeline also applies to changed keys.

Modules

Ansible

Configure Ansible for instance

Summary

This module provides Ansible integration for augmenting cloud-init's configuration of the local node. This module installs `ansible` during boot and then uses `ansible-pull` to run the playbook repository at the remote URL.

Internal name: `cc_ansible`

Module frequency: once-per-instance

Supported distros: all

Activate only on keys: `ansible`

Config schema

- **ansible:** (object)
  - **install_method:** (`distro/pip`) The type of installation for ansible. It can be one of the following values:
    - `distro`
    - `pip`
  - **run_user:** (string) User to run module commands as. If install_method: pip, the pip install runs as this user as well.
  - **ansible_config:** (string) Sets the ANSIBLE_CONFIG environment variable. If set, overrides default config.
  - **setup_controller:** (object)
    - **repositories:** (array of object) Each object in **repositories** list supports the following keys:
      - **path:** (string)
      - **source:** (string)
    - **run_ansible:** (array of object) Each object in **run_ansible** list supports the following keys:
      - **playbook_name:** (string)
      - **playbook_dir:** (string)
      - **become_password_file:** (string)
      - **connection_password_file:** (string)
      - **list_hosts:** (boolean)
      - **syntax_check:** (boolean)
      - **timeout:** (number)
      - **vault_id:** (string)
      - **vault_password_file:** (string)
      - **background:** (number)

        - **check:** (boolean)
        - **diff:** (boolean)
        - **module_path:** (string)
        - **poll:** (number)
        - **args:** (string)
        - **extra_vars:** (string)
        - **forks:** (number)
        - **inventory:** (string)
        - **scp_extra_args:** (string)
        - **sftp_extra_args:** (string)
        - **private_key:** (string)
        - **connection:** (string)
        - **module_name:** (string)
        - **sleep:** (string)
        - **tags:** (string)
        - **skip_tags:** (string)
  - **galaxy:** (object)
    - **actions:** (array of array of string)
  - **package_name:** (string)
  - **pull:** (object)
    - **accept_host_key:** (boolean)
    - **clean:** (boolean)
    - **full:** (boolean)
    - **diff:** (boolean)
    - **ssh_common_args:** (string)
    - **scp_extra_args:** (string)
    - **sftp_extra_args:** (string)
    - **private_key:** (string)
    - **checkout:** (string)
    - **module_path:** (string)
    - **timeout:** (string)
    - **url:** (string)
    - **connection:** (string)
    - **vault_id:** (string)
    - **vault_password_file:** (string)
    - **verify_commit:** (boolean)

- **inventory:** (string)
- **module_name:** (string)
- **sleep:** (string)
- **tags:** (string)
- **skip_tags:** (string)
- **playbook_name:** (string)

Examples

Example 1:

```
#cloud-config
ansible:
  package_name: ansible-core
  install_method: distro
  pull:
    url: https://github.com/holmanb/vmboot.git
    playbook_name: ubuntu.yml
```

Example 2:

```
#cloud-config
ansible:
  package_name: ansible-core
  install_method: pip
  pull:
    url: https://github.com/holmanb/vmboot.git
    playbook_name: ubuntu.yml
```

APK Configure

Configure APK repositories file

Summary

This module handles configuration of the Alpine Package Keeper (APK) `/etc/apk/repositories` file.

Note: To ensure that APK configuration is valid YAML, any strings containing special characters, especially colons, should be quoted (":").

Internal name: `cc_apk_configure`

Module frequency: once-per-instance

Supported distros: alpine

Activate only on keys: `apk_repos`

Config schema

- **apk_repos:** (object)
  - **preserve_repositories:** (boolean) By default, cloud-init will generate a new repositories file `/etc/apk/repositories` based on any valid configuration settings specified within a apk_repos section of cloud config. To disable this behavior and preserve the repositories file from the pristine image, set `preserve_repositories` to `true`.

    The `preserve_repositories` option overrides all other config keys that would alter `/etc/apk/repositories`.
  - **alpine_repo:** (object/null)
    - **base_url:** (string) The base URL of an Alpine repository, or mirror, to download official packages from. If not specified then it defaults to `https://alpine.global.ssl.fastly.net/alpine`
    - **community_enabled:** (boolean) Whether to add the Community repo to the repositories file. By default the Community repo is not included.
    - **testing_enabled:** (boolean) Whether to add the Testing repo to the repositories file. By default the Testing repo is not included. It is only recommended to use the Testing repo on a machine running the `Edge` version of Alpine as packages installed from Testing may have dependencies that conflict with those in non-Edge Main or Community repos.
    - **version:** (string) The Alpine version to use (e.g. `v3.12` or `edge`)
  - **local_repo_base_url:** (string) The base URL of an Alpine repository containing unofficial packages

Examples

Example 1: Keep the existing `/etc/apk/repositories` file unaltered.

```
#cloud-config
apk_repos:
  preserve_repositories: true
```

Example 2: Create repositories file for Alpine v3.12 main and community using default mirror site.

```
#cloud-config
apk_repos:
  alpine_repo:
    community_enabled: true
    version: 'v3.12'
```

Example 3: Create repositories file for Alpine Edge main, community, and testing using a specified mirror site and also a local repo.

```
#cloud-config
apk_repos:
  alpine_repo:
    base_url: https://some-alpine-mirror/alpine
    community_enabled: true
    testing_enabled: true
    version: edge
  local_repo_base_url: https://my-local-server/local-alpine
```

Apt Configure

Configure APT for the user

Summary

This module handles configuration of advanced package tool (APT) options and adding source lists. There are configuration options such as *apt_get_wrapper* ` and `apt_get_command` that control how cloud-init invokes `apt-get`. These configuration options are handled on a per-distro basis, so consult documentation for cloud-init's distro support for instructions on using these config options.

By default, cloud-init will generate default APT sources information in `deb822` format at `/etc/apt/sources.list.d/<distro>.sources`. When the value of `sources_list` does not appear to be `deb822` format, or stable distribution releases disable `deb822` format, `/etc/apt/sources.list` will be written instead.

Note: To ensure that APT configuration is valid YAML, any strings containing special characters, especially colons, should be quoted (":").

Note: For more information about APT configuration, see the "Additional APT configuration" example.

Internal name: `cc_apt_configure`

Module frequency: once-per-instance

Supported distros: ubuntu, debian

Config schema

- **apt:** (object)
  - **preserve_sources_list:** (boolean) By default, cloud-init will generate a new sources list in `/etc/apt/sources.list.d` based on any changes specified in cloud config. To disable this behavior and preserve the sources list from the pristine image, set `preserve_sources_list` to `true`.

    The `preserve_sources_list` option overrides all other config keys that would alter `sources.list` or `sources.list.d`, **except** for additional sources to be added to `sources.list.d`.
  - **disable_suites:** (array of string) Entries in the sources list can be disabled using `disable_suites`, which takes a list of suites to be disabled. If the string `$RELEASE` is present in a suite in the `disable_suites` list, it will be replaced with the release name. If a suite specified in `disable_suites` is not present in `sources.list` it will be ignored. For convenience, several aliases are provided for`` disable_suites``:

    * `updates` => `$RELEASE-updates`
    * `backports` => `$RELEASE-backports`
    * `security` => `$RELEASE-security`
    * `proposed` => `$RELEASE-proposed`
    * `release` => `$RELEASE`.

    When a suite is disabled using `disable_suites`, its entry in `sources.list` is not deleted; it is just commented out.

- **primary:** (array of object) The primary and security archive mirrors can be specified using the `primary` and `security` keys, respectively. Both the `primary` and `security` keys take a list of configs, allowing mirrors to be specified on a per-architecture basis. Each config is a dictionary which must have an entry for `arches`, specifying which architectures that config entry is for. The keyword `default` applies to any architecture not explicitly listed. The mirror url can be specified with the `uri` key, or a list of mirrors to check can be provided in order, with the first mirror that can be resolved being selected. This allows the same configuration to be used in different environment, with different hosts used for a local APT mirror. If no mirror is provided by `uri` or `search`, `search_dns` may be used to search for dns names in the format `<distro>-mirror` in each of the following:

  * fqdn of this host per cloud metadata,
  * localdomain,
  * domains listed in `/etc/resolv.conf`.

  If there is a dns entry for `<distro>-mirror`, then it is assumed that there is a distro mirror at `http://<distro>-mirror.<domain>/<distro>`. If the `primary` key is defined, but not the `security` key, then then configuration for `primary` is also used for `security`. If `search_dns` is used for the `security` key, the search pattern will be `<distro>-security-mirror`.

  Each mirror may also specify a key to import via any of the following optional keys:

  * `keyid`: a key to import via shortid or fingerprint.
  * `key`: a raw PGP key.
  * `keyserver`: alternate keyserver to pull `keyid` key from.

  If no mirrors are specified, or all lookups fail, then default mirrors defined in the datasource are used. If none are present in the datasource either the following defaults are used:

  * `primary` => `http://archive.ubuntu.com/ubuntu`.
  * `security` => `http://security.ubuntu.com/ubuntu` Each object in **primary** list supports the following keys:
  * **arches:** (array of string)
  * **uri:** (string)
  * **search:** (array of string)
  * **search_dns:** (boolean)
  * **keyid:** (string)
  * **key:** (string)
  * **keyserver:** (string)

- **security:** (array of object) Please refer to the primary config documentation Each object in **security** list supports the following keys:

  * **arches:** (array of string)
  * **uri:** (string)
  * **search:** (array of string)
  * **search_dns:** (boolean)
  * **keyid:** (string)
  * **key:** (string)
  * **keyserver:** (string)

- **add_apt_repo_match:** (string) All source entries in `apt-sources` that match regex in `add_apt_repo_match` will be added to the system using `add-apt-repository`. If `add_apt_repo_match` is not specified, it defaults to `^[\w-]+:\w`
- **debconf_selections:** (object) Debconf additional configurations can be specified as a dictionary under the `debconf_selections` config key, with each key in the dict representing a different set of configurations. The value of each key must be a string containing all the debconf configurations that must be applied. We will bundle all of the values and pass them to `debconf-set-selections`. Therefore, each value line must be a valid entry for `debconf-set-selections`, meaning that they must possess for distinct fields:

  `pkgname question type answer`

  Where:

  * `pkgname` is the name of the package.
  * `question` the name of the questions.
  * `type` is the type of question.
  * `answer` is the value used to answer the question.

  For example: `ippackage ippackage/ip string 127.0.01`

  * **^.+$:** (string)
- **sources_list:** (string) Specifies a custom template for rendering `sources.list` . If no `sources_list` template is given, cloud-init will use sane default. Within this template, the following strings will be replaced with the appropriate values:

  * `$MIRROR`
  * `$RELEASE`
  * `$PRIMARY`
  * `$SECURITY`
  * `$KEY_FILE`
- **conf:** (string) Specify configuration for apt, such as proxy configuration. This configuration is specified as a string. For multi-line APT configuration, make sure to follow YAML syntax.
- **https_proxy:** (string) More convenient way to specify https APT proxy. https proxy url is specified in the format `https://[[user][:pass]@]host[:port]/`.
- **http_proxy:** (string) More convenient way to specify http APT proxy. http proxy url is specified in the format `http://[[user][:pass]@]host[:port]/`.
- **proxy:** (string) Alias for defining a http APT proxy.
- **ftp_proxy:** (string) More convenient way to specify ftp APT proxy. ftp proxy url is specified in the format `ftp://[[user][:pass]@]host[:port]/`.
- **sources:** (object) Source list entries can be specified as a dictionary under the `sources` config key, with each key in the dict representing a different source file. The key of each source entry will be used as an id that can be referenced in other config entries, as well as the filename for the source's configuration under `/etc/apt/sources.list.d`. If the name does not end with `.list`, it will be appended. If there is no configuration for a key in `sources`, no file will be written, but the key may still be referred to as an id in other `sources` entries.

  Each entry under `sources` is a dictionary which may contain any of the following optional keys:

  * `source`: a sources.list entry (some variable replacements apply).
  * `keyid`: a key to import via shortid or fingerprint.

* `key`: a raw PGP key.
* `keyserver`: alternate keyserver to pull `keyid` key from.
* `filename`: specify the name of the list file.
* `append`: If `true`, append to sources file, otherwise overwrite it. Default: `true`.

The `source` key supports variable replacements for the following strings:

* `$MIRROR`
* `$PRIMARY`
* `$SECURITY`
* `$RELEASE`
* `$KEY_FILE`
* **^.+$:** (object)
  - **source:** (string)
  - **keyid:** (string)
  - **key:** (string)
  - **keyserver:** (string)
  - **filename:** (string)
  - **append:** (boolean)

Examples

Example 1:

```
#cloud-config
apt:
  preserve_sources_list: false
  disable_suites:
    - $RELEASE-updates
    - backports
    - $RELEASE
    - mysuite
  primary:
    - arches:
        - amd64
        - i386
        - default
      uri: http://us.archive.ubuntu.com/ubuntu
      search:
        - http://cool.but-sometimes-unreachable.com/ubuntu
        - http://us.archive.ubuntu.com/ubuntu
      search_dns: false
    - arches:
        - s390x
        - arm64
      uri: http://archive-to-use-for-arm64.example.com/ubuntu
```

(continues on next page)

(continued from previous page)

```
  security:
    - arches:
        - default
      search_dns: true
  sources_list: |
      deb $MIRROR $RELEASE main restricted
      deb-src $MIRROR $RELEASE main restricted
      deb $PRIMARY $RELEASE universe restricted
      deb $SECURITY $RELEASE-security multiverse
  debconf_selections:
      set1: the-package the-package/some-flag boolean true
  conf: |
      APT {
          Get {
              Assume-Yes 'true';
              Fix-Broken 'true';
          }
      }
  proxy: http://[[user][:pass]@]host[:port]/
  http_proxy: http://[[user][:pass]@]host[:port]/
  ftp_proxy: ftp://[[user][:pass]@]host[:port]/
  https_proxy: https://[[user][:pass]@]host[:port]/
  sources:
      source1:
          keyid: keyid
          keyserver: keyserverurl
          source: deb [signed-by=$KEY_FILE] http://<url>/ bionic main
      source2:
          source: ppa:<ppa-name>
      source3:
          source: deb $MIRROR $RELEASE multiverse
          key: |
              ------BEGIN PGP PUBLIC KEY BLOCK-------
              <key data>
              ------END PGP PUBLIC KEY BLOCK-------
      source4:
          source: deb $MIRROR $RELEASE multiverse
          append: false
          key: |
              ------BEGIN PGP PUBLIC KEY BLOCK-------
              <key data>
              ------END PGP PUBLIC KEY BLOCK-------
```

Example 2: Cloud-init version 23.4 will generate a `deb822`-formatted `sources` file at `/etc/apt/sources.list.d/<distro>.sources` instead of `/etc/apt/sources.list` when `sources_list` content is in `deb822` format.

```
#cloud-config
apt:
    sources_list: |
      Types: deb
      URIs: http://archive.ubuntu.com/ubuntu/
      Suites: $RELEASE
```

(continues on next page)

(continued from previous page)

```
    Components: main
```

Apt Pipelining

Configure APT pipelining

Summary

This module configures APT's `Acquire::http::Pipeline-Depth` option, which controls how APT handles HTTP pipelining. It may be useful for pipelining to be disabled, because some web servers (such as S3) do not pipeline properly (LP: #948461).

Value configuration options for this module are:

- `os`: (Default) use distro default
- `false`: Disable pipelining altogether
- `<number>`: Manually specify pipeline depth. This is not recommended.

Internal name: `cc_apt_pipelining`

Module frequency: once-per-instance

Supported distros: ubuntu, debian

Activate only on keys: `apt_pipelining`

Config schema

- **apt_pipelining:** (integer/boolean/string)

Examples

Example 1:

```
#cloud-config
apt_pipelining: false
```

Example 2:

```
#cloud-config
apt_pipelining: os
```

Example 3:

```
#cloud-config
apt_pipelining: 3
```

Bootcmd

Run arbitrary commands early in the boot process

Summary

This module runs arbitrary commands very early in the boot process, only slightly after a boothook would run. This is very similar to a boothook, but more user friendly. The environment variable `INSTANCE_ID` will be set to the current instance ID for all run commands. Commands can be specified either as lists or strings. For invocation details, see `runcmd`.

Note: `bootcmd` should only be used for things that could not be done later in the boot process.

Note: When writing files, do not use `/tmp` dir as it races with `systemd-tmpfiles-clean` (LP: #1707222). Use `/run/somedir` instead.

Internal name: `cc_bootcmd`

Module frequency: always

Supported distros: all

Activate only on keys: `bootcmd`

Config schema

- **bootcmd:** (array of array of string/string)

Examples

Example 1:

```
#cloud-config
bootcmd:
- echo 192.168.1.130 us.archive.ubuntu.com > /etc/hosts
- [cloud-init-per, once, mymkfs, mkfs, /dev/vdb]
```

Byobu

Enable/disable Byobu system-wide and for the default user

Summary

This module controls whether Byobu is enabled or disabled system-wide and for the default system user. If Byobu is to be enabled, this module will ensure it is installed. Likewise, if Byobu is to be disabled, it will be removed (if installed).

Valid configuration options for this module are:

- `enable-system`: enable Byobu system-wide
- `enable-user`: enable Byobu for the default user
- `disable-system`: disable Byobu system-wide
- `disable-user`: disable Byobu for the default user
- `enable`: enable Byobu both system-wide and for the default user
- `disable`: disable Byobu for all users
- `user`: alias for `enable-user`
- `system`: alias for `enable-system`

Internal name: `cc_byobu`

Module frequency: once-per-instance

Supported distros: ubuntu, debian

Config schema

- **byobu_by_default:** (`enable-system/enable-user/disable-system/disable-user/enable/disable/user/system`)

Examples

Example 1:

```
#cloud-config
byobu_by_default: enable-user
```

Example 2:

```
#cloud-config
byobu_by_default: disable-system
```

CA Certificates

Add CA certificates

Summary

This module adds CA certificates to the system's CA store and updates any related files using the appropriate OS-specific utility. The default CA certificates can be disabled/deleted from use by the system with the configuration option `remove_defaults`.

Note: Certificates must be specified using valid YAML. To specify a multi-line certificate, the YAML multi-line list syntax must be used.

Note: Alpine Linux requires the `ca-certificates` package to be installed in order to provide the `update-ca-certificates` command.

Internal name: `cc_ca_certs`

Module frequency: once-per-instance

Supported distros: almalinux, aosc, cloudlinux, alpine, debian, fedora, rhel, opensuse, opensuse-microos, opensuse-tumbleweed, opensuse-leap, sle_hpc, sle-micro, sles, ubuntu, photon

Activate only on keys: `ca_certs`, `ca-certs`

Config schema

- **ca_certs:** (object)
  - **remove-defaults:** (boolean)

    Deprecated in version 22.3.Use ``remove_defaults`` instead.
  - **remove_defaults:** (boolean) Remove default CA certificates if true. Default: `false`
  - **trusted:** (array of string) List of trusted CA certificates to add.
- **ca-certs:** (object)

  Deprecated in version 22.3.Use ``ca_certs`` instead.
  - **remove-defaults:** (boolean)

    Deprecated in version 22.3.Use ``remove_defaults`` instead.
  - **remove_defaults:** (boolean) Remove default CA certificates if true. Default: `false`
  - **trusted:** (array of string) List of trusted CA certificates to add.

Examples

Example 1:

```
#cloud-config
ca_certs:
  remove_defaults: true
  trusted:
  - single_line_cert
  - |
    -----BEGIN CERTIFICATE-----
```

(continues on next page)

(continued from previous page)

```
    YOUR-ORGS-TRUSTED-CA-CERT-HERE
    -----END CERTIFICATE-----
```

Chef

Module that installs, configures, and starts Chef

Summary

This module enables Chef to be installed (from packages, gems, or from omnibus). Before this occurs, Chef configuration is written to disk (`validation.pem`, `client.pem`, `firstboot.json`, `client.rb`), and required directories are created (`/etc/chef` and `/var/log/chef` and so on).

If configured, Chef will be installed and started in either daemon or non-daemon mode. If run in non-daemon mode, post-run actions are executed to do finishing activities such as removing `validation.pem`.

Internal name: `cc_chef`

Module frequency: always

Supported distros: all

Activate only on keys: `chef`

Config schema

- **chef:** (object)
  - **directories:** (array of string) Create the necessary directories for chef to run. By default, it creates the following directories:
    - `/etc/chef`
    - `/var/log/chef`
    - `/var/lib/chef`
    - `/var/cache/chef`
    - `/var/backups/chef`
    - `/var/run/chef`
  - **validation_cert:** (string) Optional string to be written to file validation_key. Special value `system` means set use existing file.
  - **validation_key:** (string) Optional path for validation_cert. default to `/etc/chef/validation.pem`
  - **firstboot_path:** (string) Path to write run_list and initial_attributes keys that should also be present in this configuration, defaults to `/etc/chef/firstboot.json`
  - **exec:** (boolean) Set true if we should run or not run chef (defaults to false, unless a gem installed is requested where this will then default to true).
  - **client_key:** (string) Optional path for client_cert. Default: `/etc/chef/client.pem`.
  - **encrypted_data_bag_secret:** (string) Specifies the location of the secret key used by chef to encrypt data items. By default, this path is set to null, meaning that chef will have to look at the path `/etc/chef/encrypted_data_bag_secret` for it.

- **environment:** (string) Specifies which environment chef will use. By default, it will use the `_default` configuration.
- **file_backup_path:** (string) Specifies the location in which backup files are stored. By default, it uses the `/var/backups/chef` location.
- **file_cache_path:** (string) Specifies the location in which chef cache files will be saved. By default, it uses the `/var/cache/chef` location.
- **json_attribs:** (string) Specifies the location in which some chef json data is stored. By default, it uses the `/etc/chef/firstboot.json` location.
- **log_level:** (string) Defines the level of logging to be stored in the log file. By default this value is set to `:info`.
- **log_location:** (string) Specifies the location of the chef log file. By default, the location is specified at `/var/log/chef/client.log`.
- **node_name:** (string) The name of the node to run. By default, we will use th instance id as the node name.
- **omnibus_url:** (string) Omnibus URL if chef should be installed through Omnibus. By default, it uses the `https://www.chef.io/chef/install.sh`.
- **omnibus_url_retries:** (integer) The number of retries that will be attempted to reach the Omnibus URL. Default: `5`.
- **omnibus_version:** (string) Optional version string to require for omnibus install.
- **pid_file:** (string) The location in which a process identification number (pid) is saved. By default, it saves in the `/var/run/chef/client.pid` location.
- **server_url:** (string) The URL for the chef server
- **show_time:** (boolean) Show time in chef logs
- **ssl_verify_mode:** (string) Set the verify mode for HTTPS requests. We can have two possible values for this parameter:
  - `:verify_none`: No validation of SSL certificates.
  - `:verify_peer`: Validate all SSL certificates.

  By default, the parameter is set as `:verify_none`.
- **validation_name:** (string) The name of the chef-validator key that Chef Infra Client uses to access the Chef Infra Server during the initial Chef Infra Client run.
- **force_install:** (boolean) If set to `true`, forces chef installation, even if it is already installed.
- **initial_attributes:** (object of string) Specify a list of initial attributes used by the cookbooks.
- **install_type:** (`packages`/`gems`/`omnibus`) The type of installation for chef. It can be one of the following values:
  - `packages`
  - `gems`
  - `omnibus`
- **run_list:** (array of string) A run list for a first boot json.
- **chef_license:** (`accept`/`accept-silent`/`accept-no-persist`) string that indicates if user accepts or not license related to some of chef products. See https://docs.chef.io/licensing/accept/

Examples

Example 1:

```
#cloud-config
chef:
  directories: [/etc/chef, /var/log/chef]
  encrypted_data_bag_secret: /etc/chef/encrypted_data_bag_secret
  environment: _default
  initial_attributes:
    apache:
      keepalive: false
      prefork: {maxclients: 100}
  install_type: omnibus
  log_level: :auto
  omnibus_url_retries: 2
  run_list: ['recipe[apache2]', 'role[db]']
  server_url: https://chef.yourorg.com:4000
  ssl_verify_mode: :verify_peer
  validation_cert: system
  validation_name: yourorg-validator
```

Disable EC2 Metadata

Disable AWS EC2 Metadata

Summary

This module can disable the EC2 datasource by rejecting the route to `169.254.169.254`, the usual route to the datasource. This module is disabled by default.

Internal name: `cc_disable_ec2_metadata`

Module frequency: always

Supported distros: all

Activate only on keys: `disable_ec2_metadata`

Config schema

- **disable_ec2_metadata:** (boolean) Set true to disable IPv4 routes to EC2 metadata. Default: `false`

Examples

Example 1:

```
#cloud-config
disable_ec2_metadata: true
```

Disk Setup

Configure partitions and filesystems

Summary

This module configures simple partition tables and filesystems.

Note: For more detail about configuration options for disk setup, see the disk setup example.

Note: If a swap partition is being created via `disk_setup`, then an `fs_entry` entry is also needed in order for `mkswap` to be run, otherwise when swap activation is later attempted it will fail.

For convenience, aliases can be specified for disks using the `device_aliases` config key, which takes a dictionary of `alias: path` mappings. There are automatic aliases for `swap` and `ephemeral<X>`, where `swap` will always refer to the active swap partition and `ephemeral<X>` will refer to the block device of the ephemeral image.

Disk partitioning is done using the `disk_setup` directive. This config directive accepts a dictionary where each key is either a path to a block device or an alias specified in `device_aliases`, and each value is the configuration options for the device. File system configuration is done using the `fs_setup` directive. This config directive accepts a list of filesystem configs.

Internal name: `cc_disk_setup`

Module frequency: once-per-instance

Supported distros: all

Activate only on keys: `disk_setup`, `fs_setup`

Config schema

- **device_aliases:** (object)
  - **<alias_name>:** (string) Path to disk to be aliased by this name.
- **disk_setup:** (object)
  - **<alias name/path>:** (object)
    - **table_type:** (`mbr`/`gpt`) Specifies the partition table type, either `mbr` or `gpt`. Default: `mbr`.
    - **layout:** (`remove`/boolean/array of integer/array of) If set to `true`, a single partition using all the space on the device will be created. If set to `false`, no partitions will be created. If set to `remove`, any existing partition table will be purged. Partitions can be specified by providing a list to `layout`, where each entry in the list is either a size or a list containing a size and the numerical value for a partition type. The size for partitions is specified in **percentage** of disk space, not in bytes (e.g. a size of 33

would take up 1/3 of the disk space). The partition type defaults to '83' (Linux partition), for other types of partition, such as Linux swap, the type must be passed as part of a list along with the size. Default: `false`.

  * **overwrite:** (boolean) Controls whether this module tries to be safe about writing partition tables or not. If `overwrite: false` is set, the device will be checked for a partition table and for a file system and if either is found, the operation will be skipped. If `overwrite: true` is set, no checks will be performed. Using `overwrite: true` is **dangerous** and can lead to data loss, so double check that the correct device has been specified if using this option. Default: `false`

- **fs_setup:** (array of object) Each object in **fs_setup** list supports the following keys:
  - **label:** (string) Label for the filesystem.
  - **filesystem:** (string) Filesystem type to create. E.g., `ext4` or `btrfs`
  - **device:** (string) Specified either as a path or as an alias in the format `<alias name>.<y>` where `<y>` denotes the partition number on the device. If specifying device using the `<alias name>.<partition number>` format, the value of `partition` will be overwritten.
  - **partition:** (string/integer) The partition can be specified by setting `partition` to the desired partition number. The `partition` option may also be set to `auto`, in which this module will search for the existence of a filesystem matching the `label`, `filesystem` and `device` of the `fs_setup` entry and will skip creating the filesystem if one is found. The `partition` option may also be set to `any`, in which case any filesystem that matches `filesystem` and `device` will cause this module to skip filesystem creation for the `fs_setup` entry, regardless of `label` matching or not. To write a filesystem directly to a device, use `partition: none`. `partition: none` will **always** write the filesystem, even when the `label` and `filesystem` are matched, and `overwrite` is `false`.
  - **overwrite:** (boolean) If `true`, overwrite any existing filesystem. Using `overwrite: true` for filesystems is **dangerous** and can lead to data loss, so double check the entry in `fs_setup`. Default: `false`
  - **replace_fs:** (string) Ignored unless `partition` is `auto` or `any`. Default `false`.
  - **extra_opts:** (['array', 'string'] of string) Optional options to pass to the filesystem creation command. Ignored if you using `cmd` directly.
  - **cmd:** (['array', 'string'] of string) Optional command to run to create the filesystem. Can include string substitutions of the other `fs_setup` config keys. This is only necessary if you need to override the default command.

Examples

Example 1:

```
#cloud-config
device_aliases: {my_alias: /dev/sdb, swap_disk: /dev/sdc}
disk_setup:
  /dev/sdd: {layout: true, overwrite: true, table_type: mbr}
  my_alias:
    layout: [50, 50]
    overwrite: true
    table_type: gpt
  swap_disk:
    layout:
    - [100, 82]
    overwrite: true
    table_type: gpt
```

(continues on next page)

(continued from previous page)

```
fs_setup:
- {cmd: mkfs -t %(filesystem)s -L %(label)s %(device)s, device: my_alias.1, filesystem:␣
↪ext4,
  label: fs1}
- {device: my_alias.2, filesystem: ext4, label: fs2}
- {device: swap_disk.1, filesystem: swap, label: swap}
- {device: /dev/sdd1, filesystem: ext4, label: fs3}
mounts:
- [my_alias.1, /mnt1]
- [my_alias.2, /mnt2]
- [swap_disk.1, none, swap, sw, '0', '0']
- [/dev/sdd1, /mnt3]
```

Fan

Configure Ubuntu fan networking

Summary

This module installs, configures and starts the Ubuntu fan network system (Read more about Ubuntu Fan).

If cloud-init sees a `fan` entry in cloud-config it will:

- Write `config_path` with the contents of the `config` key
- Install the package `ubuntu-fan` if it is not installed
- Ensure the service is started (or restarted if was previously running)

Additionally, the `ubuntu-fan` package will be automatically installed if not present.

Internal name: `cc_fan`

Module frequency: once-per-instance

Supported distros: ubuntu

Activate only on keys: `fan`

Config schema

- **fan:** (object)
  - **config:** (string) The fan configuration to use as a single multi-line string
  - **config_path:** (string) The path to write the fan configuration to. Default: `/etc/network/fan`

Examples

Example 1:

```
#cloud-config
fan:
  config: |
    # fan 240
    10.0.0.0/8 eth0/16 dhcp
    10.0.0.0/8 eth1/16 dhcp off
    # fan 241
    241.0.0.0/8 eth0/16 dhcp
  config_path: /etc/network/fan
```

Final Message

Output final message when cloud-init has finished

Summary

This module configures the final message that cloud-init writes. The message is specified as a Jinja template with the following variables set:

- `version`: cloud-init version
- `timestamp`: time at cloud-init finish
- `datasource`: cloud-init data source
- `uptime`: system uptime

This message is written to the cloud-init log (usually `/var/log/cloud-init.log`) as well as stderr (which usually redirects to `/var/log/cloud-init-output.log`).

Upon exit, this module writes the system uptime, timestamp, and cloud-init version to `/var/lib/cloud/instance/boot-finished` independent of any user data specified for this module.

Internal name: `cc_final_message`

Module frequency: always

Supported distros: all

Config schema

- **final_message:** (string) The message to display at the end of the run

Examples

Example 1:

```
#cloud-config
final_message: |
  cloud-init has finished
  version: $version
  timestamp: $timestamp
  datasource: $datasource
  uptime: $uptime
```

Growpart

Grow partitions

Summary

Growpart resizes partitions to fill the available disk space. This is useful for cloud instances with a larger amount of disk space available than the pristine image uses, as it allows the instance to automatically make use of the extra space.

Note that this only works if the partition to be resized is the last one on a disk with classic partitioning scheme (MBR, BSD, GPT). LVM, Btrfs and ZFS have no such restrictions.

The devices on which to run growpart are specified as a list under the `devices` key.

There is some functionality overlap between this module and the `growroot` functionality of `cloud-initramfs-tools`. However, there are some situations where one tool is able to function and the other is not. The default configuration for both should work for most cloud instances. To explicitly prevent `cloud-initramfs-tools` from running `growroot`, the file `/etc/growroot-disabled` can be created.

By default, both `growroot` and `cc_growpart` will check for the existence of this file and will not run if it is present. However, this file can be ignored for `cc_growpart` by setting `ignore_growroot_disabled` to `true`. Read more about `cloud-initramfs-tools`.

On FreeBSD, there is also the `growfs` service, which has a lot of overlap with `cc_growpart` and `cc_resizefs`, but only works on the root partition. In that configuration, we use it, otherwise, we fall back to `gpart`.

Note: `growfs` may insert a swap partition, if none is present, unless instructed not to via `growfs_swap_size=0` in either `kenv(1)`, or `rc.conf(5)`.

Growpart is enabled by default on the root partition. The default config for growpart is:

```
growpart:
  mode: auto
  devices: [\"/\"]
  ignore_growroot_disabled: false
```

Internal name: `cc_growpart`

Module frequency: always

Supported distros: all

Config schema

- **growpart:** (object)
  - **mode:** (`auto`/`growpart`/`gpart`/`off`/`False`) The utility to use for resizing. Default: `auto`

    Possible options:

    * `auto` - Use any available utility
    * `growpart` - Use growpart utility
    * `gpart` - Use BSD gpart utility
    * `'off'` - Take no action

    Changed in version 22.3.Specifying a boolean ``false`` value for ``mode`` is deprecated. Use the string ``"off"`` instead.
  - **devices:** (array of string) The devices to resize. Each entry can either be the path to the device's mountpoint in the filesystem or a path to the block device in '/dev'. Default: `[/]`
  - **ignore_growroot_disabled:** (boolean) If `true`, ignore the presence of `/etc/growroot-disabled`. If `false` and the file exists, then don't resize. Default: `false`

Examples

Example 1:

```
#cloud-config
growpart:
  devices: [/]
  ignore_growroot_disabled: false
  mode: auto
```

Example 2:

```
#cloud-config
growpart:
  devices: [/, /dev/vdb1]
  ignore_growroot_disabled: true
  mode: growpart
```

GRUB dpkg

Configure GRUB debconf installation device

Summary

Configure which device is used as the target for GRUB installation. This module can be enabled/disabled using the `enabled` config key in the `grub_dpkg` config dict. This module automatically selects a disk using `grub-probe` if no installation device is specified.

The value placed into the debconf database is in the format expected by the GRUB post-install script expects. Normally, this is a `/dev/disk/by-id/` value, but we do fallback to the plain disk name if a `by-id` name is not present.

If this module is executed inside a container, then the debconf database is seeded with empty values, and `install_devices_empty` is set to `true`.

Internal name: `cc_grub_dpkg`

Module frequency: once-per-instance

Supported distros: ubuntu, debian

Config schema

- **grub_dpkg:** (object)
  - **enabled:** (boolean) Whether to configure which device is used as the target for grub installation. Default: `true`
  - **grub-pc/install_devices:** (string) Device to use as target for grub installation. If unspecified, `grub-probe` of `/boot` will be used to find the device
  - **grub-pc/install_devices_empty:** (boolean/string) Sets values for `grub-pc/install_devices_empty`. If unspecified, will be set to `true` if `grub-pc/install_devices` is empty, otherwise `false`

    Changed in version 22.3.Use a boolean value instead.
  - **grub-efi/install_devices:** (string) Partition to use as target for grub installation. If unspecified, `grub-probe` of `/boot/efi` will be used to find the partition
- **grub-dpkg:** (object)

  Deprecated in version 22.2.Use ``grub_dpkg`` instead.

Examples

Example 1:

```
#cloud-config
grub_dpkg:
  enabled: true
  # BIOS mode (install_devices needs disk)
  grub-pc/install_devices: /dev/sda
  grub-pc/install_devices_empty: false
  # EFI mode (install_devices needs partition)
  grub-efi/install_devices: /dev/sda
```

Install Hotplug

Install hotplug udev rules if supported and enabled

Summary

This module will install the udev rules to enable hotplug if supported by the datasource and enabled in the userdata. The udev rules will be installed as `/etc/udev/rules.d/90-cloud-init-hook-hotplug.rules`.

When hotplug is enabled, newly added network devices will be added to the system by cloud-init. After udev detects the event, cloud-init will refresh the instance metadata from the datasource, detect the device in the updated metadata, then apply the updated network configuration.

Currently supported datasources: Openstack, EC2

Internal name: `cc_install_hotplug`

Module frequency: once-per-instance

Supported distros: all

Config schema

- **updates:** (object)
  - **network:** (object)
    - **when:** (array of `boot-new-instance/boot-legacy/boot/hotplug`)

Examples

Example 1: Enable hotplug of network devices

```
#cloud-config
updates:
  network:
    when: [hotplug]
```

Example 2: Enable network hotplug alongside boot event

```
#cloud-config
updates:
  network:
    when: [boot, hotplug]
```

Keyboard

Set keyboard layout

Summary

Handle keyboard configuration.

Internal name: `cc_keyboard`

Module frequency: once-per-instance

Supported distros: alpine, arch, debian, ubuntu, almalinux, amazon, azurelinux, centos, cloudlinux, eurolinux, fedora, mariner, miraclelinux, openmandriva, photon, rhel, rocky, virtuozzo, opensuse, opensuse-leap, opensuse-microos, opensuse-tumbleweed, sle_hpc, sle-micro, sles, suse

Activate only on keys: `keyboard`

Config schema

- **keyboard:** (object)
  - **layout:** (string) Required. Keyboard layout. Corresponds to XKBLAYOUT.
  - **model:** (string) Optional. Keyboard model. Corresponds to XKBMODEL. Default: `pc105`.
  - **variant:** (string) Required for Alpine Linux, optional otherwise. Keyboard variant. Corresponds to XKBVARIANT.
  - **options:** (string) Optional. Keyboard options. Corresponds to XKBOPTIONS.

Examples

Example 1: Set keyboard layout to "us"

```
#cloud-config
keyboard:
  layout: us
```

Example 2: Set specific keyboard layout, model, variant, options

```
#cloud-config
keyboard:
  layout: de
  model: pc105
  variant: nodeadkeys
  options: compose:rwin
```

Example 3: For Alpine Linux, set specific keyboard layout and variant, as used by `setup-keymap`. Model and options are ignored.

```
#cloud-config
keyboard:
  layout: gb
  variant: gb-extd
```

Keys to Console

Control which SSH host keys may be written to console

Summary

For security reasons it may be desirable not to write SSH host keys and their fingerprints to the console. To avoid either of them being written to the console, the `emit_keys_to_console` config key under the main `ssh` config key can be used.

To avoid the fingerprint of types of SSH host keys being written to console the `ssh_fp_console_blacklist` config key can be used. By default, all types of keys will have their fingerprints written to console.

To avoid host keys of a key type being written to console the `ssh_key_console_blacklist` config key can be used. By default, all supported host keys are written to console.

Internal name: `cc_keys_to_console`

Module frequency: once-per-instance

Supported distros: all

Config schema

- **ssh:** (object)
  - **emit_keys_to_console:** (boolean) Set false to avoid printing SSH keys to system console. Default: `true`.
- **ssh_key_console_blacklist:** (array of string) Avoid printing matching SSH key types to the system console.
- **ssh_fp_console_blacklist:** (array of string) Avoid printing matching SSH fingerprints to the system console.

Examples

Example 1: Do not print any SSH keys to system console

```
#cloud-config
ssh:
  emit_keys_to_console: false
```

Example 2: Do not print certain SSH key types to console

```
#cloud-config
ssh_key_console_blacklist: [rsa]
```

Example 3: Do not print specific SSH key fingerprints to console

```
#cloud-config
ssh_fp_console_blacklist:
- E25451E0221B5773DEBFF178ECDACB160995AA89
- FE76292D55E8B28EE6DB2B34B2D8A784F8C0AAB0
```

Landscape

Install and configure Landscape client

Summary

This module installs and configures `landscape-client`. The Landscape client will only be installed if the key `landscape` is present in config.

Landscape client configuration is given under the `client` key under the main `landscape` config key. The config parameters are not interpreted by cloud-init, but rather are converted into a `ConfigObj`-formatted file and written out to the `[client]` section in `/etc/landscape/client.conf`. The following default client config is provided, but can be overridden

```
landscape:
  client:
    log_level: "info"
    url: "https://landscape.canonical.com/message-system"
    ping_url: "http://landscape.canoncial.com/ping"
    data_path: "/var/lib/landscape/client"
```

Note: See Landscape documentation for client config keys.

Note: If `tags` is defined, its contents should be a string delimited with a comma (“,”) rather than a list.

Internal name: `cc_landscape`

Module frequency: once-per-instance

Supported distros: ubuntu

Activate only on keys: `landscape`

Config schema

- **landscape:** (object)
  - **client:** (object)
    - **url:** (string) The Landscape server URL to connect to. Default: `https://landscape.canonical.com/message-system`.
    - **ping_url:** (string) The URL to perform lightweight exchange initiation with. Default: `https://landscape.canonical.com/ping`.
    - **data_path:** (string) The directory to store data files in. Default: `/var/lib/land-scape/client/`.
    - **log_level:** (`debug`/`info`/`warning`/`error`/`critical`) The log level for the client. Default: `info`.
    - **computer_title:** (string) The title of this computer.
    - **account_name:** (string) The account this computer belongs to.
    - **registration_key:** (string) The account-wide key used for registering clients.
    - **tags:** (string) Comma separated list of tag names to be sent to the server.

- **http_proxy:** (string) The URL of the HTTP proxy, if one is needed.
- **https_proxy:** (string) The URL of the HTTPS proxy, if one is needed.

Examples

To discover additional supported client keys, run `man landscape-config`.

Example 1:

```
#cloud-config
landscape:
  client:
    url: https://landscape.canonical.com/message-system
    ping_url: http://landscape.canonical.com/ping
    data_path: /var/lib/landscape/client
    http_proxy: http://my.proxy.com/foobar
    https_proxy: https://my.proxy.com/foobar
    tags: server,cloud
    computer_title: footitle
    registration_key: fookey
    account_name: fooaccount
```

Example 2: Minimum viable config requires `account_name` and `computer_title`.

```
#cloud-config
landscape:
  client:
    computer_title: kiosk 1
    account_name: Joe's Biz
```

Example 3: To install `landscape-client` from a PPA, specify `apt.sources`.

```
#cloud-config
apt:
  sources:
    trunk-testing-ppa:
      source: ppa:landscape/self-hosted-beta
landscape:
  client:
    account_name: myaccount
    computer_title: himom
```

Locale

Set system locale

Summary

Configure the system locale and apply it system-wide. By default, use the locale specified by the datasource.

Internal name: `cc_locale`

Module frequency: once-per-instance

Supported distros: all

Config schema

- **locale:** (string) The locale to set as the system's locale (e.g. ar_PS)
- **locale_configfile:** (string) The file in which to write the locale configuration (defaults to the distro's default location)

Examples

Example 1: Set the locale to `"ar_AE"`

```
#cloud-config
locale: ar_AE
```

Example 2: Set the locale to `"fr_CA"` in `/etc/alternate_path/locale`

```
#cloud-config
locale: fr_CA
locale_configfile: /etc/alternate_path/locale
```

LXD

Configure LXD with `lxd init` and (optionally) `lxd-bridge`

Summary

This module configures LXD with user-specified options using `lxd init`.

- If `lxd` is not present on the system but LXD configuration is provided, then `lxd` will be installed.
- If the selected storage backend userspace utility is not installed, it will be installed.
- If network bridge configuration is provided, then `lxd-bridge` will be configured accordingly.

Internal name: `cc_lxd`

Module frequency: once-per-instance

Supported distros: ubuntu

Activate only on keys: `lxd`

Config schema

- **lxd:** (object)
  - **init:** (object) LXD init configuration values to provide to *lxd init –auto* command. Can not be combined with `lxd.preseed`.
    - **network_address:** (string) IP address for LXD to listen on
    - **network_port:** (integer) Network port to bind LXD to.
    - **storage_backend:** (`zfs/dir/lvm/btrfs`) Storage backend to use. Default: `dir`.
    - **storage_create_device:** (string) Setup device based storage using DEVICE
    - **storage_create_loop:** (integer) Setup loop based storage with SIZE in GB
    - **storage_pool:** (string) Name of storage pool to use or create
    - **trust_password:** (string) The password required to add new clients
  - **bridge:** (object) LXD bridge configuration provided to setup the host lxd bridge. Can not be combined with `lxd.preseed`.
    - **mode:** (`none/existing/new`) Whether to setup LXD bridge, use an existing bridge by `name` or create a new bridge. *none* will avoid bridge setup, *existing* will configure lxd to use the bring matching `name` and *new* will create a new bridge.
    - **name:** (string) Name of the LXD network bridge to attach or create. Default: `lxdbr0`.
    - **mtu:** (integer) Bridge MTU, defaults to LXD's default value
    - **ipv4_address:** (string) IPv4 address for the bridge. If set, `ipv4_netmask` key required.
    - **ipv4_netmask:** (integer) Prefix length for the `ipv4_address` key. Required when `ipv4_address` is set.
    - **ipv4_dhcp_first:** (string) First IPv4 address of the DHCP range for the network created. This value will combined with `ipv4_dhcp_last` key to set LXC `ipv4.dhcp.ranges`.
    - **ipv4_dhcp_last:** (string) Last IPv4 address of the DHCP range for the network created. This value will combined with `ipv4_dhcp_first` key to set LXC `ipv4.dhcp.ranges`.
    - **ipv4_dhcp_leases:** (integer) Number of DHCP leases to allocate within the range. Automatically calculated based on *ipv4_dhcp_first* and *ipv4_dhcp_last* when unset.
    - **ipv4_nat:** (boolean) Set `true` to NAT the IPv4 traffic allowing for a routed IPv4 network. Default: `false`.
    - **ipv6_address:** (string) IPv6 address for the bridge (CIDR notation). When set, `ipv6_netmask` key is required. When absent, no IPv6 will be configured.
    - **ipv6_netmask:** (integer) Prefix length for `ipv6_address` provided. Required when `ipv6_address` is set.
    - **ipv6_nat:** (boolean) Whether to NAT. Default: `false`.
    - **domain:** (string) Domain to advertise to DHCP clients and use for DNS resolution.
  - **preseed:** (string) Opaque LXD preseed YAML config passed via stdin to the command: lxd init –preseed. See: https://documentation.ubuntu.com/lxd/en/latest/howto/initialize/#non-interactive-configuration or lxd init –dump for viable config. Can not be combined with either `lxd.init` or `lxd.bridge`.

Examples

Example 1: Simplest working directory-backed LXD configuration.

```
#cloud-config
lxd:
  init:
    storage_backend: dir
```

Example 2: `lxd-init` showcasing cloud-init's LXD config options.

```
#cloud-config
lxd:
  init:
    network_address: 0.0.0.0
    network_port: 8443
    storage_backend: zfs
    storage_pool: datapool
    storage_create_loop: 10
  bridge:
    mode: new
    mtu: 1500
    name: lxdbr0
    ipv4_address: 10.0.8.1
    ipv4_netmask: 24
    ipv4_dhcp_first: 10.0.8.2
    ipv4_dhcp_last: 10.0.8.3
    ipv4_dhcp_leases: 250
    ipv4_nat: true
    ipv6_address: fd98:9e0:3744::1
    ipv6_netmask: 64
    ipv6_nat: true
    domain: lxd
```

Example 3: For more complex non-interactive LXD configuration of networks, storage pools, profiles, projects, clusters and core config, `lxd:preseed` config will be passed as stdin to the command: `lxd init --preseed`.

See the LXD non-interactive configuration or run `lxd init --dump` to see viable preseed YAML allowed.

Preseed settings configuring the LXD daemon for HTTPS connections on 192.168.1.1 port 9999, a nested profile which allows for LXD nesting on containers and a limited project allowing for RBAC approach when defining behavior for sub-projects.

```
#cloud-config
lxd:
  preseed: |
    config:
      core.https_address: 192.168.1.1:9999
    networks:
      - config:
          ipv4.address: 10.42.42.1/24
          ipv4.nat: true
          ipv6.address: fd42:4242:4242:4242::1/64
          ipv6.nat: true
        description: ""
```

(continues on next page)

(continued from previous page)

```
    name: lxdbr0
    type: bridge
    project: default
  storage_pools:
  - config:
      size: 5GiB
      source: /var/snap/lxd/common/lxd/disks/default.img
    description: ""
    name: default
    driver: zfs
  profiles:
  - config: {}
    description: Default LXD profile
    devices:
      eth0:
        name: eth0
        network: lxdbr0
        type: nic
      root:
        path: /
        pool: default
        type: disk
    name: default
  - config: {}
    security.nesting: true
    devices:
      eth0:
        name: eth0
        network: lxdbr0
        type: nic
      root:
        path: /
        pool: default
        type: disk
    name: nested
  projects:
  - config:
      features.images: true
      features.networks: true
      features.profiles: true
      features.storage.volumes: true
    description: Default LXD project
    name: default
  - config:
      features.images: false
      features.networks: true
      features.profiles: false
      features.storage.volumes: false
    description: Limited Access LXD project
    name: limited
```

MCollective

Install, configure and start MCollective

Summary

This module installs, configures and starts MCollective. If the `mcollective` key is present in config, then MCollective will be installed and started.

Configuration for `mcollective` can be specified in the `conf` key under `mcollective`. Each config value consists of a key-value pair and will be written to `/etc/mcollective/server.cfg`. The `public-cert` and `private-cert` keys, if present in conf may be used to specify the public and private certificates for MCollective. Their values will be written to `/etc/mcollective/ssl/server-public.pem` and `/etc/mcollective/ssl/server-private.pem`.

> **Warning:** The EC2 metadata service is a network service and thus is readable by non-root users on the system (i.e., `ec2metadata --user-data`). If security is a concern, use `include-once` and SSL URLS.

Internal name: `cc_mcollective`

Module frequency: once-per-instance

Supported distros: all

Activate only on keys: `mcollective`

Config schema

- **mcollective:** (object)
  - **conf:** (object)
    - **public-cert:** (string) Optional value of server public certificate which will be written to `/etc/mcollective/ssl/server-public.pem`
    - **private-cert:** (string) Optional value of server private certificate which will be written to `/etc/mcollective/ssl/server-private.pem`
    - **^.+$:** (boolean/integer/string) Optional config key: value pairs which will be appended to `/etc/mcollective/server.cfg`.

Examples

Example 1: Provide server private and public key, and provide the `loglevel: debug` and `plugin.stomp.host: dbhost` config settings in `/etc/mcollective/server.cfg`:

```
#cloud-config
mcollective:
  conf:
    loglevel: debug
    plugin.stomp.host: dbhost
    public-cert: |
      -------BEGIN CERTIFICATE--------
      <cert data>
      -------END CERTIFICATE--------
```

(continues on next page)

(continued from previous page)

```
  private-cert: |
    -------BEGIN CERTIFICATE--------
    <cert data>
    -------END CERTIFICATE--------
```

Mounts

Configure mount points and swap files

Summary

This module can add or remove mount points from `/etc/fstab` as well as configure swap. The `mounts` config key takes a list of `fstab` entries to add. Each entry is specified as a list of `[ fs_spec, fs_file, fs_vfstype, fs_mntops, fs-freq, fs_passno ]`.

For more information on these options, consult the manual for `/etc/fstab`. When specifying the `fs_spec`, if the device name starts with one of `xvd`, `sd`, `hd`, or `vd`, the leading `/dev` may be omitted. Any mounts that do not appear to either an attached block device or network resource will be skipped with a log like "Ignoring nonexistent mount ...".

Cloud-init will attempt to add the following mount directives if available and unconfigured in `/etc/fstab`:

```
mounts:
- ["ephemeral0", "/mnt", "auto", "defaults,nofail,x-systemd.after=cloud-init.service", "0
↪", "2"]
- ["swap", "none", "swap", "sw", "0", "0"]
```

In order to remove a previously-listed mount, an entry can be added to the `mounts` list containing `fs_spec` for the device to be removed but no mount point (i.e. `[ swap ]` or `[ swap, null ]`).

The `mount_default_fields` config key allows default values to be specified for the fields in a `mounts` entry that are not specified, aside from the `fs_spec` and the `fs_file` fields. If specified, this must be a list containing 6 values. It defaults to:

```
mount_default_fields: [none, none, "auto", "defaults,nofail,x-systemd.after=cloud-init.
↪service", "0", "2"]
```

Non-systemd init systems will vary in `mount_default_fields`.

Swap files can be configured by setting the path to the swap file to create with `filename`, the size of the swap file with `size`, maximum size of the swap file if using an `size: auto` with `maxsize`. By default, no swap file is created.

Note: If multiple mounts are specified, where a subsequent mount's mount point is inside of a previously-declared mount's mount point, (i.e. the 1st mount has a mount point of `/abc` and the 2nd mount has a mount point of `/abc/def`) then this will not work as expected – `cc_mounts` first creates the directories for all the mount points **before** it starts to perform any mounts and so the sub-mount point directory will not be created correctly inside the parent mount point.

For systems using `util-linux`'s `mount` program, this issue can be worked around by specifying `X-mount.mkdir` as part of a `fs_mntops` value for the subsequent mount entry.

Internal name: `cc_mounts`

Module frequency: once-per-instance

Supported distros: all

Config schema

- **mounts:** (array of array of string) List of lists. Each inner list entry is a list of `/etc/fstab` mount declarations of the format: [fs_spec, fs_file, fs_vfstype, fs_mntops, fs-freq, fs_passno]. A mount declaration with less than 6 items will get remaining values from `mount_default_fields`. A mount declaration with only *fs_spec* and no *fs_file* mountpoint will be skipped.
- **mount_default_fields:** (array of string/null) Default mount configuration for any mount entry with less than 6 options provided. When specified, 6 items are required and represent `/etc/fstab` entries. Default: `defaults,nofail,x-systemd.after=cloud-init.service,_netdev`
- **swap:** (object)
  - **filename:** (string) Path to the swap file to create
  - **size:** (`auto`/integer/string) The size in bytes of the swap file, 'auto' or a human-readable size abbreviation of the format <float_size><units> where units are one of B, K, M, G or T. **WARNING: Attempts to use IEC prefixes in your configuration prior to cloud-init version 23.1 will result in unexpected behavior. SI prefixes names (KB, MB) are required on pre-23.1 cloud-init, however IEC values are used. In summary, assume 1KB == 1024B, not 1000B**
  - **maxsize:** (integer/string) The maxsize in bytes of the swap file

Examples

Example 1: Mount `ephemeral0` with `noexec` flag, `/dev/sdc` with `mount_default_fields`, and `/dev/xvdh` with custom `fs_passno "0"` to avoid `fsck` on the mount.

Also provide an automatically-sized swap with a max size of 10485760 bytes.

```
#cloud-config
mounts:
- [ /dev/ephemeral0, /mnt, auto, "defaults,noexec" ]
- [ sdc, /opt/data ]
- [ xvdh, /opt/data, auto, "defaults,nofail", "0", "0" ]
mount_default_fields: [None, None, auto, "defaults,nofail", "0", "2"]
swap:
  filename: /my/swapfile
  size: auto
  maxsize: 10485760
```

Example 2: Create a 2 GB swap file at `/swapfile` using human-readable values.

```
#cloud-config
swap:
  filename: /swapfile
  size: 2G
  maxsize: 2G
```

NTP

Enable and configure NTP

Summary

Handle Network Time Protocol (NTP) configuration. If `ntp` is not installed on the system and NTP configuration is specified, `ntp` will be installed.

If there is a default NTP config file in the image or one is present in the distro's `ntp` package, it will be copied to a file with `.dist` appended to the filename before any changes are made.

A list of NTP pools and NTP servers can be provided under the `ntp` config key.

If no NTP `servers` or `pools` are provided, 4 pools will be used in the format:

`{0-3}.{distro}.pool.ntp.org`

Internal name: `cc_ntp`

Module frequency: once-per-instance

Supported distros: almalinux, alpine, aosc, azurelinux, centos, cloudlinux, cos, debian, eurolinux, fedora, freebsd, mariner, miraclelinux, openbsd, openeuler, OpenCloudOS, openmandriva, opensuse, opensuse-microos, opensuse-tumbleweed, opensuse-leap, photon, rhel, rocky, sle_hpc, sle-micro, sles, TencentOS, ubuntu, virtuozzo

Activate only on keys: `ntp`

Config schema

- **ntp:** (null/object)
  - **pools:** (array of string) List of ntp pools. If both pools and servers are empty, 4 default pool servers will be provided of the format `{0-3}.{distro}.pool.ntp.org`. NOTE: for Alpine Linux when using the Busybox NTP client this setting will be ignored due to the limited functionality of Busybox's ntpd.
  - **servers:** (array of string) List of ntp servers. If both pools and servers are empty, 4 default pool servers will be provided with the format `{0-3}.{distro}.pool.ntp.org`.
  - **peers:** (array of string) List of ntp peers.
  - **allow:** (array of string) List of CIDRs to allow
  - **ntp_client:** (string) Name of an NTP client to use to configure system NTP. When unprovided or 'auto' the default client preferred by the distribution will be used. The following built-in client names can be used to override existing configuration defaults: chrony, ntp, openntpd, ntpdate, systemd-timesyncd.
  - **enabled:** (boolean) Attempt to enable ntp clients if set to True. If set to `false`, ntp client will not be configured or installed
  - **config:** (object) Configuration settings or overrides for the `ntp_client` specified.
    - **confpath:** (string) The path to where the `ntp_client` configuration is written.
    - **check_exe:** (string) The executable name for the `ntp_client`. For example, ntp service `check_exe` is 'ntpd' because it runs the ntpd binary.
    - **packages:** (array of string) List of packages needed to be installed for the selected `ntp_client`.
    - **service_name:** (string) The systemd or sysvinit service name used to start and stop the `ntp_client` service.

* **template:** (string) Inline template allowing users to customize their `ntp_client` configuration with the use of the Jinja templating engine. The template content should start with `## template:jinja`. Within the template, you can utilize any of the following ntp module config keys: `servers`, `pools`, `allow`, and `peers`. Each cc_ntp schema config key and expected value type is defined above.

Examples

Example 1: Override NTP with chrony configuration on Ubuntu.

```
#cloud-config
ntp:
  enabled: true
  ntp_client: chrony  # Uses cloud-init default chrony configuration
```

Example 2: Provide a custom NTP client configuration.

```
#cloud-config
ntp:
  enabled: true
  ntp_client: myntpclient
  config:
    confpath: /etc/myntpclient/myntpclient.conf
    check_exe: myntpclientd
    packages:
    - myntpclient
    service_name: myntpclient
    template: |
      ## template:jinja
      # My NTP Client config
      {% if pools -%}# pools{% endif %}
      {% for pool in pools -%}
      pool {{pool}} iburst
      {% endfor %}
      {%- if servers %}# servers
      {% endif %}
      {% for server in servers -%}
      server {{server}} iburst
      {% endfor %}
      {% if peers -%}# peers{% endif %}
      {% for peer in peers -%}
      peer {{peer}}
      {% endfor %}
      {% if allow -%}# allow{% endif %}
      {% for cidr in allow -%}
      allow {{cidr}}
      {% endfor %}
  pools: [0.int.pool.ntp.org, 1.int.pool.ntp.org, ntp.myorg.org]
  servers:
  - ntp.server.local
  - ntp.ubuntu.com
  - 192.168.23.2
  allow:
  - 192.168.23.0/32
```

(continues on next page)

(continued from previous page)

```
peers:
- km001
- km002
```

Package Update Upgrade Install

Update, upgrade, and install packages

Summary

This module allows packages to be updated, upgraded or installed during boot. If any packages are to be installed or an upgrade is to be performed then the package cache will be updated first. If a package installation or upgrade requires a reboot, then a reboot can be performed if `package_reboot_if_required` is specified.

Internal name: `cc_package_update_upgrade_install`

Module frequency: once-per-instance

Supported distros: all

Activate only on keys: `apt_update`, `package_update`, `apt_upgrade`, `package_upgrade`, `packages`

Config schema

- **packages:** (array of object/array of string/string) An array containing either a package specification, or an object consisting of a package manager key having a package specification value . A package specification can be either a package name or a list with two entries, the first being the package name and the second being the specific package version to install.
- **package_update:** (boolean) Set `true` to update packages. Happens before upgrade or install. Default: `false`
- **package_upgrade:** (boolean) Set `true` to upgrade packages. Happens before install. Default: `false`
- **package_reboot_if_required:** (boolean) Set `true` to reboot the system if required by presence of */var/run/reboot-required*. Default: `false`
- **apt_update:** (boolean)

  Deprecated in version 22.2.Use ``package_update`` instead.
- **apt_upgrade:** (boolean)

  Deprecated in version 22.2.Use ``package_upgrade`` instead.
- **apt_reboot_if_required:** (boolean)

  Deprecated in version 22.2.Use ``package_reboot_if_required`` instead.

Examples

Example 1:

```
#cloud-config
package_reboot_if_required: true
package_update: true
package_upgrade: true
packages:
- pwgen
- pastebinit
- [libpython3.8, 3.8.10-0ubuntu1~20.04.2]
- snap:
  - certbot
  - [juju, --edge]
  - [lxd, --channel=5.15/stable]
- apt: [mg]
```

Phone Home

Post data to URL

Summary

This module can be used to post data to a remote host after boot is complete. If the post URL contains the string `$INSTANCE_ID` it will be replaced with the ID of the current instance.

Either all data can be posted, or a list of keys to post.

Available keys are:

- `pub_key_rsa`
- `pub_key_ecdsa`
- `pub_key_ed25519`
- `instance_id`
- `hostname`
- `fdqn`

Data is sent as `x-www-form-urlencoded` arguments.

Example HTTP POST:

```
POST / HTTP/1.1
Content-Length: 1337
User-Agent: Cloud-Init/21.4
Accept-Encoding: gzip, deflate
Accept: */*
Content-Type: application/x-www-form-urlencoded

pub_key_rsa=rsa_contents&pub_key_ecdsa=ecdsa_contents&pub_key_ed25519=ed25519_contents&
↪instance_id=i-87018aed&hostname=myhost&fqdn=myhost.internal
```

Internal name: `cc_phone_home`

Module frequency: once-per-instance

Supported distros: all

Activate only on keys: `phone_home`

Config schema

- **phone_home:** (object)
  - **url:** (string) The URL to send the phone home data to.
  - **post:** (`all`/array of `pub_key_rsa`/`pub_key_ecdsa`/`pub_key_ed25519`/`instance_id`/`hostname`/`fqdn`) A list of keys to post or `all`. Default: `all`
  - **tries:** (integer) The number of times to try sending the phone home data. Default: `10`

Examples

Example 1:

```
#cloud-config
phone_home: {post: all, url: 'http://example.com/$INSTANCE_ID/'}
```

Example 2:

```
#cloud-config
phone_home:
  post: [pub_key_rsa, pub_key_ecdsa, pub_key_ed25519, instance_id, hostname, fqdn]
  tries: 5
  url: http://example.com/$INSTANCE_ID/
```

Power State Change

Change power state

Summary

This module handles shutdown/reboot after all config modules have been run. By default it will take no action, and the system will keep running unless a package installation/upgrade requires a system reboot (e.g. installing a new kernel) and `package_reboot_if_required` is `true`.

Using this module ensures that cloud-init is entirely finished with modules that would be executed. An example to distinguish delay from timeout:

If you delay 5 (5 minutes) and have a timeout of 120 (2 minutes), the max time until shutdown will be 7 minutes, though it could be as soon as 5 minutes. Cloud-init will invoke 'shutdown +5' after the process finishes, or when 'timeout' seconds have elapsed.

Note: With Alpine Linux any message value specified is ignored as Alpine's `halt`, `poweroff`, and `reboot` commands do not support broadcasting a message.

Internal name: `cc_power_state_change`

Module frequency: once-per-instance

Supported distros: all

Activate only on keys: `power_state`

Config schema

- **power_state:** (object)
  - **delay:** (integer/string/`now`) Time in minutes to delay after cloud-init has finished. Can be `now` or an integer specifying the number of minutes to delay. Default: `now`

    Changed in version 22.3.Use of type string for this value is deprecated. Use ``now`` or integer type.
  - **mode:** (`poweroff`/`reboot`/`halt`) Must be one of `poweroff`, `halt`, or `reboot`.
  - **message:** (string) Optional message to display to the user when the system is powering off or rebooting.
  - **timeout:** (integer) Time in seconds to wait for the cloud-init process to finish before executing shutdown. Default: `30`
  - **condition:** (string/boolean/array) Apply state change only if condition is met. May be boolean true (always met), false (never met), or a command string or list to be executed. For command formatting, see the documentation for `cc_runcmd`. If exit code is 0, condition is met, otherwise not. Default: `true`

Examples

Example 1:

```
#cloud-config
power_state:
  delay: now
  mode: poweroff
  message: Powering off
  timeout: 2
  condition: true
```

Example 2:

```
#cloud-config
power_state:
  delay: 30
  mode: reboot
  message: Rebooting machine
  condition: test -f /var/tmp/reboot_me
```

Puppet

Install, configure and start Puppet

Summary

This module handles Puppet installation and configuration. If the `puppet` key does not exist in global configuration, no action will be taken.

If a config entry for `puppet` is present, then by default the latest version of Puppet will be installed. If the `puppet` config key exists in the config archive, this module will attempt to start puppet even if no installation was performed.

The module also provides keys for configuring the new Puppet 4 paths and installing the `puppet` package from the puppetlabs repositories.

The keys are `package_name`, `conf_file`, `ssl_dir` and `csr_attributes_path`. If unset, their values will default to ones that work with Puppet 3.X, and with distributions that ship modified Puppet 4.X, that use the old paths.

Internal name: `cc_puppet`

Module frequency: once-per-instance

Supported distros: all

Activate only on keys: `puppet`

Config schema

- **puppet:** (object)
  - **install:** (boolean) Whether or not to install puppet. Setting to `false` will result in an error if puppet is not already present on the system. Default: `true`
  - **version:** (string) Optional version to pass to the installer script or package manager. If unset, the latest version from the repos will be installed.
  - **install_type:** (`packages`/`aio`) Valid values are `packages` and `aio`. Agent packages from the puppetlabs repositories can be installed by setting `aio`. Based on this setting, the default config/SSL/CSR paths will be adjusted accordingly. Default: `packages`
  - **collection:** (string) Puppet collection to install if `install_type` is `aio`. This can be set to one of `puppet` (rolling release), `puppet6`, `puppet7` (or their nightly counterparts) in order to install specific release streams.
  - **aio_install_url:** (string) If `install_type` is `aio`, change the url of the install script.
  - **cleanup:** (boolean) Whether to remove the puppetlabs repo after installation if `install_type` is `aio` Default: `true`
  - **conf_file:** (string) The path to the puppet config file. Default depends on `install_type`
  - **ssl_dir:** (string) The path to the puppet SSL directory. Default depends on `install_type`
  - **csr_attributes_path:** (string) The path to the puppet csr attributes file. Default depends on `install_type`
  - **package_name:** (string) Name of the package to install if `install_type` is `packages`. Default: `puppet`
  - **exec:** (boolean) Whether or not to run puppet after configuration finishes. A single manual run can be triggered by setting `exec` to `true`, and additional arguments can be passed to `puppet agent` via the `exec_args` key (by default the agent will execute with the `--test` flag). Default: `false`

- **exec_args:** (array of string) A list of arguments to pass to 'puppet agent' if 'exec' is true Default: `['--test']`
- **start_service:** (boolean) By default, the puppet service will be automatically enabled after installation and set to automatically start on boot. To override this in favor of manual puppet execution set `start_service` to `false`
- **conf:** (object) Every key present in the conf object will be added to puppet.conf. As such, section names should be one of: `main`, `server`, `agent` or `user` and keys should be valid puppet configuration options. The configuration is specified as a dictionary containing high-level `<section>` keys and lists of `<key>=<value>` pairs within each section. The `certname` key supports string substitutions for `%i` and `%f`, corresponding to the instance id and fqdn of the machine respectively.

  `ca_cert` is a special case. It won't be added to puppet.conf. It holds the puppetserver certificate in pem format. It should be a multi-line string (using the | YAML notation for multi-line strings).

  - **main:** (object)
  - **server:** (object)
  - **agent:** (object)
  - **user:** (object)
  - **ca_cert:** (string)
- **csr_attributes:** (object) create a `csr_attributes.yaml` file for CSR attributes and certificate extension requests. See https://puppet.com/docs/puppet/latest/config_file_csr_attributes.html
  - **custom_attributes:** (object)
  - **extension_requests:** (object)

Examples

Example 1:

```
#cloud-config
puppet:
  install: true
  version: "7.7.0"
  install_type: "aio"
  collection: "puppet7"
  aio_install_url: 'https://git.io/JBhoQ'
  cleanup: true
  conf_file: "/etc/puppet/puppet.conf"
  ssl_dir: "/var/lib/puppet/ssl"
  csr_attributes_path: "/etc/puppet/csr_attributes.yaml"
  exec: true
  exec_args: ['--test']
  conf:
    agent:
      server: "puppetserver.example.org"
      certname: "%i.%f"
    ca_cert: |
      -----BEGIN CERTIFICATE-----
      MIICCTCCAXKgAwIBAgIBATANBgkqhkiG9w0BAQUFADANMQswCQYDVQQDDAJjYTAe
      Fw0xMDAyMTUxNzI5MjFaFw0xNTAyMTQxNzI5MjFaMA0xCzAJBgNVBAMMAmNhMIGf
```

(continues on next page)

(continued from previous page)

```
      MA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCu7Q40sm47/E1Pf+r8AYb/V/FWGPgc
      b014OmNoX7dgCxTDvps/h8Vw555PdAFsW5+QhsGr31IJNI3kSYprFQcYf7A8tNWu
      1MASW2CfaEiOEi9F1R3R4Qlz4ix+iNoHiUDTjazw/tZwEdxaQXQVLwgTGRwVa+aA
      qbutJKi93MILLwIDAQABo3kwdzA4BglghkgBhvhCAQ0EKxYpUHVwcGV0IFJ1Ynkv
      T3BlblNTTCBHZW5lcmF0ZWQgQ2VydGlmaWNhdGUwDwYDVR0TAQH/BAUwAwEB/zAd
      BgNVHQ4EFgQUu4+jHB+GYE5Vxo+ol1OAhevspjAwCwYDVR0PBAQDAgEGMA0GCSqG
      SIb3DQEBBQUAA4GBAH/rxlUIjwNb3n7TXJcDJ6MMHUlwjr03BDJXKb34Ulndkpaf
      +GAlzPXWa7bO908M9I8RnPfvtKnteLbvgTK+h+zX1XCty+S2EQWk29i2AdoqOTxb
      hppiGMp0tT5Havu4aceCXiy2crVcudj3NFciy8X66SoECemW9UYDCb9T5D0d
      -----END CERTIFICATE-----
  csr_attributes:
    custom_attributes:
      1.2.840.113549.1.9.7: 342thbjkt82094y0uthhor289jnqthpc2290
    extension_requests:
      pp_uuid: ED803750-E3C7-44F5-BB08-41A04433FE2E
      pp_image_name: my_ami_image
      pp_preshared_key: 342thbjkt82094y0uthhor289jnqthpc2290
```

Example 2:

```
#cloud-config
puppet:
  install_type: "packages"
  package_name: "puppet"
  exec: false
```

Resizefs

Resize filesystem

Summary

Resize a filesystem to use all available space on partition. This module is useful along with `cc_growpart` and will ensure that if the root partition has been resized, the root filesystem will be resized along with it.

By default, `cc_resizefs` will resize the root partition and will block the boot process while the `resize` command is running.

Optionally, the resize operation can be performed in the background while cloud-init continues running modules. This can be enabled by setting `resize_rootfs` to `noblock`.

This module can be disabled altogether by setting `resize_rootfs` to `false`.

Internal name: `cc_resizefs`

Module frequency: always

Supported distros: all

Config schema

- **resize_rootfs:** (`True`/`False`/`noblock`) Whether to resize the root partition. `noblock` will resize in the background. Default: `true`

Examples

Example 1: Disable root filesystem resize operation.

```
#cloud-config
resize_rootfs: false
```

Example 2: Runs resize operation in the background.

```
#cloud-config
resize_rootfs: noblock
```

Resolv Conf

Configure `resolv.conf`

Summary

You should not use this module unless manually editing `/etc/resolv.conf` is the correct way to manage nameserver information on your operating system.

Many distros have moved away from manually editing `resolv.conf` so please verify that this is the preferred nameserver management method for your distro before using this module. Note that using *Network configuration* is preferred, rather than using this module, when possible.

This module is intended to manage `resolv.conf` in environments where early configuration of `resolv.conf` is necessary for further bootstrapping and/or where configuration management such as Puppet or Chef own DNS configuration.

When using a *Config drive* and a RHEL-like system, `resolv.conf` will also be managed automatically due to the available information provided for DNS servers in the *Networking config Version 2* format. For those who wish to have different settings, use this module.

For the `resolv_conf` section to be applied, `manage_resolv_conf` must be set `true`.

Note: For Red Hat with `sysconfig`, be sure to set `PEERDNS=no` for all DHCP-enabled NICs.

Internal name: `cc_resolv_conf`

Module frequency: once-per-instance

Supported distros: alpine, azurelinux, fedora, mariner, opensuse, opensuse-leap, opensuse-microos, opensuse-tumbleweed, photon, rhel, sle_hpc, sle-micro, sles, openeuler

Activate only on keys: `manage_resolv_conf`

Config schema

- **manage_resolv_conf:** (boolean) Whether to manage the resolv.conf file. `resolv_conf` block will be ignored unless this is set to `true`. Default: `false`
- **resolv_conf:** (object)
  - **nameservers:** (array) A list of nameservers to use to be added as `nameserver` lines
  - **searchdomains:** (array) A list of domains to be added `search` line
  - **domain:** (string) The domain to be added as `domain` line
  - **sortlist:** (array) A list of IP addresses to be added to `sortlist` line
  - **options:** (object) Key/value pairs of options to go under `options` heading. A unary option should be specified as `true`

Examples

Example 1:

```
#cloud-config
manage_resolv_conf: true
resolv_conf:
  domain: example.com
  nameservers: [8.8.8.8, 8.8.4.4]
  options: {rotate: true, timeout: 1}
  searchdomains: [foo.example.com, bar.example.com]
  sortlist: [10.0.0.1/255, 10.0.0.2]
```

Red Hat Subscription

Register Red Hat Enterprise Linux-based system

Summary

Register a Red Hat system, either by username and password **or** by activation and org.

Following a successful registration, you can:

- auto-attach subscriptions
- set the service level
- add subscriptions based on pool ID
- enable/disable yum repositories based on repo ID
- alter the `rhsm_baseurl` and `server-hostname` in `/etc/rhsm/rhs.conf`.

Internal name: `cc_rh_subscription`

Module frequency: once-per-instance

Supported distros: fedora, rhel, openeuler

Activate only on keys: `rh_subscription`

Config schema

- **rh_subscription:** (object)
  - **username:** (string) The username to use. Must be used with password. Should not be used with `activation-key` or `org`
  - **password:** (string) The password to use. Must be used with username. Should not be used with `activation-key` or `org`
  - **activation-key:** (string) The activation key to use. Must be used with `org`. Should not be used with `username` or `password`
  - **org:** (string/integer) The organization to use. Must be used with `activation-key`. Should not be used with `username` or `password`

    Deprecated in version 24.2.Use of type integer for this value is deprecated. Use a string instead.
  - **auto-attach:** (boolean) Whether to attach subscriptions automatically
  - **service-level:** (string) The service level to use when subscribing to RH repositories. `auto-attach` must be true for this to be used
  - **add-pool:** (array of string) A list of pools ids add to the subscription
  - **enable-repo:** (array of string) A list of repositories to enable
  - **disable-repo:** (array of string) A list of repositories to disable
  - **rhsm-baseurl:** (string) Sets the baseurl in `/etc/rhsm/rhsm.conf`
  - **server-hostname:** (string) Sets the serverurl in `/etc/rhsm/rhsm.conf`

Examples

Example 1:

```
#cloud-config
rh_subscription:
  username: joe@foo.bar
  ## Quote your password if it has symbols to be safe
  password: '1234abcd'
```

Example 2:

```
#cloud-config
rh_subscription:
  activation-key: foobar
  org: "ABC"
```

Example 3:

```
#cloud-config
rh_subscription:
  activation-key: foobar
  org: 12345
  auto-attach: true
  service-level: self-support
  add-pool:
```

(continues on next page)

(continued from previous page)

```
    - 1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a
    - 2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b
  enable-repo:
    - repo-id-to-enable
    - other-repo-id-to-enable
  disable-repo:
    - repo-id-to-disable
    - other-repo-id-to-disable
  # Alter the baseurl in /etc/rhsm/rhsm.conf
  rhsm-baseurl: http://url
  # Alter the server hostname in /etc/rhsm/rhsm.conf
  server-hostname: foo.bar.com
```

Rsyslog

Configure system logging via rsyslog

Summary

This module configures remote system logging using rsyslog.

Configuration for remote servers can be specified in `configs`, but for convenience it can be specified as key-value pairs in `remotes`.

This module can install rsyslog if not already present on the system using the `install_rsyslog`, `packages`, and `check_exe` options. Installation may not work on systems where this module runs before networking is up.

Note: On BSD, cloud-init will attempt to disable and stop the base system syslogd. This may fail on a first run. We recommend creating images with `service syslogd disable`.

Internal name: `cc_rsyslog`

Module frequency: once-per-instance

Supported distros: all

Activate only on keys: `rsyslog`

Config schema

- **rsyslog:** (object)
  - **config_dir:** (string) The directory where rsyslog configuration files will be written. Default: `/etc/rsyslog.d`
  - **config_filename:** (string) The name of the rsyslog configuration file. Default: `20-cloud-config.conf`
  - **configs:** (array of string/object) Each entry in `configs` is either a string or an object. Each config entry contains a configuration string and a file to write it to. For config entries that are an object, `filename` sets the target filename and `content` specifies the config string to write. For config entries that are only a string, the string is used as the config string to write. If the filename to write the config to is not specified, the value of the `config_filename` key is used. A file with the selected filename will be written inside the directory specified by `config_dir`.

- **remotes:** (object) Each key is the name for an rsyslog remote entry. Each value holds the contents of the remote config for rsyslog. The config consists of the following parts:
  - filter for log messages (defaults to `*.*`)
  - optional leading `@` or `@@`, indicating udp and tcp respectively (defaults to `@`, for udp)
  - ipv4 or ipv6 hostname or address. ipv6 addresses must be in `[::1]` format, (e.g. `@[fd00::1]:514`)
  - optional port number (defaults to `514`)

  This module will provide sane defaults for any part of the remote entry that is not specified, so in most cases remote hosts can be specified just using `<name>: <address>`.
- **service_reload_command:** (`auto`/array of string) The command to use to reload the rsyslog service after the config has been updated. If this is set to `auto`, then an appropriate command for the distro will be used. This is the default behavior. To manually set the command, use a list of command args (e.g. `[systemctl, restart, rsyslog]`).
- **install_rsyslog:** (boolean) Install rsyslog. Default: `false`
- **check_exe:** (string) The executable name for the rsyslog daemon.

  For example, `rsyslogd`, or `/opt/sbin/rsyslogd` if the rsyslog binary is in an unusual path. This is only used if `install_rsyslog` is `true`. Default: `rsyslogd`
- **packages:** (array of string) List of packages needed to be installed for rsyslog. This is only used if `install_rsyslog` is `true`. Default: `[rsyslog]`

Examples

Example 1:

```
#cloud-config
rsyslog:
  remotes: {juju: 10.0.4.1, maas: 192.168.1.1}
  service_reload_command: auto
```

Example 2:

```
#cloud-config
rsyslog:
  config_dir: /opt/etc/rsyslog.d
  config_filename: 99-late-cloud-config.conf
  configs:
  - '*.* @@192.158.1.1'
  - {content: '*.*   @@192.0.2.1:10514', filename: 01-example.conf}
  - {content: '*.*   @@syslogd.example.com

      '}
  remotes: {juju: 10.0.4.1, maas: 192.168.1.1}
  service_reload_command: [your, syslog, restart, command]
```

Example 3: Default (no) configuration with package installation on FreeBSD.

```
#cloud-config
rsyslog:
  check_exe: rsyslogd
```

(continues on next page)

(continued from previous page)

```
config_dir: /usr/local/etc/rsyslog.d
install_rsyslog: true
packages: [rsyslogd]
```

Runcmd

Run arbitrary commands

Summary

Run arbitrary commands at a `rc.local`-like time-frame with output to the console. Each item can be either a list or a string. The item type affects how it is executed:

- If the item is a string, it will be interpreted by `sh`.
- If the item is a list, the items will be executed as if passed to `execve(3)` (with the first argument as the command).

The `runcmd` module only writes the script to be run later. The module that actually runs the script is `scripts_user` in the *Final boot stage*.

Note: All commands must be proper YAML, so you must quote any characters YAML would eat (":" can be problematic).

Note: When writing files, do not use `/tmp` dir as it races with `systemd-tmpfiles-clean` (LP: #1707222). Use `/run/somedir` instead.

Internal name: `cc_runcmd`

Module frequency: once-per-instance

Supported distros: all

Activate only on keys: `runcmd`

Config schema

- **runcmd:** (array of array of string/string/null)

Examples

Example 1:

```
#cloud-config
runcmd:
- [ls, -l, /]
- [sh, -xc, 'echo $(date) '': hello world!''']
- [sh, -c, echo "=========hello world'========="]
- ls -l /root
```

Salt Minion

Set up and run Salt Minion

Summary

This module installs, configures and starts Salt Minion. If the `salt_minion` key is present in the config parts, then Salt Minion will be installed and started.

Configuration for Salt Minion can be specified in the `conf` key under `salt_minion`. Any config values present there will be assigned in `/etc/salt/minion`. The public and private keys to use for Salt Minion can be specified with `public_key` and `private_key` respectively.

If you have a custom package name, service name, or config directory, you can specify them with `pkg_name`, `service_name`, and `config_dir` respectively.

Salt keys can be manually generated by `salt-key --gen-keys=GEN_KEYS`, where `GEN_KEYS` is the name of the keypair, e.g. ''minion". The keypair will be copied to `/etc/salt/pki` on the Minion instance.

Internal name: `cc_salt_minion`

Module frequency: once-per-instance

Supported distros: all

Activate only on keys: `salt_minion`

Config schema

- **salt_minion:** (object)
  - **pkg_name:** (string) Package name to install. Default: `salt-minion`
  - **service_name:** (string) Service name to enable. Default: `salt-minion`
  - **config_dir:** (string) Directory to write config files to. Default: `/etc/salt`
  - **conf:** (object) Configuration to be written to *config_dir*/minion
  - **grains:** (object) Configuration to be written to *config_dir*/grains
  - **public_key:** (string) Public key to be used by the salt minion
  - **private_key:** (string) Private key to be used by salt minion
  - **pki_dir:** (string) Directory to write key files. Default: *config_dir*/pki/minion

Examples

Example 1:

```
#cloud-config
salt_minion:
  conf:
    file_client: local
    fileserver_backend: [gitfs]
    gitfs_remotes: ['https://github.com/_user_/_repo_.git']
    master: salt.example.com
```

(continues on next page)

(continued from previous page)

```
config_dir: /etc/salt
grains:
  role: [web]
pkg_name: salt-minion
pki_dir: /etc/salt/pki/minion
private_key: '------BEGIN PRIVATE KEY------

  <key data>

  ------END PRIVATE KEY-------

  '
public_key: '------BEGIN PUBLIC KEY-------

  <key data>

  ------END PUBLIC KEY-------

  '
service_name: salt-minion
```

Scripts Per Boot

Run per-boot scripts

Summary

Any scripts in the `scripts/per-boot` directory on the datasource will be run every time the system boots. Scripts will be run in alphabetical order. This module does not accept any config keys.

Internal name: `cc_scripts_per_boot`

Module frequency: always

Supported distros: all

Config schema

No schema definitions for this module

Examples

No examples for this module

Scripts Per Instance

Run per-instance scripts

Summary

Any scripts in the `scripts/per-instance` directory on the datasource will be run when a new instance is first booted. Scripts will be run in alphabetical order. This module does not accept any config keys.

Some cloud platforms change `instance-id` if a significant change was made to the system. As a result, per-instance scripts will run again.

Internal name: `cc_scripts_per_instance`

Module frequency: once-per-instance

Supported distros: all

Config schema

No schema definitions for this module

Examples

No examples for this module

Scripts Per Once

Run one-time scripts

Summary

Any scripts in the `scripts/per-once` directory on the datasource will be run only once. Changes to the instance will not force a re-run. The only way to re-run these scripts is to run the `clean` subcommand and reboot. Scripts will be run in alphabetical order. This module does not accept any config keys.

Internal name: `cc_scripts_per_once`

Module frequency: once

Supported distros: all

Config schema

No schema definitions for this module

Examples

No examples for this module

Scripts User

Run user scripts

Summary

This module runs all user scripts present in the `scripts` directory in the instance configuration. Any cloud-config parts with a `#!` will be treated as a script and run. Scripts specified as cloud-config parts will be run in the order they are specified in the configuration. This module does not accept any config keys.

Internal name: `cc_scripts_user`

Module frequency: once-per-instance

Supported distros: all

Config schema

No schema definitions for this module

Examples

No examples for this module

Scripts Vendor

Run vendor scripts

Summary

On select Datasources, vendors have a channel for the consumption of all supported user data types via a special channel called vendor data. Any scripts in the `scripts/vendor` directory in the datasource will be run when a new instance is first booted. Scripts will be run in alphabetical order. This module allows control over the execution of vendor data.

Internal name: `cc_scripts_vendor`

Module frequency: once-per-instance

Supported distros: all

Config schema

- **vendor_data:** (object)
  - **enabled:** (boolean/string) Whether vendor data is enabled or not. Default: `true`

    Deprecated in version 22.3.Use of type string for this value is deprecated. Use a boolean instead.
  - **prefix:** ([‘array’, ‘string’] of string/integer) The command to run before any vendor scripts. Its primary use case is for profiling a script, not to prevent its run

Examples

Example 1:

```
#cloud-config
vendor_data: {enabled: true, prefix: /usr/bin/ltrace}
```

Example 2:

```
#cloud-config
vendor_data:
  enabled: true
  prefix: [timeout, 30]
```

Example 3: Vendor data will not be processed.

```
#cloud-config
vendor_data: {enabled: false}
```

Seed Random

Provide random seed data

Summary

All cloud instances started from the same image will produce similar data when they are first booted as they are all starting with the same seed for the kernel’s entropy keyring. To avoid this, random seed data can be provided to the instance, either as a string or by specifying a command to run to generate the data.

Configuration for this module is under the `random_seed` config key. If the cloud provides its own random seed data, it will be appended to `data` before it is written to `file`.

If the `command` key is specified, the given command will be executed. This will happen after `file` has been populated. That command’s environment will contain the value of the `file` key as `RANDOM_SEED_FILE`. If a command is specified that cannot be run, no error will be reported unless `command_required` is set to `true`.

Internal name: `cc_seed_random`

Module frequency: once-per-instance

Supported distros: all

Config schema

- **random_seed:** (object)
  - **file:** (string) File to write random data to. Default: `/dev/urandom`
  - **data:** (string) This data will be written to `file` before data from the datasource. When using a multi-line value or specifying binary data, be sure to follow YAML syntax and use the `|` and `!binary` YAML format specifiers when appropriate
  - **encoding:** (`raw`/`base64`/`b64`/`gzip`/`gz`) Used to decode `data` provided. Allowed values are `raw`, `base64`, `b64`, `gzip`, or `gz`. Default: `raw`
  - **command:** (array of string) Execute this command to seed random. The command will have RANDOM_SEED_FILE in its environment set to the value of `file` above.
  - **command_required:** (boolean) If true, and `command` is not available to be run then an exception is raised and cloud-init will record failure. Otherwise, only debug error is mentioned. Default: `false`

Examples

Example 1:

```
#cloud-config
random_seed:
  command: [sh, -c, dd if=/dev/urandom of=$RANDOM_SEED_FILE]
  command_required: true
  data: my random string
  encoding: raw
  file: /dev/urandom
```

Example 2: Use `pollinate` to gather data from a remote entropy server and write it to `/dev/urandom`:

```
#cloud-config
random_seed:
  command: [pollinate, '--server=http://local.pollinate.server']
  command_required: true
  file: /dev/urandom
```

Set Hostname

Set hostname and FQDN

Summary

This module handles setting the system hostname and fully qualified domain name (FQDN). If `preserve_hostname` is set, then the hostname will not be altered.

A hostname and FQDN can be provided by specifying a full domain name under the `fqdn` key. Alternatively, a hostname can be specified using the `hostname` key, and the FQDN of the cloud will be used. If a FQDN is specified with the `hostname` key, it will be handled properly, although it is better to use the `fqdn` config key. If both `fqdn` and `hostname` are set, then `prefer_fqdn_over_hostname` will force use of FQDN in all distros when true, and when false it will force the short hostname. Otherwise, the hostname to use is distro-dependent.

Note: Cloud-init performs no hostname input validation before sending the hostname to distro-specific tools, and most tools will not accept a trailing dot on the FQDN.

This module will run in the init-local stage before networking is configured if the hostname is set by metadata or user data on the local system.

This will occur on datasources like NoCloud and OVF where metadata and user data are available locally. This ensures that the desired hostname is applied before any DHCP requests are performed on these platforms where dynamic DNS is based on initial hostname.

Internal name: `cc_set_hostname`

Module frequency: once-per-instance

Supported distros: all

Config schema

- **preserve_hostname:** (boolean) If true, the hostname will not be changed. Default: `false`
- **hostname:** (string) The hostname to set
- **fqdn:** (string) The fully qualified domain name to set
- **prefer_fqdn_over_hostname:** (boolean) If true, the fqdn will be used if it is set. If false, the hostname will be used. If unset, the result is distro-dependent
- **create_hostname_file:** (boolean) If `false`, the hostname file (e.g. /etc/hostname) will not be created if it does not exist. On systems that use systemd, setting create_hostname_file to `false` will set the hostname transiently. If `true`, the hostname file will always be created and the hostname will be set statically on systemd systems. Default: `true`

Examples

Example 1:

```
#cloud-config
preserve_hostname: true
```

Example 2:

```
#cloud-config
hostname: myhost
create_hostname_file: true
fqdn: myhost.example.com
prefer_fqdn_over_hostname: true
```

Example 3: On a machine without an `/etc/hostname` file, don't create it. In most clouds, this will result in a DHCP-configured hostname provided by the cloud.

```
#cloud-config
create_hostname_file: false
```

Set Passwords

Set user passwords and enable/disable SSH password auth

Summary

This module consumes three top-level config keys: `ssh_pwauth`, `chpasswd` and `password`.

The `ssh_pwauth` config key determines whether or not sshd will be configured to accept password authentication.

The `chpasswd` config key accepts a dictionary containing either (or both) of `users` and `expire`.

- The `users` key is used to assign a password to a corresponding pre-existing user.
- The `expire` key is used to set whether to expire all user passwords specified by this module, such that a password will need to be reset on the user's next login.

Note: Prior to cloud-init 22.3, the `expire` key only applies to plain text (including `RANDOM`) passwords. Post-22.3, the `expire` key applies to both plain text and hashed passwords.

The `password` config key is used to set the default user's password. It is ignored if the `chpasswd` `users` is used. Note that the `list` keyword is deprecated in favor of `users`.

Internal name: `cc_set_passwords`

Module frequency: once-per-instance

Supported distros: all

Config schema

- **ssh_pwauth:** (boolean/string) Sets whether or not to accept password authentication. `true` will enable password auth. `false` will disable. Default: leave the value unchanged. In order for this config to be applied, SSH may need to be restarted. On systemd systems, this restart will only happen if the SSH service has already been started. On non-systemd systems, a restart will be attempted regardless of the service state.

  Changed in version 22.3.Use of non-boolean values for this field is deprecated.

- **chpasswd:** (object)
  - **expire:** (boolean) Whether to expire all user passwords such that a password will need to be reset on the user's next login. Default: `true`
  - **users:** (array of object) This key represents a list of existing users to set passwords for. Each item under users contains the following required keys: `name` and `password` or in the case of a randomly generated password, `name` and `type`. The `type` key has a default value of `hash`, and may alternatively be set to `text` or `RANDOM`. Randomly generated passwords may be insecure, use at your own risk. Each object in **users** list supports the following keys:
  - **list:** (string/array of string)

    Deprecated in version 22.2.Use ``users`` instead.

- **password:** (string) Set the default user's password. Ignored if `chpasswd` `list` is used

Examples

Example 1: Set a default password, to be changed at first login.

```
#cloud-config
{password: password1, ssh_pwauth: true}
```

Example 2:

- Disable SSH password authentication.
- Don't require users to change their passwords on next login.
- Set the password for user1 to be 'password1' (OS does hashing).
- Set the password for user2 to a pre-hashed password.
- Set the password for user3 to be a randomly generated password, which

  will be written to the system console.

```
#cloud-config
chpasswd:
  expire: false
  users:
  - {name: user1, password: password1, type: text}
  - {name: user2, password: $6$rounds=4096$5DJ8a9WMTEzIo5J4
↪$Yms6imfeBvf3Yfu84mQBerh18l7OR1Wm1BJXZqFSpJ6BVas0AYJqIjP7czkOaAZHZi1kxQ5Y1IhgWN8K9NgxR1}
↪
  - {name: user3, type: RANDOM}
ssh_pwauth: false
```

Snap

Install, configure and manage snapd and snap packages

Summary

This module provides a simple configuration namespace in cloud-init for setting up snapd and installing snaps.

Both `assertions` and `commands` values can be either a dictionary or a list. If these configs are provided as a dictionary, the keys are only used to order the execution of the assertions or commands and the dictionary is merged with any vendor data the snap configuration provided. If a list is provided by the user instead of a dict, any vendor data snap configuration is ignored.

The `assertions` configuration option is a dictionary or list of properly-signed snap assertions, which will run before any snap commands. They will be added to snapd's `assertion` database by invoking `snap ack <aggregate_assertion_file>`.

Snap `commands` is a dictionary or list of individual snap commands to run on the target system. These commands can be used to create snap users, install snaps, and provide snap configuration.

Note: If 'side-loading' private/unpublished snaps on an instance, it is best to create a snap seed directory and `seed.yaml` manifest in `/var/lib/snapd/seed/` which snapd automatically installs on startup.

Internal name: `cc_snap`

Module frequency: once-per-instance

Supported distros: ubuntu

Activate only on keys: `snap`

Config schema

- **snap:** (object)
  - **assertions:** ([‘object’, ‘array’] of string) Properly-signed snap assertions which will run before and snap `commands`.
  - **commands:** ([‘object’, ‘array’] of string/array of string) Snap commands to run on the target system

Examples

Example 1:

```
#cloud-config
snap:
  assertions:
    00: |
      signed_assertion_blob_here
    02: |
      signed_assertion_blob_here
  commands:
    00: snap create-user --sudoer --known <snap-user>@mydomain.com
    01: snap install canonical-livepatch
    02: canonical-livepatch enable <AUTH_TOKEN>
```

Example 2: For convenience, the `snap` command can be omitted when specifying commands as a list - `snap` will be automatically prepended. The following commands are all equivalent:

```
#cloud-config
snap:
  commands:
    0: [install, vlc]
    1: [snap, install, vlc]
    2: snap install vlc
    3: snap install vlc
```

Example 3: You can use a list of commands.

```
#cloud-config
snap:
  assertions:
    - signed_assertion_blob_here
    - |
      signed_assertion_blob_here
```

Example 4: You can also use a list of assertions.

```
#cloud-config
snap:
  assertions:
    - signed_assertion_blob_here
    - |
      signed_assertion_blob_here
```

Spacewalk

Install and configure spacewalk

Summary

This module installs Spacewalk and applies basic configuration. If the Spacewalk config key is present, Spacewalk will be installed. The server to connect to after installation must be provided in the `server` in Spacewalk configuration. A proxy to connect through and an activation key may optionally be specified.

For more details about spacewalk see the Fedora documentation.

Internal name: `cc_spacewalk`

Module frequency: once-per-instance

Supported distros: rhel, fedora, openeuler

Activate only on keys: `spacewalk`

Config schema

- **spacewalk:** (object)
  - **server:** (string) The Spacewalk server to use
  - **proxy:** (string) The proxy to use when connecting to Spacewalk
  - **activation_key:** (string) The activation key to use when registering with Spacewalk

Examples

Example 1:

```
#cloud-config
spacewalk: {activation_key: <key>, proxy: <proxy host>, server: <url>}
```

SSH

Configure SSH and SSH keys

Summary

This module handles most configuration for SSH, and for both host and authorized SSH keys.

Authorized keys

Authorized keys are a list of public SSH keys that are allowed to connect to a user account on a system. They are stored in `.ssh/authorized_keys` in that account's home directory. Authorized keys for the default user defined in `users` can be specified using `ssh_authorized_keys`. Keys should be specified as a list of public keys.

Note: See the `cc_set_passwords` module documentation to enable/disable SSH password authentication.

Root login can be enabled/disabled using the `disable_root` config key. Root login options can be manually specified with `disable_root_opts`.

Supported public key types for the ssh_authorized_keys are:

- rsa
- ecdsa
- ed25519
- ecdsa-sha2-nistp256-cert-v01@openssh.com
- ecdsa-sha2-nistp256
- ecdsa-sha2-nistp384-cert-v01@openssh.com
- ecdsa-sha2-nistp384
- ecdsa-sha2-nistp521-cert-v01@openssh.com
- ecdsa-sha2-nistp521
- sk-ecdsa-sha2-nistp256-cert-v01@openssh.com
- sk-ecdsa-sha2-nistp256@openssh.com
- sk-ssh-ed25519-cert-v01@openssh.com
- sk-ssh-ed25519@openssh.com
- ssh-ed25519-cert-v01@openssh.com
- ssh-ed25519
- ssh-rsa-cert-v01@openssh.com
- ssh-rsa
- ssh-xmss-cert-v01@openssh.com
- ssh-xmss@openssh.com

Note: This list has been filtered out from the supported key types of OpenSSH source, where the `sigonly` keys are removed. See `ssh_util` for more information.

rsa, ecdsa and ed25519 are added for legacy, as they are valid public keys in some older distros. They may be removed in the future when support for the older distros is dropped.

Host keys

Host keys are for authenticating a specific instance. Many images have default host SSH keys, which can be removed using ssh_deletekeys.

Host keys can be added using the ssh_keys configuration key.

When host keys are generated the output of the ssh-keygen command(s) can be displayed on the console using the ssh_quiet_keygen configuration key.

Note: When specifying private host keys in cloud-config, take care to ensure that communication between the data source and the instance is secure.

If no host keys are specified using ssh_keys, then keys will be generated using ssh-keygen. By default, one public/private pair of each supported host key type will be generated. The key types to generate can be specified using the ssh_genkeytypes config flag, which accepts a list of host key types to use. For each host key type for which this module has been instructed to create a keypair, if a key of the same type is already present on the system (i.e. if ssh_deletekeys was set to false), no key will be generated.

Supported host key types for the ssh_keys and the ssh_genkeytypes config flags are:

- ecdsa
- ed25519
- rsa

Unsupported host key types for the ssh_keys and the ssh_genkeytypes config flags are:

- ecdsa-sk
- ed25519-sk

Internal name: cc_ssh

Module frequency: once-per-instance

Supported distros: all

Config schema

- **ssh_keys:** (object) A dictionary entries for the public and private host keys of each desired key type. Entries in the ssh_keys config dict should have keys in the format <key type>_private, <key type>_public, and, optionally, <key type>_certificate, e.g. rsa_private: <key>, rsa_public: <key>, and rsa_certificate: <key>. Not all key types have to be specified, ones left unspecified will not be used. If this config option is used, then separate keys will not be automatically generated. In order to specify multi-line private host keys and certificates, use YAML multi-line syntax. **Note:** Your ssh keys might possibly be visible to unprivileged users on your system, depending on your cloud's security model.
  - **<key_type>:** (string)
- **ssh_authorized_keys:** (array of string) The SSH public keys to add .ssh/authorized_keys in the default user's home directory
- **ssh_deletekeys:** (boolean) Remove host SSH keys. This prevents re-use of a private host key from an image with default host SSH keys. Default: true

- **ssh_genkeytypes:** (array of `ecdsa/ed25519/rsa`) The SSH key types to generate. Default: `[rsa, ecdsa, ed25519]`
- **disable_root:** (boolean) Disable root login. Default: `true`
- **disable_root_opts:** (string) Disable root login options. If `disable_root_opts` is specified and contains the string `$USER`, it will be replaced with the username of the default user. Default: `no-port-forwarding, no-agent-forwarding,no-X11-forwarding,command="echo 'Please login as the user \ "$USER\" rather than the user \"$DISABLE_USER\".';echo;sleep 10;exit 142"`
- **allow_public_ssh_keys:** (boolean) If `true`, will import the public SSH keys from the datasource's metadata to the user's `.ssh/authorized_keys` file. Default: `true`
- **ssh_quiet_keygen:** (boolean) If `true`, will suppress the output of key generation to the console. Default: `false`
- **ssh_publish_hostkeys:** (object)
  - **enabled:** (boolean) If true, will read host keys from `/etc/ssh/*.pub` and publish them to the datasource (if supported). Default: `true`
  - **blacklist:** (array of string) The SSH key types to ignore when publishing. Default: `[]` to publish all SSH key types

Examples

Example 1:

```
#cloud-config
allow_public_ssh_keys: true
disable_root: true
disable_root_opts: no-port-forwarding,no-agent-forwarding,no-X11-forwarding
ssh_authorized_keys: [ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAGEA3FSyQwBI6Z+nCSjUU ..., ssh-rsa
    AAAAB3NzaC1yc2EAAAABIwAAAQEA3I7VUf2l5gSn5uavROsc5HRDpZ ...]
ssh_deletekeys: true
ssh_genkeytypes: [rsa, ecdsa, ed25519]
ssh_keys: {rsa_certificate: 'ssh-rsa-cert-v01@openssh.com AAAAIHNzaC1lZDI1NTE5LWNlcnQt
    ...

    ', rsa_private: '-----BEGIN RSA PRIVATE KEY-----

    MIIBxwIBAAJhAKD0YSHy73nUgysO13XsJmd4fHiFyQ+00R7VVu2iV9Qco

    ...

    -----END RSA PRIVATE KEY-----

    ', rsa_public: ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAGEAoPRhIfLvedSDKw7Xd ...}
ssh_publish_hostkeys:
  blacklist: [rsa]
  enabled: true
ssh_quiet_keygen: true
```

SSH AuthKey Fingerprints

Log fingerprints of user SSH keys

Summary

Write fingerprints of authorized keys for each user to log. This is enabled by default, but can be disabled using `no_ssh_fingerprints`. The hash type for the keys can be specified, but defaults to `sha256`.

Internal name: `cc_ssh_authkey_fingerprints`

Module frequency: once-per-instance

Supported distros: all

Config schema

- **no_ssh_fingerprints:** (boolean) If true, SSH fingerprints will not be written. Default: `false`
- **authkey_hash:** (string) The hash type to use when generating SSH fingerprints. Default: `sha256`

Examples

Example 1:

```
#cloud-config
no_ssh_fingerprints: true
```

Example 2:

```
#cloud-config
authkey_hash: sha512
```

SSH Import ID

Import SSH ID

Summary

This module imports SSH keys from either a public keyserver (usually Launchpad), or GitHub, using `ssh-import-id`. Keys are referenced by the username they are associated with on the keyserver. The keyserver can be specified by prepending either `lp:` for Launchpad or `gh:` for GitHub to the username.

Internal name: `cc_ssh_import_id`

Module frequency: once-per-instance

Supported distros: alpine, cos, debian, ubuntu

Config schema

- **ssh_import_id:** (array of string)

Examples

Example 1:

```
#cloud-config
ssh_import_id: [user, 'gh:user', 'lp:user']
```

Timezone

Set the system timezone

Summary

Sets the system timezone based on the value provided.

Internal name: `cc_timezone`

Module frequency: once-per-instance

Supported distros: all

Activate only on keys: `timezone`

Config schema

- **timezone:** (string) The timezone to use as represented in /usr/share/zoneinfo

Examples

Example 1:

```
#cloud-config
timezone: US/Eastern
```

Ubuntu Drivers

Interact with third party drivers in Ubuntu

Summary

This module interacts with the `ubuntu-drivers` command to install third party driver packages.

Internal name: `cc_ubuntu_drivers`

Module frequency: once-per-instance

Supported distros: ubuntu

Activate only on keys: `drivers`

Config schema

- **drivers:** (object)
  - **nvidia:** (object)
    - **license-accepted:** (boolean) Do you accept the NVIDIA driver license?
    - **version:** (string) The version of the driver to install (e.g. "390", "410"). Default: latest version.

Examples

Example 1:

```
#cloud-config
drivers:
  nvidia: {license-accepted: true}
```

Ubuntu Pro

Configure Ubuntu Pro support services

Summary

Attach machine to an existing Ubuntu Pro support contract and enable or disable support services such as Livepatch, ESM, FIPS and FIPS Updates.

When attaching a machine to Ubuntu Pro, one can also specify services to enable. When the `enable` list is present, only named services will be activated. If the `enable` list is not present, the contract's default services will be enabled.

On Pro instances, when `ubuntu_pro` config is provided to cloud-init, Pro's auto-attach feature will be disabled and cloud-init will perform the Pro auto-attach, ignoring the `token` key. The `enable` and `enable_beta` values will strictly determine what services will be enabled, ignoring contract defaults.

Note that when enabling FIPS or FIPS updates you will need to schedule a reboot to ensure the machine is running the FIPS-compliant kernel. See the Power State Change module for information on how to configure cloud-init to perform this reboot.

Internal name: `cc_ubuntu_pro`

Module frequency: once-per-instance

Supported distros: ubuntu

Activate only on keys: `ubuntu_pro`, `ubuntu-advantage`, `ubuntu_advantage`

Config schema

- **ubuntu_pro:** (object)
  - **enable:** (array of string) Optional list of Ubuntu Pro services to enable. Any of: cc-eal, cis, esm-infra, fips, fips-updates, livepatch. By default, a given contract token will automatically enable a number of services, use this list to supplement which services should additionally be enabled. Any service unavailable on a given Ubuntu release or unentitled in a given contract will remain disabled. In Ubuntu Pro instances, if this list is given, then only those services will be enabled, ignoring contract defaults. Passing beta services here will cause an error.
  - **enable_beta:** (array of string) Optional list of Ubuntu Pro beta services to enable. By default, a given contract token will automatically enable a number of services, use this list to supplement which services should additionally be enabled. Any service unavailable on a given Ubuntu release or unentitled in a given contract will remain disabled. In Ubuntu Pro instances, if this list is given, then only those services will be enabled, ignoring contract defaults.
  - **token:** (string) Contract token obtained from https://ubuntu.com/pro to attach. Required for non-Pro instances.
  - **features:** (object) Ubuntu Pro features.
    - **disable_auto_attach:** (boolean) Optional boolean for controlling if ua-auto-attach.service (in Ubuntu Pro instances) will be attempted each boot. Default: `false`
  - **config:** (object) Configuration settings or override Ubuntu Pro config.
    - **http_proxy:** (string/null) Ubuntu Pro HTTP Proxy URL or null to unset.
    - **https_proxy:** (string/null) Ubuntu Pro HTTPS Proxy URL or null to unset.
    - **global_apt_http_proxy:** (string/null) HTTP Proxy URL used for all APT repositories on a system or null to unset. Stored at `/etc/apt/apt.conf.d/90ubuntu-advantage-aptproxy`
    - **global_apt_https_proxy:** (string/null) HTTPS Proxy URL used for all APT repositories on a system or null to unset. Stored at `/etc/apt/apt.conf.d/90ubuntu-advantage-aptproxy`
    - **ua_apt_http_proxy:** (string/null) HTTP Proxy URL used only for Ubuntu Pro APT repositories or null to unset. Stored at `/etc/apt/apt.conf.d/90ubuntu-advantage-aptproxy`
    - **ua_apt_https_proxy:** (string/null) HTTPS Proxy URL used only for Ubuntu Pro APT repositories or null to unset. Stored at `/etc/apt/apt.conf.d/90ubuntu-advantage-aptproxy`
- **ubuntu_advantage:** (object)

  Deprecated in version 24.1.Use ``ubuntu_pro`` instead.
  - **enable:** (array of string) Optional list of Ubuntu Pro services to enable. Any of: cc-eal, cis, esm-infra, fips, fips-updates, livepatch. By default, a given contract token will automatically enable a number of services, use this list to supplement which services should additionally be enabled. Any service unavailable on a given Ubuntu release or unentitled in a given contract will remain disabled. In Ubuntu Pro instances, if this list is given, then only those services will be enabled, ignoring contract defaults. Passing beta services here will cause an error.
  - **enable_beta:** (array of string) Optional list of Ubuntu Pro beta services to enable. By default, a given contract token will automatically enable a number of services, use this list to supplement which services should additionally be enabled. Any service unavailable on a given Ubuntu release or unentitled in a given contract will remain disabled. In Ubuntu Pro instances, if this list is given, then only those services will be enabled, ignoring contract defaults.
  - **token:** (string) Contract token obtained from https://ubuntu.com/pro to attach. Required for non-Pro instances.

- **features:** (object) Ubuntu Pro features.
  - **disable_auto_attach:** (boolean) Optional boolean for controlling if ua-auto-attach.service (in Ubuntu Pro instances) will be attempted each boot. Default: `false`
- **config:** (object) Configuration settings or override Ubuntu Pro config.
  - **http_proxy:** (string/null) Ubuntu Pro HTTP Proxy URL or null to unset.
  - **https_proxy:** (string/null) Ubuntu Pro HTTPS Proxy URL or null to unset.
  - **global_apt_http_proxy:** (string/null) HTTP Proxy URL used for all APT repositories on a system or null to unset. Stored at `/etc/apt/apt.conf.d/90ubuntu-advantage-aptproxy`
  - **global_apt_https_proxy:** (string/null) HTTPS Proxy URL used for all APT repositories on a system or null to unset. Stored at `/etc/apt/apt.conf.d/90ubuntu-advantage-aptproxy`
  - **ua_apt_http_proxy:** (string/null) HTTP Proxy URL used only for Ubuntu Pro APT repositories or null to unset. Stored at `/etc/apt/apt.conf.d/90ubuntu-advantage-aptproxy`
  - **ua_apt_https_proxy:** (string/null) HTTPS Proxy URL used only for Ubuntu Pro APT repositories or null to unset. Stored at `/etc/apt/apt.conf.d/90ubuntu-advantage-aptproxy`

Examples

Example 1: Attach the machine to an Ubuntu Pro support contract with a Pro contract token obtained from https://ubuntu.com/pro.

```
#cloud-config
ubuntu_pro: {token: <ubuntu_pro_token>}
```

Example 2: Attach the machine to an Ubuntu Pro support contract, enabling only FIPS and ESM services. Services will only be enabled if the environment supports that service. Otherwise, warnings will be logged for incompatible services.

```
#cloud-config
ubuntu_pro:
  enable: [fips, esm]
  token: <ubuntu_pro_token>
```

Example 3: Attach the machine to an Ubuntu Pro support contract and enable the FIPS service. Perform a reboot once cloud-init has completed.

```
#cloud-config
power_state: {mode: reboot}
ubuntu_pro:
  enable: [fips]
  token: <ubuntu_pro_token>
```

Example 4: Set a HTTP(s) proxy before attaching the machine to an Ubuntu Pro support contract and enabling the FIPS service.

```
#cloud-config
ubuntu_pro:
  token: <ubuntu_pro_token>
  config:
    http_proxy: 'http://some-proxy:8088'
```

(continues on next page)

(continued from previous page)

```
    https_proxy: 'https://some-proxy:8088'
    global_apt_https_proxy: 'https://some-global-apt-proxy:8088/'
    global_apt_http_proxy: 'http://some-global-apt-proxy:8088/'
    ua_apt_http_proxy: 'http://10.0.10.10:3128'
    ua_apt_https_proxy: 'https://10.0.10.10:3128'
  enable:
  - fips
```

Example 5: On Ubuntu Pro instances, auto-attach but don't enable any Pro services.

```
#cloud-config
ubuntu_pro:
  enable: []
  enable_beta: []
```

Example 6: Enable ESM and beta Real-time Ubuntu services in Ubuntu Pro instances.

```
#cloud-config
ubuntu_pro:
  enable: [esm]
  enable_beta: [realtime-kernel]
```

Example 7: Disable auto-attach in Ubuntu Pro instances.

```
#cloud-config
ubuntu_pro:
  features: {disable_auto_attach: true}
```

Update Etc Hosts

Update the hosts file (usually `/etc/hosts`)

Summary

This module will update the contents of the local hosts database (hosts file, usually `/etc/hosts`) based on the hostname/FQDN specified in config. Management of the hosts file is controlled using `manage_etc_hosts`. If this is set to `false`, cloud-init will not manage the hosts file at all. This is the default behavior.

If set to `true`, cloud-init will generate the hosts file using the template located in `/etc/cloud/templates/hosts.tmpl`. In the `/etc/cloud/templates/hosts.tmpl` template, the strings `$hostname` and `$fqdn` will be replaced with the hostname and FQDN respectively.

If `manage_etc_hosts` is set to `localhost`, then cloud-init will not rewrite the hosts file entirely, but rather will ensure that an entry for the FQDN with a distribution-dependent IP is present (i.e., `ping <hostname>` will ping `127.0.0.1` or `127.0.1.1` or other IP).

Note: If `manage_etc_hosts` is set to `true`, the contents of the `hosts` file will be updated every boot. To make any changes to the `hosts` file persistent they must be made in `/etc/cloud/templates/hosts.tmpl`.

Note: For instructions on specifying hostname and FQDN, see documentation for the `cc_set_hostname` module.

Internal name: `cc_update_etc_hosts`

Module frequency: always

Supported distros: all

Activate only on keys: `manage_etc_hosts`

Config schema

- **manage_etc_hosts:** (`True/False/localhost/template`) Whether to manage `/etc/hosts` on the system. If `true`, render the hosts file using `/etc/cloud/templates/hosts.tmpl` replacing `$hostname` and `$fdqn`. If `localhost`, append a `127.0.1.1` entry that resolves from FQDN and hostname every boot. Default: `false`

  Changed in version 22.3.Use of ``template`` is deprecated, use ``true`` instead.

- **fqdn:** (string) Optional fully qualified domain name to use when updating `/etc/hosts`. Preferred over `hostname` if both are provided. In absence of `hostname` and `fqdn` in cloud-config, the `local-hostname` value will be used from datasource metadata.

- **hostname:** (string) Hostname to set when rendering `/etc/hosts`. If `fqdn` is set, the hostname extracted from `fqdn` overrides `hostname`.

Examples

Example 1: Do not update or manage `/etc/hosts` at all. This is the default behavior. Whatever is present at instance boot time will be present after boot. User changes will not be overwritten.

```
#cloud-config
manage_etc_hosts: false
```

Example 2: Manage `/etc/hosts` with cloud-init. On every boot, `/etc/hosts` will be re-written from `/etc/cloud/templates/hosts.tmpl`.

The strings `$hostname` and `$fqdn` are replaced in the template with the appropriate values, either from the config-config `fqdn` or `hostname` if provided. When absent, the cloud metadata will be checked for `local-hostname` which can be split into <hostname>.<fqdn>.

To make modifications persistent across a reboot, you must modify `/etc/cloud/templates/hosts.tmpl`.

```
#cloud-config
manage_etc_hosts: true
```

Example 3: Update `/etc/hosts` every boot, providing a "localhost" `127.0.1.1` entry with the latest hostname and FQDN as provided by either IMDS or cloud-config. All other entries will be left alone. `ping hostname` will ping `127.0.1.1`.

```
#cloud-config
manage_etc_hosts: localhost
```

Update Hostname

Update hostname and FQDN

Summary

This module will update the system hostname and FQDN. If `preserve_hostname` is set to `true`, then the hostname will not be altered.

Note: For instructions on specifying hostname and FQDN, see documentation for the `cc_set_hostname` module.

Internal name: `cc_update_hostname`

Module frequency: always

Supported distros: all

Config schema

- **preserve_hostname:** (boolean) Do not update system hostname when `true`. Default: `false`.
- **prefer_fqdn_over_hostname:** (boolean) By default, it is distro-dependent whether cloud-init uses the short hostname or fully qualified domain name when both `local-hostname` and `fqdn` are both present in instance metadata. When set `true`, use fully qualified domain name if present as hostname instead of short hostname. When set `false`, use `hostname` config value if present, otherwise fallback to `fqdn`.
- **create_hostname_file:** (boolean) If `false`, the hostname file (e.g. /etc/hostname) will not be created if it does not exist. On systems that use systemd, setting create_hostname_file to `false` will set the hostname transiently. If `true`, the hostname file will always be created and the hostname will be set statically on systemd systems. Default: `true`

Examples

Example 1: By default, when `preserve_hostname` is not specified, cloud-init updates `/etc/hostname` per-boot based on the cloud provided `local-hostname` setting. If you manually change `/etc/hostname` after boot cloud-init will no longer modify it.

This default cloud-init behavior is equivalent to this cloud-config:

```
#cloud-config
preserve_hostname: false
```

Example 2: Prevent cloud-init from updating the system hostname.

```
#cloud-config
preserve_hostname: true
```

Example 3: Prevent cloud-init from updating `/etc/hostname`.

```
#cloud-config
preserve_hostname: true
```

Example 4: Set hostname to `external.fqdn.me` instead of `myhost`.

```
#cloud-config
fqdn: external.fqdn.me
hostname: myhost
prefer_fqdn_over_hostname: true
create_hostname_file: true
```

Example 5: Set hostname to `external` instead of `external.fqdn.me` when cloud metadata provides the `local-hostname`: `external.fqdn.me`.

```
#cloud-config
prefer_fqdn_over_hostname: false
```

Example 6: On a machine without an `/etc/hostname` file, don"t create it. In most clouds, this will result in a DHCP-configured hostname provided by the cloud.

```
#cloud-config
create_hostname_file: false
```

Users and Groups

Configure users and groups

Summary

This module configures users and groups. For more detailed information on user options, see the *Including users and groups* config example.

Groups to add to the system can be specified under the `groups` key as a string of comma-separated groups to create, or a list. Each item in the list should either contain a string of a single group to create, or a dictionary with the group name as the key and string of a single user as a member of that group or a list of users who should be members of the group.

Note: Groups are added before users, so any users in a group list must already exist on the system.

Users to add can be specified as a string or list under the `users` key. Each entry in the list should either be a string or a dictionary. If a string is specified, that string can be comma-separated usernames to create, or the reserved string `default` which represents the primary admin user used to access the system. The `default` user varies per distribution and is generally configured in `/etc/cloud/cloud.cfg` by the `default_user key`.

Each `users` dictionary item must contain either a `name` or `snapuser` key, otherwise it will be ignored. Omission of `default` as the first item in the `users` list skips creation the default user. If no `users` key is provided, the default behavior is to create the default user via this config:

```
users:
- default
```

Note: Specifying a hash of a user's password with `passwd` is a security risk if the cloud-config can be intercepted. SSH authentication is preferred.

Note: If specifying a `doas` rule for a user, ensure that the syntax for the rule is valid, as the only checking performed by cloud-init is to ensure that the user referenced in the rule is the correct user.

Note: If specifying a `sudo` rule for a user, ensure that the syntax for the rule is valid, as it is not checked by cloud-init.

Note: Most of these configuration options will not be honored if the user already exists. The following options are the exceptions, and are applied to already-existing users; `plain_text_passwd`, `doas`, `hashed_passwd`, `lock_passwd`, `sudo`, `ssh_authorized_keys`, `ssh_redirect_user`.

The `user` key can be used to override the `default_user` configuration defined in `/etc/cloud/cloud.cfg`. The `user` value should be a dictionary which supports the same config keys as the `users` dictionary items.

Internal name: `cc_users_groups`

Module frequency: once-per-instance

Supported distros: all

Config schema

- **groups:** (['string', 'object', 'array'] of string/object)
  - **<group_name>:** (['string', 'array'] of string) Optional string of single username or a list of usernames to add to the group
  - **<group_name>:** (['string', 'array'] of string) Optional string of single username or a list of usernames to add to the group
- **user:** (string/object) The `user` dictionary values override the `default_user` configuration from `/etc/cloud/cloud.cfg`. The *user* dictionary keys supported for the default_user are the same as the `users` schema.
- **users:** (['string', 'array', 'object'] of string/array of string/object)

Examples

Example 1: Add the `default_user` from `/etc/cloud/cloud.cfg`. This is also the default behavior of cloud-init when no `users` key is provided.

```
#cloud-config
users: [default]
```

Example 2: Add the `admingroup` with members `root` and `sys`, and an empty group `cloud-users`.

```
#cloud-config
groups:
- admingroup: [root, sys]
- cloud-users
```

Example 3: Skip creation of the `default` user and only create `newsuper`. Password-based login is rejected, but the GitHub user `TheRealFalcon` and the Launchpad user `falcojr` can SSH as `newsuper`. The default shell for `newsuper` is bash instead of system default.

```
#cloud-config
users:
- gecos: Big Stuff
  groups: users, admin
  lock_passwd: true
  name: newsuper
  shell: /bin/bash
  ssh_import_id: ['lp:falcojr', 'gh:TheRealFalcon']
  sudo: ALL=(ALL) NOPASSWD:ALL
```

Example 4: Skip creation of the `default` user and only create `newsuper`. Password-based login is rejected, but the GitHub user `TheRealFalcon` and the Launchpad user `falcojr` can SSH as `newsuper`. `doas`/`opendoas` is configured to permit this user to run commands as other users (without being prompted for a password) except not as root.

```
#cloud-config
users:
- doas: [permit nopass newsuper, deny newsuper as root]
  gecos: Big Stuff
  groups: users, admin
  lock_passwd: true
  name: newsuper
  ssh_import_id: ['lp:falcojr', 'gh:TheRealFalcon']
```

Example 5: On a system with SELinux enabled, add `youruser` and set the SELinux user to `staff_u`. When omitted on SELinux, the system will select the configured default SELinux user.

```
#cloud-config
users:
- default
- {name: youruser, selinux_user: staff_u}
```

Example 6: To redirect a legacy username to the `default` user for a distribution, `ssh_redirect_user` will accept an SSH connection and emit a message telling the client to SSH as the `default` user. SSH clients will get the message;

```
#cloud-config
users:
- default
- {name: nosshlogins, ssh_redirect_user: true}
```

Example 7: Override any `default_user` config in `/etc/cloud/cloud.cfg` with supplemental config options. This config will make the default user `mynewdefault` and change the user to not have `sudo` rights.

```
#cloud-config
ssh_import_id: [chad.smith]
user: {name: mynewdefault, sudo: null}
```

Example 8: Avoid creating any `default_user`.

```
#cloud-config
users: []
```

Wireguard

Module to configure WireGuard tunnel

Summary

The WireGuard module provides a dynamic interface for configuring WireGuard (as a peer or server) in a straightforward way.

This module takes care of;

- writing interface configuration files
- enabling and starting interfaces
- installing wireguard-tools package
- loading WireGuard kernel module
- executing readiness probes

What is a readiness probe?

The idea behind readiness probes is to ensure WireGuard connectivity before continuing the cloud-init process. This could be useful if you need access to specific services like an internal APT Repository Server (e.g., Landscape) to install/update packages.

Example

An edge device can't access the internet but uses cloud-init modules which will install packages (e.g. `landscape`, `packages`, `ubuntu_advantage`). Those modules will fail due to missing internet connection. The `wireguard` module fixes that problem as it waits until all readiness probes (which can be arbitrary commands, e.g. checking if a proxy server is reachable over WireGuard network) are finished, before continuing the cloud-init `config` stage.

Note: In order to use DNS with WireGuard you have to install the `resolvconf` package or symlink it to systemd's `resolvectl`, otherwise `wg-quick` commands will throw an error message that executable `resolvconf` is missing, which leads the `wireguard` module to fail.

Internal name: `cc_wireguard`

Module frequency: once-per-instance

Supported distros: ubuntu

Activate only on keys: `wireguard`

Config schema

- **wireguard:** (null/object)
  - **interfaces:** (array of object) Each object in **interfaces** list supports the following keys:
    - **name:** (string) Name of the interface. Typically wgx (example: wg0)
    - **config_path:** (string) Path to configuration file of Wireguard interface
    - **content:** (string) Wireguard interface configuration. Contains key, peer, ...
  - **readinessprobe:** (array of string) List of shell commands to be executed as probes.

Examples

Configure one or more WireGuard interfaces and provide optional readiness probes.

```
#cloud-config
wireguard:
  interfaces:
    - name: wg0
      config_path: /etc/wireguard/wg0.conf
      content: |
        [Interface]
        PrivateKey = <private_key>
        Address = <address>
        [Peer]
        PublicKey = <public_key>
        Endpoint = <endpoint_ip>:<endpoint_ip_port>
        AllowedIPs = <allowedip1>, <allowedip2>, ...
    - name: wg1
      config_path: /etc/wireguard/wg1.conf
      content: |
        [Interface]
        PrivateKey = <private_key>
        Address = <address>
        [Peer]
        PublicKey = <public_key>
        Endpoint = <endpoint_ip>:<endpoint_ip_port>
        AllowedIPs = <allowedip1>
  readinessprobe:
    - 'systemctl restart service'
    - 'curl https://webhook.endpoint/example'
    - 'nc -zv some-service-fqdn 443'
```

Write Files

Write arbitrary files

Summary

Write out arbitrary content to files, optionally setting permissions. Parent folders in the path are created if absent. Content can be specified in plain text or binary. Data encoded with either base64 or binary gzip data can be specified and will be decoded before being written. Data can also be loaded from an arbitrary URI. For empty file creation, content can be omitted.

Note: If multi-line data is provided, care should be taken to ensure it follows YAML formatting standards. To specify binary data, use the YAML option `!!binary`.

Note: Do not write files under `/tmp` during boot because of a race with `systemd-tmpfiles-clean` that can cause temporary files to be cleaned during the early boot process. Use `/run/somedir` instead to avoid a race (LP: #1707222).

Warning: Existing files will be overridden.

Internal name: `cc_write_files`

Module frequency: once-per-instance

Supported distros: all

Activate only on keys: `write_files`

Config schema

- **write_files:** (array of object) Each object in **write_files** list supports the following keys:
  - **path:** (string) Path of the file to which `content` is decoded and written
  - **content:** (string) Optional content to write to the provided `path`. When content is present and encoding is not 'text/plain', decode the content prior to writing. Default: `''`
  - **source:** (object) Optional specification for content loading from an arbitrary URI
    - **uri:** (string) URI from which to load file content. If loading fails repeatedly, `content` is used instead.
    - **headers:** (object) Optional HTTP headers to accompany load request, if applicable
  - **owner:** (string) Optional owner:group to chown on the file and new directories. Default: `root:root`
  - **permissions:** (string) Optional file permissions to set on `path` represented as an octal string '0###'. Default: `0o644`
  - **encoding:** (`gz/gzip/gz+base64/gzip+base64/gz+b64/gzip+b64/b64/base64/text/plain`) Optional encoding type of the content. Default: `text/plain`. No decoding is performed by default. Supported encoding types are: gz, gzip, gz+base64, gzip+base64, gz+b64, gzip+b64, b64, base64
  - **append:** (boolean) Whether to append `content` to existing file if `path` exists. Default: `false`.
  - **defer:** (boolean) Defer writing the file until 'final' stage, after users were created, and packages were installed. Default: `false`.

Examples

Example 1: Write out base64-encoded content to `/etc/sysconfig/selinux`.

```
#cloud-config
write_files:
- encoding: b64
  content: CiMgVGhpcyBmaWxlIGNvbnRyb2xzIHRoZSBzdGF0ZSBvZiBTRUxpbnV4...
  owner: root:root
  path: /etc/sysconfig/selinux
  permissions: '0644'
```

Example 2: Appending content to an existing file.

```
#cloud-config
write_files:
- content: |
    15 * * * * root ship_logs
```

(continues on next page)

(continued from previous page)

```
  path: /etc/crontab
  append: true
```

Example 3: Provide gzipped binary content

```
#cloud-config
write_files:
- encoding: gzip
  content: !!binary |
      H4sIAIDb/U8C/1NW1E/KzNMvzuBKTc7IV8hIzcnJVyjPL8pJ4QIA6N+MVxsAAAA=
  path: /usr/bin/hello
  permissions: '0755'
```

Example 4: Create an empty file on the system

```
#cloud-config
write_files:
- path: /root/CLOUD_INIT_WAS_HERE
```

Example 5: Defer writing the file until after the package (Nginx) is installed and its user is created.

```
#cloud-config
write_files:
- path: /etc/nginx/conf.d/example.com.conf
  content: |
    server {
        server_name example.com;
        listen 80;
        root /var/www;
        location / {
            try_files $uri $uri/ $uri.html =404;
        }
    }
  owner: 'nginx:nginx'
  permissions: '0640'
  defer: true
```

Example 6: Retrieve file contents from a URI source, rather than inline. Especially useful with an external config-management repo, or for large binaries.

```
#cloud-config
write_files:
- source:
    uri: https://gitlab.example.com/some_ci_job/artifacts/hello
    headers:
      Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
      User-Agent: cloud-init on myserver.example.com
  path: /usr/bin/hello
  permissions: '0755'
```

Yum Add Repo

Add yum repository configuration to the system

Summary

Add yum repository configuration to `/etc/yum.repos.d`. Configuration files are named based on the opaque dictionary key under the `yum_repos` they are specified with. If a config file already exists with the same name as a config entry, the config entry will be skipped.

Internal name: `cc_yum_add_repo`

Module frequency: once-per-instance

Supported distros: almalinux, azurelinux, centos, cloudlinux, eurolinux, fedora, mariner, openeuler, OpenCloudOS, openmandriva, photon, rhel, rocky, TencentOS, virtuozzo

Activate only on keys: `yum_repos`

Config schema

- **yum_repo_dir:** (string) The repo parts directory where individual yum repo config files will be written. Default: `/etc/yum.repos.d`
- **yum_repos:** (object)
  - **<repo_name>:** (object) Object keyed on unique yum repo IDs. The key used will be used to write yum repo config files in `yum_repo_dir`/<repo_key_id>.repo.
    - **baseurl:** (string) URL to the directory where the yum repository's 'repodata' directory lives
    - **metalink:** (string) Specifies a URL to a metalink file for the repomd.xml
    - **mirrorlist:** (string) Specifies a URL to a file containing a baseurls list
    - **name:** (string) Optional human-readable name of the yum repo.
    - **enabled:** (boolean) Whether to enable the repo. Default: `true`.
    - **<yum_config_option>:** (integer/boolean/string) Any supported yum repository configuration options will be written to the yum repo config file. See: man yum.conf

Examples

Example 1:

```
#cloud-config
yum_repos:
  my_repo:
    baseurl: http://blah.org/pub/epel/testing/5/$basearch/
yum_repo_dir: /store/custom/yum.repos.d
```

Example 2: Enable cloud-init upstream's daily testing repo for EPEL 8 to install the latest cloud-init from tip of `main` for testing.

```
#cloud-config
yum_repos:
  cloud-init-daily:
    name: Copr repo for cloud-init-dev owned by @cloud-init
    baseurl: https://download.copr.fedorainfracloud.org/results/@cloud-init/cloud-init-
↪dev/epel-8-$basearch/
    type: rpm-md
    skip_if_unavailable: true
    gpgcheck: true
    gpgkey: https://download.copr.fedorainfracloud.org/results/@cloud-init/cloud-init-
↪dev/pubkey.gpg
    enabled_metadata: 1
```

Example 3: Add the file `/etc/yum.repos.d/epel_testing.repo` which can then subsequently be used by yum for later operations.

```
#cloud-config
yum_repos:
# The name of the repository
 epel-testing:
   baseurl: https://download.copr.fedorainfracloud.org/results/@cloud-init/cloud-init-
↪dev/pubkey.gpg
   enabled: false
   failovermethod: priority
   gpgcheck: true
   gpgkey: file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL
   name: Extra Packages for Enterprise Linux 5 - Testing
```

Example 4: Any yum repo configuration can be passed directly into the repository file created. See `man yum.conf` for supported config keys.

Write `/etc/yum.conf.d/my-package-stream.repo` with `gpgkey` checks on the repo data of the repository enabled.

```
#cloud-config
yum_repos:
  my package stream:
    baseurl: http://blah.org/pub/epel/testing/5/$basearch/
    mirrorlist: http://some-url-to-list-of-baseurls
    repo_gpgcheck: 1
    enable_gpgcheck: true
    gpgkey: https://url.to.ascii-armored-gpg-key
```

Zypper Add Repo

Configure Zypper behavior and add Zypper repositories

Summary

Zypper behavior can be configured using the `config` key, which will modify `/etc/zypp/zypp.conf`. The configuration writer will only append the provided configuration options to the configuration file. Any duplicate options will be resolved by the way the `zypp.conf` INI file is parsed.

Note: Setting `configdir` is not supported and will be skipped.

The `repos` key may be used to add repositories to the system. Beyond the required `id` and `baseurl` attributions, no validation is performed on the `repos` entries.

It is assumed the user is familiar with the Zypper repository file format. This configuration is also applicable for systems with transactional-updates.

Internal name: `cc_zypper_add_repo`

Module frequency: always

Supported distros: opensuse, opensuse-microos, opensuse-tumbleweed, opensuse-leap, sle_hpc, sle-micro, sles

Activate only on keys: `zypper`

Config schema

- **zypper:** (object)
  - **repos:** (array of object) Each object in **repos** list supports the following keys:
    - **id:** (string) The unique id of the repo, used when writing /etc/zypp/repos.d/<id>.repo.
    - **baseurl:** (string) The base repositoy URL
  - **config:** (object) Any supported zypo.conf key is written to `/etc/zypp/zypp.conf`

Examples

Example 1:

```
#cloud-config
zypper:
  config: {download.use_deltarpm: true, reposdir: /etc/zypp/repos.dir, servicesdir: /etc/
↪zypp/services.d}
  repos:
  - {autorefresh: 1, baseurl: 'http://dl.opensuse.org/dist/leap/v/repo/oss/', enabled: 1,
    id: opensuse-oss, name: os-oss}
  - {baseurl: 'http://dl.opensuse.org/dist/leap/v/update', id: opensuse-oss-update,
    name: os-oss-up}
```

2.4.2 Cloud config examples

Including users and groups

```
#cloud-config
# Add groups to the system
# The following example adds the 'admingroup' group with members 'root' and 'sys'
# and the empty group cloud-users.
groups:
  - admingroup: [root,sys]
  - cloud-users

# Add users to the system. Users are added after groups are added.
# Note: Most of these configuration options will not be honored if the user
#       already exists. Following options are the exceptions and they are
#       applicable on already-existing users:
#       - 'plain_text_passwd', 'hashed_passwd', 'lock_passwd', 'sudo',
#         'ssh_authorized_keys', 'ssh_redirect_user'.
users:
  - default
  - name: foobar
    gecos: Foo B. Bar
    primary_group: foobar
    groups: users
    selinux_user: staff_u
    expiredate: '2032-09-01'
    ssh_import_id:
      - lp:falcojr
      - gh:TheRealFalcon
    lock_passwd: false
    passwd: $6$j212wezy$7H/1LT4f9/N3wpgNunhsIqtMj62OKiS3nyNwuizouQc3u7MbYCarYeAHWYPYb2FT.
↪lbioDm2RrkJPb9BZMN1O/
  - name: barfoo
    gecos: Bar B. Foo
    sudo: ALL=(ALL) NOPASSWD:ALL
    groups: users, admin
    ssh_import_id:
      - lp:falcojr
      - gh:TheRealFalcon
    lock_passwd: true
    ssh_authorized_keys:
      - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDSL7uWGj8cgWyIOaspgKdVy0cKJ+UTjfv7jBOjG2H/
↪GN8bJVXy72XAvnhM0dUM+CCs8FOf0YlPX+Frvz2hKInrmRhZVwRSL129PasD12MlI3l44u6IwS1o/
↪W86Q+tkQYEljtqDOo0a+cOsaZkvUNzUyEXUwz/
↪lmYa6G4hMKZH4NBj7nbAAF96wsMCoyNwbWryBnDYUr6wMbjRR1J9Pw7Xh7WRC73wy4Va2YuOgbD3V/
↪5ZrFPLbWZW/7TFXVrql04QVbyei4aiFR5n//GvoqwQDNe58LmbzX/xvxyKJYdny2zXmdAhMxbrpFQsfpkJ9E/
↪H5w0yOdSvnWbUoG5xNGoOB csmith@fringe
  - name: testuser
    gecos: Mr. Test
    homedir: /local/testdir
    sudo: ["ALL=(ALL) NOPASSWD:ALL"]
  - name: cloudy
    gecos: Magic Cloud App Daemon User
```

(continues on next page)

(continued from previous page)

```
    inactive: '5'
    system: true
  - name: fizzbuzz
    sudo: false
    shell: /bin/bash
    ssh_authorized_keys:
      - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDSL7uWGj8cgWyIOaspgKdVy0cKJ+UTjfv7jBOjG2H/
↪GN8bJVXy72XAvnhM0dUM+CCs8FOf0YlPX+Frvz2hKInrmRhZVwRSL129PasD12MlI3l44u6IwS1o/
↪W86Q+tkQYEljtqDOo0a+cOsaZkvUNzUyEXUwz/
↪lmYa6G4hMKZH4NBj7nbAAF96wsMCoyNwbWryBnDYUr6wMbjRR1J9Pw7Xh7WRC73wy4Va2YuOgbD3V/
↪5ZrFPLbWZW/7TFXVrql04QVbyei4aiFR5n//GvoqwQDNe58LmbzX/xvxyKJYdny2zXmdAhMxbrpFQsfpkJ9E/
↪H5w0yOdSvnWbUoG5xNGoOB csmith@fringe
  - snapuser: joe@joeuser.io
  - name: nosshlogins
    ssh_redirect_user: true

# Valid Values:
#   name: The user's login name
#   expiredate: Date on which the user's account will be disabled.
#   gecos: The user name's real name, i.e. "Bob B. Smith"
#   homedir: Optional. Set to the local path you want to use. Defaults to
#            /home/<username>
#   primary_group: define the primary group. Defaults to a new group created
#            named after the user.
#   groups:  Optional. Additional groups to add the user to. Defaults to none
#   selinux_user:  Optional. The SELinux user for the user's login, such as
#            "staff_u". When this is omitted the system will select the default
#            SELinux user.
#   lock_passwd: Defaults to true. Lock the password to disable password login
#   inactive: Number of days after password expires until account is disabled
#   passwd: The hash -- not the password itself -- of the password you want
#            to use for this user. You can generate a hash via:
#                mkpasswd --method=SHA-512 --rounds=4096
#            (the above command would create from stdin an SHA-512 password hash
#            with 4096 salt rounds)
#
#            Please note: while the use of a hashed password is better than
#                plain text, the use of this feature is not ideal. Also,
#                using a high number of salting rounds will help, but it should
#                not be relied upon.
#
#                To highlight this risk, running John the Ripper against the
#                example hash above, with a readily available wordlist, revealed
#                the true password in 12 seconds on a i7-2620QM.
#
#                In other words, this feature is a potential security risk and is
#                provided for your convenience only. If you do not fully trust the
#                medium over which your cloud-config will be transmitted, then you
#                should not use this feature.
#
#   no_create_home: When set to true, do not create home directory.
#   no_user_group: When set to true, do not create a group named after the user.
```

(continues on next page)

(continued from previous page)

```
#   no_log_init: When set to true, do not initialize lastlog and faillog database.
#   ssh_import_id: Optional. Import SSH ids
#   ssh_authorized_keys: Optional. [list] Add keys to user's authorized keys file
#                        An error will be raised if no_create_home or system is
#                        also set.
#   ssh_redirect_user: Optional. [bool] Set true to block ssh logins for cloud
#       ssh public keys and emit a message redirecting logins to
#       use <default_username> instead. This option only disables cloud
#       provided public-keys. An error will be raised if ssh_authorized_keys
#       or ssh_import_id is provided for the same user.
#
#   sudo: Defaults to none. Accepts a sudo rule string, a list of sudo rule
#         strings or False to explicitly deny sudo usage. Examples:
#
#         Allow a user unrestricted sudo access.
#             sudo:  ALL=(ALL) NOPASSWD:ALL
#                       or
#             sudo: ["ALL=(ALL) NOPASSWD:ALL"]
#
#         Adding multiple sudo rule strings.
#             sudo:
#               - ALL=(ALL) NOPASSWD:/bin/mysql
#               - ALL=(ALL) ALL
#
#         Prevent sudo access for a user.
#             sudo: False
#
#         Note: Please double check your syntax and make sure it is valid.
#               cloud-init does not parse/check the syntax of the sudo
#               directive.
#   system: Create the user as a system user. This means no home directory.
#   snapuser: Create a Snappy (Ubuntu-Core) user via the snap create-user
#             command available on Ubuntu systems.  If the user has an account
#             on the Ubuntu SSO, specifying the email will allow snap to
#             request a username and any public ssh keys and will import
#             these into the system with username specified by SSO account.
#             If 'username' is not set in SSO, then username will be the
#             shortname before the email domain.
#

# Default user creation:
#
# Unless you define users, you will get a 'ubuntu' user on Ubuntu systems with the
# legacy permission (no password sudo, locked user, etc). If however, you want
# to have the 'ubuntu' user in addition to other users, you need to instruct
# cloud-init that you also want the default user. To do this use the following
# syntax:
#   users:
#     - default
#     - bob
#     - ....
#  foobar: ...
```

(continues on next page)

(continued from previous page)

```
#
# users[0] (the first user in users) overrides the user directive.
#
# The 'default' user above references the distro's config set in
# /etc/cloud/cloud.cfg.
```

Writing out arbitrary files

```
#cloud-config
# vim: syntax=yaml
#
# This is the configuration syntax that the write_files module
# will know how to understand. Encoding can be given b64 or gzip or (gz+b64).
# The content will be decoded accordingly and then written to the path that is
# provided.
#
# Note: Content strings here are truncated for example purposes.
write_files:
- encoding: b64
  content: CiMgVGhpcyBmaWxlIGNvbnRyb2xzIHRoZSBzdGF0ZSBvZiBTRUxpbnV4...
  owner: root:root
  path: /etc/sysconfig/selinux
  permissions: '0644'
- content: |
    # My new /etc/sysconfig/samba file

    SMBDOPTIONS="-D"
  path: /etc/sysconfig/samba
- content: !!binary |
    f0VMRgIBAQAAAAAAAAAAAAIAPgABAAAAwARAAAAAAABAAAAAAAAAAJAVAAAAAAAAAAAAAEAAOAAI
    AEAAHgAdAAYAAAAFAAAAQAAAAAAAAABAAEAAAAAAAEAAQAAAAAAAwAEAAAAAAADAAQAAAAAAAAgA
    AAAAAAAAwAAAAQAAAAAgAAAAAAAAACQAAAAAAAAAJAAAAAAAAcAAAAAAAAABwAAAAAAAAAQAA
    ....
  path: /bin/arch
  permissions: '0555'
- encoding: gzip
  content: !!binary |
    H4sIAIDb/U8C/1NW1E/KzNMvzuBKTc7IV8hIzcnJVyjPL8pJ4QIA6N+MVxsAAAA=
  path: /usr/bin/hello
  permissions: '0755'
```

Adding a yum repository

```
#cloud-config
# vim: syntax=yaml
#
# Add yum repository configuration to the system
#
# The following example adds the file /etc/yum.repos.d/epel_testing.repo
# which can then subsequently be used by yum for later operations.
yum_repos:
  # The name of the repository
  epel-testing:
    # Any repository configuration options
    # See: man yum.conf
    #
    # At least one of 'baseurl' or 'metalink' or 'mirrorlist' is required!
    baseurl: http://download.fedoraproject.org/pub/epel/testing/5/$basearch
    metalink: https://mirrors.fedoraproject.org/metalink?repo=epel-$releasever&arch=
↪$basearch&infra=$infra&content=$contentdir
    mirrorlist: https://mirrors.fedoraproject.org/metalink?repo=fedora-$releasever&
    enabled: false
    failovermethod: priority
    gpgcheck: true
    gpgkey: file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL
    name: Extra Packages for Enterprise Linux 5 - Testing
```

Configure an instance's trusted CA certificates

```
#cloud-config
#
# This is an example file to configure an instance's trusted CA certificates
# system-wide for SSL/TLS trust establishment when the instance boots for the
# first time.
#
# Make sure that this file is valid yaml before starting instances.
# It should be passed as user-data when starting the instance.

ca_certs:
  # If present and set to True, the 'remove_defaults' parameter will either
  # disable all the trusted CA certifications normally shipped with
  # Alpine, Debian or Ubuntu. On RedHat, this action will delete those
  # certificates.
  # This is mainly for very security-sensitive use cases - most users will not
  # need this functionality.
  remove_defaults: true

  # If present, the 'trusted' parameter should contain a certificate (or list
  # of certificates) to add to the system as trusted CA certificates.
  # Pay close attention to the YAML multiline list syntax.  The example shown
  # here is for a list of multiline certificates.
  trusted:
```

(continues on next page)

(continued from previous page)

```
  - |
   -----BEGIN CERTIFICATE-----
   YOUR-ORGS-TRUSTED-CA-CERT-HERE
   -----END CERTIFICATE-----
  - |
   -----BEGIN CERTIFICATE-----
   YOUR-ORGS-TRUSTED-CA-CERT-HERE
   -----END CERTIFICATE-----
```

Install and run chef recipes

```
#cloud-config
#
# This is an example file to automatically install chef-client and run a
# list of recipes when the instance boots for the first time.
# Make sure that this file is valid yaml before starting instances.
# It should be passed as user-data when starting the instance.

# The default is to install from packages.

# Key from https://packages.chef.io/chef.asc
apt:
  sources:
    source1:
      source: "deb http://packages.chef.io/repos/apt/stable $RELEASE main"
      key: |
        -----BEGIN PGP PUBLIC KEY BLOCK-----
        Version: GnuPG v1.4.12 (Darwin)
        Comment: GPGTools - http://gpgtools.org

        mQGiBEppC7QRBADfsOkZU6KZK+YmKw4wev5mjKJEkVGlus+NxW8wItX5sGa6kdUu
        twAyj7Yr92rF+ICFEP3gGU6+lGo0Nve7KxkN/1W7/m3G4zuk+ccIKmjp8KS3qn99
        dxy64vcji9jIllVa+XXOGIp0G8GEaj7mbkixL/bMeGfdMlv8Gf2XPpp9vwCgn/GC
        JKacfnw7MpLKUHOYSlb//JsEAJqao3ViNfav83jJKEkD8cf59Y8xKia5OpZqTK5W
        ShVnNWS3U5IVQk10ZDH97Qn/YrK387H4CyhLE9mxPXs/ul18ioiaars/q2MEKU2I
        XKfV21eMLO9LYd6Ny/Kqj8o5WQK2J6+NAhSwvthZcIEphcFignIuobP+B5wNFQpe
        DbKfA/0WvN2OwFeWRcmmd3Hz7nHTpcnSF+4QX6yHRF/5BgxkG6IqBIACQbzPn6Hm
        sMtm/SVf11izmDqSsQptCrOZILfLX/mE+YOl+CwWSHhl+YsFts1WOuh1EhQD26aO
        Z84HuHV5HFRWjDLw9LriltBVQcXbpfSrRP5bdr7Wh8vhqJTPjrQnT3BzY29kZSBQ
        YWNrYWdlcyA8cGFja2FnZXNAb3BzY29kZS5jb20+iGAEExECACAFAkppC7QCGwMG
        CwkIBwMCBBUCCAMEFgIDAQIeAQIXgAAKCRApQKupg++Caj8sAKCOXmdG36gWji/K
        +o+XtBfvdMnFYQCfTCEWxRy2BnzLoBBFCjDSK6sJqCu0IENIRUYgUGFja2FnZXMg
        PHBhY2thZ2VzQGNoZWYuaW8+iGIEExECACIFAlQwYFECGwMGCwkIBwMCBhUIAgkK
        CwQWAgMBAh4BAheAAAoJECClAq6mD74JqX94An26z99XOHWpLN8ahzm7cp13t4Xid
        AJ9wVcgoUBzvgg91lKfv/34cmemZn7kCDQRKaQu0EAgAg7ZLCVGVTmLqBM6njZEd
        Zbv+mZbvwLBSomdiqddE6u3eH0X3GuwaQfQWHUVG2yedyDMiG+EMtCdEeeRebTCz
        SNXQ8Xvi22hRPoEsBSwWLZI8/XNg0n0f1+GEr+mOKO0BxDB2DG7DA0nnEISxwFkK
        OFJFebR3fRsrWjj0KjDxkhse2ddU/jVz1BY7Nf8toZmwpBmdozETMOTx3LJy1HZ/
        Te9FJXJMUaB2lRyluv15MVWCKQJro4MQG/7QGcIfrIZNfAGJ32DDSjV7/YO+IpRY
        IL4CUBQ65suY4gYUG4jhRH6u7H1p99sdwsg5OIpBe/v2Vbc/tbwAB+eJJAp89Zeu
```

(continues on next page)

(continued from previous page)

```
        twADBQf/ZcGoPhTGFuzbkcNRSIz+boaeWPoSxK2DyfScyCAuG41CY9+g0HIw9Sq8
        DuxQvJ+vrEJjNvNE3EAEdKl/zkXMZDb1EXjGwDi845TxEMhhD1dDw2qpHqnJ2mtE
        WpZ7juGwA3sGhi6FapO04tIGacCfNNHmlRGipyq5ZiKIRq9mLEndlECr8cwaKgkS
        0wWu+xmMZe7N5/t/TK19HXNh4tVacv0F3fYK54GUjt2FjCQV75USnmNY4KPTYLXA
        dzC364hEMlXpN21siIFgB04w+TXn5UF3B4FfAy5hevvr4DtV4MvMiGLu0oWjpaLC
        MpmrR3Ny2wkmO0h+vgri9uIP06ODWIhJBBgRAgAJBQJKaQu0AhsMAAoJECIAq6mD
        74Jq4hIAoJ5KrYS8kCwj26SAGzglwggpvt3CAJ0bekyky56vNqoegB+y4PQVDv4K
        zA==
        =IxPr
        -----END PGP PUBLIC KEY BLOCK-----

chef:

  # Valid values are 'accept' and 'accept-no-persist'
  chef_license: "accept"

  # Valid values are 'gems' and 'packages' and 'omnibus'
  install_type: "packages"

  # Boolean: run 'install_type' code even if chef-client
  #          appears already installed.
  force_install: false

  # Chef settings
  server_url: "https://chef.yourorg.com"

  # Node Name
  # Defaults to the instance-id if not present
  node_name: "your-node-name"

  # Environment
  # Defaults to '_default' if not present
  environment: "production"

  # Default validation name is chef-validator
  validation_name: "yourorg-validator"
  # if validation_cert's value is "system" then it is expected
  # that the file already exists on the system.
  validation_cert: |
    -----BEGIN RSA PRIVATE KEY-----
    YOUR-ORGS-VALIDATION-KEY-HERE
    -----END RSA PRIVATE KEY-----

  # A run list for a first boot json, an example (not required)
  run_list:
    - "recipe[apache2]"
    - "role[db]"

  # Specify a list of initial attributes used by the cookbooks
  initial_attributes:
    apache:
      prefork:
```

(continues on next page)

(continued from previous page)

```
      maxclients: 100
    keepalive: "off"

  # if install_type is 'omnibus', change the url to download
  omnibus_url: "https://www.chef.io/chef/install.sh"

  # if install_type is 'omnibus', pass pinned version string
  # to the install script
  omnibus_version: "12.3.0"

  # If encrypted data bags are used, the client needs to have a secrets file
  # configured to decrypt them
  encrypted_data_bag_secret: "/etc/chef/encrypted_data_bag_secret"
```

Install and run *ansible-pull*

```
#cloud-config
package_update: true
package_upgrade: true

# if you're already installing other packages, you may
# wish to manually install ansible to avoid multiple calls
# to your package manager
packages:
  - git
ansible:
  install_method: pip
  pull:
    url: "https://github.com/holmanb/vmboot.git"
    playbook_name: ubuntu.yml
```

Configure instance to be managed by Ansible

```
#cloud-config
#
# A common use-case for cloud-init is to bootstrap user and ssh
# settings to be managed by a remote configuration management tool,
# such as ansible.
#
# This example assumes a default Ubuntu cloud image, which should contain
# the required software to be managed remotely by Ansible.
#
ssh_pwauth: false

users:
- name: ansible
  gecos: Ansible User
  groups: users,admin,wheel
  sudo: ALL=(ALL) NOPASSWD:ALL
```

(continues on next page)

(continued from previous page)

```
  shell: /bin/bash
  lock_passwd: true
  ssh_authorized_keys:
    - "ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAADAQABAAABgQDRCJCQ1UD9QslWDSw5Pwsvba0Wsf1pO4how5BtNaZn0xLZpTq2nqFEJshUkd/
↪zCWF7DWyhmNphQ8c+U+wcmdNVcg2pI1kPxq0VZzBfZ7cDwhjgeLsIvTXvU+HVRtsXh4c5FlUXpRjf/
↪x+a3vqFRvNsRd1DE+5ZqQHbOVbnsStk3PZppaByMg+AZZMx56OUk2pZCgvpCwj6LIixqwuxNKPxmJf45RyOsPUXwCwkq9UD4me5jk
↪141moWsGxSnd5NxCbPUWGoRfYcHc865E70nN4WrZkM7RFI/s5mvQtuj8dRL67JUEwvdvEDO0EBz21FV/
↪iOracXd2omlTUSK+wYrWGtiwQwEgr4r5bimxDKy9L8UlaJZ+ONhLTP8ecTHYkaU1C75sLX9ZYd5YtqjiNGsNF+wdW6WrXrQiWeyrG
↪"


# use the following passwordless demonstration key for testing or
# replace with your own key pair
#
# -----BEGIN OPENSSH PRIVATE KEY-----
# b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
# NhAAAAAwEAAQAAAYEA0QiQkNVA/ULJVg0sOT8LL22tFrH9aTuIaMOQbTWmZ9MS2aU6tp6h
# RCbIVJHf8wlhew1soZjaYUPHPlPsHJnTVXINqSNZD8atFWcwX2e3A8IY4Hi7CL0171Ph1U
# bbF4eHORZVF6UY3/8fmt76hUbzbEXdQxPuWakB2zlW57ErZNz2aaWgcjIPgGWTMeejlJNq
# WQoL6QsI+iyIsasLsTSj8ZiX+OUcjrD1F8AsJKvVA+JnuY5LEzz5Ld6HlFsNWUkhfBf9eN
# ZqFrBsUp3eTcQmz1FhqEX2HB3POuRO9JzeFq2ZDO0RSP7OZr0Lbo/HUS+uyVBML3bxAztB
# Ac9tRVf4jq2nF3dqJpU1EivsGK1hrYsEMBIK+K+W4psQysvS/FJWiWfjjYS0z/HnEx2JGl
# NQu+bC1/WWHeWLao4jRrDRfsHVulq160Ilnsqxiu2cGwO5WoEsSGu8nqpyg43ZHCb0FwmB
# izPQDASlniWjqcKmfnTrpzAy3eVWawwlNpaQkidTAAAFgGKSj8diko/HAAAAB3NzaC1yc2
# EAAAGBANEIkJDVQP1CyVYNLDk/Cy9trRax/Wk7iGjDkG01pmfTEtmlOraeoUQmyFSR3/MJ
# YXsNbKGY2mFDxz5T7ByZ01VyDakjWQ/GrRVnMF9ntwPCGOB4uwi9Ne9T4dVG2xeHhzkWVR
# elGN//H5re+oVG82xF3UMT7lmpAds5VuexK2Tc9mmloHIyD4BlkzHno5STalkKC+kLCPos
# iLGrC7E0o/GYl/jlHI6w9RfALCSr1QPiZ7mOSxM8+S3eh5RbDVlJIXwX/XjWahawbFKd3k
# 3EJs9RYahF9hwdzzrkTvSc3hatmQztEUj+zma9C26Px1EvrslQTC928QM7QQHPbUVX+I6t
# pxd3aiaVNRIr7BitYa2LBDASCvivluKbEMrL0vxSVoln442EtM/x5xMdiRpTULvmwtf1lh
# 3li2qOI0aw0X7B1bpatetCJZ7KsYrtnBsDuVqBLEhrvJ6qcoON2Rwm9BcJgYsz0AwEpZ4l
# o6nCpn5066cwMt3lVmsMJTaWkJInUwAAAAMBAAEAAAGAEuz77Hu9EEZyujLOdTnAW9afRv
# XDOZA6pS7yWEufjw5CSlMLwisR83yww09t1QWyvhRqEyYmvOBecsXgaSUtnYfftWz44apy
# /gQYvMVELGKaJAC/q7vjMpGyrxUPkyLMhckALU2KYgV+/rj/j6pBMeVlchmk3pikYrffUX
# JDY990WVO194Dm0buLRzJvfMKYF2BcfF4TvarjOXWAxSuR8www050oJ8HdKahW7Cm5S0po
# FRnNXFGMnLA62vN00vJW8V7j7vui9ukBbhjRWaJuY5rdG/UYmzAe4wvdIEnpk9xIn6JGCp
# FRYTRn7lTh5+/QlQ6FXRP8Ir1vXZFnhKzl0K8Vqh2sf4M79MsIUGAqGxg9xdhjIa5dmgp8
# N18IEDoNEVKUbKuKe/Z5yf8Z9tmexfH1YttjmXMOojBvUHIjRS5hdI9NxnPGRLY2kjAzcm
# gV9Rv3vtdF/+zalk3fAVLeK8hXK+di/7XTvYpfJ2EZBWiNrTeagfNNGiYydsQy3zjZAAAA
# wBNRak7UrqnIHMZn7pkCTgceb1MfByaFtlNzd+Obah54HYIQj5WdZTBAITReMZNt9S5NAR
# M8sQB8UoZPaVSC3ppILIOfLhs6KYj6RrGdiYwyIhMPJ5kRWF8xGCLUX5CjwH2EOq7XhIWt
# MwEFtd/gF2Du7HUNFPsZGnzJ3e7pDKDnE7w2khZ8CIpTFgD769uBYGAtk45QYTDo5JroVM
# ZPDq08Gb/RhIgJLmIpMwyreVpLLLe8SwoMJJ+rihmnJZxO8gAAAMEA0lhiKezeTshht4xu
# rWc0NxxD84a29gSGfTphDPOrlKSEYbkSXhjqCsAZHd8S8kMr3iF6poOk3IWSvFJ6mbd3ie
# qdRTgXH9Thwk4KgpjUhNsQuYRHBbI59Mo+BxSI1B1qzmJSGdmCBL54wwzZmFKDQPQKPxiL
# n0Mlc7GooiDMjT1tbuW/O1EL5EqTRqwgWPTKhBA6r4PnGF150hZRIMooZkD2zX6b1sGojk
# QpvKkEykTwnKCzF5TXO8+wJ3qbcEo9AAAAwQD+Z0r68c2YMNpsmyj3ZKtZNPSvJNcLmyD/
# lWoNJq3djJN4s2JbK8l5ARUdW3xSFEDI9yx/wpfsXoaqWnygP3PoFw2CM4i0EiJiyvrLFU
# r3JLfDUFRy3EJ24RsqbigmEsgQOzTl3xfzeFPfxFoOhokSvTG88PQji1AYHz5kA7p6Zfaz
# Ok11rJYIe7+e9B0lhku0AFwGyqlWQmS/MhIpnjHIk5tP4heHGSmzKQWJDbTskNWd6aq1G7
# 6HWfDpX4HgoM8AAAALaG9sbWFuYkBhcmM=
```

(continues on next page)

(continued from previous page)

```
# -----END OPENSSH PRIVATE KEY-----
#
```

Configure instance to be an Ansible controller

```
#cloud-config
#
# Demonstrate setting up an ansible controller host on boot.
# This example installs a playbook repository from a remote private repository
# and then runs two of the plays.

package_update: true
package_upgrade: true
packages:
  - git
  - python3-pip

# Set up an ansible user
# ----------------------
# In this case I give the local ansible user passwordless sudo so that ansible
# may write to a local root-only file.
users:
- name: ansible
  gecos: Ansible User
  shell: /bin/bash
  groups: users,admin,wheel,lxd
  sudo: ALL=(ALL) NOPASSWD:ALL

# Initialize lxd using cloud-init.
# --------------------------------
# In this example, a lxd container is
# started using ansible on boot, so having lxd initialized is required.
lxd:
  init:
    storage_backend: dir

# Configure and run ansible on boot
# ---------------------------------
# Install ansible using pip, ensure that community.general collection is
# installed [1].
# Use a deploy key to clone a remote private repository then run two playbooks.
# The first playbook starts a lxd container and creates a new inventory file.
# The second playbook connects to and configures the container using ansible.
# The public version of the playbooks can be inspected here [2]
#
# [1] community.general is likely already installed by pip
# [2] https://github.com/holmanb/ansible-lxd-public
#
ansible:
  install_method: pip
```

(continues on next page)

(continued from previous page)

```
  package_name: ansible
  run_user: ansible
  galaxy:
    actions:
      - ["ansible-galaxy", "collection", "install", "community.general"]

  setup_controller:
    repositories:
      - path: /home/ansible/my-repo/
        source: git@github.com:holmanb/ansible-lxd-private.git
    run_ansible:
      - playbook_dir: /home/ansible/my-repo
        playbook_name: start-lxd.yml
        timeout: 120
        forks: 1
        private_key: /home/ansible/.ssh/id_rsa
      - playbook_dir: /home/ansible/my-repo
        playbook_name: configure-lxd.yml
        become_user: ansible
        timeout: 120
        forks: 1
        private_key: /home/ansible/.ssh/id_rsa
        inventory: new_ansible_hosts

# Write a deploy key to the filesystem for ansible.
# ------------------------------------------------
# This deploy key is tied to a private github repository [1]
# This key exists to demonstrate deploy key usage in ansible
# a duplicate public copy of the repository exists here[2]
#
# [1] https://github.com/holmanb/ansible-lxd-private
# [2] https://github.com/holmanb/ansible-lxd-public
#
write_files:
  - path: /home/ansible/.ssh/known_hosts
    owner: ansible:ansible
    permissions: 0o600
    defer: true
    content: |
      |1|YJEFAk6JjnXpUjUSLFiBQS55W9E=|OLNePOn3eBa1PWhBBmt5kXsbGM4= ssh-ed25519␣
↪AAAAC3NzaC1lZDI1NTE5AAAAIOMqqnkVzrm0SdG6UOoqKLsabgH5C9okWi0dh2l9GKJl
      |1|PGGnpCpqi0aakERS4BWnYxMkMwM=|Td0piZoS4ZVC0OzeuRwKcH1MusM= ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAQEAq2A7hRGmdnm9tUDbO9IDSwBK6TbQa+PXYPCPy6rbTrTtw7PHkccKrpp0yVhp5HdEIcKr6pLlVD
↪yMf+Se8xhHTvKSCZIFImWwoG6mbUoWf9nzpIoaSjB+weqqUUmpaaasXVal72J+UX2B+2RPW3RcT0eOzQgqlJL3RKrTJvdsjE3JEAvG
↪w4yCE6gbODqnTWlg7+wC604ydGXA8VJiS5ap43JXiUFFAaQ==
      |1|OJ89KrsNcFTOvoCP/fPGKpyUYFo=|cu7mNzF+QB/5kR0spiYmUJL7DAI= ecdsa-sha2-nistp256␣
↪AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBEmKSENjQEezOmxkZMy7opKgwFB9nkt5YRrYMjNuG5N87uRgg
↪y6v0mKV0U2w0WZ2YB/++Tpockg=

  - path: /home/ansible/.ssh/id_rsa
    owner: ansible:ansible
    permissions: 0o600
```

(continues on next page)

(continued from previous page)

```
    defer: true
    encoding: base64
    content: |
      LS0tLS1CRUdJTiBPUEVOU1NIIFBSSVZBVEUgS0VZLS0tLS0KYjNCbGJuTnphQzFyWlhrdGRqRUFB
      QUFBQkc1dmJtVUFBQUFFYm05dVpRQUFBQUFBQUFBQkFBQUJsd0FBQUFkemMyZ3RjbgpOaEFBQUFB
      d0VBQVFBQUFZRUEwUWlRa05WQS9VTEpWZzBzT1Q4TEwyMnRGckg5YVR1SWFNT1FiVFdtWjlNUzJh
      VTZ0cDZoClJDYklWSkhmOHdsaGV3MXNvWmphWVVQSFBsUHNISm5UVlhJTnFTTlpEOGF0Rldjd1gy
      ZTNBOElZNEhpN0NMMDE3MVBoMVUKYmJGNGVIT1JaVkY2VVkzLzhmbXQ3NmhVYnpiRVhkUXhQdVdh
      a0IyemxXNTdFclpOejJhYVdnY2pJUGdHV1RNZWVqbEpOcQpXUW9MNlFzSStpeUlzYXNMc1RTajha
      aVgrT1VjanJEMUY4QXNKS3ZWQStKbnVZNUxFeno1TGQ2SGxGc05XVWtoZkJmOWVOClpxRnJCc1Vw
      M2VUY1FtejFGaHFFWDJIQjNQT3VSTzlKemVGcTJaRE8wUlNQN09acjBMYm8vSFVTK3V5VkJNTDNi
      eEF6dEIKQWM5dFJWZjRqcTJuRjNkcUpwVTFFaXZzR0sxaHJZc0VNQklLK0srVzRwc1F5c3ZTL0ZK
      V2lXZmpqWVMwei9IbkV4MkpHbApOUXUrYkMxL1dXSGVXTGFvNGpSckRSZnNIVnVscTE2MElsbnNx
      eGl1MmNHd081V29Fc1NHdThucXB5ZzQzWkhDYjBGd21CCml6UFFEQVNsbmlXanFjS21mblRycHpB
      eTNlVldhd3dsTnBhUWtpZFRBQUFGZ0dLU2o4ZGlrby9IQUFBQUIzTnphQzF5YzIKRUFBQUdCQU5F
      SWtKRFZRUDFDeVZZTkxEay9DeTl0clJheC9XazdpR2pEa0cwMXBtZlRFdG1sT3JhZW9VUW15RlNS
      My9NSgpZWHNOYktHWTJtRkR4ejVUN0J5WjAxVnlEYWtqV1EvR3JSVm5NRjludHdQQ0dPQjR1d2k5
      TmU5VDRkVkcyeGVIaHprV1ZSCmVsR04vL0g1cmUrb1ZHODJ4RjNVTVQ3bG1wQWRzNVZ1ZXhLMlRj
      OW1tbG9ISXlENEJsa3pIbm81U1RhbGtLQytrTENQb3MKaUxHckM3RTBvL0dZbC9qbEhJNnc5UmZB
      TENTcjFRUGlaN21PU3hNOCtTM2VoNVJiRFZsSklYd1gvWGpXYWhhd2JGS2QzawozRUpzOVJZYWhG
      OWh3ZHp6cmtUdlNjM2hhdG1RenRFVWorem1hOUMyNlB4MUV2cnNsUVRDOTI4UU03UVFIUGJVVlgr
      STZ0CnB4ZDNhaWFWTlJJcjdCaXRZYTJMQkRBU0N2aXZsdUtiRU1yTDB2eFNWb2xuNDQyRXRNL3g1
      eE1kaVJwVFVMdm13dGYxbGgKM2xpMnFPSTBhdzBYN0IxYnBhdGV0Q0paN0tzWXJ0bkJzRHVWcUJM
      RWhydko2cWNvT04yUndtOUJjSmdZc3owQXdFcFo0bApvNm5DcG41MDY2Y3dNdDNsVm1zTUpUYVdr
      SkluVXdBQUFBTUJBQUVBQUFHQUV1ejc3SHU5RUVaeXVqTE9kVG5BVzlhZlJ2ClhET1pBNnBTN3lX
      RXVmanc1Q1NsTUx3aXNSODN5d3cwOXQxUVd5dmhScUV5WW12T0JlY3NYZ2FTVXRuWWZmdFd6NDRh
      cHkKL2dRWXZNVkVMR0thSkFDL3E3dmpNcEd5cnhVUGt5TE1oY2tBTFUyS1lnVisvcmovajZwQk1l
      VmxjaG1rM3Bpa1lyZmZVWApKRFk5OTBXVk8xOTREbTBidUxSekp2Zk1LWUYyQmNmRjRUdmFyak9Y
      V0F4U3VSOHd3dzA1MG9KOEhkS2FoVzdDbTVTMHBvCkZSbk5YRkdNbkxBNjJ2TjAwdkpXOFY3ajd2
      dWk5dWtCYmhqUldhSnVZNXJkRy9VWW16QWU0d3ZkSUVucGs5eEluNkpHQ3AKRlJZVFJuN2xUaDUr
      L1FsUTZGWFJQOElyMXZYWkZuaEt6bDBLOFZxaDJzZjRNNzlNc0lVR0FxR3hnOXhkaGpJYTVkbWdw
      OApOMThJRURvTkVWS1ViS3VLZS9aNXlmOFo5dG1leGZIMVl0dGptWE1Pb2pCdlVISWpSUzVoZEk5
      TnhuUEdSTFkya2pBemNtCmdWOVJ2M3Z0ZEYvK3phbGszZkFWTGVLOGhYSytkaS83WFR2WXBmSjJF
      WkJXaU5yVGVhZ2ZOTkdpWXlkc1F5M3pqWkFBQUEKd0JOUmFrN1VycW5JSE1abjdwa0NUZ2NlYjFN
      ZkJ5YUZ0bE56ZCtPYmFoNTRIWUlRajVXZFpUQkFJVFJlTVpOdDlTNU5BUgpNOHNRQjhVb1pQYVZT
      QzNwcElMSU9mTGhzNktZajZSckdkaVl3eUloTVBKNWtSV0Y4eEdDTFVYNUNqd0gyRU9xN1hoSVd0
      Ck13RUZ0ZC9nRjJEdTdIVU5GUHNaR256SjNlN3BES0RuRTd3MmtoWjhDSXBURmdENzY5dUJZR0F0
      azQ1UVlURG81SnJvVk0KWlBEcTA4R2IvUmhJZ0pMbUlwTXd5cmVWcExMTGU4U3dvTUpKK3JpaG1u
      Slp4TzhnQUFBTUVBMGxoaUtlemVUc2hodDR4dQpyV2MwTnh4RDg0YTI5Z1NHZlRwaERQT3JsS1NF
      WWJrU1hoanFDc0FaSGQ4UzhrTXIzaUY2cG9PazNJV1N2Rko2bWJkM2llCnFkUlRnWEg5VGh3azRL
      Z3BqVWhOc1F1WVJIQmJJNTlNbytCeFNJMUIxcXptSlNHZG1DQkw1NHd3elptRktEUVBRS1B4aUwK
      bjBNbGM3R29vaURNalQxdGJ1Vy9PMUVMNUVxVFJxd2dXUFRLaEJBNnI0UG5HRjE1MGhaUklNb29a
      a0Qyelg2YjFzR29qawpRcHZLa0V5a1R3bktDekY1VFhPOCt3SjNxYmNFbzlBQUFBd1FEK1owcjY4
      YzJZTU5wc215ajNaS3RaTlBTdkpOY0xteUQvCmxXb05KcTNkakpONHMySmJLOGw1QVJVZFczeFNG
      RURJOXl4L3dwZnNYb2FxV255Z1AzUG9GdzJDTTRpMEVpSml5dnJMRlUKcjNKTGZEVUZSeTNFSjI0
      UnNxYmlnbUVzZ1FPelRsM3hmemVGUGZ4Rm9PaG9rU3ZURzg4UFFqaTFBWUh6NWtBN3A2WmZhegpP
      azExckpZSWU3K2U5QjBsaGt1MEFGd0d5cWxXUW1TL01oSXBuakhJazV0UDRoZUhHU216S1FXSkRi
      VHNrTldkNmFxMUc3CjZIV2ZEcFg0SGdvTThBQUFBTGFHOXNiV0Z1WWtCaGNtTT0KLS0tLS1FTkQg
      T1BFTlNTSCBQUklWQVRFIEtFWS0tLS0tCg==
```

Add primary apt repositories

```
#cloud-config

# Add primary apt repositories
#
# To add 3rd party repositories, see cloud-config-apt.txt or the
# Additional apt configuration and repositories section.
#
#
# Default: auto select based on cloud metadata
#  in ec2, the default is <region>.archive.ubuntu.com
# apt:
#   primary:
#     - arches: [default]
#       uri:
#     use the provided mirror
#       search:
#     search the list for the first mirror.
#     this is currently very limited, only verifying that
#     the mirror is dns resolvable or an IP address
#
# if neither mirror is set (the default)
# then use the mirror provided by the DataSource found.
# In EC2, that means using <region>.ec2.archive.ubuntu.com
#
# if no mirror is provided by the DataSource, but 'search_dns' is
# true, then search for dns names '<distro>-mirror' in each of
# - fqdn of this host per cloud metadata
# - localdomain
# - no domain (which would search domains listed in /etc/resolv.conf)
# If there is a dns entry for <distro>-mirror, then it is assumed that there
# is a distro mirror at http://<distro>-mirror.<domain>/<distro>
#
# That gives the cloud provider the opportunity to set mirrors of a distro
# up and expose them only by creating dns entries.
#
# if none of that is found, then the default distro mirror is used
apt:
  primary:
    - arches: [default]
      uri: http://us.archive.ubuntu.com/ubuntu/
# or
apt:
  primary:
    - arches: [default]
      search:
        - http://local-mirror.mydomain
        - http://archive.ubuntu.com
# or
apt:
  primary:
    - arches: [default]
```

(continues on next page)

(continued from previous page)

```
      search_dns: True
```

Run commands on first boot

```
#cloud-config

# boot commands
# default: none
# This is very similar to runcmd, but commands run very early
# in the boot process, only slightly after a 'boothook' would run.
# - bootcmd will run on every boot
# - INSTANCE_ID variable will be set to the current instance ID
# - 'cloud-init-per' command can be used to make bootcmd run exactly once
bootcmd:
  - echo 192.168.1.130 us.archive.ubuntu.com >> /etc/hosts
  - [ cloud-init-per, once, mymkfs, mkfs, /dev/vdb ]
```

```
#cloud-config

# run commands
# default: none
# runcmd contains a list of either lists or a string
# each item will be executed in order at rc.local like level with
# output to the console
# - runcmd only runs during the first boot
# - if the item is a list, the items will be properly executed as if
#   passed to execve(3) (with the first arg as the command).
# - if the item is a string, it will be simply written to the file and
#   will be interpreted by 'sh'
#
# Note, that the list has to be proper yaml, so you have to quote
# any characters yaml would eat (':' can be problematic)
runcmd:
 - [ ls, -l, / ]
 - [ sh, -xc, "echo $(date) ': hello world!'" ]
 - [ sh, -c, echo "==========hello world==========" ]
 - ls -l /root
 # Note: Don't write files to /tmp from cloud-init use /run/somedir instead.
 # Early boot environments can race systemd-tmpfiles-clean LP: #1707222.
 - mkdir /run/mydir
 - [ wget, "http://slashdot.org", -O, /run/mydir/index.html ]
```

Run commands on very early at every boot

```
#cloud-boothook
#!/bin/sh
echo 192.168.1.130 us.archive.ubuntu.com > /etc/hosts
```

Install arbitrary packages

```
#cloud-config

# Install additional packages on first boot
#
# Default: none
#
# if packages are specified, then package_update will be set to true
#
# packages may be supplied as a single package name or as a list
# with the format [<package>, <version>] wherein the specific
# package version will be installed.
packages:
 - pwgen
 - pastebinit
 - [libpython2.7, 2.7.3-0ubuntu3.1]
```

Update apt database on first boot

```
#cloud-config
# Update apt database on first boot (run 'apt-get update').
# Note, if packages are given, or package_upgrade is true, then
# update will be done independent of this setting.
#
# Default: false
package_update: true
```

Run apt or yum upgrade

```
#cloud-config

# Upgrade the instance on first boot
#
# Default: false
package_upgrade: true
```

Adjust mount points mounted

```
#cloud-config

# set up mount points
# 'mounts' contains a list of lists
#  the inner list are entries for an /etc/fstab line
#  ie : [ fs_spec, fs_file, fs_vfstype, fs_mntops, fs-freq, fs_passno ]
#
# default:
# mounts:
#  - [ ephemeral0, /mnt ]
#  - [ swap, none, swap, sw, 0, 0 ]
#
# in order to remove a previously listed mount (ie, one from defaults)
# list only the fs_spec.  For example, to override the default, of
# mounting swap:
# - [ swap ]
# or
# - [ swap, null ]
#
# - if a device does not exist at the time, an entry will still be
#   written to /etc/fstab.
# - '/dev' can be omitted for device names that begin with: xvd, sd, hd, vd
# - if an entry does not have all 6 fields, they will be filled in
#   with values from 'mount_default_fields' below.
#
# Note, that you should set 'nofail' (see man fstab) for volumes that may not
# be attached at instance boot (or reboot).
#
mounts:
 - [ ephemeral0, /mnt, auto, "defaults,noexec" ]
 - [ sdc, /opt/data ]
 - [ xvdh, /opt/data, "auto", "defaults,nofail", "0", "0" ]
 - [ dd, /dev/zero ]

# mount_default_fields
# These values are used to fill in any entries in 'mounts' that are not
# complete.  This must be an array, and must have 6 fields.
mount_default_fields: [ None, None, "auto", "defaults,nofail", "0", "2" ]


# swap can also be set up by the 'mounts' module
# default is to not create any swap files, because 'size' is set to 0
swap:
  filename: /swap.img
  size: "auto" # or size in bytes
  maxsize: 10485760   # size in bytes
```

Configure instance's SSH keys

```
#cloud-config

# add each entry to ~/.ssh/authorized_keys for the configured user or the
# first user defined in the user definition directive.
ssh_authorized_keys:
  - ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAGEA3FSyQwBI6Z+nCSjUUk8EEAnnkhXlukKoUPND/
↪RRClWz2s5TCzIkd3Ou5+Cyz71X0XmazM3l5WgeErvtIwQMyT1KjNoMhoJMrJnWqQPOt5Q8zWd9qG7PBl9+eiH5qV7NZ␣
↪mykey@host
  - ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAQEA3I7VUf2l5gSn5uavROsc5HRDpZdQueUq5ozemNSj8T7enqKHOEaFoU2VoPgGEWC9RyzSQVeyD
↪+i1D+ey3ONkZLN+LQ714cgj8fRS4Hj29SCmXp5Kt5/82cD/VN3NtHw== smoser@brickies

# Send pre-generated SSH private keys to the server
# If these are present, they will be written to /etc/ssh and
# new random keys will not be generated
#  in addition to 'rsa' as shown below, 'ecdsa' is also supported
ssh_keys:
  rsa_private: |
    -----BEGIN RSA PRIVATE KEY-----
    MIIBxwIBAAJhAKD0YSHy73nUgysO13XsJmd4fHiFyQ+00R7VVu2iV9Qcon2LZS/x
    1cydPZ4pQpfjEha6WxZ6o8ci/Ea/w0n+0HGPwaxlEG2Z9inNtj3pgFrYcRztfECb
    1j6HCibZbAzYtwIBIwJgO8h72WjcmvcpZ8OvHSvTwAguO2TkR6mPgHsgSaKy6GJo
    PUJnaZRWuba/HX0KGyhz19nPzLpzG5f0fYahlMJAyc13FV7K6kMBPXTRR6FxgHEg
    L0MPC7cdqAwOVNcPY6A7AjEA1bNaIjOzFN2sfZX0j7OMhQuc4zP7r80zaGc5oy6W
    p58hRAncFKEvnEq2CeL3vtuZAjEAwNBHpbNsBYTRPCHM7rZuG/iBtwp8Rxhc9I5w
    ixvzMgi+HpGLWzUIBS+P/XhekIjPAjA285rVmEP+DR255Ls65QbgYhJmTzIXQ2T9
    luLvcmFBC6l35Uc4gTgg4ALsmXLn71MCMGMpSWspEvuGInayTCL+vEjmNBT+FAdO
    W7D4zCpI43jRS9U06JVOeSc9CDk2lwiA3wIwCTB/6uc8Cq85D9YqpM10FuHjKpnP
    REPPOyrAspdeOAV+6VKRavstea7+2DZmSUgE
    -----END RSA PRIVATE KEY-----

  rsa_public: ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAGEAoPRhIfLvedSDKw7XdewmZ3h8eIXJD7TRHtVW7aJX1ByifYtlL/
↪HVzJ09nilCl+MSFrpbFnqjxyL8Rr/DSf7QcY/BrGUQbZn2Kc22PemAWthxHO18QJvWPocKJtlsDNi3␣
↪smoser@localhost

# By default, the fingerprints of the authorized keys for the users
# cloud-init adds are printed to the console. Setting
# no_ssh_fingerprints to true suppresses this output.
no_ssh_fingerprints: false

# By default, (most) ssh host keys are printed to the console. Setting
# emit_keys_to_console to false suppresses this output.
ssh:
  emit_keys_to_console: false
```

Additional apt configuration and repositories

```
#cloud-config
# apt_pipelining (configure Acquire::http::Pipeline-Depth)
# Default: disables HTTP pipelining. Certain web servers, such
# as S3 do not pipeline properly (LP: #948461).
# Valid options:
#   False/default: Disables pipelining for APT
#   None/Unchanged: Use OS default
#   Number: Set pipelining to some number (not recommended)
apt_pipelining: False

# Install additional packages on first boot
#
# Default: none
#
# if packages are specified, then package_update will be set to true

packages: ['pastebinit']

apt:
  # The apt config consists of two major "areas".
  #
  # On one hand there is the global configuration for the apt feature.
  #
  # On one hand (down in this file) there is the source dictionary which allows
  # to define various entries to be considered by apt.

  ############################################################################
  # Section 1: global apt configuration
  #
  # The following examples number the top keys to ease identification in
  # discussions.

  # 1.1 preserve_sources_list
  #
  # Preserves the existing /etc/apt/sources.list
  # Default: false - do overwrite sources_list. If set to true then any
  # "mirrors" configuration will have no effect.
  # Set to true to avoid affecting sources.list. In that case only
  # "extra" source specifications will be written into
  # /etc/apt/sources.list.d/*
  preserve_sources_list: true

  # 1.2 disable_suites
  #
  # This is an empty list by default, so nothing is disabled.
  #
  # If given, those suites are removed from sources.list after all other
  # modifications have been made.
  # Suites are even disabled if no other modification was made,
  # but not if is preserve_sources_list is active.
  # There is a special alias "$RELEASE" as in the sources that will be replace
```

(continues on next page)

(continued from previous page)

```
  # by the matching release.
  #
  # To ease configuration and improve readability the following common ubuntu
  # suites will be automatically mapped to their full definition.
  # updates   => $RELEASE-updates
  # backports => $RELEASE-backports
  # security  => $RELEASE-security
  # proposed  => $RELEASE-proposed
  # release   => $RELEASE
  #
  # There is no harm in specifying a suite to be disabled that is not found in
  # the source.list file (just a no-op then)
  #
  # Note: Lines don't get deleted, but disabled by being converted to a comment.
  # The following example disables all usual defaults except $RELEASE-security.
  # On top it disables a custom suite called "mysuite"
  disable_suites: [$RELEASE-updates, backports, $RELEASE, mysuite]

  # 1.3 primary/security archives
  #
  # Default: none - instead it is auto select based on cloud metadata
  # so if neither "uri" nor "search", nor "search_dns" is set (the default)
  # then use the mirror provided by the DataSource found.
  # In EC2, that means using <region>.ec2.archive.ubuntu.com
  #
  # define a custom (e.g. localized) mirror that will be used in sources.list
  # and any custom sources entries for deb / deb-src lines.
  #
  # One can set primary and security mirror to different uri's
  # the child elements to the keys primary and secondary are equivalent
  primary:
    # arches is list of architectures the following config applies to
    # the special keyword "default" applies to any architecture not explicitly
    # listed.
    - arches: [amd64, i386, default]
      # uri is just defining the target as-is
      uri: http://us.archive.ubuntu.com/ubuntu
      #
      # via search one can define lists that are tried one by one.
      # The first with a working DNS resolution (or if it is an IP) will be
      # picked. That way one can keep one configuration for multiple
      # subenvironments that select the working one.
      search:
        - http://cool.but-sometimes-unreachable.com/ubuntu
        - http://us.archive.ubuntu.com/ubuntu
      # if no mirror is provided by uri or search but 'search_dns' is
      # true, then search for dns names '<distro>-mirror' in each of
      # - fqdn of this host per cloud metadata
      # - localdomain
      # - no domain (which would search domains listed in /etc/resolv.conf)
      # If there is a dns entry for <distro>-mirror, then it is assumed that
      # there is a distro mirror at http://<distro>-mirror.<domain>/<distro>
```

(continues on next page)

(continued from previous page)

```
    #
    # That gives the cloud provider the opportunity to set mirrors of a distro
    # up and expose them only by creating dns entries.
    #
    # if none of that is found, then the default distro mirror is used
    search_dns: true
    #
    # If multiple of a category are given
    #   1. uri
    #   2. search
    #   3. search_dns
    # the first defining a valid mirror wins (in the order as defined here,
    # not the order as listed in the config).
    #
    # Additionally, if the repository requires a custom signing key, it can be
    # specified via the same fields as for custom sources:
    #   'keyid': providing a key to import via shortid or fingerprint
    #   'key': providing a raw PGP key
    #   'keyserver': specify an alternate keyserver to pull keys from that
    #                were specified by keyid
  - arches: [s390x, arm64]
    # as above, allowing to have one config for different per arch mirrors
# security is optional, if not defined it is set to the same value as primary
security:
  - uri: http://security.ubuntu.com/ubuntu
    arches: [default]
# If search_dns is set for security the searched pattern is:
#   <distro>-security-mirror

# if no mirrors are specified at all, or all lookups fail it will try
# to get them from the cloud datasource and if those neither provide one fall
# back to:
#   primary: http://archive.ubuntu.com/ubuntu
#   security: http://security.ubuntu.com/ubuntu

# 1.4 sources_list
#
# Provide a custom template for rendering sources.list
# without one provided cloud-init uses builtin templates for
# ubuntu and debian.
# Within these sources.list templates you can use the following replacement
# variables (all have sane Ubuntu defaults, but mirrors can be overwritten
# as needed (see above)):
# => $RELEASE, $MIRROR, $PRIMARY, $SECURITY
sources_list: | # written by cloud-init custom template
  deb $MIRROR $RELEASE main restricted
  deb-src $MIRROR $RELEASE main restricted
  deb $PRIMARY $RELEASE universe restricted
  deb $SECURITY $RELEASE-security multiverse

# 1.5 conf
#
```

(continues on next page)

(continued from previous page)

```
  # Any apt config string that will be made available to apt
  # see the APT.CONF(5) man page for details what can be specified
  conf: | # APT config
    APT {
      Get {
        Assume-Yes "true";
        Fix-Broken "true";
      };
    };

  # 1.6 (http_|ftp_|https_)proxy
  #
  # Proxies are the most common apt.conf option, so that for simplified use
  # there is a shortcut for those. Those get automatically translated into the
  # correct Acquire::*::Proxy statements.
  #
  # note: proxy actually being a short synonym to http_proxy
  proxy: http://[[user][:pass]@]host[:port]/
  http_proxy: http://[[user][:pass]@]host[:port]/
  ftp_proxy: ftp://[[user][:pass]@]host[:port]/
  https_proxy: https://[[user][:pass]@]host[:port]/

  # 1.7 add_apt_repo_match
  #
  # 'source' entries in apt-sources that match this python regex
  # expression will be passed to add-apt-repository
  # The following example is also the builtin default if nothing is specified
  add_apt_repo_match: '^[\w-]+:\w'


  ############################################################################
  # Section 2: source list entries
  #
  # This is a dictionary (unlike most block/net which are lists)
  #
  # The key of each source entry is the filename and will be prepended by
  # /etc/apt/sources.list.d/ if it doesn't start with a '/'.
  # If it doesn't end with .list it will be appended so that apt picks up its
  # configuration.
  #
  # Whenever there is no content to be written into such a file, the key is
  # not used as filename - yet it can still be used as index for merging
  # configuration.
  #
  # The values inside the entries consist of the following optional entries:
  #   'source': a sources.list entry (some variable replacements apply)
  #   'keyid': providing a key to import via shortid or fingerprint
  #   'key': providing a raw PGP key
  #   'keyserver': specify an alternate keyserver to pull keys from that
  #                were specified by keyid

  # This allows merging between multiple input files than a list like:
```

(continues on next page)

(continued from previous page)

```
# cloud-config1
# sources:
#   s1: {'key': 'key1', 'source': 'source1'}
# cloud-config2
# sources:
#   s2: {'key': 'key2'}
#   s1: {'keyserver': 'foo'}
# This would be merged to
# sources:
#   s1:
#     keyserver: foo
#     key: key1
#     source: source1
#   s2:
#     key: key2
#
# The following examples number the subfeatures per sources entry to ease
# identification in discussions.


sources:
  curtin-dev-ppa.list:
    # 2.1 source
    #
    # Creates a file in /etc/apt/sources.list.d/ for the sources list entry
    # based on the key: "/etc/apt/sources.list.d/curtin-dev-ppa.list"
    source: "deb http://ppa.launchpad.net/curtin-dev/test-archive/ubuntu bionic main"

    # 2.2 keyid
    #
    # Importing a gpg key for a given key id. Used keyserver defaults to
    # keyserver.ubuntu.com
    keyid: F430BBA5 # GPG key ID published on a key server

  ignored1:
    # 2.3 PPA shortcut
    #
    # Setup correct apt sources.list line and Auto-Import the signing key
    # from LP
    #
    # See https://help.launchpad.net/Packaging/PPA for more information
    # this requires 'add-apt-repository'. This will create a file in
    # /etc/apt/sources.list.d automatically, therefore the key here is
    # ignored as filename in those cases.
    source: "ppa:curtin-dev/test-archive"    # Quote the string

  my-repo2.list:
    # 2.4 replacement variables
    #
    # sources can use $MIRROR, $PRIMARY, $SECURITY, $RELEASE and $KEY_FILE
    # replacement variables.
    # They will be replaced with the default or specified mirrors and the
```

(continues on next page)

(continued from previous page)

```
    # running release.
    # The entry below would be possibly turned into:
    #   source: deb http://archive.ubuntu.com/ubuntu bionic multiverse
    source: deb [signed-by=$KEY_FILE] $MIRROR $RELEASE multiverse
    keyid: F430BBA5

  my-repo3.list:
    # this would have the same end effect as 'ppa:curtin-dev/test-archive'
    source: "deb http://ppa.launchpad.net/curtin-dev/test-archive/ubuntu bionic main"
    keyid: F430BBA5 # GPG key ID published on the key server
    filename: curtin-dev-ppa.list

  ignored2:
    # 2.5 key only
    #
    # this would only import the key without adding a ppa or other source spec
    # since this doesn't generate a source.list file the filename key is ignored
    keyid: F430BBA5 # GPG key ID published on a key server

  ignored3:
    # 2.6 key id alternatives
    #
    # Keyid's can also be specified via their long fingerprints
    keyid: B59D 5F15 97A5 04B7 E230  6DCA 0620 BBCF 0368 3F77

  ignored4:
    # 2.7 alternative keyservers
    #
    # One can also specify alternative keyservers to fetch keys from.
    keyid: B59D 5F15 97A5 04B7 E230  6DCA 0620 BBCF 0368 3F77
    keyserver: pgp.mit.edu

  ignored5:
    # 2.8 signed-by
    #
    # One can specify [signed-by=$KEY_FILE] in the source definition, which
    # will make the key be installed in the directory /etc/cloud-init.gpg.d/
    # and the $KEY_FILE replacement variable will be replaced with the path
    # to the specified key. If $KEY_FILE is used, but no key is specified,
    # apt update will (rightfully) fail due to an invalid value.
    source: deb [signed-by=$KEY_FILE] $MIRROR $RELEASE multiverse
    keyid: B59D 5F15 97A5 04B7 E230  6DCA 0620 BBCF 0368 3F77

  my-repo4.list:
    # 2.9 raw key
    #
    # The apt signing key can also be specified by providing a pgp public key
    # block. Providing the PGP key this way is the most robust method for
    # specifying a key, as it removes dependency on a remote key server.
    #
    # As with keyid's this can be specified with or without some actual source
    # content.
```

(continues on next page)

(continued from previous page)

```
      key: | # The value needs to start with -----BEGIN PGP PUBLIC KEY BLOCK-----
        -----BEGIN PGP PUBLIC KEY BLOCK-----
        Version: SKS 1.0.10

        mI0ESpA3UQEEALdZKVIMq0j6qWAXAyxSlF63SvPVIgxHPb9Nk0DZUixn+akqytxG4zKCONz6
        qLjoBBfHnynyVLfT4ihg9an1PqxRnTO+JKQxl8NgKGz6Pon569GtAOdWNKw15XKinJTDLjnj
        9y96ljJqRcpV9t/WsIcdJPcKFR5voHTEoABE2aEXABEBAAG0GUxhdW5jaHBhZCBQUEEgZm9y
        IEFsZXN0aWOItgQTAQIAIAUCSpA3UQIbAwYLCQgHAwIEFQIIAwQWAgMBAh4BAheAAAoJEA7H
        5Qi+CcVxWZ8D/1MyYvfj3FJPZUm2Yo1zZsQ657vHI9+pPouqflWOayRR9jbiyUFIn0VdQBrP
        t0FwvnOFArUovUWoKAEdqR8hPy3M3APUZjl5K4cMZR/xaMQeQRZ5CHpS4DBKURKAHC0ltS5o
        uBJKQOZm5iltJp15cgyIkBkGe8Mx18VFyVglAZey
        =Y2oI
        -----END PGP PUBLIC KEY BLOCK-----
```

Disk setup

```
#cloud-config
# Cloud-init supports the creation of simple partition tables and filesystems
# on devices.

# Default disk definitions for AWS
# --------------------------------
# (Not implemented yet, but provided for future documentation)

disk_setup:
  ephemeral0:
    table_type: 'mbr'
    layout: True
    overwrite: False

fs_setup:
  - label: None,
    filesystem: ext3
    device: ephemeral0
    partition: auto

# Default disk definitions for Microsoft Azure
# ------------------------------------------

device_aliases: {'ephemeral0': '/dev/sdb'}
disk_setup:
  ephemeral0:
    table_type: mbr
    layout: True
    overwrite: False

fs_setup:
  - label: ephemeral0
    filesystem: ext4
    device: ephemeral0.1
```

(continues on next page)

(continued from previous page)

```
    replace_fs: ntfs


# Data disks definitions for Microsoft Azure
# ------------------------------------------

disk_setup:
  /dev/disk/azure/scsi1/lun0:
    table_type: gpt
    layout: True
    overwrite: True

fs_setup:
  - device: /dev/disk/azure/scsi1/lun0
    partition: 1
    filesystem: ext4


# Default disk definitions for SmartOS
# ------------------------------------

device_aliases: {'ephemeral0': '/dev/vdb'}
disk_setup:
  ephemeral0:
    table_type: mbr
    layout: False
    overwrite: False

fs_setup:
  - label: ephemeral0
    filesystem: ext4
    device: ephemeral0.0

# Caveat for SmartOS: if ephemeral disk is not defined, then the disk will
#    not be automatically added to the mounts.


# The default definition is used to make sure that the ephemeral storage is
# setup properly.

# "disk_setup": disk partitioning
# --------------------------------

# The disk_setup directive instructs Cloud-init to partition a disk. The format is:

disk_setup:
  ephemeral0:
    table_type: 'mbr'
    layout: true
  /dev/xvdh:
    table_type: 'mbr'
    layout:
```

(continues on next page)

(continued from previous page)

```
      - 33
      - [33, 82]
      - 33
    overwrite: True

# The format is a list of dicts of dicts. The first value is the name of the
# device and the subsequent values define how to create and layout the
# partition.
# The general format is:
#   disk_setup:
#     <DEVICE>:
#       table_type: 'mbr'
#       layout: <LAYOUT|BOOL>
#       overwrite: <BOOL>
#
# Where:
#   <DEVICE>: The name of the device. 'ephemeralX' and 'swap' are special
#               values which are specific to the cloud. For these devices
#               Cloud-init will look up what the real devices is and then
#               use it.
#
#               For other devices, the kernel device name is used. At this
#               time only simply kernel devices are supported, meaning
#               that device mapper and other targets may not work.
#
#               Note: At this time, there is no handling or setup of
#               device mapper targets.
#
#   table_type=<TYPE>: Currently the following are supported:
#                   'mbr': default and setups a MS-DOS partition table
#                   'gpt': setups a GPT partition table
#
#               Note: At this time only 'mbr' and 'gpt' partition tables
#                   are allowed. It is anticipated in the future that
#                   we'll also have "RAID" to create a mdadm RAID.
#
#   layout={...}: The device layout. This is a list of values, with the
#               percentage of disk that partition will take.
#               Valid options are:
#                   [<SIZE>, [<SIZE>, <PART_TYPE]]
#
#               Where <SIZE> is the _percentage_ of the disk to use, while
#               <PART_TYPE> is the numerical value of the partition type.
#
#               The following setups two partitions, with the first
#               partition having a swap label, taking 1/3 of the disk space
#               and the remainder being used as the second partition.
#                 /dev/xvdh':
#                   table_type: 'mbr'
#                   layout:
#                     - [33,82]
#                     - 66
```

(continues on next page)

(continued from previous page)

```
#                   overwrite: True
#
#               When layout is "true" it means single partition the entire
#               device.
#
#               When layout is "false" it means don't partition or ignore
#               existing partitioning.
#
#               If layout is set to "true" and overwrite is set to "false",
#               it will skip partitioning the device without a failure.
#
#   overwrite=<BOOL>: This describes whether to ride with safetys on and
#               everything holstered.
#
#               'false' is the default, which means that:
#                   1. The device will be checked for a partition table
#                   2. The device will be checked for a filesystem
#                   3. If either a partition of filesystem is found, then
#                       the operation will be _skipped_.
#
#               'true' is cowboy mode. There are no checks and things are
#                   done blindly. USE with caution, you can do things you
#                   really, really don't want to do.
#
#
# fs_setup: Setup the filesystem
# ------------------------------
#
# fs_setup describes the how the filesystems are supposed to look.

fs_setup:
  - label: ephemeral0
    filesystem: 'ext3'
    device: 'ephemeral0'
    partition: 'auto'
  - label: mylabl2
    filesystem: 'ext4'
    device: '/dev/xvda1'
  - cmd: mkfs -t %(filesystem)s -L %(label)s %(device)s
    label: mylabl3
    filesystem: 'btrfs'
    device: '/dev/xvdh'

# The general format is:
#   fs_setup:
#     - label: <LABEL>
#       filesystem: <FS_TYPE>
#       device: <DEVICE>
#       partition: <PART_VALUE>
#       overwrite: <OVERWRITE>
#       replace_fs: <FS_TYPE>
#
```

(continues on next page)

(continued from previous page)

```
# Where:
#   <LABEL>: The filesystem label to be used. If set to None, no label is
#     used.
#
#   <FS_TYPE>: The filesystem type. It is assumed that the there
#     will be a "mkfs.<FS_TYPE>" that behaves likes "mkfs". On a standard
#     Ubuntu Cloud Image, this means that you have the option of ext{2,3,4},
#     and vfat by default.
#
#   <DEVICE>: The device name. Special names of 'ephemeralX' or 'swap'
#     are allowed and the actual device is acquired from the cloud datasource.
#     When using 'ephemeralX' (i.e. ephemeral0), make sure to leave the
#     label as 'ephemeralX' otherwise there may be issues with the mounting
#     of the ephemeral storage layer.
#
#     If you define the device as 'ephemeralX.Y' then Y will be interpetted
#     as a partition value. However, ephermalX.0 is the _same_ as ephemeralX.
#
#   <PART_VALUE>:
#     Partition definitions are overwritten if you use the '<DEVICE>.Y' notation.
#
#     The valid options are:
#     "auto|any": tell cloud-init not to care whether there is a partition
#       or not. Auto will use the first partition that does not contain a
#       filesystem already. In the absence of a partition table, it will
#       put it directly on the disk.
#
#       "auto": If a filesystem that matches the specification in terms of
#       label, filesystem and device, then cloud-init will skip the creation
#       of the filesystem.
#
#       "any": If a filesystem that matches the filesystem type and device,
#       then cloud-init will skip the creation of the filesystem.
#
#       Devices are selected based on first-detected, starting with partitions
#       and then the raw disk. Consider the following:
#           NAME     FSTYPE LABEL
#           xvdb
#           |-xvdb1  ext4
#           |-xvdb2
#           |-xvdb3  btrfs  test
#           \-xvdb4  ext4   test
#
#         If you ask for 'auto', label of 'test, and filesystem of 'ext4'
#         then cloud-init will select the 2nd partition, even though there
#         is a partition match at the 4th partition.
#
#         If you ask for 'any' and a label of 'test', then cloud-init will
#         select the 1st partition.
#
#         If you ask for 'auto' and don't define label, then cloud-init will
#         select the 1st partition.
```

(continues on next page)

(continued from previous page)

```
#
#         In general, if you have a specific partition configuration in mind,
#         you should define either the device or the partition number. 'auto'
#         and 'any' are specifically intended for formatting ephemeral storage
#         or for simple schemes.
#
#       "none": Put the filesystem directly on the device.
#
#       <NUM>: where NUM is the actual partition number.
#
#   <OVERWRITE>: Defines whether or not to overwrite any existing
#     filesystem.
#
#     "true": Indiscriminately destroy any pre-existing filesystem. Use at
#         your own peril.
#
#     "false": If an existing filesystem exists, skip the creation.
#
#   <REPLACE_FS>: This is a special directive, used for Microsoft Azure that
#     instructs cloud-init to replace a filesystem of <FS_TYPE>. NOTE:
#     unless you define a label, this requires the use of the 'any' partition
#     directive.
#
# Behavior Caveat: The default behavior is to _check_ if the filesystem exists.
#   If a filesystem matches the specification, then the operation is a no-op.
```

Configure data sources

```
#cloud-config

# Documentation on data sources configuration options
datasource:
  # Ec2
  Ec2:
    # timeout: the timeout value for a request at metadata service
    timeout : 50
    # The length in seconds to wait before giving up on the metadata
    # service.  The actual total wait could be up to
    #   len(resolvable_metadata_urls)*timeout
    max_wait : 120

    #metadata_url: a list of URLs to check for metadata services
    metadata_urls:
     - http://169.254.169.254:80
     - http://instance-data:8773

  MAAS:
    timeout : 50
    max_wait : 120

```

(continues on next page)

(continued from previous page)

```
  # there are no default values for metadata_url or oauth credentials
  # If no credentials are present, non-authed attempts will be made.
  metadata_url: http://mass-host.localdomain/source
  consumer_key: Xh234sdkljf
  token_key: kjfhgb3n
  token_secret: 24uysdfx1w4

NoCloud:
  # default seedfrom is None
  # if found, then it should contain a url with:
  #    <url>/user-data and <url>/meta-data
  # seedfrom: http://my.example.com/i-abcde/
  seedfrom: None

  # these are optional, but allow you to basically provide a datasource
  # right here
  user-data: |
    # This is the user-data verbatim
  meta-data: |
    instance-id: i-87018aed
    local-hostname: myhost.internal

SmartOS:
  # For KVM guests:
  # Smart OS datasource works over a serial console interacting with
  # a server on the other end. By default, the second serial console is the
  # device. SmartOS also uses a serial timeout of 60 seconds.
  serial_device: /dev/ttyS1
  serial_timeout: 60

  # For LX-Brand Zones guests:
  # Smart OS datasource works over a socket interacting with
  # the host on the other end. By default, the socket file is in
  # the native .zoncontrol directory.
  metadata_sockfile: /native/.zonecontrol/metadata.sock

  # a list of keys that will not be base64 decoded even if base64_all
  no_base64_decode: ['root_authorized_keys', 'motd_sys_info',
                     'iptables_disable']
  # a plaintext, comma delimited list of keys whose values are b64 encoded
  base64_keys: []
  # a boolean indicating that all keys not in 'no_base64_decode' are encoded
  base64_all: False
```

Create partitions and filesystems

```
#cloud-config
# Cloud-init supports the creation of simple partition tables and filesystems
# on devices.

# Default disk definitions for AWS
# --------------------------------
# (Not implemented yet, but provided for future documentation)

disk_setup:
  ephemeral0:
    table_type: 'mbr'
    layout: True
    overwrite: False

fs_setup:
  - label: None,
    filesystem: ext3
    device: ephemeral0
    partition: auto

# Default disk definitions for Microsoft Azure
# ------------------------------------------

device_aliases: {'ephemeral0': '/dev/sdb'}
disk_setup:
  ephemeral0:
    table_type: mbr
    layout: True
    overwrite: False

fs_setup:
  - label: ephemeral0
    filesystem: ext4
    device: ephemeral0.1
    replace_fs: ntfs


# Data disks definitions for Microsoft Azure
# ------------------------------------------

disk_setup:
  /dev/disk/azure/scsi1/lun0:
    table_type: gpt
    layout: True
    overwrite: True

fs_setup:
  - device: /dev/disk/azure/scsi1/lun0
    partition: 1
    filesystem: ext4

```

(continues on next page)

(continued from previous page)

```
52
53  # Default disk definitions for SmartOS
54  # ------------------------------------
55
56  device_aliases: {'ephemeral0': '/dev/vdb'}
57  disk_setup:
58    ephemeral0:
59      table_type: mbr
60      layout: False
61      overwrite: False
62
63  fs_setup:
64    - label: ephemeral0
65      filesystem: ext4
66      device: ephemeral0.0
67
68  # Caveat for SmartOS: if ephemeral disk is not defined, then the disk will
69  #    not be automatically added to the mounts.
70
71
72  # The default definition is used to make sure that the ephemeral storage is
73  # setup properly.
74
75  # "disk_setup": disk partitioning
76  # --------------------------------
77
78  # The disk_setup directive instructs Cloud-init to partition a disk. The format is:
79
80  disk_setup:
81    ephemeral0:
82      table_type: 'mbr'
83      layout: true
84    /dev/xvdh:
85      table_type: 'mbr'
86      layout:
87        - 33
88        - [33, 82]
89        - 33
90      overwrite: True
91
92  # The format is a list of dicts of dicts. The first value is the name of the
93  # device and the subsequent values define how to create and layout the
94  # partition.
95  # The general format is:
96  #   disk_setup:
97  #     <DEVICE>:
98  #       table_type: 'mbr'
99  #       layout: <LAYOUT|BOOL>
100 #       overwrite: <BOOL>
101 #
102 # Where:
103 #   <DEVICE>: The name of the device. 'ephemeralX' and 'swap' are special
```

(continues on next page)

(continued from previous page)

```
#                values which are specific to the cloud. For these devices
#                Cloud-init will look up what the real devices is and then
#                use it.
#
#                For other devices, the kernel device name is used. At this
#                time only simply kernel devices are supported, meaning
#                that device mapper and other targets may not work.
#
#                Note: At this time, there is no handling or setup of
#                device mapper targets.
#
#   table_type=<TYPE>: Currently the following are supported:
#                    'mbr': default and setups a MS-DOS partition table
#                    'gpt': setups a GPT partition table
#
#                Note: At this time only 'mbr' and 'gpt' partition tables
#                    are allowed. It is anticipated in the future that
#                    we'll also have "RAID" to create a mdadm RAID.
#
#   layout={...}: The device layout. This is a list of values, with the
#                percentage of disk that partition will take.
#                Valid options are:
#                    [<SIZE>, [<SIZE>, <PART_TYPE]]
#
#                Where <SIZE> is the _percentage_ of the disk to use, while
#                <PART_TYPE> is the numerical value of the partition type.
#
#                The following setups two partitions, with the first
#                partition having a swap label, taking 1/3 of the disk space
#                and the remainder being used as the second partition.
#                  /dev/xvdh':
#                    table_type: 'mbr'
#                    layout:
#                      - [33,82]
#                      - 66
#                    overwrite: True
#
#                When layout is "true" it means single partition the entire
#                device.
#
#                When layout is "false" it means don't partition or ignore
#                existing partitioning.
#
#                If layout is set to "true" and overwrite is set to "false",
#                it will skip partitioning the device without a failure.
#
#   overwrite=<BOOL>: This describes whether to ride with safetys on and
#                everything holstered.
#
#                'false' is the default, which means that:
#                    1. The device will be checked for a partition table
#                    2. The device will be checked for a filesystem
```

(continues on next page)

(continued from previous page)

```
#                   3. If either a partition of filesystem is found, then
#                       the operation will be _skipped_.
#
#               'true' is cowboy mode. There are no checks and things are
#                   done blindly. USE with caution, you can do things you
#                   really, really don't want to do.
#
#
# fs_setup: Setup the filesystem
# ------------------------------
#
# fs_setup describes the how the filesystems are supposed to look.

fs_setup:
  - label: ephemeral0
    filesystem: 'ext3'
    device: 'ephemeral0'
    partition: 'auto'
  - label: mylabl2
    filesystem: 'ext4'
    device: '/dev/xvda1'
  - cmd: mkfs -t %(filesystem)s -L %(label)s %(device)s
    label: mylabl3
    filesystem: 'btrfs'
    device: '/dev/xvdh'

# The general format is:
#   fs_setup:
#     - label: <LABEL>
#       filesystem: <FS_TYPE>
#       device: <DEVICE>
#       partition: <PART_VALUE>
#       overwrite: <OVERWRITE>
#       replace_fs: <FS_TYPE>
#
# Where:
#   <LABEL>: The filesystem label to be used. If set to None, no label is
#     used.
#
#   <FS_TYPE>: The filesystem type. It is assumed that the there
#     will be a "mkfs.<FS_TYPE>" that behaves likes "mkfs". On a standard
#     Ubuntu Cloud Image, this means that you have the option of ext{2,3,4},
#     and vfat by default.
#
#   <DEVICE>: The device name. Special names of 'ephemeralX' or 'swap'
#     are allowed and the actual device is acquired from the cloud datasource.
#     When using 'ephemeralX' (i.e. ephemeral0), make sure to leave the
#     label as 'ephemeralX' otherwise there may be issues with the mounting
#     of the ephemeral storage layer.
#
#     If you define the device as 'ephemeralX.Y' then Y will be interpetted
#     as a partition value. However, ephermalX.0 is the _same_ as ephemeralX.
```

(continues on next page)

(continued from previous page)

```
#
#   <PART_VALUE>:
#     Partition definitions are overwritten if you use the '<DEVICE>.Y' notation.
#
#     The valid options are:
#     "auto|any": tell cloud-init not to care whether there is a partition
#       or not. Auto will use the first partition that does not contain a
#       filesystem already. In the absence of a partition table, it will
#       put it directly on the disk.
#
#       "auto": If a filesystem that matches the specification in terms of
#       label, filesystem and device, then cloud-init will skip the creation
#       of the filesystem.
#
#       "any": If a filesystem that matches the filesystem type and device,
#       then cloud-init will skip the creation of the filesystem.
#
#       Devices are selected based on first-detected, starting with partitions
#       and then the raw disk. Consider the following:
#           NAME     FSTYPE LABEL
#           xvdb
#           |-xvdb1  ext4
#           |-xvdb2
#           |-xvdb3  btrfs  test
#           \-xvdb4  ext4   test
#
#         If you ask for 'auto', label of 'test, and filesystem of 'ext4'
#         then cloud-init will select the 2nd partition, even though there
#         is a partition match at the 4th partition.
#
#         If you ask for 'any' and a label of 'test', then cloud-init will
#         select the 1st partition.
#
#         If you ask for 'auto' and don't define label, then cloud-init will
#         select the 1st partition.
#
#         In general, if you have a specific partition configuration in mind,
#         you should define either the device or the partition number. 'auto'
#         and 'any' are specifically intended for formatting ephemeral storage
#         or for simple schemes.
#
#       "none": Put the filesystem directly on the device.
#
#       <NUM>: where NUM is the actual partition number.
#
#   <OVERWRITE>: Defines whether or not to overwrite any existing
#     filesystem.
#
#     "true": Indiscriminately destroy any pre-existing filesystem. Use at
#         your own peril.
#
#     "false": If an existing filesystem exists, skip the creation.
```

(continues on next page)

(continued from previous page)

```
#
#   <REPLACE_FS>: This is a special directive, used for Microsoft Azure that
#     instructs cloud-init to replace a filesystem of <FS_TYPE>. NOTE:
#     unless you define a label, this requires the use of the 'any' partition
#     directive.
#
# Behavior Caveat: The default behavior is to _check_ if the filesystem exists.
#   If a filesystem matches the specification, then the operation is a no-op.
```

2.4.3 CLI commands

For the latest list of subcommands and arguments use `cloud-init`'s `--help` option. This can be used against `cloud-init` itself, or on any of its subcommands.

```
$ cloud-init --help
```

Example output:

```
usage: cloud-init [-h] [--version] [--debug] [--force]
                                                    {init,modules,single,query,
↪features,analyze,devel,collect-logs,clean,status,schema} ...

 options:
   -h, --help            show this help message and exit
   --version, -v         Show program's version number and exit.
   --debug, -d           Show additional pre-action logging (default: False).
   --force               Force running even if no datasource is found (use at your own␣
↪risk).

 Subcommands:
   {init,modules,single,query,features,analyze,devel,collect-logs,clean,status,schema}
     init                Initialize cloud-init and perform initial modules.
     modules             Activate modules using a given configuration key.
     single              Run a single module.
     query               Query standardized instance metadata from the command line.
     features            List defined features.
     analyze             Devel tool: Analyze cloud-init logs and data.
     devel               Run development tools.
     collect-logs        Collect and tar all cloud-init debug info.
     clean               Remove logs and artifacts so cloud-init can re-run.
     status              Report cloud-init status or wait on completion.
     schema              Validate cloud-config files using jsonschema.
```

The rest of this document will give an overview of each of the subcommands.

analyze

Get detailed reports of where `cloud-init` spends its time during the boot process. For more complete reference see *Performance*.

Possible subcommands include:

- **blame**: report ordered by most costly operations.
- **dump**: machine-readable JSON dump of all `cloud-init` tracked events.
- **show**: show time-ordered report of the cost of operations during each boot stage.
- **boot**: show timestamps from kernel initialisation, kernel finish initialisation, and `cloud-init` start.

clean

Remove `cloud-init` artifacts from `/var/lib/cloud` and config files (best effort) to simulate a clean instance. On reboot, `cloud-init` will re-run all stages as it did on first boot.

- **--logs**: Optionally remove all `cloud-init` log files in `/var/log/`.
- **--reboot**: Reboot the system after removing artifacts.
- **--machine-id**: Set `/etc/machine-id` to `uninitialized\n` on this image for systemd environments. On distributions without systemd, remove the file. Best practice when cloning a golden image, to ensure the next boot of that image auto-generates a unique machine ID. More details on machine-id.
- **--configs [all | ssh_config | network]**: Optionally remove all `cloud-init` generated config files. Argument *ssh_config* cleans config files for ssh daemon. Argument *network* removes all generated config files for network. *all* removes config files of all types.

Note: The operations performed by *clean* can be supplemented / customized. See: *Custom Clean Scripts*.

collect-logs

Collect and tar `cloud-init`-generated logs, data files, and system information for triage. This subcommand is integrated with apport.

Logs collected include:

- `/var/log/cloud-init.log`
- `/var/log/cloud-init-output.log`
- `/run/cloud-init`
- `/var/lib/cloud/instance/user-data.txt`
- `cloud-init` package version
- `dmesg` output
- `journalctl` output

Note: Ubuntu users can file bugs using **ubuntu-bug cloud-init** to automatically attach these logs to a bug report.

devel

Collection of development tools under active development. These tools will likely be promoted to top-level subcommands when stable.

Do **NOT** rely on the output of these commands as they can and will change.

Current subcommands:

net-convert

Manually use `cloud-init`'s network format conversion. Useful for testing configuration or testing changes to the network conversion logic itself.

render

Use `cloud-init`'s jinja template render to process **#cloud-config** or **custom-scripts**, injecting any variables from `/run/cloud-init/instance-data.json`. It accepts a user data file containing the jinja template header `## template: jinja` and renders that content with any `instance-data.json` variables present.

hotplug-hook

Hotplug related subcommands. This command is intended to be called via a `systemd` service and is not considered user-accessible except for debugging purposes.

query

Query if hotplug is enabled for a given subsystem.

handle

Respond to newly added system devices by retrieving updated system metadata and bringing up/down the corresponding device.

enable

Enable hotplug for a given subsystem. This is a last resort command for administrators to enable hotplug in running instances. The recommended method is configuring *Events and updates*, if not enabled by default in the active datasource.

features

Print out each feature supported. If `cloud-init` does not have the **`features`** subcommand, it also does not support any features described in this document.

```
$ cloud-init features
```

Example output:

```
NETWORK_CONFIG_V1
NETWORK_CONFIG_V2
```

init

Generally run by OS init systems to execute `cloud-init`'s stages: *init* and *init-local*. See *Boot stages* for more info. Can be run on the command line, but is deprecated, because incomplete configuration can be applied when run later in boot. The boot stages are generally gated to run only once due to semaphores in `/var/lib/cloud/instance/sem/` and `/var/lib/cloud/sem`.

- **`--local`**: Run *init-local* stage instead of *init*.
- **`--file`** : Use additional yaml configuration files.

modules

Generally run by OS init systems to execute `modules:config` and `modules:final` boot stages. This executes cloud config *Module reference* configured to run in the Init, Config and Final stages. Can be run on the command line, but this is not recommended and will generate a warning because incomplete configuration can be applied when run later in boot. The modules are declared to run in various boot stages in the file `/etc/cloud/cloud.cfg` under keys:

- `cloud_init_modules`
- `cloud_config_modules`
- `cloud_final_modules`

Can be run on the command line, but is deprecated, because incomplete configuration can be applied when run later in boot. Each module is gated to run only once due to semaphores in `/var/lib/cloud/`.

- **`--mode [init|config|final]`**: Run `modules:init`, `modules:config` or `modules:final` `cloud-init` stages. See *Boot stages* for more info.
- **`--file`** : Use additional yaml configuration files.

Warning: *–mode init* is deprecated in 24.1 and scheduled to be removed in 29.1. Use **`cloud-init init`** instead.

query

Query standardised cloud instance metadata crawled by `cloud-init` and stored in `/run/cloud-init/instance-data.json`. This is a convenience command-line interface to reference any cached configuration metadata that `cloud-init` crawls when booting the instance. See *Instance metadata* for more info.

- **`--all`**: Dump all available instance data as JSON which can be queried.
- **`--instance-data`**: Optional path to a different `instance-data.json` file to source for queries.
- **`--list-keys`**: List available query keys from cached instance data.
- **`--format`**: A string that will use jinja-template syntax to render a string replacing.
- **`<varname>`**: A dot-delimited variable path into the `instance-data.json` object.

Below demonstrates how to list all top-level query keys that are standardised aliases:

```
$ cloud-init query --list-keys
```

Example output:

```
_beta_keys
availability_zone
base64_encoded_keys
cloud_name
ds
instance_id
local_hostname
platform
public_ssh_keys
region
sensitive_keys
subplatform
userdata
v1
vendordata
```

Here are a few examples of how to query standardised metadata from clouds:

```
$ cloud-init query v1.cloud_name
```

Example output:

```
aws  # or openstack, azure, gce etc.
```

Any standardised `instance-data` under a <v#> key is aliased as a top-level key for convenience:

```
$ cloud-init query cloud_name
```

Example output:

```
aws  # or openstack, azure, gce etc.
```

One can also query datasource-specific metadata on EC2, e.g.:

```
$ cloud-init query ds.meta_data.public_ipv4
```

Note: The standardised instance data keys under **v#** are guaranteed not to change behaviour or format. If using top-level convenience aliases for any standardised instance data keys, the most value (highest **v#**) of that key name is what is reported as the top-level value. So these aliases act as a 'latest'.

This data can then be formatted to generate custom strings or data. For example, we can generate a custom hostname FQDN based on `instance-id`, cloud and region:

```
$ cloud-init query --format 'custom-{{instance_id}}.{{region}}.{{v1.cloud_name}}.com'
```

```
custom-i-0e91f69987f37ec74.us-east-2.aws.com
```

schema

Validate cloud-config files using jsonschema.

- **-h, --help**: Show this help message and exit.
- **-c CONFIG_FILE, --config-file CONFIG_FILE**: Path of the cloud-config YAML file to validate.
- **-t SCHEMA_TYPE, --schema-type SCHEMA_TYPE**: The schema type to validate –config-file against. One of: cloud-config, network-config. Default: cloud-config.
- **--system**: Validate the system cloud-config user data.
- **-d DOCS [cc_module ...], --docs DOCS [cc_module ...]**: Print schema module docs. Choices are: "all" or "space-delimited" `cc_names`.
- **--annotate**: Annotate existing cloud-config file with errors.

The following example checks a config file and annotates the config file with errors on `stdout`.

```
$ cloud-init schema -c ./config.yml --annotate
```

single

Attempt to run a single, named, cloud config module.

- **--name**: The cloud-config module name to run.
- **--frequency**: Module frequency for this run. One of (`always``|``instance``|``once`).
- **--report**: Enable reporting.
- **--file** : Use additional yaml configuration files.

The following example re-runs the `cc_set_hostname` module ignoring the module default frequency of `instance`:

```
$ cloud-init single --name set_hostname --frequency always
```

Note: Mileage may vary trying to re-run each `cloud-config` module, as some are not idempotent.

status

Report cloud-init's current status.

Exits 1 if `cloud-init` crashes, 2 if `cloud-init` finishes but experienced recoverable errors, and 0 if `cloud-init` ran without error.

- **`--long`**: Detailed status information.
- **`--wait`**: Block until `cloud-init` completes.
- **`--format [yaml|json]`**: Machine-readable JSON or YAML detailed output.

The **`status`** command can be used simply as follows:

```
$ cloud-init status
```

Which shows whether `cloud-init` is currently running, done, disabled, or in error. Note that the `extended_status` key in `--long` or `--format json` contains more accurate and complete status information. Example output:

```
status: running
```

The **`--long`** option, shown below, provides a more verbose output.

```
$ cloud-init status --long
```

Example output when `cloud-init` is running:

```
status: running
extended_status: running
boot_status_code: enabled-by-generator
last_update: Wed, 13 Mar 2024 18:46:26 +0000
detail: DataSourceLXD
errors: []
recoverable_errors: {}
```

Example output when `cloud-init` is done:

```
status: done
extended_status: done
boot_status_code: enabled-by-generator
last_update: Wed, 13 Mar 2024 18:46:26 +0000
detail: DataSourceLXD
errors: []
recoverable_errors: {}
```

The detailed output can be shown in machine-readable JSON or YAML with the **`format`** option, for example:

```
$ cloud-init status --format=json
```

Which would produce the following example output:

```
{
  "boot_status_code": "enabled-by-generator",
  "datasource": "lxd",
  "detail": "DataSourceLXD",
  "errors": [],
```

(continues on next page)

(continued from previous page)


```
  "extended_status": "done",
  "init": {
    "errors": [],
    "finished": 1710355584.3603137,
    "recoverable_errors": {},
    "start": 1710355584.2216876
  },
  "init-local": {
    "errors": [],
    "finished": 1710355582.279756,
    "recoverable_errors": {},
    "start": 1710355582.2255273
  },
  "last_update": "Wed, 13 Mar 2024 18:46:26 +0000",
  "modules-config": {
    "errors": [],
    "finished": 1710355585.5042186,
    "recoverable_errors": {},
    "start": 1710355585.334438
  },
  "modules-final": {
    "errors": [],
    "finished": 1710355586.9038777,
    "recoverable_errors": {},
    "start": 1710355586.8076844
  },
  "recoverable_errors": {},
  "stage": null,
  "status": "done"
}
```


2.4.4 Availability

Below outlines the current availability of `cloud-init` across distributions and clouds, both public and private.

Note: If a distribution or cloud does not show up in the list below, contact them and ask for images to be generated using `cloud-init`!

Distributions

`Cloud-init` has support across all major Linux distributions, FreeBSD, NetBSD, OpenBSD and DragonFlyBSD:

- Alpine Linux
- Arch Linux
- Debian
- DragonFlyBSD
- Fedora

- FreeBSD
- Gentoo Linux
- NetBSD
- OpenBSD
- Photon OS
- RHEL/CentOS/AlmaLinux/Rocky Linux/EuroLinux
- SLES/openSUSE
- Ubuntu

Clouds

`Cloud-init` provides support across a wide ranging list of execution environments in the public cloud:

- Amazon Web Services
- Microsoft Azure
- Google Cloud Platform
- Oracle Cloud Infrastructure
- Softlayer
- Rackspace Public Cloud
- IBM Cloud
- DigitalOcean
- Bigstep
- Hetzner
- Joyent
- CloudSigma
- Alibaba Cloud
- OVH
- OpenNebula
- Exoscale
- Scaleway
- CloudStack
- AltCloud
- SmartOS
- UpCloud
- Vultr
- Zadara Edge Cloud Platform
- 3DS Outscale
- Akamai

Additionally, `cloud-init` is supported on these private clouds:

- Bare metal installs
- OpenStack
- LXD
- KVM
- Metal-as-a-Service (MAAS)
- VMware

2.4.5 FAQ

How do I get help?

Having trouble? We would like to help!

- First go through this page with answers to common questions
- Use the search bar at the upper left to search our documentation
- Ask questions in the `#cloud-init` IRC channel on Libera
- Join and ask questions on the `cloud-init` mailing list
- Find a bug? Check out the *Reporting bugs* topic to find out how to report one

`autoinstall`

Other projects, such as Subiquity autoinstaller, use cloud-init to implement a subset of their features and have a YAML configuration format which combines `cloud-init`'s cloud-config with additional keys.

If you are an autoinstall user, please direct questions to their IRC channel (`#ubuntu-server` on Libera).

Can I use cloud-init as a library?

Please don't. Some projects do. However, `cloud-init` does not currently make any API guarantees to either external consumers or out-of-tree datasources / modules. Current users of cloud-init as a library are projects that have close contact with `cloud-init`, which is why this (fragile) model currently works.

For those that choose to ignore this advice, logging in cloud-init is configured in `cloud-init/cmd/main.py`, and reconfigured in the `cc_rsyslog` module for obvious reasons.

Where can I learn more?

Below are some videos, blog posts, and white papers about `cloud-init` from a variety of sources.

Videos:

- cloud-init - The Good Parts
- Perfect Proxmox Template with Cloud Image and Cloud Init [proxmox, cloud-init, template]
- cloud-init - Building clouds one Linux box at a time (Video)
- Metadata and cloud-init

- Introduction to cloud-init

Blog Posts:

- cloud-init - The cross-cloud Magic Sauce (PDF)
- cloud-init - Building clouds one Linux box at a time (PDF)
- The beauty of cloud-init
- Cloud-init Getting Started [fedora, libvirt, cloud-init]
- Build Azure Devops Agents With Linux cloud-init for Dotnet Development [terraform, azure, devops, docker, dotnet, cloud-init]
- Cloud-init Getting Started [fedora, libvirt, cloud-init]
- Setup Neovim cloud-init Completion [neovim, yaml, Language Server Protocol, jsonschema, cloud-init]

Events:

- cloud-init Summit 2019
- cloud-init Summit 2018
- cloud-init Summit 2017

Whitepapers:

- Utilising cloud-init on Microsoft Azure (Whitepaper)
- Cloud Instance Initialization with cloud-init (Whitepaper)

2.4.6 Merging user data sections

The ability to merge user data sections allows a way to specify how cloud-config YAML "dictionaries" provided as user data are handled when there are multiple YAML files to be merged together (e.g., when performing an #include).

For example merging these two configurations:

```
#cloud-config (1)
runcmd:
  - bash1
  - bash2

#cloud-config (2)
runcmd:
  - bash3
  - bash4
```

Yields the following merged config:

```
#cloud-config (merged)
runcmd:
  - bash1
  - bash2
  - bash3
  - bash4
```

Built-in mergers

`Cloud-init` provides merging for the following built-in types:

- **`Dict`**
- **`List`**
- **`String`**

`Dict`

The **`Dict`** merger has the following options, which control what is done with values contained within the config.

- **`allow_delete`**: Existing values not present in the new value can be deleted. Defaults to `False`.
- **`no_replace`**: Do not replace an existing value if one is already present. Enabled by default.
- **`replace`**: Overwrite existing values with new ones.

`List`

The **`List`** merger has the following options, which control what is done with the values contained within the config.

- **`append`**: Add new value to the end of the list. Defaults to `False`.
- **`prepend`**: Add new values to the start of the list. Defaults to `False`.
- **`no_replace`**: Do not replace an existing value if one is already present. Enabled by default.
- **`replace`**: Overwrite existing values with new ones.

String

The **`Str`** merger has the following options, which control what is done with the values contained within the config.

- **`append`**: Add new value to the end of the string. Defaults to False.

Common options

These are the common options for all merge types, which control how recursive merging is done on other types.

- **`recurse_dict`**: If `True`, merge the new values of the dictionary. Defaults to `True`.
- **`recurse_list`**: If `True`, merge the new values of the list. Defaults to `False`.
- **`recurse_array`**: Alias for `recurse_list`.
- **`recurse_str`**: If `True`, merge the new values of the string. Defaults to False.

Customisation

Custom 3rd party mergers can be defined, for more info visit *Custom Mergers*.

How to activate

There are a few ways to activate the merging algorithms, and to customise them for your own usage.

1. The first way involves the usage of MIME messages in `cloud-init` to specify multi-part documents (this is one way in which multiple cloud-config can be joined together into a single cloud-config). Two new headers are looked for, both of which can define the way merging is done (the first header to exist "wins"). These new headers (in lookup order) are `'Merge-Type'` and `'X-Merge-Type'`. The value should be a string which will satisfy the new merging format definition (see below for this format).
2. The second way is to specify the *merge type* in the body of the cloud-config dictionary. There are two ways to specify this; either as a string, or as a dictionary (see format below). The keys that are looked up for this definition are the following (in order): `'merge_how'`, `'merge_type'`.

String format

The following string format is expected:

```
classname1(option1,option2)+classname2(option3,option4)....
```

The `class name` will be connected to class names used when looking for the class that can be used to merge, and options provided will be given to the class upon construction of that class.

The following example shows the default string that gets used when none is otherwise provided:

```
list()+dict()+str()
```

Dictionary format

A dictionary can be used when it specifies the same information as the string format (i.e., the second option above). For example:

```
{'merge_how': [{'name': 'list', 'settings': ['append']},
               {'name': 'dict', 'settings': ['no_replace', 'recurse_list']},
               {'name': 'str', 'settings': ['append']}]}
```

This would be the dictionary equivalent of the default string format.

Specifying multiple types, and what this does

Now you may be asking yourself: "What exactly happens if I specify a `merge-type` header or dictionary for every cloud-config I provide?"

The answer is that when merging, a stack of `'merging classes'` is kept. The first one in the stack is the default merging class. This set of mergers will be used when the first cloud-config is merged with the initial empty cloud-config dictionary. If the cloud-config that was just merged provided a set of merging classes (via the above formats) then those merging classes will be pushed onto the stack. Now if there is a second cloud-config to be merged then the merging classes from the cloud-config before the first will be used (not the default) and so on. In this way a cloud-config can decide how it will merge with a cloud-config dictionary coming after it.

Other uses

In addition to being used for merging user data sections, the default merging algorithm for merging `'conf.d'` YAML files (which form an initial YAML config for `cloud-init`) was also changed to use this mechanism, to take advantage of the full benefits (and customisation) here as well. Other places that used the previous merging are also, similarly, now extensible (metadata merging, for example).

Note, however, that merge algorithms are not used *across* configuration types. As was the case before merging was implemented, user data will overwrite `'conf.d'` configuration without merging.

Example cloud-config

A common request is to include multiple `runcmd` directives in different files and merge all of the commands together. To achieve this, we must modify the default merging to allow for dictionaries to join list values.

The first config:

```
#cloud-config
merge_how:
 - name: list
   settings: [append]
 - name: dict
   settings: [no_replace, recurse_list]

runcmd:
  - bash1
  - bash2
```

The second config:

```
#cloud-config
merge_how:
 - name: list
   settings: [append]
 - name: dict
   settings: [no_replace, recurse_list]

runcmd:
  - bash3
  - bash4
```

2.4.7 Datasources

Datasources are sources of configuration data for `cloud-init` that typically come from the user (i.e., user data) or come from the cloud that created the configuration drive (i.e., metadata). Typical user data includes files, YAML, and shell scripts whereas typical metadata includes server name, instance id, display name, and other cloud specific details.

Any metadata processed by `cloud-init`'s datasources is persisted as `/run/cloud-init/instance-data.json`. `Cloud-init` provides tooling to quickly introspect some of that data. See *Instance metadata* for more information.

How to configure which datasource to use

By default `cloud-init` should automatically determine which datasource it is running on. Therefore, in most cases, users of `cloud-init` should not have to configure `cloud-init` to specify which datasource cloud-init is running on; `cloud-init` should "figure it out".

There are exceptions, however, when the *datasource does not identify* itself to `cloud-init`. For these exceptions, one can override datasource detection either by configuring a single datasource in the *datasource_list*, or by using *kernel command line arguments*.

Datasources:

The following is a list of documentation for each supported datasource:

Akamai

The Akamai datasource provides an interface to consume metadata on the Akamai Connected Cloud. This service is available at `169.254.169.254` and `fd00:a9fe:a9fe::1` from within the instance.

Configuration

The Akamai datasource supports the following configuration, although in normal use no changes to the defaults should be necessary:

```
datasource:
  Akamai:
    base_urls:
      ipv4: http://169.254.169.254
      ipv6: http://[fd00:a9fe:a9fe::1]
    paths:
        token: /v1/token
        metadata: /v1/instance
        userdata: /v1/user-data
    allow_local_stage: True
    allow_init_stage: True
    allow_dhcp: True
    allow_ipv4: True
    allow_ipv6: True
    preferred_mac_prefixes:
    - f2:3
```

- `base_urls`

  The URLs used to access the metadata service over IPv4 and IPv6 respectively.

- `paths`

  The paths used to reach specific endpoints within the service.

- `allow_local_stage`

  Allows this datasource to fetch data during the local stage. This can be disabled if your image does not want ephemeral networking used.

- `allow_init_stage`

  Allows this datasource to fetch data during the init stage, once networking is online.

- `allow_dhcp`

  Allows this datasource to use dhcp to find an IPv4 address to fetch metadata with during the local stage.

- `allow_ipv4`

  Allow the use of IPv4 when fetching metadata during any stage.

- `allow_ipv6`

  Allows the use of IPv6 when fetching metadata during any stage.

- `preferred_mac_prefixes`

  A list of MAC Address prefixes that will be preferred when selecting an interface to use for ephemeral networking. This is ignored during the init stage.

Configuration Overrides

In some circumstances, the Akamai platform may send configurations overrides to instances via dmi data to prevent certain behavior that may not be supported based on the instance's region or configuration. For example, if deploying an instance in a region that does not yet support metadata, both the local and init stages will be disabled, preventing cloud-init from attempting to fetch metadata. Configuration overrides sent this way will appears in the `baseboard-serial-number` field.

Alibaba Cloud (AliYun)

The `AliYun` datasource reads data from Alibaba Cloud ECS. Support is present in `cloud-init` since 0.7.9.

Metadata service

The Alibaba Cloud metadata service is available at the well known URL `http://100.100.100.200/`. For more information see Alibaba Cloud ECS on metadata.

Configuration

The following configuration can be set for the datasource in system configuration (in `/etc/cloud/cloud.cfg` or `/etc/cloud/cloud.cfg.d/`).

An example configuration with the default values is provided below:

```
datasource:
  AliYun:
    metadata_urls: ["http://100.100.100.200"]
    timeout: 50
    max_wait: 120
```

Versions

Like the EC2 metadata service, Alibaba Cloud's metadata service provides versioned data under specific paths. As of April 2018, there are only `2016-01-01` and `latest` versions.

It is expected that the dated version will maintain a stable interface but `latest` may change content at a future date.

`Cloud-init` uses the `2016-01-01` version.

You can list the versions available to your instance with:

```
$ curl http://100.100.100.200/
```

Example output:

```
2016-01-01
latest
```

Metadata

Instance metadata can be queried at `http://100.100.100.200/2016-01-01/meta-data`:

```
$ curl http://100.100.100.200/2016-01-01/meta-data
```

Example output:

```
dns-conf/
eipv4
hostname
image-id
instance-id
instance/
mac
network-type
network/
ntp-conf/
owner-account-id
private-ipv4
public-keys/
region-id
serial-number
source-address
sub-private-ipv4-list
vpc-cidr-block
vpc-id
```

Userdata

If provided, user data will appear at `http://100.100.100.200/2016-01-01/user-data`. If no user data is provided, this will return a 404.

```
$ curl http://100.100.100.200/2016-01-01/user-data
```

Example output:

```
#!/bin/sh
echo "Hello World."
```

AltCloud

The datasource AltCloud will be used to pick up user data on RHEVm and vSphere.

RHEVm

For RHEVm v3.0 the user data is injected into the VM using floppy injection via the RHEVm dashboard "Custom Properties".

The format of the "Custom Properties" entry must be:

```
floppyinject=user-data.txt:<base64 encoded data>
```

For example, to pass a simple bash script:

```
$ cat simple_script.bash
#!/bin/bash
echo "Hello Joe!" >> /tmp/JJV_Joe_out.txt

$ base64 < simple_script.bash
IyEvYmluL2Jhc2gKZWNobyAiSGVsbG8gSm9lISIgPj4gL3RtcC9KSlZfSm9lX291dC50eHQK
```

To pass this example script to `cloud-init` running in a RHEVm v3.0 VM set the "Custom Properties" when creating the RHEMv v3.0 VM to:

```
floppyinject=user-data.
↪txt:IyEvYmluL2Jhc2gKZWNobyAiSGVsbG8gSm9lISIgPj4gL3RtcC9KSlZfSm9lX291dC50eHQK
```

Note: The prefix with file name must be: `floppyinject=user-data.txt:`

It is also possible to launch a RHEVm v3.0 VM and pass optional user data to it using the Delta Cloud.

vSphere

For VMWare's vSphere the user data is injected into the VM as an ISO via the CD-ROM. This can be done using the vSphere dashboard by connecting an ISO image to the CD/DVD drive.

To pass this example script to `cloud-init` running in a vSphere VM set the CD/DVD drive when creating the vSphere VM to point to an ISO on the data store.

Note: The ISO must contain the user data.

For example, to pass the same `simple_script.bash` to vSphere:

Create the ISO

```
$ mkdir my-iso
```

Note: The file name on the ISO must be: `user-data.txt`

```
$ cp simple_script.bash my-iso/user-data.txt
$ genisoimage -o user-data.iso -r my-iso
```

Verify the ISO

```
$ sudo mkdir /media/vsphere_iso
$ sudo mount -o loop user-data.iso /media/vsphere_iso
$ cat /media/vsphere_iso/user-data.txt
$ sudo umount /media/vsphere_iso
```

Then, launch the vSphere VM the ISO `user-data.iso` attached as a CD-ROM.

It is also possible to launch a vSphere VM and pass optional user data to it using the Delta Cloud.

Amazon EC2

The EC2 datasource is the oldest and most widely used datasource that `cloud-init` supports. This datasource interacts with a *magic* IP provided to the instance by the cloud provider (typically this IP is `169.254.169.254`). At this IP a http server is provided to the instance so that the instance can make calls to get instance user data and instance metadata.

Metadata is accessible via the following URL:

```
GET http://169.254.169.254/2009-04-04/meta-data/
ami-id
ami-launch-index
ami-manifest-path
block-device-mapping/
hostname
instance-id
instance-type
```

(continues on next page)

(continued from previous page)

```
local-hostname
local-ipv4
placement/
public-hostname
public-ipv4
public-keys/
reservation-id
security-groups
```

User data is accessible via the following URL:

```
GET http://169.254.169.254/2009-04-04/user-data
1234,fred,reboot,true | 4512,jimbo, | 173,,,
```

Note that there are multiple EC2 Metadata versions of this data provided to instances. `Cloud-init` attempts to use the most recent API version it supports in order to get the latest API features and instance-data. If a given API version is not exposed to the instance, those API features will be unavailable to the instance.

EC2 version	supported instance-data/feature
2021-03-23	Required for Instance tag support. This feature must be enabled individually on each instance. See the EC2 tags user guide.
2016-09-02	Required for secondary IP address support.
2009-04-04	Minimum supports EC2 API version for metadata and user data.

To see which versions are supported by your cloud provider use the following URL:

```
GET http://169.254.169.254/
1.0
2007-01-19
2007-03-01
2007-08-29
2007-10-10
2007-12-15
2008-02-01
2008-09-01
2009-04-04
...
latest
```

Configuration settings

The following configuration can be set for the datasource in system configuration (in `/etc/cloud/cloud.cfg` or `/etc/cloud/cloud.cfg.d/`).

The settings that may be configured are:

metadata_urls

This list of URLs will be searched for an EC2 metadata service. The first entry that successfully returns a 200 response for `<url>/<version>/meta-data/instance-id` will be selected.

Default: `['http://169.254.169.254', 'http://[fd00:ec2::254]', 'http://instance-data.:8773']`.

max_wait

The maximum amount of clock time in seconds that should be spent searching `metadata_urls`. A value less than zero will result in only one request being made, to the first in the list.

Default: 120

timeout

The timeout value provided to `urlopen` for each individual http request. This is used both when selecting a `metadata_url` and when crawling the metadata service.

Default: 50

apply_full_imds_network_config

Boolean (default: True) to allow `cloud-init` to configure any secondary NICs and secondary IPs described by the metadata service. All network interfaces are configured with DHCP (v4) to obtain a primary IPv4 address and route. Interfaces which have a non-empty `ipv6s` list will also enable DHCPv6 to obtain a primary IPv6 address and route. The DHCP response (v4 and v6) return an IP that matches the first element of `local-ipv4s` and `ipv6s` lists respectively. All additional values (secondary addresses) in the static IP lists will be added to the interface.

An example configuration with the default values is provided below:

```
datasource:
  Ec2:
    metadata_urls: ["http://169.254.169.254:80", "http://instance-data:8773"]
    max_wait: 120
    timeout: 50
    apply_full_imds_network_config: true
```

Notes

- There are 2 types of EC2 instances, network-wise: Virtual Private Cloud (VPC) ones and Classic ones (also known as non-VPC). One major difference between them is that Classic instances have their MAC address changed on stop/restart operations, so `cloud-init` will recreate the network config file for EC2 Classic instances every boot. On VPC instances this file is generated only on the first boot of the instance. The check for the instance type is performed by `is_classic_instance()` method.
- For EC2 instances with multiple network interfaces (NICs) attached, DHCP4 will be enabled to obtain the primary private IPv4 address of those NICs. Wherever DHCP4 or DHCP6 is enabled for a NIC, a DHCP route-metric will be added with the value of `<device-number + 1> * 100` to ensure DHCP routes on the primary NIC are preferred to any secondary NICs. For example: the primary NIC will have a DHCP route-metric of 100, the next NIC will have 200.
- For EC2 instances with multiple NICs, policy-based routing will be configured on secondary NICs / secondary IPs to ensure outgoing packets are routed via the correct interface. This network configuration is only applied on distros using Netplan and at first boot only but it can be configured to be applied on every boot and when NICs are hotplugged, see *Events and updates*.

Azure

This datasource finds metadata and user data from the Azure cloud platform.

The Azure cloud platform provides initial data to an instance via an attached CD formatted in UDF. This CD contains a `ovf-env.xml` file that provides some information. Additional information is obtained via interaction with the "endpoint".

IMDS

Azure provides the instance metadata service (IMDS), which is a REST service on `169.254.169.254` providing additional configuration information to the instance. `Cloud-init` uses the IMDS for:

- Network configuration for the instance which is applied per boot.
- A pre-provisioning gate which blocks instance configuration until Azure fabric is ready to provision.
- Retrieving SSH public keys. `Cloud-init` will first try to utilise SSH keys returned from IMDS, and if they are not provided from IMDS then it will fall back to using the OVF file provided from the CD-ROM. There is a large performance benefit to using IMDS for SSH key retrieval, but in order to support environments where IMDS is not available then we must continue to all for keys from OVF[?]

Configuration

The following configuration can be set for the datasource in system configuration (in `/etc/cloud/cloud.cfg` or `/etc/cloud/cloud.cfg.d/`).

The settings that may be configured are:

- **`apply_network_config`**

  Boolean set to True to use network configuration described by Azure's IMDS endpoint instead of fallback network config of DHCP on eth0. Default is True.

- **`apply_network_config_for_secondary_ips`**

  Boolean to configure secondary IP address(es) for each NIC per IMDS configuration. Default is True.

- **data_dir**

  Path used to read metadata files and write crawled data.

- **disk_aliases**

  A dictionary defining which device paths should be interpreted as ephemeral images. See *cc_disk_setup* module for more info.

Configuration for the datasource can also be read from a `dscfg` entry in the `LinuxProvisioningConfigurationSet`. Content in `dscfg` node is expected to be base64 encoded YAML content, and it will be merged into the `'datasource: Azure'` entry.

An example configuration with the default values is provided below:

```
datasource:
  Azure:
    apply_network_config: true
    apply_network_config_for_secondary_ips: true
    data_dir: /var/lib/waagent
    disk_aliases:
      ephemeral0: /dev/disk/cloud/azure_resource
```

User data

User data is provided to `cloud-init` inside the `ovf-env.xml` file. `Cloud-init` expects that user data will be provided as a base64 encoded value inside the text child of an element named `UserData` or `CustomData`, which is a direct child of the `LinuxProvisioningConfigurationSet` (a sibling to `UserName`).

If both `UserData` and `CustomData` are provided, the behaviour is undefined on which will be selected. In the example below, user data provided is `'this is my userdata'`.

Example:

```
<wa:ProvisioningSection>
 <wa:Version>1.0</wa:Version>
 <LinuxProvisioningConfigurationSet
    xmlns="http://schemas.microsoft.com/windowsazure"
    xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <ConfigurationSetType>LinuxProvisioningConfiguration</ConfigurationSetType>
  <HostName>myHost</HostName>
  <UserName>myuser</UserName>
  <UserPassword/>
  <CustomData>dGhpcyBpcyBteSB1c2VyZGF0YQ===</CustomData>
  <dscfg>eyJhZ2VudF9jb21tYW5kIjogWyJzdGFydCIsICJ3YWxpbnV4YWdlbnQiXX0=</dscfg>
  <DisableSshPasswordAuthentication>true</DisableSshPasswordAuthentication>
  <SSH>
   <PublicKeys>
    <PublicKey>
     <Fingerprint>6BE7A7C3C8A8F4B123CCA5D0C2F1BE4CA7B63ED7</Fingerprint>
     <Path>this-value-unused</Path>
    </PublicKey>
   </PublicKeys>
  </SSH>
  </LinuxProvisioningConfigurationSet>
</wa:ProvisioningSection>
```

HostName

When the user launches an instance, they provide a hostname for that instance. The hostname is provided to the instance in the `ovf-env.xml` file as `HostName`.

Whatever value the instance provides in its DHCP request will resolve in the domain returned in the 'search' request.

A generic image will already have a hostname configured. The Ubuntu cloud images have `ubuntu` as the hostname of the system, and the initial DHCP request on eth0 is not guaranteed to occur after the datasource code has been run. So, on first boot, that initial value will be sent in the DHCP request and *that* value will resolve.

In order to make the `HostName` provided in the `ovf-env.xml` resolve, a DHCP request must be made with the new value. `Cloud-init` handles this by setting the hostname in the datasource's `get_data` method via `hostname $HostName`, and then bouncing the interface. This behaviour can be configured or disabled in the datasource config. See 'Configuration' above.

CloudSigma

This datasource finds metadata and user data from the CloudSigma cloud platform. Data transfer occurs through a virtual serial port of the CloudSigma's VM, and the presence of a network adapter is **NOT** a requirement. See server context in their public documentation for more information.

Setting a hostname

By default, the name of the server will be applied as a hostname on the first boot.

Providing user data

You can provide user data to the VM using the dedicated meta field in the server context `cloudinit-user-data`. By default, *cloud-config* format is expected there, and the `#cloud-config` header can be omitted. However, since this is a raw-text field you could provide any of the valid *config formats*.

You have the option to encode your user data using Base64. In order to do that you have to add the `cloudinit-user-data` field to the `base64_fields`. The latter is a comma-separated field with all the meta fields having Base64-encoded values.

If your user data does not need an internet connection you can create a meta field in the server context `cloudinit-dsmode` and set "local" as the value. If this field does not exist, the default value is "net".

CloudStack

Apache CloudStack exposes user data, metadata, user password, and account SSH key through the `virtual router`. The datasource obtains the `virtual router` address via DHCP lease information given to the instance. For more details on metadata and user data, refer to the CloudStack Administrator Guide.

The following URLs provide to access user data and metadata from the Virtual Machine. `data-server.` is a well-known hostname provided by the CloudStack `virtual router` that points to the next `UserData` server (which is usually also the `virtual router`).

```
http://data-server./latest/user-data
http://data-server./latest/meta-data
http://data-server./latest/meta-data/{metadata type}
```

If `data-server.` cannot be resolved, `cloud-init` will try to obtain the `virtual router`'s address from the system's DHCP leases. If that fails, it will use the system's default gateway.

Configuration

The following configuration can be set for the datasource in system configuration (in `/etc/cloud/cloud.cfg` or `/etc/cloud/cloud.cfg.d/`).

The settings that may be configured are:

- **`max_wait`**

  The maximum amount of clock time in seconds that should be spent searching `metadata_urls`. A value less than zero will result in only one request being made, to the first in the list.

  Default: 120

- **`timeout`**

  The timeout value provided to `urlopen` for each individual http request. This is used both when selecting a `metadata_url` and when crawling the metadata service.

  Default: 50

Example

An example configuration with the default values is provided below:

```
datasource:
  CloudStack:
    max_wait: 120
    timeout: 50
```

Config drive

The configuration drive datasource supports the OpenStack configuration drive disk.

By default, `cloud-init` *always* considers this source to be a fully-fledged datasource. Instead, the typical behavior is to assume it is really only present to provide networking information. `Cloud-init` will copy the network information, apply it to the system, and then continue on. The "full" datasource could then be found in the EC2 metadata service. If this is not the case then the files contained on the located drive must provide equivalents to what the EC2 metadata service would provide (which is typical of the version 2 support listed below).

Note: See the config drive extension and metadata introduction in the public documentation for more information.

Version 1 (deprecated)

Note: Version 1 is legacy and should be considered deprecated. Version 2 has been supported in OpenStack since 2012.2 (Folsom).

The following criteria are required to use a config drive:

1. Must be formatted with vfat filesystem.
2. Must contain *one* of the following files:

```
/etc/network/interfaces
/root/.ssh/authorized_keys
/meta.js
```

`/etc/network/interfaces`

This file is laid down by nova in order to pass static networking information to the guest. `Cloud-init` will copy it off of the config-drive and into /etc/network/interfaces (or convert it to RH format) as soon as it can, and then attempt to bring up all network interfaces.

`/root/.ssh/authorized_keys`

This file is laid down by nova, and contains the ssk keys that were provided to nova on instance creation (nova-boot –key ....)

`/meta.js`

meta.js is populated on the config-drive in response to the user passing "meta flags" (nova boot –meta key=value ...). It is expected to be json formatted.

Version 2

The following criteria are required to use a config drive:

1. Must be formatted with vfat or iso9660 filesystem, or have a *filesystem* label of `config-2` or `CONFIG-2`.
2. The files that will typically be present in the config drive are:

```
openstack/
  - 2012-08-10/ or latest/
    - meta_data.json
    - user_data (not mandatory)
  - content/
    - 0000 (referenced content files)
    - 0001
    - ....
ec2
  - latest/
    - meta-data.json (not mandatory)
```

Keys and values

`Cloud-init`'s behaviour can be modified by keys found in the `meta.js` (version 1 only) file in the following ways.

ds-mode

```
dsmode:
  values: local, net, pass
  default: pass
```

This is what indicates if config drive is a final datasource or not. By default it is 'pass', meaning this datasource should not be read. Set it to 'local' or 'net' to stop `cloud-init` from continuing to search for other datasources after network config.

The difference between 'local' and 'net' is that local will not require networking to be up before user-data actions are run.

instance-id

```
instance-id:
  default: iid-dsconfigdrive
```

This is utilised as the metadata's instance-id. It should generally be unique, as it is what is used to determine "is this a new instance?".

public-keys

```
public-keys:
  default: None
```

If present, these keys will be used as the public keys for the instance. This value overrides the content in `authorized_keys`.

Note: It is likely preferable to provide keys via user data.

user-data

```
user-data:
  default: None
```

This provides `cloud-init` user data. See *examples* for details of what needs to be present here.

DigitalOcean

Warning: Deprecated in version 23.2. Use `DataSourceConfigDrive` instead.

The DigitalOcean datasource consumes the content served from DigitalOcean's metadata service. This metadata service serves information about the running droplet via http over the link local address `169.254.169.254`. The metadata API endpoints are fully described in the DigitalOcean metadata documentation.

Configuration

DigitalOcean's datasource can be configured as follows:

```
datasource:
  DigitalOcean:
    retries: 3
    timeout: 2
```

- `retries`

  Specifies the number of times to attempt connection to the metadata service.

- `timeout`

  Specifies the timeout (in seconds) to wait for a response from the metadata service.

E24Cloud

E24Cloud platform provides an AWS EC2 metadata service clone. It identifies itself to guests using the DMI system-manufacturer (`/sys/class/dmi/id/sys_vendor`).

Exoscale

This datasource supports reading from the metadata server used on the Exoscale platform. Use of the Exoscale datasource is recommended to benefit from new features of the Exoscale platform.

The datasource relies on the availability of a compatible metadata server (`http://169.254.169.254` is used by default) and its companion password server, reachable at the same address (by default on port 8080).

Crawling of metadata

The metadata service and password server are crawled slightly differently:

- The "metadata service" is crawled every boot.
- The password server is also crawled every boot (the Exoscale datasource forces the password module to run with "frequency always").

In the password server case, the following rules apply in order to enable the "restore instance password" functionality:

- If a password is returned by the password server, it is then marked "saved" by the `cloud-init` datasource. Subsequent boots will skip setting the password (the password server will return `saved_password`).

- When the instance password is reset (via the Exoscale UI), the password server will return the non-empty password at next boot, therefore causing `cloud-init` to reset the instance's password.

Configuration

Users of this datasource are discouraged from changing the default settings unless instructed to by Exoscale support.

The following settings are available and can be set for the *datasource base configuration* (in `/etc/cloud/cloud.cfg.d/`).

The settings available are:

- `metadata_url`: The URL for the metadata service.

  Defaults to `http://169.254.169.254`.

- `api_version`: The API version path on which to query the instance metadata.

  Defaults to `1.0`.

- `password_server_port`: The port (on the metadata server) on which the password server listens.

  Defaults to `8080`.

- `timeout`: The timeout value provided to `urlopen` for each individual http request.

  Defaults to `10`.

- `retries`: The number of retries that should be done for a http request.

  Defaults to `6`.

Example

An example configuration with the default values is provided below:

```
datasource:
  Exoscale:
    metadata_url: "http://169.254.169.254"
    api_version: "1.0"
    password_server_port: 8080
    timeout: 10
    retries: 6
```

Fallback/no datasource

This is the fallback datasource when no other datasource can be selected. It is the equivalent of an empty datasource, in that it provides an empty string as user data, and an empty dictionary as metadata.

It is useful for testing, as well as for occasions when you do not need an actual datasource to meet your instance requirements (i.e. you just want to run modules that are not concerned with any external data).

It is typically put at the end of the datasource search list so that if all other datasources are not matched, then this one will be so that the user is not left with an inaccessible instance.

Note: The instance id that this datasource provides is `iid-datasource-none`.

Google Compute Engine

The GCE datasource gets its data from the internal compute metadata server. Metadata can be queried at the URL `http://metadata.google.internal/computeMetadata/v1/` from within an instance. For more information see the GCE metadata docs.

Currently, the default project and instance level metadata keys `project/attributes/sshKeys` and `instance/attributes/ssh-keys` are merged to provide `public-keys`.

`user-data` and `user-data-encoding` can be provided to `cloud-init` by setting those custom metadata keys for an *instance*.

Configuration

The following configuration can be set for the datasource in system configuration (in `/etc/cloud/cloud.cfg` or `/etc/cloud/cloud.cfg.d/`).

The settings that may be configured are:

- `retries`

  The number of retries that should be attempted for a http request. This value is used only after `metadata_url` is selected.

  Default: 5

- `sec_between_retries`

  The amount of wait time between retries when crawling the metadata service.

  Default: 1

Example

An example configuration with the default values is provided below:

```
datasource:
  GCE:
    retries: 5
    sec_between_retries: 1
```

LXD

The LXD datasource allows the user to provide custom user data, vendor data, metadata and network-config to the instance without running a network service (or even without having a network at all). This datasource performs HTTP GETs against the LXD socket device which is provided to each running LXD container and VM as `/dev/lxd/sock` and represents all instance-metadata as versioned HTTP routes such as:

- 1.0/meta-data
- 1.0/config/user.meta-data
- 1.0/config/user.vendor-data
- 1.0/config/user.user-data
- 1.0/config/user.<any-custom-key>

The LXD socket device `/dev/lxd/sock` is only present on containers and VMs when the instance configuration has `security.devlxd=true` (default). Disabling the `security.devlxd` configuration setting at initial launch will ensure that `cloud-init` uses the *NoCloud* datasource. Disabling `security.devlxd` over the life of the container will result in warnings from `cloud-init`, and `cloud-init` will keep the originally-detected LXD datasource.

The LXD datasource is detected as viable by `ds-identify` during the *detect stage* when either `/dev/lxd/sock` exists or `/sys/class/dmi/id/board_name` matches "LXD".

The LXD datasource provides `cloud-init` with the ability to react to metadata, vendor data, user data and network-config changes, and to render the updated configuration across a system reboot.

To modify which metadata, vendor data or user data are provided to the launched container, use either LXD profiles or `lxc launch ... -c <key>="<value>"` at initial container launch, by setting one of the following keys:

- `user.meta-data`: YAML metadata which will be appended to base metadata.
- `user.vendor-data`: YAML which overrides any metadata values.
- `user.network-config`: YAML representing either *Networking config Version 1* or *Networking config Version 2* format.
- `user.user-data`: YAML which takes precedence and overrides both metadata and vendor data values.
- `user.any-key`: Custom user configuration key and value pairs, which can be passed to `cloud-init`. Those keys/values will be present in instance-data which can be used by both *#template: jinja* #cloud-config templates and the **`cloud-init query`** command.

Note: LXD version 4.22 introduced a new scope of config keys prefaced by `cloud-init.`, which are preferred above the related `user.*` keys:

- `cloud-init.meta-data`
- `cloud-init.vendor-data`
- `cloud-init.network-config`
- `cloud-init.user-data`

Configuration

By default, network configuration from this datasource will be:

```
version: 1
config:
    - type: physical
      name: eth0
      subnets:
          - type: dhcp
            control: auto
```

This datasource is intended to replace *NoCloud* datasource for LXD instances with a more direct support for LXD APIs instead of static NoCloud seed files.

Hotplug

Network hotplug functionality is supported for the LXD datasource as described in the *Events and updates* documentation. As hotplug functionality relies on the cloud-provided network metadata, the LXD datasource will only meaningfully react to a hotplug event if it has the configuration necessary to respond to the change. Practically, this means that even with hotplug enabled, **the default behavior for adding a new virtual NIC will result in no change**.

To update the configuration to be used by hotplug, first pass the network configuration via the `cloud-init.network-config` (or `user.network-config` on older versions).

Example

Given an LXD instance named `my-lxd` with hotplug enabled and an LXD bridge named `my-bridge`, the following will allow for additional DHCP configuration of `eth1`:

```
$ cat /tmp/cloud-network-config.yaml
version: 2
ethernets:
    eth0:
        dhcp4: true
    eth1:
        dhcp4: true

$ lxc config set my-lxd cloud-init.network-config="$(cat /tmp/cloud-network-config.yaml)"
$ lxc config device add my-lxd eth1 nic name=eth1 nictype=bridged parent=my-bridge
Device eth1 added to my-lxd
```

MAAS

For now see: https://maas.io/docs

NoCloud

The data source `NoCloud` is a flexible datasource that can be used in multiple different ways.

With NoCloud, one can provide configuration to the instance locally (without network access) or alternatively NoCloud can fetch the configuration from a remote server.

Much of the following documentation describes how to tell cloud-init where to get its configuration.

Runtime configurations

Cloud-init discovers four types of configuration at runtime. The source of these configuration types is configurable with a discovery configuration. This discovery configuration can be delivered to cloud-init in different ways, but is different from the configurations that cloud-init uses to configure the instance at runtime.

user data

User data is a *configuration format* that allows a user to configure an instance.

metadata

The `meta-data` file is a YAML-formatted file.

vendor data

Vendor data may be used to provide default cloud-specific configurations which may be overriden by user data. This may be useful, for example, to configure an instance with a cloud provider's repository mirror for faster package installation.

network config

Network configuration typically comes from the cloud provider to set cloud-specific network configurations, or a reasonable default is set by cloud-init (typically cloud-init brings up an interface using DHCP).

Since NoCloud is a generic datasource, network configuration may be set the same way as user data, metadata, vendor data.

See the *network configuration* documentation for information on network configuration formats.

Discovery configuration

The purpose of the discovery configuration is to tell cloud-init where it can find the runtime configurations described above.

There are two methods for cloud-init to receive a discovery configuration.

Method 1: Line configuration

The "line configuration" is a single string of text which is passed to an instance at boot time via either the kernel command line or in the serial number exposed via DMI (sometimes called SMBIOS).

Example:

```
ds=nocloud;s=https://10.42.42.42/configs/
```

In the above line configuration, `ds=nocloud` tells cloud-init to use the NoCloud datasource, and `s=https://10.42.42.42/configs/` tells cloud-init to fetch configurations using `https` from the URI `https://10.42.42.42/configs/`.

We will describe the possible values in a line configuration in the following sections. See *this section* for more details on line configuration.

Note: If using kernel command line arguments with GRUB, note that an unescaped semicolon is intepreted as the end of a statement. See: GRUB quoting

---

Method 2: System configuration

System configurations are YAML-formatted files and have names that end in `.cfg`. These are located under `/etc/cloud/cloud.cfg.d/`.

Example:

```
datasource:
  NoCloud:
    seedfrom: https://10.42.42.42/configs/
```

The above system configuration tells cloud-init that it is using NoCloud and that it can find configurations at `https://10.42.42.42/configs/`.

The scope of this section is limited to its use for selecting the source of its configuration, however it is worth mentioning that the system configuration provides more than just the discovery configuration.

In addition to defining where cloud-init can find runtime configurations, the system configuration also controls many of cloud-init's default behaviors. Most users shouldn't need to modify these defaults, however it is worth noting that downstream distributions often use them to set reasonable default behaviors for cloud-init. This includes things such as which distro to behave as and which networking backend to use.

The default values in `/etc/cloud/cloud.cfg` may be overriden by drop-in files which are stored in `/etc/cloud/cloud.cfg.d`.

Configuration sources

User-data, metadata, network config, and vendor data may be sourced from one of several possible locations, either locally or remotely.

Source 1: Local filesystem

System configuration may provide cloud-init runtime configuration directly

```
datasource:
  NoCloud:
    meta-data: |
      instance-id: l-eadfbe
    user-data: |
      #cloud-config
      runcmd: [ echo "it worked!" > /tmp/example.txt ]
```

Local filesystem: custom location

Cloud-init makes it possible to find system configuration in a custom filesystem path for those that require more flexibility. This may be done with a line configuration:

```
ds=nocloud;s=file://path/to/directory/
```

Or a system configuration:

```
datasource:
  NoCloud:
    seedfrom: file://path/to/directory
```

Source 2: Drive with labeled filesystem

A labeled vfat or *iso9660* filesystem may be used. The filesystem volume must be labelled `CIDATA`. The *configuration files* must be in the root directory of the filesystem.

Source 3: Custom webserver

Configuration files can be provided to cloud-init over HTTP(S) using a line configuration:

```
ds=nocloud;s=https://10.42.42.42/cloud-init/configs/
```

or using system configuration:

```
datasource:
  NoCloud:
    seedfrom: https://10.42.42.42/cloud-init/configs/
```

Source 4: FTP Server

Configuration files can be provided to cloud-init over unsecured FTP or alternatively with FTP over TLS using a line configuration

```
ds=nocloud;s=ftps://10.42.42.42/cloud-init/configs/
```

or using system configuration

```
datasource:
  NoCloud:
    seedfrom: ftps://10.42.42.42/cloud-init/configs/
```

Source files

The base path pointed to by the URI in the above sources provides content using the following final path components:

- `user-data`
- `meta-data`
- `vendor-data`
- `network-config`

For example, if the `seedfrom` value of `seedfrom` is `https://10.42.42.42/`, then the following files will be fetched from the webserver at first boot:

```
https://10.42.42.42/user-data
https://10.42.42.42/vendor-data
https://10.42.42.42/meta-data
https://10.42.42.42/network-config
```

If the required files don't exist, this datasource will be skipped.

Line configuration in detail

The line configuration has several options.

Permitted keys (DMI and kernel command line)

Currently three keys (and their aliases) are permitted in cloud-init's kernel command line and DMI (sometimes called SMBIOS) serial number.

There is only one required key in a line configuration:

- `seedfrom` (alternatively `s`)

A valid `seedfrom` value consists of a URI which must contain a trailing `/`.

Some optional keys may be used, but their use is discouraged and may be removed in the future.

- `local-hostname` (alternatively `h`)
- `instance-id` (alternatively `i`)

Both of these can be set in `meta-data` instead.

Seedfrom: HTTP and HTTPS

The URI elements supported by NoCloud's HTTP and HTTPS implementations include:

```
<scheme>://<host>/<path>/
```

Where `scheme` can be `http` or `https` and `host` can be an IP address or DNS name.

Seedfrom: FTP and FTP over TLS

The URI elements supported by NoCloud's FTP and FTPS implementation include:

```
<scheme>://<userinfo>@<host>:<port>/<path>/
```

Where `scheme` can be `ftp` or `ftps`, `userinfo` will be `username:password` (defaults is `anonymous` and an empty password), `host` can be an IP address or DNS name, and `port` is which network port to use (default is `21`).

Discovery configuration considerations

Above, we describe the two methods of providing discovery configuration (system configuration and line configuration). Two methods exist because there are advantages and disadvantages to each option, neither is clearly a better choice - so it is left to the user to decide.

Line configuration

Advantages

- it may be possible to set kernel command line and DMI variables at boot time without modifying the base image

Disadvantages

- requires control and modification of the hypervisor or the bootloader
- DMI / SMBIOS is architecture specific

System configuration

Advantages

- simple: requires only modifying a file

Disadvantages

- requires modifying the filesystem prior to booting an instance

DMI-specific kernel command line

Cloud-init performs variable expansion of the `seedfrom` URL for any DMI kernel variables present in `/sys/class/dmi/id` (kenv on FreeBSD). Your `seedfrom` URL can contain variable names of the format `__dmi.varname__` to indicate to the `cloud-init` NoCloud datasource that `dmi.varname` should be expanded to the value of the DMI system attribute wanted.

Table 1: Available DMI variables for expansion in `seedfrom` URL

`dmi.baseboard-asset-tag`	`dmi.baseboard-manufacturer`	`dmi.baseboard-version`
`dmi.bios-release-date`	`dmi.bios-vendor`	`dmi.bios-version`
`dmi.chassis-asset-tag`	`dmi.chassis-manufacturer`	`dmi.chassis-serial-number`
`dmi.chassis-version`	`dmi.system-manufacturer`	`dmi.system-product-name`
`dmi.system-serial-number`	`dmi.system-uuid`	`dmi.system-version`

For example, you can pass this line configuration to QEMU:

```
-smbios type=1,serial=ds=nocloud;s=http://10.10.0.1:8000/__dmi.chassis-serial-number__/
```

This will cause NoCloud to fetch the full metadata from a URL based on YOUR_SERIAL_NUMBER as seen in `/sys/class/dmi/id/chassis_serial_number` (kenv on FreeBSD) from http://10.10.0.1:8000/YOUR_SERIAL_NUMBER/meta-data after the network initialisation is complete.

Example: Creating a disk

Given a disk Ubuntu cloud image in `disk.img`, you can create a sufficient disk by following the following example.

1. Create the `user-data` and `meta-data` files that will be used to modify the image on first boot.

```
$ echo -e "instance-id: iid-local01\nlocal-hostname: cloudimg" > meta-data
$ echo -e "#cloud-config\npassword: passw0rd\nchpasswd: { expire: False }\nssh_pwauth:␣
↪True\ncreate_hostname_file: true\n" > user-data
```

2. At this stage you have three options:

   a. Create a disk to attach with some user data and metadata:

   ```
   $ genisoimage  -output seed.iso -volid cidata -joliet -rock user-data meta-data
   ```

   b. Alternatively, create a `vfat` filesystem with the same files:

   ```
   $ truncate --size 2M seed.iso
   $ mkfs.vfat -n cidata seed.iso
   ```

   - 2b) Option 1: mount and copy files:

   ```
   $ sudo mount -t vfat seed.iso /mnt
   $ sudo cp user-data meta-data /mnt
   $ sudo umount /mnt
   ```

   - 2b) Option 2: the `mtools` package provides `mcopy`, which can access `vfat` filesystems without mounting them:

   ```
   $ mcopy -oi seed.iso user-data meta-data ::
   ```

3. Create a new qcow image to boot, backed by your original image:

```
$ qemu-img create -f qcow2 -b disk.img -F qcow2 boot-disk.img
```

4. Boot the image and log in as "Ubuntu" with password "passw0rd":

```
$ kvm -m 256 \
   -net nic -net user,hostfwd=tcp::2222-:22 \
   -drive file=boot-disk.img,if=virtio \
   -drive driver=raw,file=seed.iso,if=virtio
```

Note: Note that "passw0rd" was set as password through the user data above. There is no password set on these images.

Note: The `instance-id` provided (`iid-local01` above) is what is used to determine if this is "first boot". So, if you are making updates to user data you will also have to change the `instance-id`, or start the disk fresh.

Example `meta-data`

```
instance-id: iid-abcdefg
network-interfaces: |
  iface eth0 inet static
  address 192.168.1.10
  network 192.168.1.0
  netmask 255.255.255.0
  broadcast 192.168.1.255
  gateway 192.168.1.254
hostname: myhost
```

`network-config`

Network configuration can also be provided to `cloud-init` in either *Networking config Version 1* or *Networking config Version 2* by providing that YAML formatted data in a file named `network-config`.

Example network v1:

```
version: 1
config:
   - type: physical
     name: interface0
     mac_address: "52:54:00:12:34:00"
     subnets:
        - type: static
          address: 192.168.1.10
          netmask: 255.255.255.0
          gateway: 192.168.1.254
```

Example network v2:

```
version: 2
ethernets:
  interface0:
    match:
      macaddress: "52:54:00:12:34:00"
    set-name: interface0
    addresses:
      - 192.168.1.10/255.255.255.0
    gateway4: 192.168.1.254
```

None

The data source `None` may be used when no other viable datasource is present on disk. This has two primary use cases:

1. Providing user data to cloud-init from on-disk configuration when no other datasource is present.
2. As a fallback for when a datasource is otherwise intermittently unavailable.

When the datasource is `None`, cloud-init is unable to obtain or render networking configuration. Additionally, when cloud-init completes, a warning is logged that DataSourceNone is being used.

Configuration

User data and meta data may be passed to cloud-init via system configuration in `/etc/cloud/cloud.cfg` or `/etc/cloud/cloud.cfg.d/*.cfg`.

userdata_raw

A **string** containing the user data (including header) to be used by cloud-init.

metadata

The metadata to be used by cloud-init.

Example configuration

```
datasource:
  None:
    metadata:
      local-hostname: "myhost.internal"
    userdata_raw: |
      #cloud-config
      runcmd:
      - echo 'mydata' > /var/tmp/mydata.txt
```

NWCS

The NWCS datasource retrieves basic configuration values from the locally accessible metadata service. All data is served over HTTP from the address `169.254.169.254`.

Configuration

The NWCS datasource can be configured as follows:

```
datasource:
  NWCS:
    url: 'http://169.254.169.254'
    retries: 3
    timeout: 2
    wait: 2
```

- `url`: The URL used to acquire the metadata configuration.
- `retries`: Determines the number of times to attempt to connect to the metadata service.
- `timeout`: Determines the timeout (in seconds) to wait for a response from the metadata service
- `wait`: Determines the timeout in seconds to wait before retrying after accessible failure.

OpenNebula

The OpenNebula (ON) datasource supports the contextualisation disk.

OpenNebula's virtual machines are contextualised (parametrised) by CD-ROM image, which contains a shell script `context.sh`, with custom variables defined on virtual machine start. There are no fixed contextualisation variables, but the datasource accepts many used and recommended across the documentation.

Datasource configuration

Datasource accepts the following configuration options:

```
dsmode:
  values: local, net, disabled
  default: net
```

These specify whether the datasource will be processed in `local` (pre-networking) stage, `net` (post-networking) stage or be `disabled`.

```
parseuser:
  default: nobody
```

Unprivileged system user used for contextualisation script processing.

Contextualisation disk

The following criteria are required:

1. Must be formatted with iso9660 filesystem or have a *filesystem* label of `CONTEXT` or `CDROM`.
2. Must contain the file `context.sh` with contextualisation variables. The file is generated by OpenNebula and has a `KEY='VALUE'` format that can be easily read by bash.

Contextualisation variables

There are no fixed or standard contextualisation variables in OpenNebula. The following variables were found in various places and in revisions of the OpenNebula documentation. Where multiple similar variables are specified, only the one found first is taken.

- `DSMODE`: Datasource mode configuration override. Values are: `local`, `net`, `disabled`.

```
DNS
ETH<x>_IP
ETH<x>_NETWORK
ETH<x>_MASK
ETH<x>_GATEWAY
ETH<x>_GATEWAY6
ETH<x>_DOMAIN
ETH<x>_DNS
ETH<x>_SEARCH_DOMAIN
ETH<x>_MTU
ETH<x>_IP6
ETH<x>_IP6_ULA
ETH<x>_IP6_PREFIX_LENGTH
ETH<x>_IP6_GATEWAY
```

Static network configuration.

```
SET_HOSTNAME
HOSTNAME
```

Instance hostname.

```
PUBLIC_IP
IP_PUBLIC
ETH0_IP
```

If no hostname has been specified, `cloud-init` will try to create a hostname from the instance's IP address in `local` dsmode. In `net` dsmode, `cloud-init` tries to resolve one of its IP addresses to get the hostname.

```
SSH_KEY
SSH_PUBLIC_KEY
```

One or multiple SSH keys (separated by newlines) can be specified.

```
USER_DATA
USERDATA
```

`Cloud-init` user data.

Example configuration

This example `cloud-init` configuration (`cloud.cfg`) enables OpenNebula datasource only in `net` mode.

```
disable_ec2_metadata: True
datasource_list: ['OpenNebula']
datasource:
  OpenNebula:
    dsmode: net
    parseuser: nobody
```

Example VM's context section

```
CONTEXT=[
  SSH_KEY="$USER[SSH_KEY]
$USER[SSH_KEY1]
$USER[SSH_KEY2]",
  PUBLIC_IP="$NIC[IP]",
  USER_DATA="#cloud-config
# see https://help.ubuntu.com/community/CloudInit

packages: []

mounts:
- [vdc,none,swap,sw,0,0]
runcmd:
- echo 'Instance has been configured by cloud-init.' | wall
" ]
```

OpenStack

This datasource supports reading data from the OpenStack Metadata Service.

Discovery

To determine whether a platform looks like it may be OpenStack, `cloud-init` checks the following environment attributes as a potential OpenStack platform:

- May be OpenStack **if**:
  - `non-x86 cpu architecture`: because DMI data is buggy on some arches.
- Is OpenStack **if** x86 architecture and **ANY** of the following:
  - `/proc/1/environ`: `Nova-lxd` contains `product_name=OpenStack Nova`.
  - `DMI product_name`: Either `Openstack Nova` or `OpenStack Compute`.
  - `DMI chassis_asset_tag` is `HUAWEICLOUD`, `OpenTelekomCloud`, `SAP CCloud VM`, `OpenStack Nova` (since 19.2) or `OpenStack Compute` (since 19.2).

Configuration

The following configuration can be set for the datasource in system configuration (in `/etc/cloud/cloud.cfg` or `/etc/cloud/cloud.cfg.d/`).

The settings that may be configured are as follows:

`metadata_urls`

This list of URLs will be searched for an OpenStack metadata service. The first entry that successfully returns a 200 response for `<url>/openstack` will be selected.

Default: ['http://169.254.169.254'])

`max_wait`

The maximum amount of clock time (in seconds) that should be spent searching `metadata_urls`. A value less than zero will result in only one request being made, to the first in the list.

Default: -1

`timeout`

The timeout value provided to `urlopen` for each individual http request. This is used both when selecting a `metadata_url` and when crawling the metadata service.

Default: 10

`retries`

The number of retries that should be attempted for an http request. This value is used only after `metadata_url` is selected.

Default: 5

`apply_network_config`

A boolean specifying whether to configure the network for the instance based on `network_data.json` provided by the metadata service. When False, only configure DHCP on the primary NIC for this instance.

Default: True

Example configuration

An example configuration with the default values is provided below:

```
datasource:
  OpenStack:
    metadata_urls: ["http://169.254.169.254"]
    max_wait: -1
    timeout: 10
    retries: 5
    apply_network_config: True
```

Vendor Data

The OpenStack metadata server can be configured to serve up vendor data, which is available to all instances for consumption. OpenStack vendor data is generally a JSON object.

`Cloud-init` will look for configuration in the `cloud-init` attribute of the vendor data JSON object. `Cloud-init` processes this configuration using the same handlers as user data, so any formats that work for user data should work for vendor data.

For example, configuring the following as vendor data in OpenStack would upgrade packages and install `htop` on all instances:

```
{"cloud-init": "#cloud-config\npackage_upgrade: True\npackages:\n - htop"}
```

For more general information about how `cloud-init` handles vendor data, including how it can be disabled by users on instances, see our *explanation topic*.

OpenStack can also be configured to provide "dynamic vendordata" which is provided by the DynamicJSON provider and appears under a different metadata path, `/vendor_data2.json`.

`Cloud-init` will look for a `cloud-init` at the `vendor_data2` path; if found, settings are applied after (and, hence, overriding) the settings from static vendor data. Both sets of vendor data can be overridden by user data.

OpenStack Ironic Bare Metal

During boot, cloud-init typically has to identify which platform it is running on. Since OpenStack Ironic Bare Metal doesn't provide a method for cloud-init to discover that it is running on Ironic, extra user configuration is required.

Cloud-init provides two methods to do this:

Method 1: Configuration file

Explicitly set `datasource_list` to only `openstack`, such as:

```
datasource_list: ["openstack"]
```

Method 2: Kernel command line

Set the kernel command line to configure *datasource override*.

Example using Ubuntu + GRUB2:

```
$ echo 'ds=openstack' >> /etc/default/grub
$ grub-mkconfig -o /boot/efi/EFI/ubuntu/grub.cfg
```

Oracle

This datasource reads metadata, vendor data and user data from Oracle Compute Infrastructure (OCI).

Oracle platform

OCI provides bare metal and virtual machines. In both cases, the platform identifies itself via DMI data in the chassis asset tag with the string `'OracleCloud.com'`.

Oracle's platform provides a metadata service that mimics the `2013-10-17` version of OpenStack metadata service. Initially, support for Oracle was done via the OpenStack datasource.

`Cloud-init` has a specific datasource for Oracle in order to:

a. Allow and support the future growth of the OCI platform.

b. **Address small differences between OpenStack and Oracle metadata** implementation.

Configuration

The following configuration can be set for the datasource in system configuration (in `/etc/cloud/cloud.cfg` or `/etc/cloud/cloud.cfg.d/`).

`configure_secondary_nics`

A boolean, defaulting to False. If set to True on an OCI Virtual Machine, `cloud-init` will fetch networking metadata from Oracle's IMDS and use it to configure the non-primary network interface controllers in the system. If set to True on an OCI Bare Metal Machine, it will have no effect (though this may change in the future).

`max_wait`

An integer, defaulting to 30. The maximum time in seconds to wait for the metadata service to become available. If the metadata service is not available within this time, the datasource will fail.

timeout

An integer, defaulting to 5. The time in seconds to wait for a response from the metadata service before retrying.

Example configuration

An example configuration with the default values is provided below:

```
datasource:
 Oracle:
  configure_secondary_nics: false
  max_wait: 30
  timeout: 5
```

OVF

The OVF datasource provides a generic datasource for reading data from an Open Virtualization Format ISO transport.

What platforms support OVF

OFV is an open standard which is supported by various virtualization platforms, including (but not limited to):

GCP OpenShift Proxmox vSphere VirtualBox Xen

While these (and many more) platforms support OVF, in some cases cloud-init has alternative datasources which provide better platform integration. Make sure to check whether another datasource is exists which is specific to your platform of choice before trying to use OVF.

Configuration

Cloud-init gets configurations from an OVF XML file. User-data and network configuration are provided by properties in the XML which contain key / value pairs. The user-data is provided by a key named `user-data`, and network configuration is provided by a key named `network-config`.

Graceful rpctool fallback

The datasource initially attempts to use the program `vmware-rpctool` if it is available. However, if the program returns a non-zero exit code, then the datasource falls back to using the program `vmtoolsd` with the `--cmd` argument.

On some older versions of ESXi and open-vm-tools, the `vmware-rpctool` program is much more performant than `vmtoolsd`. While this gap was closed, it is not reasonable to expect the guest where cloud-init is running to know whether the underlying hypervisor has the patch.

Additionally, vSphere VMs may have the following present in their VMX file:

```
guest_rpc.rpci.auth.cmd.info-set = "TRUE"
guest_rpc.rpci.auth.cmd.info-get = "TRUE"
```

The above configuration causes the `vmware-rpctool` command to return a non-zero exit code with the error message `Permission denied`. If this should occur, the datasource falls back to using `vmtoolsd`.

Additional information

For further information see a full working example in `cloud-init`'s source code tree in `doc/sources/ovf`.

Rbx Cloud

The Rbx datasource consumes the metadata drive available on the HyperOne and Rootbox platforms.

This datasource supports network configurations, hostname, user accounts and user metadata.

Metadata drive

Drive metadata is a FAT-formatted partition with the `CLOUDMD` or `cloudmd` label on the system disk. Its contents are refreshed each time the virtual machine is restarted, if the partition exists. For more information see HyperOne Virtual Machine docs.

Scaleway

Scaleway datasource uses data provided by the Scaleway metadata service to do initial configuration of the network services.

The metadata service is reachable at the following addresses :

- IPv4: `169.254.42.42`
- IPv6: `fd00:42::42`

Configuration

Scaleway datasource may be configured in system configuration (in */etc/cloud cloud.cfg*) or by adding a file with the .cfg suffix containing the following information in the */etc/cloud.cfg.d* directory:

```
datasource:
  Scaleway:
    retries: 3
    timeout: 10
    max_wait: 2
    metadata_urls:
      - alternate_url
```

- `retries`

  Controls the maximum number of attempts to reach the metadata service.

- `timeout`

  Controls the number of seconds to wait for a response from the metadata service for one protocol.

- `max_wait`

  Controls the number of seconds to wait for a response from the metadata service for all protocols.

- `metadata_urls`

  List of additional URLs to be used in an attempt to reach the metadata service in addition to the existing ones.

User Data

cloud-init fetches user data using the metadata service using the */user_data* endpoint. Scaleway's documentation provides a detailed description on how to use userdata. One can also interact with it using the userdata api.

SmartOS Datasource

This datasource finds metadata and user data from the SmartOS virtualisation platform (i.e., Joyent).

Please see http://smartos.org/ for information about SmartOS.

SmartOS platform

The SmartOS virtualisation platform uses metadata from the instance via the second serial console. On Linux, this is `/dev/ttyS1`. The data is provided via a simple protocol:

- Something queries for the data,
- the console responds with the status, and
- if "SUCCESS" returns until a single ".n".

New versions of the SmartOS tooling will include support for Base64-encoded data.

Metadata channels

`Cloud-init` supports three modes of delivering user data and metadata via the flexible channels of SmartOS.

1. User data is written to `/var/db/user-data`:
   - As per the spec, user data is for consumption by the end user, not provisioning tools.
   - `Cloud-init` ignores this channel, other than writing it to disk.
   - Removal of the `meta-data` key means that `/var/db/user-data` gets removed.
   - A backup of previous metadata is maintained as `/var/db/user-data.<timestamp>`. `<timestamp>` is the epoch time when `cloud-init` ran.
2. `user-script` is written to `/var/lib/cloud/scripts/per-boot/99_user_data`:
   - This is executed each boot.
   - A link is created to `/var/db/user-script`.
   - Previous versions of `user-script` is written to `/var/lib/cloud/scripts/per-boot.backup/99_user_script.<timestamp>.`
   - <timestamp> is the epoch time when `cloud-init` ran.
   - When the `user-script` metadata key goes missing, `user-script` is removed from the file system, although a backup is maintained.
   - If the script does not start with a shebang (i.e., it starts with #!<executable>), or it is not an executable, `cloud-init` will add a shebang of "#!/bin/bash".
3. `Cloud-init` user data is treated like on other Clouds.
   - This channel is used for delivering `_all_ cloud-init` instructions.
   - Scripts delivered over this channel must be well formed (i.e., they must have a shebang).

Cloud-init supports reading the traditional metadata fields supported by the SmartOS tools. These are:

- root_authorized_keys
- hostname
- enable_motd_sys_info
- iptables_disable

Note: At this time, iptables_disable and enable_motd_sys_info are read but are not actioned.

Disabling user-script

Cloud-init uses the per-boot script functionality to handle the execution of the user-script. If you want to prevent this, use a cloud-config of:

```
#cloud-config
cloud_final_modules:
 - scripts_per_once
 - scripts_per_instance
 - scripts_user
 - ssh_authkey_fingerprints
 - keys_to_console
 - phone_home
 - final_message
 - power_state_change
```

Alternatively you can use the JSON patch method:

```
#cloud-config-jsonp
[
     { "op": "replace",
       "path": "/cloud_final_modules",
       "value": ["scripts_per_once",
                 "scripts_per_instance",
                 "scripts_user",
                 "ssh_authkey_fingerprints",
                 "keys_to_console",
                 "phone_home",
                 "final_message",
                 "power_state_change"]
     }
]
```

The default cloud-config includes "script-per-boot". Cloud-init will still ingest and write the user data, but will not execute it when you disable the per-boot script handling.

The cloud-config needs to be delivered over the cloud-init:user-data channel in order for cloud-init to ingest it.

Note: Unless you have an explicit use-case, it is recommended that you do not disable the per-boot script execution, especially if you are using any of the life-cycle management features of SmartOS.

Base64

The following are exempt from Base64 encoding, owing to the fact that they are provided by SmartOS:

- `root_authorized_keys`
- `enable_motd_sys_info`
- `iptables_disable`
- `user-data`
- `user-script`

This list can be changed through the *datasource base configuration* variable `no_base64_decode`.

This means that `user-script`, `user-data` and other values can be Base64 encoded. Since `cloud-init` can only guess whether or not something is truly Base64 encoded, the following metadata keys are hints as to whether or not to Base64 decode something:

- `base64_all`: Except for excluded keys, attempt to Base64 decode the values. If the value fails to decode properly, it will be returned in its text.
- `base64_keys`: A comma-delimited list of which keys are Base64 encoded.
- `b64-<key>`: For any key, if an entry exists in the metadata for `'b64-<key>'`, then `'b64-<key>'` is expected to be a plain-text boolean indicating whether or not its value is encoded.
- `no_base64_decode`: This is a configuration setting (i.e., `/etc/cloud/cloud.cfg.d`) that sets which values should not be Base64 decoded.

`disk_aliases` and ephemeral disk

By default, SmartOS only supports a single ephemeral disk. That disk is completely empty (un-partitioned, with no filesystem).

The SmartOS datasource has built-in cloud-config which instructs the `disk_setup` module to partition and format the ephemeral disk.

You can control the `disk_setup` in 2 ways:

1. Through the datasource config, you can change the 'alias' of `ephemeral0` to reference another device. The default is:

   ```
   'disk_aliases': {'ephemeral0': '/dev/vdb'}
   ```

   This means that anywhere `disk_setup` sees a device named 'ephemeral0', then `/dev/vdb` will be substituted.

2. You can provide `disk_setup` or `fs_setup` data in `user-data` to overwrite the datasource's built-in values.

See `doc/examples/cloud-config-disk-setup.txt` for information on `disk_setup`.

UpCloud

The UpCloud datasource consumes information from UpCloud's metadata service. This metadata service serves information about the running server via HTTP over the address `169.254.169.254` available in every DHCP-configured interface. The metadata API endpoints are fully described in UpCloud API documentation.

Providing user data

When creating a server, user data is provided by specifying it as `user_data` in the API or via the server creation tool in the control panel. User data is immutable during the server's lifetime, and can be removed by deleting the server.

VMware

This datasource is for use with systems running on a VMware platform such as vSphere and currently supports the following data transports:

- Guest OS Customization
- GuestInfo keys

The configuration method is dependent upon the transport.

Guest OS customization

The following configuration can be set for this datasource in `cloud-init` configuration (in `/etc/cloud/cloud.cfg` or `/etc/cloud/cloud.cfg.d/`).

System configuration

- `disable_vmware_customization`: true (disable) or false (enable) the VMware traditional Linux guest customization. Traditional Linux guest customization is customizing a Linux virtual machine with a traditional Linux customization specification. Setting this configuration to false is required to make sure this datasource is found in `ds-identify` when using Guest OS customization transport. VMware Tools only checks this configuration in `/etc/cloud/cloud.cfg`.

  Default: true

Datasource configuration

- `allow_raw_data`: true (enable) or false (disable) the VMware customization using `cloud-init` metadata and user data directly. Since vSphere 7.0 Update 3 version, users can create a Linux customization specification with minimal `cloud-init` metadata and user data, and apply this specification to a virtual machine. This datasource will parse the metadata and user data and configure the virtual machine with them. See Guest customization using cloud-init for more information.

  Default: true

- `vmware_cust_file_max_wait`: The maximum amount of clock time (in seconds) that should be spent waiting for VMware customization files.

  Default: 15

Configuration examples

1. Enable VMware customization and set the maximum waiting time for the VMware customization file to 10 seconds:

   Set `disable_vmware_customization` in the `/etc/cloud/cloud.cfg`

   ```
   disable_vmware_customization: false
   ```

   Create a `/etc/cloud/cloud.cfg.d/99-vmware-guest-customization.cfg` with the following content

   ```
   datasource:
     VMware:
       vmware_cust_file_max_wait: 10
   ```

2. Enable VMware customization but only try to apply a traditional Linux Guest Customization configuration, and set the maximum waiting time for the VMware customization file to 10 seconds:

   Set `disable_vmware_customization` in the `/etc/cloud/cloud.cfg`

   ```
   disable_vmware_customization: false
   ```

   Create a `/etc/cloud/cloud.cfg.d/99-vmware-guest-customization.cfg` with the following content

   ```
   datasource:
     VMware:
       allow_raw_data: false
       vmware_cust_file_max_wait: 10
   ```

VMware Tools configuration

VMware Tools is required for this datasource's configuration settings, as well as vCloud and vSphere admin configuration. Users can change the VMware Tools configuration options with the following command:

```
vmware-toolbox-cmd config set <section> <key> <value>
```

The following VMware Tools configuration option affects this datasource's behaviour when applying customization configuration with custom scripts:

- `[deploypkg] enable-custom-scripts`: If this option is absent in VMware Tools configuration, the custom script is disabled by default for security reasons. Some VMware products could change this default behaviour (for example: enabled by default) via customization of the specification settings.

  VMware admins can refer to customization configuration and set the customization specification settings.

For more information, see VMware vSphere Product Documentation and specific VMware Tools configuration options.

GuestInfo keys

One method of providing meta, user, and vendor data is by setting the following key/value pairs on a VM's `extraConfig` property:

Property	Description
`guestinfo.metadata`	A YAML or JSON document containing the `cloud-init` metadata.
`guestinfo.metadata.encoding`	The encoding type for `guestinfo.metadata`.
`guestinfo.userdata`	A YAML document containing the `cloud-init` user data.
`guestinfo.userdata.encoding`	The encoding type for `guestinfo.userdata`.
`guestinfo.vendordata`	A YAML document containing the `cloud-init` vendor data.
`guestinfo.vendordata.encoding`	The encoding type for `guestinfo.vendordata`.

All `guestinfo.*.encoding` values may be set to `base64` or `gzip+base64`.

Features

This section reviews several features available in this datasource.

Graceful rpctool fallback

The datasource initially attempts to use the program `vmware-rpctool` if it is available. However, if the program returns a non-zero exit code, then the datasource falls back to using the program `vmtoolsd` with the `--cmd` argument.

On some older versions of ESXi and open-vm-tools, the `vmware-rpctool` program is much more performant than `vmtoolsd`. While this gap was closed, it is not reasonable to expect the guest where cloud-init is running to know whether the underlying hypervisor has the patch.

Additionally, vSphere VMs may have the following present in their VMX file:

```
guest_rpc.rpci.auth.cmd.info-set = "TRUE"
guest_rpc.rpci.auth.cmd.info-get = "TRUE"
```

The above configuration causes the `vmware-rpctool` command to return a non-zero exit code with the error message `Permission denied`. If this should occur, the datasource falls back to using `vmtoolsd`.

Instance data and lazy networks

One of the hallmarks of `cloud-init` is *its use of instance-data and JINJA queries* – the ability to write queries in user and vendor data that reference runtime information present in `/run/cloud-init/instance-data.json`. This works well when the metadata provides all of the information up front, such as the network configuration. For systems that rely on DHCP, however, this information may not be available when the metadata is persisted to disk.

This datasource ensures that even if the instance is using DHCP to configure networking, the same details about the configured network are available in `/run/cloud-init/instance-data.json` as if static networking was used. This information collected at runtime is easy to demonstrate by executing the datasource on the command line. From the root of this repository, run the following command:

```
PYTHONPATH="$(pwd)" python3 cloudinit/sources/DataSourceVMware.py
```

The above command will result in output similar to the below JSON:

```
{
    "hostname": "akutz.localhost",
    "local-hostname": "akutz.localhost",
    "local-ipv4": "192.168.0.188",
    "local_hostname": "akutz.localhost",
    "network": {
        "config": {
            "dhcp": true
        },
        "interfaces": {
            "by-ipv4": {
                "172.0.0.2": {
                    "netmask": "255.255.255.255",
                    "peer": "172.0.0.2"
                },
                "192.168.0.188": {
                    "broadcast": "192.168.0.255",
                    "mac": "64:4b:f0:18:9a:21",
                    "netmask": "255.255.255.0"
                }
            },
            "by-ipv6": {
                "fd8e:d25e:c5b6:1:1f5:b2fd:8973:22f2": {
                    "flags": 208,
                    "mac": "64:4b:f0:18:9a:21",
                    "netmask": "ffff:ffff:ffff:ffff::/64"
                }
            },
            "by-mac": {
                "64:4b:f0:18:9a:21": {
                    "ipv4": [
                        {
                            "addr": "192.168.0.188",
                            "broadcast": "192.168.0.255",
                            "netmask": "255.255.255.0"
                        }
                    ],
                    "ipv6": [
                        {
                            "addr": "fd8e:d25e:c5b6:1:1f5:b2fd:8973:22f2",
                            "flags": 208,
                            "netmask": "ffff:ffff:ffff:ffff::/64"
                        }
                    ]
                },
                "ac:de:48:00:11:22": {
                    "ipv6": []
                }
            }
        }
    },
    "wait-on-network": {
        "ipv4": true,
```


(continues on next page)

(continued from previous page)

```
        "ipv6": "false"
    }
}
```

Redacting sensitive information (GuestInfo keys transport only)

Sometimes the `cloud-init` user data might contain sensitive information, and it may be desirable to have the `guestinfo.userdata` key (or other `guestinfo` keys) redacted as soon as its data is read by the datasource. This is possible by adding the following to the metadata:

```
redact: # formerly named cleanup-guestinfo, which will also work
- userdata
- vendordata
```

When the above snippet is added to the metadata, the datasource will iterate over the elements in the `redact` array and clear each of the keys. For example, when the `guestinfo` transport is used, the above snippet will cause the following commands to be executed:

```
vmware-rpctool "info-set guestinfo.userdata ---"
vmware-rpctool "info-set guestinfo.userdata.encoding  "
vmware-rpctool "info-set guestinfo.vendordata ---"
vmware-rpctool "info-set guestinfo.vendordata.encoding  "
```

Please note that keys are set to the valid YAML string `---` as it is not possible remove an existing key from the `guestinfo` key-space. A key's analogous encoding property will be set to a single white-space character, causing the datasource to treat the actual key value as plain-text, thereby loading it as an empty YAML doc (hence the aforementioned `---`).

Reading the local IP addresses

This datasource automatically discovers the local IPv4 and IPv6 addresses for a guest operating system based on the default routes. However, when inspecting a VM externally, it's not possible to know what the *default* IP address is for the guest OS. That's why this datasource sets the discovered, local IPv4 and IPv6 addresses back in the `guestinfo` namespace as the following keys:

- `guestinfo.local-ipv4`
- `guestinfo.local-ipv6`

It is possible that a host may not have any default, local IP addresses. It's also possible the reported, local addresses are link-local addresses. But these two keys may be used to discover what this datasource determined were the local IPv4 and IPv6 addresses for a host.

Waiting on the network

Sometimes `cloud-init` may bring up the network, but it will not finish coming online before the datasource's `setup` function is called, resulting in a `/var/run/cloud-init/instance-data.json` file that does not have the correct network information. It is possible to instruct the datasource to wait until an IPv4 or IPv6 address is available before writing the instance data with the following metadata properties:

```
wait-on-network:
  ipv4: true
  ipv6: true
```

If either of the above values are true, then the datasource will sleep for a second, check the network status, and repeat until one or both addresses from the specified families are available.

Walkthrough of GuestInfo keys transport

The following series of steps is a demonstration of how to configure a VM with this datasource using the GuestInfo keys transport:

1. Create the metadata file for the VM. Save the following YAML to a file named `metadata.yaml`:

   ```
   instance-id: cloud-vm
   local-hostname: cloud-vm
   network:
     version: 2
     ethernets:
       nics:
         match:
           name: ens*
         dhcp4: yes
   ```

2. Create the userdata file `userdata.yaml`:

   ```
   #cloud-config

   users:
   - default
   - name: akutz
     primary_group: akutz
     sudo: ALL=(ALL) NOPASSWD:ALL
     groups: sudo, wheel
     lock_passwd: true
     ssh_authorized_keys:
     - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQDE0c5FczvcGSh/tG4iw+Fhfi/O5/EvUM/
    ↪96js65tly4++YTXK1d9jcznPS5ruDlbIZ30oveCBd3kT8LLVFwzh6hepYTf0YmCTpF4eDunyqmpCXDvVscQYRXyasEm5olGmV
    ↪mt2PcPjooeX00vAj81jjU2f3XKrjjz2u2+KIt9eba+vOQ6HiC8c2IzRkUAJ5i1atLy8RIbejo23+0P4N2jjk17QySFOVHwPBI
    ↪0M/4ideeU74EN/
    ↪CgVsvO6JrLsPBR4dojkV5qNbMNxIVv5cUwIy2ThlLgqpNCeFIDLCWNZEFKlEuNeSQ2mPtIO7ETxEL2Cz5y/
    ↪7AIuildzYMc6wi2bofRC8HmQ7rMXRWdwLKWsR0L7SKjHblIwarxOGqLnUI+k2E71YoP7SZSlxaKi17pqkr0OMCF+kKqvcvHAC
    ↪tZKSpjYtjMb5+RonfhaFRNzvj7cCE1f3Kp8UVqAdcGBTtReoE8eRUT63qIxjw03a7VwAyB2w+9cu1R9/
    ↪vAo8SBeRqw== sakutz@gmail.com
   ```

3. Please note this step requires that the VM be powered off. All of the commands below use the VMware CLI tool, govc.

Go ahead and assign the path to the VM to the environment variable `VM`:

```
export VM="/inventory/path/to/the/vm"
```

4. Power off the VM:

   To ensure the next power-on operation results in a first-boot scenario for `cloud-init`, it may be necessary to run the following command just before powering off the VM:

   ```
   cloud-init clean --logs --machine-id
   ```

   Otherwise `cloud-init` may not run in first-boot mode. For more information on how the boot mode is determined, please see the *First Boot Documentation*.

   ```
   govc vm.power -off "${VM}"
   ```

5. Export the environment variables that contain the `cloud-init` metadata and user data:

   ```
   export METADATA=$(gzip -c9 <metadata.yaml | { base64 -w0 2>/dev/null || base64; }) \
        USERDATA=$(gzip -c9 <userdata.yaml | { base64 -w0 2>/dev/null || base64; })
   ```

6. Assign the metadata and user data to the VM:

   ```
   govc vm.change -vm "${VM}" \
   -e guestinfo.metadata="${METADATA}" \
   -e guestinfo.metadata.encoding="gzip+base64" \
   -e guestinfo.userdata="${USERDATA}" \
   -e guestinfo.userdata.encoding="gzip+base64"
   ```

   Note: Please note the above commands include specifying the encoding for the properties. This is important as it informs the datasource how to decode the data for `cloud-init`. Valid values for `metadata.encoding` and `userdata.encoding` include:

   - `base64`
   - `gzip+base64`

   ---

7. Power on the VM:

   ```
   govc vm.power -on "${VM}"
   ```

If all went according to plan, the CentOS box is:

- Locked down, allowing SSH access only for the user in the user data.
- Configured for a dynamic IP address via DHCP.
- Has a hostname of `cloud-vm`.

Examples of common configurations

Setting the hostname

The hostname is set by way of the metadata key `local-hostname`.

Setting the instance ID

The instance ID may be set by way of the metadata key `instance-id`. However, if this value is absent then the instance ID is read from the file `/sys/class/dmi/id/product_uuid`.

Providing public SSH keys

The public SSH keys may be set by way of the metadata key `public-keys-data`. Each newline-terminated string will be interpreted as a separate SSH public key, which will be placed in distro's default user's `~/.ssh/authorized_keys`. If the value is empty or absent, then nothing will be written to `~/.ssh/authorized_keys`.

Configuring the network

The network is configured by setting the metadata key `network` with a value consistent with Network Config *Version 1* or *Version 2*, depending on the Linux distro's version of `cloud-init`.

The metadata key `network.encoding` may be used to indicate the format of the metadata key `network`. Valid encodings are `base64` and `gzip+base64`.

Vultr

The Vultr datasource retrieves basic configuration values from the locally accessible metadata service. All data is served over HTTP from the address `169.254.169.254`. The endpoints are documented in the metadata service documentation.

Configuration

Vultr's datasource can be configured as follows:

```
datasource:
  Vultr:
    url: 'http://169.254.169.254'
    retries: 3
    timeout: 2
    wait: 2
```

- `url`: The URL used to acquire the metadata configuration.
- `retries`: Determines the number of times to attempt to connect to the metadata service.
- `timeout`: Determines the timeout (in seconds) to wait for a response from the metadata service.
- `wait`: Determines the timeout (in seconds) to wait before retrying after accessible failure.

WSL

The Windows Subsystem for Linux (WSL) somewhat resembles a container hypervisor. A Windows user may have as many Linux distro instances as they wish, either created by the distro-launcher workflow (for the distros delivered through MS Store) or by importing a tarball containing a root filesystem. This page assumes the reader is familiar with WSL. To learn more about that, please visit the Microsoft documentation.

Requirements

1. **WSL interoperability must be enabled**. The datasource needs to execute some Windows binaries to compute the possible locations of the user data files.
2. **WSL automount must be enabled**. The datasource needs to access files in the Windows host filesystem.
3. **The init system must be aware of cloud-init**. WSL has opt-in support for systemd, thus for distros that rely on it, such as Ubuntu, cloud-init will run automatically if systemd is enabled via the `/etc/wsl.conf`. The Ubuntu applications distributed via Microsoft Store enable systemd in the first boot, so no action is required if the user sets up a new instance by using them. Users of other distros may find it surprising that cloud-init doesn't run automatically by default. At the time of this writing, only systemd distros are supported by the WSL datasource, although there is nothing hard-coded in the implementation code that requires it, so non-systemd distros may find ways to run cloud-init and make it just work.

Notice that requirements 1 and 2 are met by default, i.e. WSL grants those features enabled. Users can disable those features, though. That would prevent the datasource from working. For more information about how to configure WSL, check the official documentation.

User data configuration

The WSL datasource relies exclusively on the Windows filesystem as the provider of user data. Access to those files is provided by WSL itself unless disabled by the user, thus the datasource doesn't require any special component running on the Windows host to provide such data.

User data can be supplied in any *format supported by cloud-init*, such as YAML cloud-config files or shell scripts. At runtime, the WSL datasource looks for user data in the following locations inside the Windows host filesystem, in the order specified below.

First, configurations from Ubuntu Pro/Landscape are checked for in the following paths:

1. `%USERPROFILE%\.ubuntupro\.cloud-init\<InstanceName>.user-data` holds data provided by Landscape to configure a specific WSL instance. If this file is present, normal user-provided configurations are not looked for. This file is merged with (2) on a per-module basis. If this file is not present, then the first user-provided configuration will be used in its place.
2. `%USERPROFILE%\.ubuntupro\.cloud-init\agent.yaml` holds data provided by the Ubuntu Pro for WSL agent. If this file is present, its modules will be merged with (1), overriding any conflicting modules. If (1) is not provided, then this file will be merged with any valid user-provided configuration instead. Exception is made for Landscape client config computer tags. If user provided data contains a value for `landscape.client.tags` it will be used instead of the one provided by the `agent.yaml`, which is treated as a default.

Then, if a file from (1) is not found, a user-provided configuration will be looked for instead in the following order:

1. `%USERPROFILE%\.cloud-init\<InstanceName>.user-data` holds user data for a specific instance configuration. The datasource resolves the name attributed by WSL to the instance being initialized and looks for this file before any of the subsequent alternatives. Example: `sid-mlkit.user-data` matches an instance named `Sid-MLKit`.

2. %USERPROFILE%\.cloud-init\<ID>-<VERSION_ID>.user-data for the distro-specific configuration, matched by the distro ID and VERSION_ID entries as specified in /etc/os-release. If VERSION_ID is not present, then VERSION_CODENAME will be used instead. Example: ubuntu-22.04.user-data will affect any instance created from an Ubuntu 22.04 Jammy Jellyfish image if a more specific configuration file does not match.
3. %USERPROFILE%\.cloud-init\<ID>-all.user-data for the distro-specific configuration, matched by the distro ID entry in /etc/os-release, regardless of the release version. Example: debian-all.user-data will affect any instance created from any Debian GNU/Linux image, regardless of which release, if a more specific configuration file does not match.
4. %USERPROFILE%\.cloud-init\default.user-data for the configuration affecting all instances, regardless of which distro and release version, if a more specific configuration file does not match. That could be used, for example, to automatically create a user with the same name across all WSL instances a user may have.

Only the first match is loaded, and no config merging is done, even in the presence of errors. That avoids unexpected behaviour due to surprising merge scenarios. Also, notice that the file name casing is irrelevant since both the Windows file names, as well as the WSL distro names, are case-insensitive by default. If none are found, cloud-init remains disabled if no other configurations from previous steps were found.

Note: Some users may have configured case sensitivity for file names on Windows. Note that user data files will still be matched case-insensitively. If there are both *InstanceName.user-data* and *instancename.user-data*, which one will be chosen is arbitrary and should not be relied on. Thus it's recommended to avoid that scenario to prevent confusion.

Since WSL instances are scoped by the Windows user, having the user data files inside the %USERPROFILE% directory (typically C:\Users\<USERNAME>) ensures that WSL instance initialization won't be subject to naming conflicts if the Windows host is shared by multiple users.

Vendor and metadata

The current implementation doesn't allow supplying vendor data. The reasoning is that vendor data adds layering, thus complexity, for no real benefit to the user. Supplying vendor data could be relevant to WSL itself, if the subsystem was aware of cloud-init and intended to leverage it, which is not the case to the best of our knowledge at the time of this writing.

Most of what metadata is intended for is not applicable under WSL, such as setting a hostname. Yet, the knowledge of metadata.instance-id is vital for cloud-init. So, this datasource provides a default value but also supports optionally sourcing metadata from a per-instance specific configuration file: %USERPROFILE%\.cloud-init\<InstanceName>.meta-data. If that file exists, it is a YAML-formatted file minimally providing a value for instance ID such as: instance-id: x-y-z. Advanced users looking to share snapshots or relaunch a snapshot where cloud-init is re-triggered, must run sudo cloud-init clean --logs on the instance before snapshot/export, or create the appropriate .meta-data file containing instance-id: some-new-instance-id.

Unsupported or restricted modules and features

Certain features of cloud-init and its modules either require further customization in the code to better fit the WSL platform or cannot be supported at all due to the constraints of that platform. When writing user-data config files, please check the following restrictions:

- File paths in an include file must be Linux absolute paths.

  Users may be surprised with that requirement since the user data files are inside the Windows file system. But remember that cloud-init is still running inside a Linux instance, and the files referenced in the include user data file will be read by cloud-init, thus they must be represented with paths understandable inside the Linux instance.

Most users will find their Windows system drive mounted as */mnt/c*, so let's consider that assumption in the following example:

`C:\Users\Me\.cloud-init\noble-cpp.user-data`

```
#include
/mnt/c/Users/me/.cloud-init/config.user-data
/mnt/c/Users/me/Downloads/cpp.yaml
```

When initializing an instance named `Noble-Cpp` cloud-init will find that include file, referring to files inside the Windows file system, and will load them effectively. A failure would happen if Windows paths were otherwise in the include file.

- Network configuration is not supported.

  WSL has full control of the instances' networking features and configuration. A limited set of options for networking is exposed to the user via `/etc/wsl.conf`. Those options don't fit well with the networking model cloud-init expects or understands.

- Set hostname.

  WSL automatically assigns the instance hostname and any attempt to change it will take effect only until the next boot when WSL takes over again. The user can set the desired hostname via `/etc/wsl.conf`, if necessary.

- Default user.

  While creating users through cloud-init works as in any other platform, WSL has the concept of the *default user*, which is the user logged in by default. So, to create the default user with cloud-init, one must supply user data to the *Users and Groups module* and write the entry in `/etc/wsl.conf` to make that user the default. See the example:

```
#cloud-config
users:
- name: j
  gecos: Agent J
  groups: users,sudo,netdev,audio
  sudo: ALL=(ALL) NOPASSWD:ALL
  shell: /bin/bash
  lock_passwd: true

write_files:
- path: /etc/wsl.conf
  append: true
  contents: |
    [user]
    default=j
```

- Disk setup, Growpart, Mounts and Resizefs.

  The root filesystem must have the layout expected by WSL. Other mount points may work, depending on how the hardware devices are exposed by the Windows host, and fstab processing during boot is subject to configuration via `/etc/wsl.conf`, so users should expect limited functionality.

- GRUB dpkg.

  WSL controls the boot process, meaning that attempts to install and configure GRUB as any other bootloader won't be effective.

- Resolv conf and update etc/ hosts.

WSL automatically generates those files by default, unless configured to behave otherwise in `/etc/wsl.conf`. Overwriting may work, but only until the next reboot.

ZStack

ZStack platform provides an AWS EC2 metadata service, but with different datasource identity. More information about ZStack can be found at ZStack.

Discovery

To determine whether a VM is running on the ZStack platform, `cloud-init` checks DMI information via `dmidecode -s chassis-asset-tag`. If the output ends with `.zstack.io`, it's running on the ZStack platform.

Metadata

The same way as with EC2, instance metadata can be queried at:

```
GET http://169.254.169.254/2009-04-04/meta-data/
instance-id
local-hostname
```

User data

The same way as with EC2, instance user data can be queried at:

```
GET http://169.254.169.254/2009-04-04/user-data/
meta_data.json
user_data
password
```

2.4.8 Supported distros

Cloud-init has support for multiple different operating systems. Currently support includes various different distributions within the Unix family of operating systems. See the complete list below.

- AlmaLinux
- Alpine Linux
- AOSC OS
- Arch Linux
- CentOS
- CloudLinux
- Container-Optimized OS
- Debian
- DragonFlyBSD
- EuroLinux

- Fedora
- FreeBSD
- Gentoo
- MarinerOS
- MIRACLE LINUX
- NetBSD
- OpenBSD
- openEuler
- OpenCloudOS
- OpenMandriva
- PhotonOS
- Red Hat Enterprise Linux
- Rocky
- SLES/openSUSE
- TencentOS
- Ubuntu
- Virtuozzo

If you would like to add support for another distributions, start by taking a look at another distro module in `cloudinit/distros/`.

Note: While BSD variants are not typically referred to as "distributions", cloud-init has an abstraction to account for operating system differences, which should be contained in cloudinit/distros/.

2.4.9 Network configuration

Default behaviour

`Cloud-init` searches for network configuration in order of increasing precedence; each item overriding the previous.

- **Datasource**: For example, OpenStack may provide network config in the MetaData Service.
- **System config**: A `network:` entry in `/etc/cloud/cloud.cfg.d/*` configuration files.
- **Kernel command line**: `ip=` or `network-config=<Base64 encoded YAML config string>`

Cloud-init will write out the following files representing the network-config processed:

- `/run/cloud-init/network-config.json`: world-readable JSON containing the selected source network-config JSON used by cloud-init network renderers.

User data cannot change an instance's network configuration. In the absence of network configuration in any of the above sources, `cloud-init` will write out a network configuration that will issue a DHCP request on a "first" network interface.

Note: The `network-config` value is expected to be a Base64 encoded YAML string in *Networking config Version 1* or *Networking config Version 2* format. Optionally, it can be compressed with `gzip` prior to Base64 encoding.

Disabling network configuration

Users may disable `cloud-init`'s network configuration capability and rely on other methods, such as embedded configuration or other customisations.

`cloud-init` supports the following methods for disabling `cloud-init`.

Kernel command line

`Cloud-init` will check for the parameter `network-config=disabled`, which will automatically disable any network configuration.

Example disabling kernel command line entry:

```
network-config=disabled
```

Cloud config

In the combined `cloud-init` configuration dictionary, merged from `/etc/cloud/cloud.cfg` and `/etc/cloud/cloud.cfg.d/*`:

```
network:
  config: disabled
```

If `cloud-init`'s networking config has not been disabled, and no other network information is found, then it will proceed to generate a fallback networking configuration.

Disabling network activation

Some datasources may not be initialised until after the network has been brought up. In this case, `cloud-init` will attempt to bring up the interfaces specified by the datasource metadata using a network activator discovered by cloudinit.net.activators.select_activator.

This behaviour can be disabled in the `cloud-init` configuration dictionary, merged from `/etc/cloud/cloud.cfg` and `/etc/cloud/cloud.cfg.d/*`:

```
disable_network_activation: true
```

Fallback network configuration

`Cloud-init` will attempt to determine which, of any attached network devices, is most likely to have a connection and then generate a network configuration to issue a DHCP request on that interface.

`Cloud-init` runs during early boot and does not expect composed network devices (such as Bridges) to be available. `Cloud-init` does not consider the following interface devices as likely "first" network interfaces for fallback configuration; they are filtered out from being selected.

- **loopback**: `name=lo`
- **Virtual Ethernet**: `name=veth*`
- **Software Bridges**: `type=bridge`
- **Software VLANs**: `type=vlan`

`Cloud-init` will prefer network interfaces that indicate they are connected via the Linux `carrier` flag being set. If no interfaces are marked as connected, then all unfiltered interfaces are potential connections.

Of the potential interfaces, `cloud-init` will attempt to pick the "right" interface given the information it has available.

Finally, after selecting the "right" interface, a configuration is generated and applied to the system.

Note: PhotonOS disables fallback networking configuration by default, leaving network unrendered when no other network config is provided. If fallback config is still desired on PhotonOS, it can be enabled by providing `disable_fallback_netcfg: false` in `/etc/cloud/cloud.cfg:sys_config` settings.

Network configuration sources

`Cloud-init` accepts a number of different network configuration formats in support of different cloud substrates. The datasource for these clouds in `cloud-init` will detect and consume datasource-specific network configuration formats for use when writing an instance's network configuration.

The following datasources optionally provide network configuration:

- *Config drive*
  - OpenStack Metadata Service Network
- *DigitalOcean*
  - DigitalOcean JSON metadata
- *LXD*
  - LXD
- *NoCloud*
  - *Networking config Version 1*
  - *Networking config Version 2*
- *OpenStack*
  - OpenStack Metadata Service Network
- *SmartOS Datasource*
  - SmartOS JSON Metadata
- *UpCloud*

  - UpCloud JSON metadata
- *Vultr*
  - Vultr JSON metadata

For more information on network configuration formats:

Networking config Version 1

This network configuration format lets users customise their instance's networking interfaces by assigning subnet configuration, virtual device creation (bonds, bridges, VLANs) routes and DNS configuration.

Required elements of a *network config Version 1* are `config` and `version`.

`Cloud-init` will read this format from *Base configuration*.

For example, the following could be present in `/etc/cloud/cloud.cfg.d/custom-networking.cfg`:

```
network:
  version: 1
  config:
  - type: physical
    name: eth0
    subnets:
      - type: dhcp
```

The *NoCloud* datasource can also provide `cloud-init` networking configuration in this format.

Configuration types

Within the network `config` portion, users include a list of configuration types. The current list of support `type` values are as follows:

- `physical`: Physical
- `bond`: Bond
- `bridge`: Bridge
- `vlan`: VLAN
- `nameserver`: Nameserver
- `route`: Route

Physical, Bond, Bridge and VLAN types may also include IP configuration under the key `subnets`.

- `subnets`: Subnet/IP

Physical

The `physical` type configuration represents a "physical" network device, typically Ethernet-based. At least one of these entries is required for external network connectivity. Type `physical` requires only one key: `name`. A `physical` device may contain some or all of the following keys:

`name: <desired device name>`

A device's name must be less than 15 characters. Names exceeding the maximum will be truncated. This is a limitation of the Linux kernel network-device structure.

`mac_address: <MAC Address>`

The MAC Address is a device unique identifier that most Ethernet-based network devices possess. Specifying a MAC Address is optional. Letters must be lowercase.

Note: It is best practice to "quote" all MAC addresses, since an unquoted MAC address might be incorrectly interpreted as an integer in YAML.

Note: `Cloud-init` will handle the persistent mapping between a device's `name` and the `mac_address`.

`mtu: <MTU SizeBytes>`

The MTU key represents a device's Maximum Transmission Unit, which is the largest size packet or frame, specified in octets (eight-bit bytes), that can be sent in a packet- or frame-based network. Specifying `mtu` is optional.

Note: The possible supported values of a device's MTU are not available at configuration time. It's possible to specify a value too large or to small for a device, and may be ignored by the device.

`accept-ra: <boolean>`

The `accept-ra` key is a boolean value that specifies whether or not to accept Router Advertisements (RA) for this interface. Specifying `accept-ra` is optional.

Physical example

```
network:
  version: 1
  config:
    # Simple network adapter
    - type: physical
      name: interface0
      mac_address: '00:11:22:33:44:55'
```

(continues on next page)

(continued from previous page)

```
    # Second nic with Jumbo frames
    - type: physical
      name: jumbo0
      mac_address: 'aa:11:22:33:44:55'
      mtu: 9000
    # 10G pair
    - type: physical
      name: gbe0
      mac_address: 'cd:11:22:33:44:00'
    - type: physical
      name: gbe1
      mac_address: 'cd:11:22:33:44:02'
```

Bond

A `bond` type will configure a Linux software Bond with one or more network devices. A `bond` type requires the following keys:

`name: <desired device name>`

A device's name must be less than 15 characters. Names exceeding the maximum will be truncated. This is a limitation of the Linux kernel network-device structure.

`mac_address: <MAC Address>`

When specifying MAC Address on a bond this value will be assigned to the bond device and may be different than the MAC address of any of the underlying bond interfaces. Specifying a MAC Address is optional. If `mac_address` is not present, then the bond will use one of the MAC Address values from one of the bond interfaces.

Note: It is best practice to "quote" all MAC addresses, since an unquoted MAC address might be incorrectly interpreted as an integer in YAML.

`bond_interfaces: <List of network device names>`

The `bond_interfaces` key accepts a list of network device `name` values from the configuration. This list may be empty.

mtu: <MTU SizeBytes>

The MTU key represents a device's Maximum Transmission Unit, the largest size packet or frame, specified in octets (eight-bit bytes), that can be sent in a packet- or frame-based network. Specifying `mtu` is optional.

Note: The possible supported values of a device's MTU are not available at configuration time. It's possible to specify a value too large or to small for a device, and may be ignored by the device.

params: <Dictionary of key: value bonding parameter pairs>

The `params` key in a bond holds a dictionary of bonding parameters. This dictionary may be empty. For more details on what the various bonding parameters mean please read the Linux Kernel `Bonding.txt`.

Valid `params` keys are:

- `active_slave`: Set bond attribute
- `ad_actor_key`: Set bond attribute
- `ad_actor_sys_prio`: Set bond attribute
- `ad_actor_system`: Set bond attribute
- `ad_aggregator`: Set bond attribute
- `ad_num_ports`: Set bond attribute
- `ad_partner_key`: Set bond attribute
- `ad_partner_mac`: Set bond attribute
- `ad_select`: Set bond attribute
- `ad_user_port_key`: Set bond attribute
- `all_slaves_active`: Set bond attribute
- `arp_all_targets`: Set bond attribute
- `arp_interval`: Set bond attribute
- `arp_ip_target`: Set bond attribute
- `arp_validate`: Set bond attribute
- `downdelay`: Set bond attribute
- `fail_over_mac`: Set bond attribute
- `lacp_rate`: Set bond attribute
- `lp_interval`: Set bond attribute
- `miimon`: Set bond attribute
- `mii_status`: Set bond attribute
- `min_links`: Set bond attribute
- `mode`: Set bond attribute
- `num_grat_arp`: Set bond attribute
- `num_unsol_na`: Set bond attribute

- `packets_per_slave`: Set bond attribute
- `primary`: Set bond attribute
- `primary_reselect`: Set bond attribute
- `queue_id`: Set bond attribute
- `resend_igmp`: Set bond attribute
- `slaves`: Set bond attribute
- `tlb_dynamic_lb`: Set bond attribute
- `updelay`: Set bond attribute
- `use_carrier`: Set bond attribute
- `xmit_hash_policy`: Set bond attribute

Bond example

```
network:
 version: 1
 config:
   # Simple network adapter
   - type: physical
     name: interface0
     mac_address: '00:11:22:33:44:55'
   # 10G pair
   - type: physical
     name: gbe0
     mac_address: 'cd:11:22:33:44:00'
   - type: physical
     name: gbe1
     mac_address: 'cd:11:22:33:44:02'
   - type: bond
     name: bond0
     bond_interfaces:
       - gbe0
       - gbe1
     params:
       bond-mode: active-backup
```

Bridge

Type `bridge` requires the following keys:

- `name`: Set the name of the bridge.
- `bridge_interfaces`: Specify the ports of a bridge via their `name`. This list may be empty.
- `params`: A list of bridge params. For more details, please read the `bridge-utils-interfaces` manpage.

Valid keys are:

- `bridge_ageing`: Set the bridge's ageing value.
- `bridge_bridgeprio`: Set the bridge device network priority.

- `bridge_fd`: Set the bridge's forward delay.
- `bridge_hello`: Set the bridge's hello value.
- `bridge_hw`: Set the bridge's MAC address.
- `bridge_maxage`: Set the bridge's maxage value.
- `bridge_maxwait`: Set how long network scripts should wait for the bridge to be up.
- `bridge_pathcost`: Set the cost of a specific port on the bridge.
- `bridge_portprio`: Set the priority of a specific port on the bridge.
- `bridge_ports`: List of devices that are part of the bridge.
- `bridge_stp`: Set spanning tree protocol on or off.
- `bridge_waitport`: Set amount of time in seconds to wait on specific ports to become available.

Bridge example

```
network:
 version: 1
 config:
   # Simple network adapter
   - type: physical
     name: interface0
     mac_address: '00:11:22:33:44:55'
   # Second nic with Jumbo frames
   - type: physical
     name: jumbo0
     mac_address: 'aa:11:22:33:44:55'
     mtu: 9000
   - type: bridge
     name: br0
     bridge_interfaces:
       - jumbo0
     params:
       bridge_ageing: 250
       bridge_bridgeprio: 22
       bridge_fd: 1
       bridge_hello: 1
       bridge_maxage: 10
       bridge_maxwait: 0
       bridge_pathcost:
         - jumbo0 75
       bridge_portprio:
         - jumbo0 28
       bridge_stp: 'off'
       bridge_maxwait: 15
```

VLAN

Type `vlan` requires the following keys:

- `name`: Set the name of the VLAN
- `vlan_link`: Specify the underlying link via its `name`.
- `vlan_id`: Specify the VLAN numeric id.

The following optional keys are supported:

```
mtu:  <MTU SizeBytes>
```

The MTU key represents a device's Maximum Transmission Unit, the largest size packet or frame, specified in octets (eight-bit bytes), that can be sent in a packet- or frame-based network. Specifying `mtu` is optional.

Note: The possible supported values of a device's MTU are not available at configuration time. It's possible to specify a value too large or to small for a device and may be ignored by the device.

VLAN example

```
network:
  version: 1
  config:
    # Physical interfaces.
    - type: physical
      name: eth0
      mac_address: 'c0:d6:9f:2c:e8:80'
    # VLAN interface.
    - type: vlan
      name: eth0.101
      vlan_link: eth0
      vlan_id: 101
      mtu: 1500
```

Nameserver

Users can specify a `nameserver` type. Nameserver dictionaries include the following keys:

- `address`: List of IPv4 or IPv6 address of nameservers.
- `search`: Optional. List of hostnames to include in the search path.
- `interface`: Optional. Ties the nameserver definition to the specified interface. The value specified here must match the `name` of an interface defined in this config. If unspecified, this nameserver will be considered a global nameserver.

Nameserver example

```
network:
  version: 1
  config:
    - type: physical
      name: interface0
      mac_address: '00:11:22:33:44:55'
      subnets:
         - type: static
           address: 192.168.23.14/27
           gateway: 192.168.23.1
    - type: nameserver
      interface: interface0  # Ties nameserver to interface0 only
      address:
        - 192.168.23.2
        - 8.8.8.8
      search:
        - exemplary
```

Route

Users can include static routing information as well. A `route` dictionary has the following keys:

- `destination`: IPv4 network address with CIDR netmask notation.
- `gateway`: IPv4 gateway address with CIDR netmask notation.
- `metric`: Integer which sets the network metric value for this route.

Route example

```
network:
  version: 1
  config:
    - type: physical
      name: interface0
      mac_address: '00:11:22:33:44:55'
      subnets:
         - type: static
           address: 192.168.23.14/24
           gateway: 192.168.23.1
    - type: route
      destination: 192.168.24.0/24
      gateway: 192.168.24.1
      metric: 3
```

Subnet/IP

For any network device (one of the "config types") users can define a list of `subnets` which contain ip configuration dictionaries. Multiple subnet entries will create interface aliases, allowing a single interface to use different ip configurations.

Valid keys for `subnets` include the following:

- `type`: Specify the subnet type.
- `control`: Specify 'manual', 'auto' or 'hotplug'. Indicates how the interface will be handled during boot.
- `address`: IPv4 or IPv6 address. It may include CIDR netmask notation.
- `netmask`: IPv4 subnet mask in dotted format or CIDR notation.
- `broadcast` : IPv4 broadcast address in dotted format. This is only rendered if `/etc/network/interfaces` is used.
- `gateway`: IPv4 address of the default gateway for this subnet.
- `dns_nameservers`: Specify a list of IPv4 DNS server IPs.
- `dns_search`: Specify a list of DNS search paths.
- `routes`: Specify a list of routes for a given interface.

Subnet types are one of the following:

- `dhcp4`: Configure this interface with IPv4 dhcp.
- `dhcp`: Alias for `dhcp4`.
- `dhcp6`: Configure this interface with IPv6 dhcp.
- `static`: Configure this interface with a static IPv4.
- `static6`: Configure this interface with a static IPv6.
- `ipv6_dhcpv6-stateful`: Configure this interface with `dhcp6`.
- `ipv6_dhcpv6-stateless`: Configure this interface with SLAAC and DHCP.
- `ipv6_slaac`: Configure address with SLAAC.

When making use of `dhcp` or either of the `ipv6_dhcpv6` types, no additional configuration is needed in the subnet dictionary.

Using `ipv6_dhcpv6-stateless` or `ipv6_slaac` allows the IPv6 address to be automatically configured with State-Less Address AutoConfiguration (SLAAC). SLAAC requires support from the network, so verify that your cloud or network offering has support before trying it out. With `ipv6_dhcpv6-stateless`, DHCPv6 is still used to fetch other subnet details such as gateway or DNS servers. If you only want to discover the address, use `ipv6_slaac`.

Subnet DHCP example

```
network:
  version: 1
  config:
    - type: physical
      name: interface0
      mac_address: '00:11:22:33:44:55'
```

(continues on next page)

(continued from previous page)

```
    subnets:
      - type: dhcp
```

Subnet static example

```
network:
  version: 1
  config:
    - type: physical
      name: interface0
      mac_address: '00:11:22:33:44:55'
      subnets:
        - type: static
          address: 192.168.23.14/27
          gateway: 192.168.23.1
          dns_nameservers:
            - 192.168.23.2
            - 8.8.8.8
          dns_search:
            - exemplary.maas
```

Multiple subnet example

The following will result in an `interface0` using DHCP and `interface0:1` using the static subnet configuration:

```
network:
  version: 1
  config:
    - type: physical
      name: interface0
      mac_address: '00:11:22:33:44:55'
      subnets:
        - type: dhcp
        - type: static
          address: 192.168.23.14/27
          gateway: 192.168.23.1
          dns_nameservers:
            - 192.168.23.2
            - 8.8.8.8
          dns_search:
            - exemplary
```

Subnet with routes example

```
network:
  version: 1
  config:
    - type: physical
      name: interface0
      mac_address: '00:11:22:33:44:55'
      subnets:
        - type: dhcp
        - type: static
          address: 10.184.225.122
          netmask: 255.255.255.252
          routes:
            - gateway: 10.184.225.121
              netmask: 255.240.0.0
              destination: 10.176.0.0
            - gateway: 10.184.225.121
              netmask: 255.240.0.0
              destination: 10.208.0.0
```

Multi-layered configurations

Complex networking sometimes uses layers of configuration. The syntax allows users to build those layers one at a time. All of the virtual network devices supported allow specifying an underlying device by their `name` value.

Bonded VLAN example

```
network:
  version: 1
  config:
    # 10G pair
    - type: physical
      name: gbe0
      mac_address: 'cd:11:22:33:44:00'
    - type: physical
      name: gbe1
      mac_address: 'cd:11:22:33:44:02'
    # Bond.
    - type: bond
      name: bond0
      bond_interfaces:
        - gbe0
        - gbe1
      params:
        bond-mode: 802.3ad
        bond-lacp_rate: fast
    # A Bond VLAN.
    - type: vlan
      name: bond0.200
```

(continues on next page)

(continued from previous page)

```
      vlan_link: bond0
      vlan_id: 200
      subnets:
        - type: dhcp4
```

Multiple VLAN example

```
network:
  version: 1
  config:
  - name: eth0
    mac_address: 'd4:be:d9:a8:49:13'
    mtu: 1500
    subnets:
    - address: 10.245.168.16/21
      dns_nameservers:
      - 10.245.168.2
      gateway: 10.245.168.1
      type: static
    type: physical
  - name: eth1
    mac_address: 'd4:be:d9:a8:49:15'
    mtu: 1500
    subnets:
    - address: 10.245.188.2/24
      dns_nameservers: []
      type: static
    type: physical
  - name: eth1.2667
    mtu: 1500
    subnets:
    - address: 10.245.184.2/24
      dns_nameservers: []
      type: static
    type: vlan
    vlan_id: 2667
    vlan_link: eth1
  - name: eth1.2668
    mtu: 1500
    subnets:
    - address: 10.245.185.1/24
      dns_nameservers: []
      type: static
    type: vlan
    vlan_id: 2668
    vlan_link: eth1
  - name: eth1.2669
    mtu: 1500
    subnets:
    - address: 10.245.186.1/24
```

(continues on next page)

(continued from previous page)

```
    dns_nameservers: []
    type: static
  type: vlan
  vlan_id: 2669
  vlan_link: eth1
- name: eth1.2670
  mtu: 1500
  subnets:
  - address: 10.245.187.2/24
    dns_nameservers: []
    type: static
  type: vlan
  vlan_id: 2670
  vlan_link: eth1
- address: [10.245.168.2]
  search:
  - dellstack
  type: nameserver
```

Networking config Version 2

`Cloud-init`'s support for Version 2 network config is a subset of the Version 2 format defined for the Netplan tool. `Cloud-init` supports both reading and writing of Version 2. Writing support requires a distro with Netplan present.

Netplan passthrough

On a system with Netplan present, `cloud-init` will pass Version 2 configuration through to Netplan without modification. On such systems, you do not need to limit yourself to the below subset of Netplan's configuration format.

Warning: If you are writing or generating network configuration that may be used on non-netplan systems, you **must** limit yourself to the subset described in this document, or you will see network configuration failures on non-netplan systems.

Version 2 configuration format

The `network` key has at least two required elements. First, it must include `version: 2` and one or more of possible device `types`.

`Cloud-init` will read this format from *Base configuration*.

For example the following could be present in `/etc/cloud/cloud.cfg.d/custom-networking.cfg`:

```
network:
  version: 2
  ethernets: {}
```

It may also be provided in other locations including the *NoCloud*. See *Network configuration* for other places.

Supported device `types` values are as follows:

- `ethernets`: Ethernets
- `bonds`: Bonds
- `bridges`: Bridges
- `vlans`: VLANs

Each `type` block contains device definitions as a map (where the keys are called "configuration IDs"). Each entry under the `types` may include IP and/or device configuration.

Device configuration IDs

The key names below the per-device-type definition maps (like `ethernets:`) are called "ID"s. They must be unique throughout the entire set of configuration files. Their primary purpose is to serve as anchor names for composite devices, for example to enumerate the members of a bridge that is currently being defined.

There are two physically/structurally different classes of device definitions, and the ID field has a different interpretation for each:

Physical devices (e.g., ethernet, wifi)

These can dynamically come and go between reboots and even during runtime (hotplugging). In the generic case, they can be selected by `match:` rules on desired properties, such as name/name pattern, MAC address, or driver. In general these will match any number of devices (unless they refer to properties which are unique such as the full path or MAC address), so without further knowledge about the hardware, these will always be considered as a group.

It is valid to specify no match rules at all, in which case the ID field is simply the interface name to be matched. This is mostly useful if you want to keep simple cases simple, and it's how network device configuration has been done for a long time.

If there are `match:` rules, then the ID field is a purely opaque name which is only being used for references from definitions of compound devices in the config.

Virtual devices (e.g., veth, bridge, bond)

These are fully under the control of the config file(s) and the network stack, i.e., these devices are being created instead of matched. Thus `match:` and `set-name:` are not applicable for these, and the ID field is the name of the created virtual device.

Common properties for physical device types

`match: <(mapping)>`

This selects a subset of available physical devices by various hardware properties. The following configuration will then apply to all matching devices, as soon as they appear. *All* specified properties must match. The following properties for creating matches are supported:

name: <(scalar)>

Current interface name. Globs are supported, and the primary use case for matching on names, as selecting one fixed name can be more easily achieved with having no `match:` at all and just using the ID (see above). Note that currently only `networkd` supports globbing, `NetworkManager` does not.

Example:

```
# all cards on second PCI bus
match:
  name: enp2*
```

macaddress: <(scalar)>

Device's MAC address in the form xx:xx:xx:xx:xx:xx. Globs are not allowed. Letters must be lowercase.

Example:

```
# fixed MAC address
match:
  macaddress: "11:22:33:aa:bb:ff"
```

Note: It is best practice to "quote" all MAC addresses, since an unquoted MAC address might be incorrectly interpreted as an integer in YAML.

driver: <(scalar)>

Kernel driver name, corresponding to the `DRIVER` udev property. Globs are supported. Matching on driver is *only* supported with `networkd`.

Example:

```
# first card of driver ``ixgbe``
match:
  driver: ixgbe
  name: en*s0
```

set-name: <(scalar)>

When matching on unique properties such as path or MAC, or with additional assumptions such as "there will only ever be one wifi device", match rules can be written so that they only match one device. Then this property can be used to give that device a more specific/desirable/nicer name than the default from udev's `ifnames`. Any additional device that satisfies the match rules will then fail to get renamed and keep the original kernel name (and dmesg will show an error).

`wakeonlan: <(bool)>`

Enable wake on LAN. Off by default.

Common properties for all device types

`renderer: <(scalar)>`

Use the given networking backend for this definition. Currently supported are `networkd` and `NetworkManager`. This property can be specified globally in `networks:`, for a device type (e.g., in `ethernets:`) or for a particular device definition. Default is `networkd`.

Note: `Cloud-init` only supports networkd backend if rendering `version2` config to the instance.

`dhcp4: <(bool)>`

Enable DHCP for IPv4. Off by default.

`dhcp6: <(bool)>`

Enable DHCP for IPv6. Off by default.

`dhcp4-overrides and dhcp6-overrides: <(mapping)>`

DHCP behaviour overrides. Overrides will only have an effect if the corresponding DHCP type is enabled. Refer to Netplan#dhcp-overrides for more documentation.

Note: These properties are only consumed on `netplan` and `networkd` renderers.

The `netplan` renderer *passes through* everything and the `networkd` renderer consumes the following sub-properties:

- `hostname` *
- `route-metric` *
- `send-hostname` *
- `use-dns`
- `use-domains`
- `use-hostname`
- `use-mtu` *
- `use-ntp`
- `use-routes` *

Note: Sub-properties marked with a * are unsupported for `dhcp6-overrides` when used with the `networkd` renderer.

Example:

```
dhcp4-overrides:
  hostname: hal
  route-metric: 1100
  send-hostname: false
  use-dns: false
  use-domains: false
  use-hostname: false
  use-mtu: false
  use-ntp: false
  use-routes: false
```

addresses: <(sequence of scalars)>

Add static addresses to the interface in addition to the ones received through DHCP or RA. Each sequence entry is in CIDR notation, i.e., of the form `addr/prefixlen`. `addr` is an IPv4 or IPv6 address as recognised by `inet_pton(3)` and `prefixlen` the number of bits of the subnet.

Example: `addresses: [192.168.14.2/24, 2001:1::1/64]`

gateway4: or gateway6: <(scalar)>

Deprecated, see Netplan#default-routes. Set default gateway for IPv4/6, for manual address configuration. This requires setting `addresses` too. Gateway IPs must be in a form recognised by `inet_pton(3)`

Example for IPv4: `gateway4: 172.16.0.1` Example for IPv6: `gateway6: 2001:4::1`

mtu: <MTU SizeBytes>

The MTU key represents a device's Maximum Transmission Unit, the largest size packet or frame, specified in octets (eight-bit bytes), that can be sent in a packet- or frame-based network. Specifying `mtu` is optional.

nameservers: <(mapping)>

Set DNS servers and search domains, for manual address configuration. There are two supported fields: `addresses:` is a list of IPv4 or IPv6 addresses similar to `gateway*`, and `search:` is a list of search domains.

Example:

```
nameservers:
  search: [lab, home]
  addresses: [8.8.8.8, FEDC::1]
```

routes: *<(sequence of mapping)>*

Add device specific routes. Each mapping includes a `to`, `via` key with an IPv4 or IPv6 address as value. `metric` is an optional value.

Example:

```
routes:
 - to: 0.0.0.0/0
   via: 10.23.2.1
   metric: 3
```

Ethernets

Ethernet device definitions do not support any specific properties beyond the common ones described above.

Bonds

interfaces: *<(sequence of scalars)>*

All devices matching this ID list will be added to the bond.

Example:

```
ethernets:
  switchports:
    match: {name: "enp2*"}
[...]
bonds:
  bond0:
    interfaces: [switchports]
```

parameters: *<(mapping)>*

Customisation parameters for special bonding options. Time values are specified in seconds unless otherwise specified.

mode: *<(scalar)>*

Set the bonding mode used for the interfaces. The default is `balance-rr` (round robin). Possible values are `balance-rr`, `active-backup`, `balance-xor`, `broadcast`, `802.3ad`, `balance-tlb`, and `balance-alb`.

lacp-rate: <(scalar)>

Set the rate at which LACPDUs are transmitted. This is only useful in 802.3ad mode. Possible values are `slow` (30 seconds, default), and `fast` (every second).

mii-monitor-interval: <(scalar)>

Specifies the interval for MII monitoring (verifying if an interface of the bond has carrier). The default is `0`; which disables MII monitoring.

min-links: <(scalar)>

The minimum number of links up in a bond to consider the bond interface to be up.

transmit-hash-policy: <(scalar)>

Specifies the transmit hash policy for the selection of slaves. This is only useful in balance-xor, 802.3ad and balance-tlb modes. Possible values are `layer2`, `layer3+4`, `layer2+3`, `encap2+3`, and `encap3+4`.

ad-select: <(scalar)>

Set the aggregation selection mode. Possible values are `stable`, `bandwidth`, and `count`. This option is only used in 802.3ad mode.

all-slaves-active: <(bool)>

If the bond should drop duplicate frames received on inactive ports, set this option to `false`. If they should be delivered, set this option to `true`. The default value is false, and is the desirable behaviour in most situations.

arp-interval: <(scalar)>

Set the interval value for how frequently ARP link monitoring should happen. The default value is `0`, which disables ARP monitoring.

arp-ip-targets: <(sequence of scalars)>

IPs of other hosts on the link which should be sent ARP requests in order to validate that a slave is up. This option is only used when `arp-interval` is set to a value other than `0`. At least one IP address must be given for ARP link monitoring to function. Only IPv4 addresses are supported. You can specify up to 16 IP addresses. The default value is an empty list.

`arp-validate: <(scalar)>`

Configure how ARP replies are to be validated when using ARP link monitoring. Possible values are `none`, `active`, `backup`, and `all`.

`arp-all-targets: <(scalar)>`

Specify whether to use any ARP IP target being up as sufficient for a slave to be considered up; or if all the targets must be up. This is only used for `active-backup` mode when `arp-validate` is enabled. Possible values are `any` and `all`.

`up-delay: <(scalar)>`

Specify the delay before enabling a link once the link is physically up. The default value is `0`.

`down-delay: <(scalar)>`

Specify the delay before disabling a link once the link has been lost. The default value is `0`.

`fail-over-mac-policy: <(scalar)>`

Set whether to set all slaves to the same MAC address when adding them to the bond, or how else the system should handle MAC addresses. The possible values are `none`, `active`, and `follow`.

`gratuitous-arp: <(scalar)>`

Specify how many ARP packets to send after failover. Once a link is up on a new slave, a notification is sent and possibly repeated if this value is set to a number greater than `1`. The default value is `1` and valid values are between `1` and `255`. This only affects `active-backup` mode.

`packets-per-slave: <(scalar)>`

In `balance-rr` mode, specifies the number of packets to transmit on a slave before switching to the next. When this value is set to `0`, slaves are chosen at random. Allowable values are between `0` and `65535`. The default value is `1`. This setting is only used in `balance-rr` mode.

`primary-reselect-policy: <(scalar)>`

Set the reselection policy for the primary slave. On failure of the active slave, the system will use this policy to decide how the new active slave will be chosen and how recovery will be handled. The possible values are `always`, `better`, and `failure`.

learn-packet-interval: <(scalar)>

Specify the interval between sending Learning packets to each slave. The value range is between `1` and `0x7fffffff`. The default value is `1`. This option only affects `balance-tlb` and `balance-alb` modes.

Bridges

interfaces: <(sequence of scalars)>

All devices matching this ID list will be added to the bridge.

Example:

```
ethernets:
  switchports:
    match: {name: "enp2*"}
[...]
bridges:
  br0:
    interfaces: [switchports]
```

parameters: <(mapping)>

Customisation parameters for special bridging options. Time values are specified in seconds unless otherwise stated.

ageing-time: <(scalar)>

Set the period of time to keep a MAC address in the forwarding database after a packet is received.

priority: <(scalar)>

Set the priority value for the bridge. This value should be a number between `0` and `65535`. Lower values mean higher priority. The bridge with the higher priority will be elected as the root bridge.

forward-delay: <(scalar)>

Specify the period of time the bridge will remain in Listening and Learning states before getting to the Forwarding state. This value should be set in seconds for the `systemd` backend, and in milliseconds for the `NetworkManager` backend.

hello-time: <(scalar)>

Specify the interval between two hello packets being sent out from the root and designated bridges. Hello packets communicate information about the network topology.

max-age: <(scalar)>

Set the maximum age of a hello packet. If the last hello packet is older than that value, the bridge will attempt to become the root bridge.

path-cost: <(scalar)>

Set the cost of a path on the bridge. Faster interfaces should have a lower cost. This allows a finer control on the network topology so that the fastest paths are available whenever possible.

stp: <(bool)>

Define whether the bridge should use Spanning Tree Protocol. The default value is "true", which means that Spanning Tree should be used.

VLANs

id: <(scalar)>

VLAN ID, a number between 0 and 4094.

link: <(scalar)>

ID of the underlying device definition on which this VLAN gets created.

Example:

```
ethernets:
  eno1: {...}
vlans:
  en-intra:
    id: 1
    link: eno1
    dhcp4: yes
  en-vpn:
    id: 2
    link: eno1
    address: ...
```

Examples

Configure an ethernet device with `networkd`, identified by its name, and enable DHCP:

```
network:
  version: 2
  ethernets:
    eno1:
      dhcp4: true
```

This is a complex example which shows most available features:

```
network:
  version: 2
  ethernets:
    # opaque ID for physical interfaces, only referred to by other stanzas
    id0:
      match:
        macaddress: '00:11:22:33:44:55'
      wakeonlan: true
      dhcp4: true
      addresses:
        - 192.168.14.2/24
        - 2001:1::1/64
      gateway4: 192.168.14.1
      gateway6: 2001:1::2
      nameservers:
        search: [foo.local, bar.local]
        addresses: [8.8.8.8]
      # static routes
      routes:
        - to: 192.0.2.0/24
          via: 11.0.0.1
          metric: 3
    lom:
      match:
        driver: ixgbe
      # you are responsible for setting tight enough match rules
      # that only match one device if you use set-name
      set-name: lom1
      dhcp6: true
    switchports:
      # all cards on second PCI bus; unconfigured by themselves, will be added
      # to br0 below
      match:
        name: enp2*
      mtu: 1280
  bonds:
    bond0:
      interfaces: [id0, lom]
  bridges:
    # the key name is the name for virtual (created) interfaces; no match: and
    # set-name: allowed
```

(continues on next page)

(continued from previous page)

```
    br0:
      # IDs of the components; switchports expands into multiple interfaces
      interfaces: [wlp1s0, switchports]
      dhcp4: true
  vlans:
    en-intra:
      id: 1
      link: id0
      dhcp4: yes
```

Network configuration outputs

`Cloud-init` converts various forms of user-supplied or automatically generated configuration into an internal network configuration state. From this state, `cloud-init` delegates rendering of the configuration to distro-supported formats. The following `renderers` are supported in `cloud-init`:

NetworkManager

NetworkManager is the standard Linux network configuration tool suite. It supports a wide range of networking setups. Configuration is typically stored in `/etc/NetworkManager`.

It is the default for a number of Linux distributions; notably Fedora, CentOS/RHEL, and their derivatives.

ENI

`/etc/network/interfaces` or `ENI` is supported by the `ifupdown` package found in Alpine Linux, Debian and Ubuntu.

Netplan

Introduced in Ubuntu 16.10 (Yakkety Yak), Netplan has been the default network configuration tool in Ubuntu since 17.10 (Artful Aardvark). Netplan consumes *Networking config Version 2* input and renders network configuration for supported backends such as `systemd-networkd` and `NetworkManager`.

Sysconfig

Sysconfig format is used by RHEL, CentOS, Fedora and other derivatives.

NetBSD, OpenBSD, FreeBSD

Network renders supporting BSD releases, which typically write configuration to `/etc/rc.conf`. Unique to BSD renderers is that each renderer also calls something akin to FreeBSD.start_services which will invoke applicable network services to setup the network, making network activators unneeded for BSD flavors at the moment.

Network output policy

Note: These are **upstream** defaults and are known to be overridden by downstream distributions.

The default policy for selecting a network `renderer` (in order of preference) is as follows:

- ENI
- Sysconfig
- Netplan
- NetworkManager
- FreeBSD
- NetBSD
- OpenBSD
- Networkd

The default policy for selecting a network `activator` (in order of preference) is as follows:

- **ENI**: using `ifup`, `ifdown` to manage device setup/teardown
- **Netplan**: using `netplan apply` to manage device setup/teardown
- **NetworkManager**: using `nmcli` to manage device setup/teardown
- **Networkd**: using `ip` to manage device setup/teardown

When applying the policy, `cloud-init` checks if the current instance has the correct binaries and paths to support the renderer. The first renderer that can be used is selected. Users may override the network renderer policy by supplying an updated configuration in cloud-config.

```
system_info:
  network:
    renderers: ['netplan', 'network-manager', 'eni', 'sysconfig', 'freebsd', 'netbsd',
↪'openbsd']
    activators: ['eni', 'netplan', 'network-manager', 'networkd']
```

Network configuration tools

`Cloud-init` contains a command used to test input/output conversion between formats. The `tools/net-convert.py` in the `cloud-init` source repository is helpful in examining expected output for a given input format. If running these commands from the cloud-init source directory, make sure to set the correct path `PYTHON_PATH=.`

CLI Interface:

```
$ cloud-init devel net-convert --help
```

Example output:

```
usage: /usr/bin/cloud-init devel net-convert [-h] -p PATH -k {eni,network_data.json,yaml,
↪azure-imds,vmware-imc} -d PATH -D
                                             {alpine,arch,azurelinux,debian,ubuntu,
↪freebsd,dragonfly,gentoo,cos,netbsd,openbsd,almalinux,amazon,centos,cloudlinux,
↪eurolinux,fedora,mariner,miraclelinux,openmandriva,photon,rhel,rocky,virtuozzo,
↪opensuse,sles,openEuler}
                                             [-m name,mac] [--debug] -O {eni,netplan,
↪networkd,sysconfig,network-manager}

options:
  -h, --help            show this help message and exit
  -p PATH, --network-data PATH
                        The network configuration to read
  -k {eni,network_data.json,yaml,azure-imds,vmware-imc}, --kind {eni,network_data.json,
↪yaml,azure-imds,vmware-imc}
                        The format of the given network config
  -d PATH, --directory PATH
                        directory to place output in
  -D {alpine,arch,azurelinux,debian,ubuntu,freebsd,dragonfly,gentoo,cos,netbsd,openbsd,
↪almalinux,amazon,centos,cloudlinux,eurolinux,fedora,mariner,miraclelinux,openmandriva,
↪photon,rhel,rocky,virtuozzo,opensuse,sles,openeuler}, --distro {alpine,arch,azurelinux,
↪debian,ubuntu,freebsd,dragonfly,gentoo,cos,netbsd,openbsd,almalinux,amazon,centos,
↪cloudlinux,eurolinux,fedora,mariner,miraclelinux,openmandriva,photon,rhel,rocky,
↪virtuozzo,opensuse,sles,openEuler}
  -m name,mac, --mac name,mac
                        interface name to mac mapping
  --debug               enable debug logging to stderr.
  -O {eni,netplan,networkd,sysconfig,network-manager}, --output-kind {eni,netplan,
↪networkd,sysconfig,network-manager}
                        The network config format to emit
```

Example of converting V2 to sysconfig:

```
$ cloud-init devel net-convert --network-data v2.yaml --kind yaml \
   --output-kind sysconfig -d target
$ cat target/etc/sysconfig/network-scripts/ifcfg-eth*
```

Example output:

```
# Created by cloud-init automatically, do not edit.
#
BOOTPROTO=static
```

(continues on next page)

(continued from previous page)

```
DEVICE=eth7
IPADDR=192.168.1.5/255.255.255.0
ONBOOT=yes
TYPE=Ethernet
USERCTL=no
# Created by cloud-init automatically, do not edit.
#
BOOTPROTO=dhcp
DEVICE=eth9
ONBOOT=yes
TYPE=Ethernet
USERCTL=no
```

2.4.10 Base configuration

Warning: This documentation is intended for custom image creators, such as distros and cloud providers, not end users. Modifying the base configuration should not be necessary for end users and can result in a system that may be unreachable or may no longer boot.

`Cloud-init` base config is primarily defined in two places:

- `/etc/cloud/cloud.cfg`
- `/etc/cloud/cloud.cfg.d/*.cfg`

See the *configuration sources explanation* for more information on how these files get sourced and combined with other configuration.

Generation

`cloud.cfg` isn't present in any of `cloud-init`'s source files. The configuration is templated and customised for each distribution supported by `cloud-init`.

Base configuration keys

Module keys

Modules are grouped into the following keys:

- `cloud_init_modules`: Modules run during *network* timeframe.
- `cloud_config_modules`: Modules run during *config* timeframe.
- `cloud_final_modules`: Modules run during *final* timeframe.

Each `module` definition contains an array of strings, where each string is the name of the module. Each name is taken directly from the module filename, with the `cc_` prefix and `.py` suffix removed, and with `-` and `_` being interchangeable.

Alternatively, in place of the module name, an array of `<name>, <frequency>[, <args>]` args may be specified. See *the module creation guidelines* for more information on `frequency` and `args`.

Note: Most modules won't run at all if they're not triggered via a respective user data key, so removing modules or changing the run frequency is **not** a recommended way to reduce instance boot time.

Examples

To specify that only cc_final_message.py run during final timeframe:

```
cloud_final_modules:
- final_message
```

To change the frequency from the default of `ALWAYS` to `ONCE`:

```
cloud_final_modules:
- [final_message, once]
```

To include default arguments to the module (that may be overridden by user data):

```
cloud_final_modules:
- [final_message, once, "my final message"]
```

Datasource keys

Many datasources allow configuration of the datasource for use in querying the datasource for metadata using the `datasource` key. This configuration is datasource dependent and can be found under each datasource's respective *documentation*. It will generally take the form of:

```
datasource:
  <datasource_name>:
    ...
```

System info keys

These keys are used for setup of `cloud-init` itself, or the datasource or distro. Anything under `system_info` cannot be overridden by vendor data, user data, or any other handlers or transforms. In some cases there may be a `system_info` key used for the distro, while the same key is used outside of `system_info` for a user data module. Both keys will be processed independently.

- `system_info`: Top-level key.
  - `paths`: Definitions of common paths used by `cloud-init`.
    - `cloud_dir`: Default: `/var/lib/cloud`.
    - `templates_dir`: Default: `/etc/cloud/templates`.
  - `distro`: Name of distro being used.
  - `default_user`: Defines the default user for the system using the same user configuration as *Users and Groups*. Note that this CAN be overridden if a `users` configuration is specified without a `- default` entry.
  - `ntp_client`: The default NTP client for the distro. Takes the same form as `ntp_client` defined in *NTP*.

- `package_mirrors`: Defines the package mirror info for apt.
- `ssh_svcname`: The SSH service name. For most distros this will be either `ssh` or `sshd`.
- `network`: Top-level key for distro-specific networking configuration.
  - `renderers`: Prioritised list of networking configurations to try on this system. The first valid entry found will be used. Options are:
    - `eni`: For `/etc/network/interfaces`.
    - `network-manager`
    - `netplan`
    - `networkd`: For `systemd-networkd`.
    - `freebsd`
    - `netbsd`
    - `openbsd`
  - `activators`: Prioritised list of networking tools to try to activate network on this system. The first valid entry found will be used. Options are:
    - `eni`: For `ifup/ifdown`.
    - `netplan`: For `netplan generate/netplan apply`.
    - `network-manager`: For `nmcli connection load/ nmcli connection up`.
    - `networkd`: For `ip link set up/ip link set down`.
- `apt_get_command`: Command used to interact with APT repositories. Default: `apt-get`.
- `apt_get_upgrade_subcommand`: APT subcommand used to upgrade system. Default: `dist-upgrade`.
- `apt_get_wrapper`: Command used to wrap the apt-get command.
  - `enabled`: Whether to use the specified `apt_wrapper` command. If set to `auto`, use the command if it exists on the `PATH`. Default: `true`.
  - `command`: Command used to wrap any `apt-get` calls. Default: `eatmydata`.

Logging keys

See *the logging explanation* for a comprehensive logging explanation. Note that `cloud-init` has a default logging definition that shouldn't need to be altered. It is defined in this instance at `/etc/cloud/cloud.cfg.d/05_logging.cfg`.

The logging keys used in the base configuration are as follows:

logcfg

A standard python fileConfig formatted log configuration. This is the primary logging configuration key and will take precedence over `log_cfgs` or `log_basic` keys.

log_cfgs

A list of logging configs in fileConfig format to apply when running `cloud-init`. Note that `log_cfgs` is used in `/etc/cloud.cfg.d/05_logging.cfg`.

log_basic

Boolean value to determine if `cloud-init` should apply a basic default logging configuration if none has been provided. Defaults to `true` but only takes effect if `logcfg` or `log_cfgs` hasn't been defined.

output

If and how to redirect `stdout/stderr`. Defined in `/etc/cloud.cfg.d/05_logging.cfg` and explained in *the logging explanation*.

syslog_fix_perms

Takes a list of <`owner:group`> strings and will set the owner of `def_log_file` accordingly.

def_log_file

Only used in conjunction with `syslog_fix_perms`. Specifies the filename to be used for setting permissions. Defaults to `/var/log/cloud-init.log`.

Other keys

network

The *network configuration* to be applied to this instance.

datasource_pkg_list

Prioritised list of python packages to search when finding a datasource. Automatically includes `cloudinit.sources`.

datasource_list

This key contains a prioritised list of datasources that `cloud-init` attempts to discover on boot. By default, this is defined in `/etc/cloud/cloud.cfg.d`.

There are a few reasons to modify the `datasource_list`:

1. Override default datasource discovery priority order
2. Force cloud-init to use a specific datasource: A single entry in the list (or a single entry and `None`) will override datasource discovery, which will force the specified datasource to run.
3. Remove known invalid datasources: this might improve boot speed on distros that do not use `ds-identify` to detect and select the datasource,

> **Warning:** This key is unique in that it uses a subset of YAML syntax. It **requires** that the key and its contents, a list, must share a single line - no newlines.

vendor_data/vendor_data2

Allows the user to disable `vendor_data` or `vendor_data2` along with providing a prefix for any executed scripts.

Format is a dict with `enabled` and `prefix` keys:

- `enabled`: A boolean indicating whether to enable or disable the `vendor_data`.
- `prefix`: A path to prepend to any `vendor_data`-provided script.

manual_cache_clean

By default, cloud-init searches for a datasource on every boot. Setting this to `true` will disable this behaviour. This is useful if your datasource information will not be present every boot. Default: `false`.

Example

On an Ubuntu system, `/etc/cloud/cloud.cfg` should look similar to:

```
# The top level settings are used as module and base configuration.
# A set of users which may be applied and/or used by various modules
# when a 'default' entry is found it will reference the 'default_user'
# from the distro configuration specified below
users:
- default


# If this is set, 'root' will not be able to ssh in and they
# will get a message to login instead as the default $user
disable_root: true

# This will cause the set+update hostname module to not operate (if true)
preserve_hostname: false
```

(continues on next page)

(continued from previous page)

```
# If you use datasource_list array, keep array items in a single line.
# If you use multi line array, ds-identify script won't read array items.
# Example datasource config
# datasource:
#    Ec2:
#      metadata_urls: [ 'blah.com' ]
#      timeout: 5 # (defaults to 50 seconds)
#      max_wait: 10 # (defaults to 120 seconds)

# The modules that run in the 'init' stage
cloud_init_modules:
- seed_random
- bootcmd
- write_files
- growpart
- resizefs
- disk_setup
- mounts
- set_hostname
- update_hostname
- update_etc_hosts
- ca_certs
- rsyslog
- users_groups
- ssh

# The modules that run in the 'config' stage
cloud_config_modules:
- wireguard
- snap
- ubuntu_autoinstall
- ssh_import_id
- keyboard
- locale
- set_passwords
- grub_dpkg
- apt_pipelining
- apt_configure
- ubuntu_pro
- ntp
- timezone
- disable_ec2_metadata
- runcmd
- byobu

# The modules that run in the 'final' stage
cloud_final_modules:
- package_update_upgrade_install
- fan
- landscape
- lxd
- ubuntu_drivers
```

(continues on next page)

(continued from previous page)

```
- write_files_deferred
- puppet
- chef
- ansible
- mcollective
- salt_minion
- reset_rmc
- scripts_vendor
- scripts_per_once
- scripts_per_boot
- scripts_per_instance
- scripts_user
- ssh_authkey_fingerprints
- keys_to_console
- install_hotplug
- phone_home
- final_message
- power_state_change

# System and/or distro specific settings
# (not accessible to handlers/transforms)
system_info:
  # This will affect which distro class gets used
  distro: ubuntu
  # Default user name + that default users groups (if added/used)
  default_user:
    name: ubuntu
    doas:
      - permit nopass ubuntu
    lock_passwd: True
    gecos: Ubuntu
    groups: [adm, cdrom, dip, lxd, sudo]
    sudo: ["ALL=(ALL) NOPASSWD:ALL"]
    shell: /bin/bash
  network:
    dhcp_client_priority: [dhclient, dhcpcd, udhcpc]
    renderers: ['netplan', 'eni', 'sysconfig']
    activators: ['netplan', 'eni', 'network-manager', 'networkd']
  # Automatically discover the best ntp_client
  ntp_client: auto
  # Other config here will be given to the distro class and/or path classes
  paths:
    cloud_dir: /var/lib/cloud/
    templates_dir: /etc/cloud/templates/
  package_mirrors:
    - arches: [i386, amd64]
    failsafe:
        primary: http://archive.ubuntu.com/ubuntu
        security: http://security.ubuntu.com/ubuntu
    search:
        primary:
        - http://%(ec2_region)s.ec2.archive.ubuntu.com/ubuntu/
```

(continues on next page)

(continued from previous page)

```
        - http://%(availability_zone)s.clouds.archive.ubuntu.com/ubuntu/
        - http://%(region)s.clouds.archive.ubuntu.com/ubuntu/
        security: []
    - arches: [arm64, armel, armhf]
    failsafe:
        primary: http://ports.ubuntu.com/ubuntu-ports
        security: http://ports.ubuntu.com/ubuntu-ports
    search:
        primary:
        - http://%(ec2_region)s.ec2.ports.ubuntu.com/ubuntu-ports/
        - http://%(availability_zone)s.clouds.ports.ubuntu.com/ubuntu-ports/
        - http://%(region)s.clouds.ports.ubuntu.com/ubuntu-ports/
        security: []
    - arches: [default]
    failsafe:
        primary: http://ports.ubuntu.com/ubuntu-ports
        security: http://ports.ubuntu.com/ubuntu-ports
  ssh_svcname: ssh

# configure where output will go
output:
  init: "> /var/log/my-cloud-init.log"
  config: [ ">> /tmp/foo.out", "> /tmp/foo.err" ]
  final:
    output: "| tee /tmp/final.stdout | tee /tmp/bar.stdout"
    error: "&1"

# Set `true` to enable the stop searching for a datasource on boot.
manual_cache_clean: False

# def_log_file and syslog_fix_perms work together
# if
# - logging is set to go to a log file 'L' both with and without syslog
# - and 'L' does not exist
# - and syslog is configured to write to 'L'
# then 'L' will be initially created with root:root ownership (during
# cloud-init), and then at cloud-config time (when syslog is available)
# the syslog daemon will be unable to write to the file.
#
# to remedy this situation, 'def_log_file' can be set to a filename
# and syslog_fix_perms to a string containing "<user>:<group>"
def_log_file: /var/log/my-logging-file.log
syslog_fix_perms: syslog:root
```

2.4.11 Datasource dsname

Each datasource has an attribute called dsname. This may be used in the kernel command line to *override datasource detection*. The `dsname` on the kernel command line may be a case-insensitive match. See the mapping between datasource module names and `dsname` in the table below.

Datasource Module	dsname
DataSourceRbxCloud.py	RbxCloud
DataSourceConfigDrive.py	ConfigDrive
DataSourceNoCloud.py	NoCloud
DataSourceVultr.py	Vultr
DataSourceEc2.py	Ec2
DataSourceOracle.py	Oracle
DataSourceMAAS.py	MAAS
DataSourceDigitalOcean.py	DigitalOcean
DataSourceNone.py	None
DataSourceSmartOS.py	Joyent
DataSourceHetzner.py	Hetzner
DataSourceLXD.py	LXD
DataSourceOpenNebula.py	OpenNebula
DataSourceAzure.py	Azure
DataSourceGCE.py	GCE
DataSourceScaleway.py	Scaleway
DataSourceAltCloud.py	AltCloud
DataSourceCloudSigma.py	CloudSigma
DataSourceBigstep.py	Bigstep
DataSourceIBMCloud.py	IBMCloud
DataSourceOVF.py	OVF
DataSourceUpCloud.py	UpCloud
DataSourceOpenStack.py	OpenStack
DataSourceVMware.py	VMware
DataSourceCloudStack.py	CloudStack
DataSourceExoscale.py	Exoscale
DataSourceAliYun.py	AliYun
DataSourceNWCS.py	NWCS
DataSourceAkamai.py	Akamai

2.4.12 Performance analysis

Occasionally, instances don't perform as well as expected, and so we provide a simple tool to inspect which operations took the longest during boot and setup.

cloud-init analyze

The *cloud-init* command has an analysis sub-command, **analyze**, which parses any `cloud-init.log` file into formatted and sorted events. This analysis reveals the most costly cloud-init operations and which configuration options are responsible. These subcommands default to reading `/var/log/cloud-init.log`.

analyze show

Parse and organise `cloud-init.log` events by stage and include each sub-stage granularity with time delta reports.

```
$ cloud-init analyze show -i my-cloud-init.log
```

Example output:

```
-- Boot Record 01 --
The total time elapsed since completing an event is printed after the "@"
character.
The time the event takes is printed after the "+" character.

Starting stage: modules-config
|`->config-snap_config ran successfully @05.47700s +00.00100s
|`->config-ssh-import-id ran successfully @05.47800s +00.00200s
|`->config-locale ran successfully @05.48000s +00.00100s
...
```

analyze dump

Parse `cloud-init.log` into event records and return a list of dictionaries that can be consumed for other reporting needs.

```
$ cloud-init analyze dump -i my-cloud-init.log
```

Example output:

```
[
 {
  "description": "running config modules",
  "event_type": "start",
  "name": "modules-config",
  "origin": "cloudinit",
  "timestamp": 1510807493.0
 },...
```

analyze blame

Parse `cloud-init.log` into event records and sort them based on the highest time cost for a quick assessment of areas of cloud-init that may need improvement.

```
$ cloud-init analyze blame -i my-cloud-init.log
```

Example output:

```
-- Boot Record 11 --
     00.01300s (modules-final/config-scripts-per-boot)
     00.00400s (modules-final/config-final-message)
     ...
```

analyze boot

Make subprocess calls to the kernel in order to get relevant pre-cloud-init timestamps, such as the kernel start, kernel finish boot, and cloud-init start.

```
$ cloud-init analyze boot
```

Example output:

```
-- Most Recent Boot Record --
    Kernel Started at: 2019-06-13 15:59:55.809385
    Kernel ended boot at: 2019-06-13 16:00:00.944740
    Kernel time to boot (seconds): 5.135355
    Cloud-init start: 2019-06-13 16:00:05.738396
    Time between Kernel boot and Cloud-init start (seconds): 4.793656
```

2.4.13 Stable Release Updates (SRU)

Once upstream cloud-init has released a new version, the Ubuntu Server team backports cloud-init to previous releases via a special procedure called a "Stable Release Update" (SRU). This helps ensure that new versions of cloud-init on existing releases of Ubuntu will not experience breaking changes. Breaking changes are allowed when transitioning from one Ubuntu series to the next (Focal -> Jammy).

SRU package version

Ubuntu cloud-init packages follow the SRU release version format.

SRU testing for cloud-init

The cloud-init project has a specific process it follows when validating a cloud-init SRU, which is documented in the CloudinitUpdates wiki page.

An SRU test of cloud-init performs the following:

> For each Ubuntu SRU, the Ubuntu Server team validates the new version of cloud-init on these platforms: **Amazon EC2, Azure, GCE, OpenStack, Oracle, Softlayer (IBM), LXD using the integration test suite.**

Test process:

The *integration test suite* used for validation follows these steps:

- *Install a pre-release version of cloud-init* from the **-proposed** APT pocket (e.g., **jammy-proposed**).
- Upgrade cloud-init and attempt a clean run of cloud-init to assert that the new version works properly on the specific platform and Ubuntu series.
- Check for tracebacks and errors in behaviour.

2.4.14 Breaking changes

This section provides guidance on specific breaking changes to cloud-init releases.

Note: These changes may not be present in all distributions of cloud-init as many operating system vendors patch out breaking changes in cloud-init to ensure consistent behavior on their platform.

24.1

Removal of `--file` top-level option

The `--file` top-level option has been removed from cloud-init. It only applied to a handful of subcommands so it did not make sense as a top-level option. Instead, `--file` may be passed to a subcommand that supports it. For example, the following command will no longer work:

```
cloud-init --file=userdata.yaml modules --mode config
```

Instead, use:

```
cloud-init modules --file=userdata.yaml --mode config
```

Removed Ubuntu's ordering dependency on snapd.seeded

In Ubuntu releases, cloud-init will no longer wait on `snapd` pre-seeding to run. If a user-provided script relies on a snap, it must now be prefixed with `snap wait system seed.loaded` to ensure the snaps are ready for use. For example, a cloud config that previously included:

```
runcmd:
  - [ snap, install, mc-installer ]
```

Will now need to be:

```
runcmd:
  - [ snap, wait, system, seed.loaded ]
  - [ snap, install, mc-installer ]
```

23.2-24.1 - Datasource identification

23.2
If the detected `datasource_list` contains a single datasource or that datasource plus `None`, automatically use that datasource without checking to see if it is available. This allows for using datasources that don't have a way to be deterministically detected.

23.4
If the detected `datasource_list` contains a single datasource plus `None`, no longer automatically use that datasource because `None` is a valid datasource that may be used if the primary datasource is not available.

24.1
ds-identify no longer automatically appends `None` to a datasource list with a single entry provided under `/etc/cloud`. If `None` is desired as a fallback, it must be explicitly added to the customized datasource list.

23.4 - added status code for recoverable error

Cloud-init return codes have been extended with a new error code (2), which will be returned when cloud-init experiences an error that it can recover from. See *this page which documents the change*.

23.2 - kernel command line

The `ds=` kernel command line value is used to forcibly select a specific datasource in cloud-init. Prior to 23.2, this only optionally selected the `NoCloud` datasource.

Anyone that previously had a matching `ds=nocloud*` in their kernel command line that did not want to use the `NoCloud` datasource may experience broken behavior as a result of this change.

Workarounds include updating the kernel command line and optionally configuring a `datasource_list` in `/etc/cloud/cloud.cfg.d/*.cfg`.

2.4.15 Log and configuration files

Cloud-init uses the filesystem to read inputs and write outputs. These files are configuration and log files, respectively. If other methods of *debugging cloud-init* fail, then digging into log files is your next step in debugging.

Cloud-init log files

Cloud-init's early boot logic runs before system loggers are available or filesystems are mounted. Runtime logs and early boot logs have different locations.

Runtime logs

While booting, `cloud-init` logs to two different files:

- `/var/log/cloud-init-output.log`: Captures the output from each stage of `cloud-init` when it runs.
- `/var/log/cloud-init.log`: Very detailed log with debugging output, describing each action taken.

Be aware that each time a system boots, new logs are appended to the files in `/var/log`. Therefore, the files may contain information from more than one boot.

When reviewing these logs, look for errors or Python tracebacks.

Early boot logs

Prior to initialization, `cloud-init` runs early detection and enablement / disablement logic.

- `/run/cloud-init/cloud-init-generator.log`: On systemd systems, this log file describes early boot enablement of cloud-init via the systemd generator. These logs are most useful if trying to figure out why cloud-init did not run.
- `/run/cloud-init/ds-identify.log`: Contains logs about platform / datasource detection. These logs are most useful if cloud-init did not identify the correct datasource (cloud) to run on.

Configuration files

`Cloud-init` configuration files are provided in two places:

- `/etc/cloud/cloud.cfg`
- `/etc/cloud/cloud.cfg.d/*.cfg`

These files can define the modules that run during instance initialisation, the datasources to evaluate on boot, as well as other settings.

See the *configuration sources explanation* and *configuration reference* pages for more details.

2.4.16 Custom Modules

This includes reference documentation on how to extend cloud-init with custom / out-of-tree functionality.

Custom Formats

One can define custom data formats by presenting a *#part-handler* config via user-data or vendor-data.

Custom Clean Scripts

Cloud-init provides the directory `/etc/cloud/clean.d/` for third party applications which need additional configuration artifact cleanup from the filesystem when the *cloud-init clean* command is invoked.

The **`clean`** operation is typically performed by image creators when preparing a golden image for clone and redeployment. The clean command removes any cloud-init internal state, allowing cloud-init to treat the next boot of this image as the "first boot". Any executable scripts in this subdirectory will be invoked in lexicographical order when running the **`clean`** command.

Example

```
$ cat /etc/cloud/clean.d/99-live-installer
#!/bin/sh
sudo rm -rf /var/lib/installer_imgs/
sudo rm -rf /var/log/installer/
```

Custom Configuration Module

Custom 3rd-party out-of-tree configuration modules can be added to cloud-init by:

1. *Implement a config module* in a Python file with its name starting with `cc_`.
2. Place the file where the rest of config modules are located. On Ubuntu this path is typically: */usr/lib/python3/dist-packages/cloudinit/config/*.
3. Extend the *base-configuration's* `cloud_init_modules`, `cloud_config_modules` or `cloud_final_modules` to let the config module run on one of those stages.

Warning: The config jsonschema validation functionality is going to complain about unknown config keys introduced by custom modules and there is not an easy way for custom modules to define their keys schema-wise.

Custom DataSource

Custom 3rd-party out-of-tree DataSources can be added to cloud-init by:

1. *Implement a DataSource* in a Python file.
2. Place that file in as a single Python module or package in folder included in `$PYTHONPATH`.
3. Extend the base configuration's *datasource_pkg_list* to include the Python package where the DataSource is located.
4. Extend the *base-configuration*'s *datasource_list* to include the name of the custom DataSource.

Custom Mergers

It is possible for users to inject their own *merging* files to handle specific types of merging as they choose (the basic ones included will handle lists, dicts, and strings).

A *merge class* is a class definition providing functions that can be used to merge a given type with another given type.

An example of one of these *merging classes* is the following:

```
class Merger:
    def __init__(self, merger, opts):
        self._merger = merger
        self._overwrite = 'overwrite' in opts

    # This merging algorithm will attempt to merge with
    # another dictionary, on encountering any other type of object
    # it will not merge with said object, but will instead return
    # the original value
    #
    # On encountering a dictionary, it will create a new dictionary
    # composed of the original and the one to merge with, if 'overwrite'
    # is enabled then keys that exist in the original will be overwritten
    # by keys in the one to merge with (and associated values). Otherwise
    # if not in overwrite mode the 2 conflicting keys themselves will
    # be merged.
    def _on_dict(self, value, merge_with):
```

(continues on next page)

(continued from previous page)

```
        if not isinstance(merge_with, (dict)):
            return value
        merged = dict(value)
        for (k, v) in merge_with.items():
            if k in merged:
                if not self._overwrite:
                    merged[k] = self._merger.merge(merged[k], v)
                else:
                    merged[k] = v
            else:
                merged[k] = v
        return merged
```

There is an `_on_dict` method here that will be given a source value, and a value to merge with. The result will be the merged object.

This code itself is called by another merging class which "directs" the merging to happen by analysing the object types to merge, and attempting to find a known object that will merge that type. An example of this can be found in the `mergers/__init__.py` file (see `LookupMerger` and `UnknownMerger`).

Note how each merge can have options associated with it, which affect how the merging is performed. For example, a dictionary merger can be told to overwrite instead of attempting to merge, or a string merger can be told to append strings instead of discarding other strings to merge with.

2.5 How to contribute to cloud-init

Thank you for wanting to help us improve cloud-init! There are a variety of ways you can contribute to this project, including no-code and low-code options. This documentation will help orient you with our processes.

Please make sure that you read this guide before starting to contribute. It contains all the details you need to know to give your contribution the best chance of being accepted.

Cloud-init is hosted and managed on GitHub. If you're not familiar with how GitHub works, their quickstart documentation provides an excellent introduction to all the tools and processes you'll need to know.

2.5.1 Prerequisites

Before you can begin, you will need to:

- Read and agree to abide by our Code of Conduct.
- Sign the Canonical contributor license agreement. This grants us your permission to use your contributions in the project.
- Create (or have) a GitHub account. We will refer to your GitHub username as `GH_USER`.

2.5.2 Getting help

We use IRC and have a dedicated *#cloud-init* channel where you can contact us for help and guidance. This link will take you directly to our IRC channel on Libera.

2.5.3 Getting started

Find issues to work on

We track bugs and issues with GitHub issues, and use labels to categorise them. You can filter the list of open issues by different labels according to what you're interested in contributing.

For new contributors, we especially recommend using the "good first issue" label, and during Hacktoberfest we also add the "hacktoberfest" label to smaller, self-contained issues suitable for participants.

If you would like to work on documentation specifically, you can also use the "documentation" label to filter the list of issues.

Claiming an issue

You can express your interest in an issue by posting a comment on it. You should only work on one open issue at a time to avoid overloading yourself.

If you have not submitted a PR or commented with an updated status within a week of leaving your initial comment, other contributors should consider the issue to be available for them to work on.

When you submit your proposed fix for an issue, including the issue number in the PR commit message will link the issue to your proposed fix. This is the `Fixes GH-0000` line in the template PR commit message.

Creating an issue

If you've spotted something that doesn't already have an issue, you can always create one in GitHub.

For documentation issues, you can submit an issue in GitHub using the "Give feedback" button at the top of each documentation page. It will automatically include the URL of the page you came from, so all you need to do is describe the issue you've found.

No-code or low-code options

Contributing to our documentation is a great way to get involved with no or minimal coding experience.

If you can't find a documentation issue you want to work on, you can always check out the documentation for yourself and see what improvements you think can be made. This might be related to:

- spelling and grammar
- the user interface/experience (UI/UX)
- accessibility
- the CSS theming
- or even just highlighting things you found confusing or unclear

Feel free to contact us on IRC if you have other ideas about contributions you might want to make, such as blog posts, guides, or tutorials.

Submit your first pull request

Follow these steps prior to submitting your first pull request to cloud-init:

Setup Git and GitHub appropriately

Understanding how to use Git and GitHub is a prerequisite for contributing to cloud-init. Please refer to the GitHub quickstart documentation for more information.

Sign the CLA

To contribute to cloud-init, you must first sign the Canonical contributor license agreement (CLA).

If you have already signed it as an individual, your Launchpad username will be listed in the contributor-agreement-canonical group. Unfortunately there is no easy way to check if the organisation or company you are working for has signed it.

When you sign:

- ensure that you fill in the GitHub username field,
- when prompted for a 'Project contact' or 'Canonical Project Manager', enter 'James Falcon'.

If your company has signed the CLA for you, please contact us to help in verifying which Launchpad/GitHub accounts are associated with the company.

For any questions or help with the process, email James Falcon with the subject: "Cloud-init CLA". You can also contact user `falcojr` in the #cloud-init channel on the Libera IRC network.

Add your name to the CLA signers list

As part of your first PR to cloud-init, you should also add your GitHub username (alphabetically) to the in-repository list that we use to track CLA signatures: tools/.github-cla-signers.

PR #344 and PR #345 are good examples of what this should look like in your pull request, though please do not use a separate PR for this step.

Create a sandbox environment

It is very often helpful to create a safe and sandboxed environment to test your changes in while you work. If you are not sure how to do this, check out *our QEMU tutorial*, which walks through this process step-by-step.

Format the code

Apply the `black` and `isort` formatting rules with tox:

```
tox -e do_format
```

Run unit tests

Run unit tests and lint/formatting checks with tox:

```
tox
```

Read our code review process

Once you have submitted your PR (if not earlier!) you will want to read the cloud-init *Code Review Process*, so you can understand how your changes will end up in cloud-init's codebase.

Code review process

Code is reviewed for acceptance by at least one core team member (later referred to as committers), but comments and suggestions from others are encouraged and welcome.

Goals

This process aims to:

- provide timely and actionable feedback on every submission,
- make sure incoming PRs are handled efficiently, and
- get PRs accepted within a reasonable time frame.

Asking for help

Cloud-init contributors, community members and users are encouraged to ask for help if they need it. If you have questions about the code review process, or need advice on an open PR, these are the available avenues:

- Open a PR, add "WIP:" to the title, and leave a comment on that PR
- join the `#cloud-init` channel on the Libera IRC network
- post on the `#cloud-init` Discourse topic
- send an email to the cloud-init mailing list:

  ```
  cloud-init@lists.launchpad.net
  ```

These are listed in order of our preference, but please use whichever of them you are most comfortable with.

Role definitions

There are three roles involved in code reviews:

- **Proposer**

  The person(s) submitting the PR

- **Reviewer**

  A person who is reviewing the PR

- **Committer**

  A cloud-init core developer (i.e., someone with permission to merge PRs into `main`)

PR acceptance conditions

Before a PR can be accepted and merged into `main` ("landed"), the following conditions **must** be met:

- The CLA must be signed **by the proposer** (unless the proposer is covered by an entity-level CLA signature),
- All required status checks must be passing,
- At least one "Approve" review must be given **by a committer**, and
- No "Request changes" reviews from a committer can be outstanding.

The following conditions **should** be met:

- Any Python functions/methods/classes have docstrings added/updated,
- Any changes to config module behaviour are captured in that module's documentation,
- Any Python code added has corresponding *unit tests*, and
- No "Request changes" reviews from any **reviewer** are outstanding.

These conditions can be relaxed at the discretion of the committers on a case-by-case basis. For accountability, this should not be the decision of a single committer, and the decision should be documented in comments on the PR.

To take a specific example, the `cc_phone_home` module had no tests at the time PR #237 was submitted, so the **proposer** was not expected to write a full set of tests for their minor modification, but they *were* expected to update the config module docs.

Non-committer reviews

Reviews from non-committer reviewers are *always* welcome. Please feel empowered to review PRs and leave your thoughts and comments on any submitted PR, regardless of the proposer.

Much of the below process is written in terms of the **committers**. This does not mean that reviews should only come from that group, but rather acknowledges that we are ultimately responsible for maintaining the standards of the codebase. It is reasonable (and very welcome) for a reviewer to only examine part of a PR, but a committer must not merge a PR without full scrutiny.

Opening phase

Proposer opens a PR

In this phase, the proposer opens a pull request and needs to ensure they meet the criteria laid out above. If they need help understanding or meeting these criteria, then they can (and should!) ask for help.

CI runs automatically

- If CI fails:

  The **proposer** is expected to fix CI failures. If they don't understand the failures, they should comment on the PR to ask for help (or use another way of *Asking for help*). If they don't ask for help, the committers will assume the proposer is working on addressing the failures.

- If CI passes:

  Move on to the **review phase**.

Review phase

In this phase, the **proposer** and the **reviewers** will work iteratively together to get the PR merged into the cloud-init codebase.

There are three potential outcomes: **merged**, **rejected permanently**, and **temporarily closed**. The first two are covered in this section; see the *inactive pull requests* section for details about temporary closure.

A committer is assigned

The committers assign a committer to the PR. This committer is expected to shepherd the PR to completion (and to merge it, if that is the outcome reached).

They perform an initial review, and monitor the PR to ensure the proposer is receiving help if they need it. The committers perform this assignment on a regular basis for any new PRs submitted.

Committer's initial review

The assigned committer performs an initial review of the PR, resulting in one of the following.

Approve

If the submitted PR meets all of the *PR acceptance conditions* and passes code review, then the committer will squash merge immediately.

Sometimes, a PR should not be merged immediately. The *wip* label will be applied to PRs for which this is true. Only committers are able to apply labels to PRs, so anyone who thinks this label should be applied to a PR should request it in a comment on the PR.

- The review process is **DONE**.

Approve (with nits)

A "nit" is understood to be something like a minor style issue or a spelling error, generally confined to a single line of code.

If the proposer submits their PR with *"Allow edits from maintainer"* enabled, and the only changes the committer requests are minor nits, the committer can push fixes for those nits and immediately squash merge.

If the committer does not wish to fix these nits but believes they should block a straightforward *Approve*, then their review should be *Needs Changes* instead.

If a committer is unsure whether their requested change is a nit, they should not treat it as a nit.

If a proposer wants to opt-out of this, they should uncheck *"Allow edits from maintainer"* when submitting their PR.

- The review process is **DONE**.

Outright rejection

The committer will close the PR with a message for the proposer to explain why.

This is reserved for cases where the proposed change is unfit for landing and there is no reasonable path forward. This should only be used sparingly, as there are very few cases where proposals are *completely* unfit.

If a different approach to the same problem is planned, it should be submitted as a separate PR. The committer should include this information in their message when the PR is closed.

- The review process is **DONE**.

Needs Changes

The committer will give the proposer clear feedback on what is needed for an *Approve* vote or, for more complex PRs, what the next steps towards an *Approve* vote are.

The proposer can ask questions if they don't understand, or disagree with, the committer's review comments.

Once agreement has been reached, the proposer will address the review comments.

Once the review comments are addressed, CI will run. If CI fails, the proposer is expected to fix any CI failures. If CI passes, the proposer should indicate that the PR is ready for re-review (by @ mentioning the assigned reviewer), effectively moving back to the start of the *Review phase*.

Inactive pull requests

PRs will be temporarily closed if they have been waiting on proposer action for a certain amount of time without activity. A PR will be marked as **stale** (with an explanatory comment) after 14 days of inactivity.

It will be closed after a further 7 days of inactivity.

These closes are not considered permanent, and the closing message should reflect this for the proposer. However, if a PR is re-opened, it should effectively re-enter the *Opening phase*, as it may need some work done to get CI passing again.

2.5.4 Contribute

Pull request checklist

Before any pull request can be accepted, remember to do the following:

- Make sure your GitHub username is added (alphabetically) to the in-repository list that we use to track CLA signatures: tools/.github-cla-signers.
- Add or update any *unit tests* accordingly.
- Add or update any *Integration testing* (if applicable).
- Format code (using `black` and `isort`) with `tox -e do_format`.
- Ensure unit tests and/or linting checks pass using `tox`.
- Submit a PR against the `main` branch of the cloud-init repository.

Debugging and reporting

Logging

Cloud-init supports both local and remote logging configurable through multiple configurations:

- Python's built-in logging configuration
- Cloud-init's event reporting system
- The `cloud-init` `rsyslog` module

Python logging

Cloud-init uses the Python logging module, and can accept config for this module using the standard Python `fileConfig` format. Cloud-init looks for config for the logging module under the `logcfg` key.

Note: The logging configuration is not YAML, it is Python `fileConfig` format, and is passed through directly to the Python logging module. Please use the correct syntax for a multi-line string in YAML.

By default, cloud-init uses the logging configuration provided in `/etc/cloud/cloud.cfg.d/05_logging.cfg`. The default Python logging configuration writes all cloud-init events with a priority of `WARNING` or higher to console, and writes all events with a level of `DEBUG` or higher to `/var/log/cloud-init.log` and via `syslog`.

Python's `fileConfig` format consists of sections with headings in the format `[title]` and key value pairs in each section. Configuration for Python logging must contain the sections `[loggers]`, `[handlers]`, and `[formatters]`, which name the entities of their respective types that will be defined. The section name for each defined logger, handler and formatter will start with its type, followed by an underscore (_) and the name of the entity. For example, if a logger was specified with the name `log01`, config for the logger would be in the section `[logger_log01]`.

Logger config entries contain basic logging setup. They may specify a list of handlers to send logging events to as well as the lowest priority level of events to handle. A logger named `root` must be specified and its configuration (under `[logger_root]`) must contain a level and a list of handlers. A level entry can be any of the following: `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`, or `NOTSET`. For the `root` logger the `NOTSET` option will allow all logging events to be recorded.

Each configured handler must specify a class under Python's `logging` package namespace. A handler may specify a message formatter to use, a priority level, and arguments for the handler class. Common handlers are `StreamHandler`,

which handles stream redirects (i.e., logging to `stderr`), and `FileHandler` which outputs to a log file. The logging module also supports logging over net sockets, over http, via smtp, and additional complex configurations. For full details about the handlers available for Python logging, see the python logging handlers documentation.

Log messages are formatted using the `logging.Formatter` class, which is configured using `formatter` config entities. A default format of `%(message)s` is given if no formatter configs are specified. Formatter config entities accept a format string that supports variable replacements. These may also accept a `datefmt` string which may be used to configure the timestamp used in the log messages. The format variables `%(asctime)s`, `%(levelname)s` and `%(message)s` are commonly used and represent the timestamp, the priority level of the event and the event message. For additional information on logging formatters see python logging formatters.

Note: By default, the format string used in the logging formatter are in Python's old style `%s` form. The `str.format()` and `string.Template` styles can also be used by using `{` or `$` in place of `%` by setting the `style` parameter in formatter config.

A simple (but functional) Python logging configuration for cloud-init is below. It will log all messages of priority `DEBUG` or higher to both `stderr` and `/tmp/my.log` using a `StreamHandler` and a `FileHandler`, using the default format string `%(message)s`:

```
logcfg: |
 [loggers]
 keys=root,cloudinit
 [handlers]
 keys=ch,cf
 [formatters]
 keys=
 [logger_root]
 level=DEBUG
 handlers=
 [logger_cloudinit]
 level=DEBUG
 qualname=cloudinit
 handlers=ch,cf
 [handler_ch]
 class=StreamHandler
 level=DEBUG
 args=(sys.stderr,)
 [handler_cf]
 class=FileHandler
 level=DEBUG
 args=('/tmp/my.log',)
```

For additional information about configuring Python's logging module, please see the documentation for python logging config.

Command output

Cloud-init can redirect its `stdout` and `stderr` based on config given under the `output` config key. The output of any commands run by cloud-init and any user or vendor scripts provided will also be included here. The `output` key accepts a dictionary for configuration. Output files may be specified individually for each stage (`init`, `config`, and `final`), or a single key `all` may be used to specify output for all stages.

The output for each stage may be specified as a dictionary of `output` and `error` keys, for `stdout` and `stderr` respectively, as a tuple with `stdout` first and `stderr` second, or as a single string to use for both. The strings passed to all of these keys are handled by the system shell, so any form of redirection that can be used in bash is valid, including piping cloud-init's output to `tee`, or `logger`. If only a filename is provided, cloud-init will append its output to the file as though >> was specified.

By default, cloud-init loads its output configuration from `/etc/cloud/cloud.cfg.d/05_logging.cfg`. The default config directs both `stdout` and `stderr` from all cloud-init stages to `/var/log/cloud-init-output.log`. The default config is given as:

```
output: { all: "| tee -a /var/log/cloud-init-output.log" }
```

For a more complex example, the following configuration would output the init stage to `/var/log/cloud-init.out` and `/var/log/cloud-init.err`, for `stdout` and `stderr` respectively, replacing anything that was previously there. For the config stage, it would pipe both `stdout` and `stderr` through **`tee -a /var/log/cloud-config.log`**. For the final stage it would append the output of `stdout` and `stderr` to `/var/log/cloud-final.out` and `/var/log/cloud-final.err` respectively.

```
output:
    init:
        output: "> /var/log/cloud-init.out"
        error: "> /var/log/cloud-init.err"
    config: "tee -a /var/log/cloud-config.log"
    final:
        - ">> /var/log/cloud-final.out"
        - "/var/log/cloud-final.err"
```

Event reporting

Cloud-init contains an eventing system that allows events to be emitted to a variety of destinations.

Three configurations are available for reporting events:

- `webhook`: POST to a web server.
- `log`: Write to the cloud-init log at configurable log level.
- `stdout`: Print to `stdout`.

The default configuration is to emit events to the cloud-init log file at `DEBUG` level.

Event reporting can be configured using the `reporting` key in cloud-config user data.

Configuration

webhook

```
reporting:
  <user-defined name>:
    type: webhook
    endpoint: <url>
    timeout: <timeout in seconds>
    retries: <number of retries>
    consumer_key: <OAuth consumer key>
    token_key: <OAuth token key>
    token_secret: <OAuth token secret>
    consumer_secret: <OAuth consumer secret>
```

`endpoint` is the only additional required key when specifying `type: webhook`.

log

```
reporting:
  <user-defined name>:
    type: log
    level: <DEBUG|INFO|WARN|ERROR|FATAL>
```

`level` is optional and defaults to "DEBUG".

print

```
reporting:
  <user-defined name>:
    type: print
```

Example

The follow example shows configuration for all three sources:

```
#cloud-config
reporting:
  webserver:
    type: webhook
    endpoint: "http://10.0.0.1:55555/asdf"
    timeout: 5
    retries: 3
    consumer_key: <consumer_key>
    token_key: <token_key>
    token_secret: <token_secret>
    consumer_secret: <consumer_secret>
  info_log:
```

(continues on next page)

(continued from previous page)

```
    type: log
    level: WARN
  stdout:
    type: print
```

`rsyslog` module

Cloud-init's `cc_rsyslog` module allows for fully customizable `rsyslog` configuration under the `rsyslog` config key. The simplest way to use the `rsyslog` module is by specifying remote servers under the `remotes` key in `rsyslog` config. The `remotes` key takes a dictionary where each key represents the name of an `rsyslog` server and each value is the configuration for that server. The format for server config is:

- optional filter for log messages (defaults to `*.*`)
- optional leading `@` or `@@`, indicating UDP and TCP respectively (defaults to `@`, for UDP)
- IPv4 or IPv6 hostname or address. IPv6 addresses must be in `[::1]` format (e.g., `@[fd00::1]:514`)
- optional port number (defaults to `514`)

For example, to send logging to an `rsyslog` server named `log_serv` with address `10.0.4.1`, using port number `514`, over UDP, with all log messages enabled one could use either of the following.

With all options specified:

```
rsyslog:
    remotes:
        log_serv: "*.* @10.0.4.1:514"
```

With defaults used:

```
rsyslog:
    remotes:
        log_serv: "10.0.4.1"
```

For more information on `rsyslog` configuration, see *our module reference page*.

Internal Files: data

Cloud-init uses the filesystem to store its own internal state. These files are not intended for user consumption, but may prove helpful to debug unexpected cloud-init failures.

Data files

Inside the `/var/lib/cloud/` directory there are two important subdirectories:

instance

The /var/lib/cloud/instance directory is a symbolic link that points to the most recently used instance-id directory. This folder contains the information cloud-init received from datasources, including vendor and user data. This can help to determine that the correct data was passed.

It also contains the datasource file that contains the full information about which datasource was identified and used to set up the system.

Finally, the boot-finished file is the last thing that cloud-init creates.

data

The /var/lib/cloud/data directory contains information related to the previous boot:

- instance-id: ID of the instance as discovered by cloud-init. Changing this file has no effect.
- result.json: JSON file showing both the datasource used to set up the instance, and whether any errors occurred.
- status.json: JSON file showing the datasource used, a breakdown of all four stages, whether any errors occurred, and the start and stop times of the stages.

2.6 Contribute to the code

For a run-through of the entire process, the following pages will be your best starting point:

- *Find issues to work on*
- *Build your first pull request*
- *Our code review process*

On the rest of this page you'll find the key resources you'll need to start contributing to the cloud-init codebase.

2.6.1 Testing

Submissions to cloud-init must include testing. Unit testing and integration testing are integral parts of contributing code.

Testing

Cloud-init has both unit tests and integration tests. Unit tests can be found at tests/unittests. Integration tests can be found at tests/integration_tests. Documentation specifically for integration tests can be found on the *Integration testing* page, but the guidelines specified below apply to both types of tests.

Cloud-init uses pytest to run its tests, and has tests written both as unittest.TestCase sub-classes and as un-subclassed pytest tests.

Guidelines

The following guidelines should be followed.

Test layout

- For ease of organisation and greater accessibility for developers unfamiliar with `pytest`, all `cloud-init` unit tests must be contained within test classes. In other words, module-level test functions should not be used.
- Since all tests are contained within classes, it is acceptable to mix `TestCase` test classes and `pytest` test classes within the same test file.
  - These can be easily distinguished by their definition: `pytest` classes will not use inheritance at all (e.g., TestGetPackageMirrorInfo), whereas `TestCase` classes will subclass (indirectly) from `TestCase` (e.g., TestPrependBaseCommands).
- Unit tests and integration tests are located under `cloud-init/tests`.
  - For consistency, unit test files should have a matching name and directory location under `tests/unittests`.
  - E.g., the expected test file for code in `cloudinit/path/to/file.py` is `tests/unittests/path/to/test_file.py`.

`pytest` tests

- `pytest` test classes should use pytest fixtures to share functionality instead of inheritance.
- `pytest` tests should use bare `assert` statements, to take advantage of `pytest`'s assertion introspection.

`pytest` version "gotchas"

As we still support Ubuntu 18.04 (Bionic Beaver), we can only use `pytest` features that are available in v3.3.2. This is an inexhaustive list of ways in which this may catch you out:

- Only the following built-in fixtures are available[1]:
  - `cache`
  - `capfd`
  - `capfdbinary`
  - `caplog`
  - `capsys`
  - `capsysbinary`
  - `doctest_namespace`
  - `monkeypatch`
  - `pytestconfig`

[1] This list of fixtures (with markup) can be reproduced by running:

```
python3 -m pytest  --fixtures -q | grep "^[^ -]" | grep -v 'no tests ran in' | sort | sed 's/ \[session scope\
↪]//g;s/.*/* ``\0``/g'
```

in an ubuntu lxd container with python3-pytest installed.

- `record_xml_property`
- `recwarn`
- `tmpdir_factory`
- `tmpdir`

Mocking and assertions

- Variables/parameter names for `Mock` or `MagicMock` instances should start with `m_` to clearly distinguish them from non-mock variables. For example, `m_readurl` (which would be a mock for `readurl`).
- The `assert_*` methods that are available on `Mock` and `MagicMock` objects should be avoided, as typos in these method names may not raise `AttributeError` (and so can cause tests to silently pass).
  - **An important exception:** if a `Mock` is autospecced then misspelled assertion methods *will* raise an `AttributeError`, so these assertion methods may be used on autospecced `Mock` objects.
- For a non-autospecced `Mock`, these substitutions can be used (`m` is assumed to be a `Mock`):
  - `m.assert_any_call(*args, **kwargs)` => `assert mock.call(*args, **kwargs) in m.call_args_list`
  - `m.assert_called()` => `assert 0 != m.call_count`
  - `m.assert_called_once()` => `assert 1 == m.call_count`
  - `m.assert_called_once_with(*args, **kwargs)` => `assert [mock.call(*args, **kwargs)] == m.call_args_list`
  - `m.assert_called_with(*args, **kwargs)` => `assert mock.call(*args, **kwargs) == m.call_args_list[-1]`
  - `m.assert_has_calls(call_list, any_order=True)` => `for call in call_list: assert call in m.call_args_list`
    - `m.assert_has_calls(...)` and `m.assert_has_calls(..., any_order=False)` are not easily replicated in a single statement, so their use when appropriate is acceptable.
  - `m.assert_not_called()` => `assert 0 == m.call_count`
- When there are multiple patch calls in a test file for the module it is testing, it may be desirable to capture the shared string prefix for these patch calls in a module-level variable. If used, such variables should be named `M_PATH` or, for datasource tests, `DS_PATH`.

Test argument ordering

- Test arguments should be ordered as follows:
  - `mock.patch` arguments. When used as a decorator, `mock.patch` partially applies its generated `Mock` object as the first argument, so these arguments must go first.
  - `pytest.mark.parametrize` arguments, in the order specified to the `parametrize` decorator. These arguments are also provided by a decorator, so it's natural that they sit next to the `mock.patch` arguments.
  - Fixture arguments, alphabetically. These are not provided by a decorator, so they are last, and their order has no defined meaning, so we default to alphabetical.
- It follows from this ordering of test arguments (so that we retain the property that arguments left-to-right correspond to decorators bottom-to-top) that test decorators should be ordered as follows:

- `pytest.mark.parametrize`
- `mock.patch`

Integration testing

Overview

Integration tests are written using `pytest` and are located at `tests/integration_tests`. General design principles laid out in *Testing* should be followed for integration tests.

Setup is accomplished via a set of fixtures located in `tests/integration_tests/conftest.py`.

Test definition

Tests are defined like any other `pytest` test. The `user_data` mark can be used to supply the cloud-config user data. Platform-specific marks can be used to limit tests to particular platforms. The `client` fixture can be used to interact with the launched test instance.

See *Examples* section for examples.

Test execution

Test execution happens via `pytest`. A `tox` definition exists to run integration tests. To run all integration tests, you would run:

```
$ tox -e integration-tests
```

`pytest` arguments may also be passed. For example:

```
$ tox -e integration-tests tests/integration_tests/modules/test_combined.py
```

Configuration

All possible configuration values are defined in tests/integration_tests/integration_settings.py. Defaults can be overridden by supplying values in `tests/integration_tests/user_settings.py` or by providing an environment variable of the same name prepended with `CLOUD_INIT_`. For example, to set the `PLATFORM` setting:

```
CLOUD_INIT_PLATFORM='ec2' pytest tests/integration_tests/
```

Cloud interaction

Cloud interaction happens via the pycloudlib library. In order to run integration tests, pycloudlib must first be configured.

For a minimal setup using LXD, write the following to `~/.config/pycloudlib.toml`:

```
[lxd]
```

Image selection

Each integration testing run uses a single image as its basis. This image is configured using the `OS_IMAGE` variable; see *Configuration* for details of how configuration works.

`OS_IMAGE` can take two types of value: an Ubuntu series name (e.g. "focal"), or an image specification. If an Ubuntu series name is given, then the most recent image for that series on the target cloud will be used. For other use cases, an image specification is used.

In its simplest form, an image specification can simply be a cloud's image ID (e.g., "ami-deadbeef", "ubuntu:focal"). In this case, the identified image will be used as the basis for this testing run.

This has a drawback, however. As we do not know what OS or release is within the image, the integration testing framework will run *all* tests against the image in question. If it's a RHEL8 image, then we would expect Ubuntu-specific tests to fail (and vice versa).

To address this, a full image specification can be given. This is of the form: `<image_id>[::<os>[::<release>]]` where `image_id` is a cloud's image ID, `os` is the OS name, and `release` is the OS release name. So, for example, Ubuntu 18.04 (Bionic Beaver) on LXD is `ubuntu:bionic::ubuntu::bionic` or RHEL8 on Amazon is `ami-justanexample::rhel::8`. When a full specification is given, only tests which are intended for use on that OS and release will be executed.

Image setup

Image setup occurs once when a test session begins and is implemented via fixture. Image setup roughly follows these steps:

- Launch an instance on the specified test platform.
- Install the version of `cloud-init` under test.
- Run **`cloud-init clean`** on the instance so subsequent boots resemble "out of the box" behaviour.
- Take a snapshot of the instance to be used as a new image from which new instances can be launched.

Test setup

Test setup occurs between image setup and test execution. Test setup is implemented via one of the `client` fixtures. When a `client` fixture is used, a test instance from which to run tests is launched prior to test execution, and then torn down after.

Continuous integration

A subset of the integration tests are run when a pull request is submitted on GitHub. The tests run on these continuous integration (CI) runs are given a `pytest` mark:

```
@pytest.mark.ci
```

Most new tests should *not* use this mark, so be aware that having a successful CI run does not necessarily mean that your test passed successfully.

Fixtures

Integration tests rely heavily on fixtures to do initial test setup. One or more of these fixtures will be used in almost every integration test.

Details such as the cloud platform or initial image to use are determined via what is specified in the *Configuration*.

client

The `client` fixture should be used for most test cases. It ensures:

- All setup performed by *session_cloud* and *setup_image*.
- Pytest marks used during instance creation are obtained and applied.
- The test instance is launched.
- Test failure status is determined after test execution.
- Logs are collected (if configured) after test execution.
- The test instance is torn down after test execution.

`module_client` and `class_client` fixtures also exist for the purpose of running multiple tests against a single launched instance. They provide the exact same functionality as `client`, but are scoped to the module or class respectively.

session_cloud

The `session_cloud` session-scoped fixture will provide an IntegrationCloud instance for the currently configured cloud. The fixture also ensures that any custom cloud session cleanup is performed.

setup_image

The `setup_image` session-scope fixture will create a new image to launch all further cloud instances during this test run. It ensures:

- A cloud instance is launched on the configured platform.
- The version of `cloud-init` under test is installed on the instance.
- **`cloud-init clean --logs`** is run on the instance.
- A snapshot of the instance is taken to be used as the basis for future instance launches.
- The originally launched instance is torn down.
- The custom created image is torn down after all tests finish.

Examples

A simple test case using the `client` fixture:

```
USER_DATA = """\
#cloud-config
bootcmd:
- echo 'hello!' > /var/tmp/hello.txt
"""


@pytest.mark.user_data(USER_DATA)
def test_bootcmd(client):
    log = client.read_from_file("/var/log/cloud-init.log")
    assert "Shellified 1 commands." in log
    assert client.execute('cat /var/tmp/hello.txt').strip() == "hello!"
```

Customizing the launch arguments before launching an instance manually:

```
def test_launch(session_cloud: IntegrationCloud, setup_image):
    with session_cloud.launch(launch_kwargs={"wait": False}) as client:
        client.instance.wait()
        assert client.execute("echo hello world").strip() == "hello world"
```

- *Unit testing overview and design principles*
- *Integration testing*

2.6.2 Popular contributions

Module creation

Much of `cloud-init`'s functionality is provided by *modules*. All modules follow a similar layout in order to provide consistent execution and documentation. Use the example provided here to create a new module.

Your Python module

Modules are located in the `cloudinit/config/` directory, where the naming convention for modules is to use `cc_<module_name>` (with underscores as the separators).

The handle function

Your module must include a `handle` function. The arguments are:

- `name`: The module name specified in the configuration.
- `cfg`: A configuration object that is the result of the merging of cloud-config configuration with any datasource-provided configuration.
- `cloud`: A cloud object that can be used to access various datasource and paths for the given distro and data provided by the various datasource instance types.

- `args`: An argument list. This is usually empty and is only populated if the module is called independently from the command line or if the module definition in `/etc/cloud/cloud.cfg[.d]` has been modified to pass arguments to this module.

Schema definition

If your module introduces any new cloud-config keys, you must provide a schema definition in cloud-init-schema.json.

- The `meta` variable must exist and be of type MetaSchema.
  - `id`: The module ID. In most cases this will be the filename without the `.py` extension.
  - `distros`: Defines the list of supported distros. It can contain any of the values (not keys) defined in the OSFAMILIES map or `[ALL_DISTROS]` if there is no distro restriction.
  - `frequency`: Defines how often module runs. It must be one of:
    - `PER_ALWAYS`: Runs on every boot.
    - `ONCE`: Runs only on first boot.
    - `PER_INSTANCE`: Runs once per instance. When exactly this happens is dependent on the datasource, but may be triggered any time there would be a significant change to the instance metadata. An example could be an instance being moved to a different subnet.
  - `activate_by_schema_keys`: Optional list of cloud-config keys that will activate this module. When this list not empty, the config module will be skipped unless one of the `activate_by_schema_keys` are present in merged cloud-config instance-data.

Example module.py file

```
# This file is part of cloud-init. See LICENSE file for license information.
"""Example Module: Shows how to create a module"""

import logging
from cloudinit.cloud import Cloud
from cloudinit.config import Config
from cloudinit.config.schema import MetaSchema
from cloudinit.distros import ALL_DISTROS
from cloudinit.settings import PER_INSTANCE

LOG = logging.getLogger(__name__)

meta: MetaSchema = {
    "id": "cc_example",
    "distros": [ALL_DISTROS],
    "frequency": PER_INSTANCE,
    "activate_by_schema_keys": ["example_key, example_other_key"],
} # type: ignore

def handle(
    name: str, cfg: Config, cloud: Cloud, args: list
) -> None:
    LOG.debug(f"Hi from module {name}")
```

Module documentation

Every module has a folder in the `doc/module-docs/` directory, containing a `data.yaml` file, and one or more `example*.yaml` files.

- The `data.yaml` file contains most of the documentation fields. At a minimum, your module should be provided with this file. Examples are not strictly required, but are helpful to readers of the documentation so it is preferred for at least one example to be included.
- The `example*.yaml` files are illustrative demonstrations of using the module, but should be self-contained and in correctly-formatted YAML. These will be automatically tested against the defined schema.

Example data.yaml file

```
cc_module_name:
  description: >
    This module provides some functionality, which you can describe here.

    For straightforward text examples, use a greater-than (``>``) symbol
    next to ``description: `` to ensure proper rendering in the
    documentation. Empty lines will be respected, but line-breaks are
    folded together to create proper paragraphs.

    If you need to use call-outs or code blocks, use a pipe (``|``) symbol
    instead of ``>`` so that reStructuredText formatting (e.g. for
    directives, which take varying levels of indentation) is respected.
  examples:
  - comment: |
      Example 1: (optional) description of the expected behavior of the example
    file: cc_module_name/example1.yaml
  - comment: |
      Example 2: (optional) description of a second example.
    file: cc_module_name/example2.yaml
  name: Module Name
  title: Very brief (1 sentence) tag line describing what your module does
```

Rendering the module docs

The module documentation is auto-generated via the `doc/rtd/reference/modules.rst` file.

For your module documentation to be shown in the cloud-init docs, you will need to add an entry to this page. Modules are listed in alphabetical order. The entry should be in the following reStructuredText format:

```
.. datatemplate:yaml:: ../../module-docs/cc_ansible/data.yaml
   :template: modules.tmpl
```

The template pulls information from both your `module.py` file, and from its corresponding entry in the the `module-docs` directory.

Module execution

For a module to be run, it must be defined in a module run section in `/etc/cloud/cloud.cfg` or `/etc/cloud/cloud.cfg.d` on the launched instance. The three module sections are cloud_init_modules, cloud_config_modules, and cloud_final_modules, corresponding to the *Network*, *Config*, and *Final* boot stages respectively.

Add your module to cloud.cfg.tmpl under the appropriate module section. Each module gets run in the order listed, so ensure your module is defined in the correct location based on dependencies. If your module has no particular dependencies or is not necessary for a later boot stage, it should be placed in the `cloud_final_modules` section before the `final-message` module.

Benefits of including your config module in upstream cloud-init

Config modules included in upstream cloud-init benefit from ongoing maintenance, compatibility with the rest of the codebase, and security fixes by the upstream development team.

If this is not possible, one can add *custom out-of-tree config modules* to cloud-init.

Supporting your cloud or platform

The upstream cloud-init project regularly accepts code contributions for new platforms that wish to support cloud-init.

Ways to add platform support

To add cloud-init support for a new platform, there are two possible approaches:

1. Provide platform compatibility with one of the existing datasource definitions, such as nocloud via DatasourceNoCloud.py. Several platforms, including Libvirt and Proxmox use this approach.
2. Add a new datasource definition to upstream cloud-init. This provides tighter integration, a better debugging experience, and more control and flexibility of cloud-init's interaction with the datasource. This option is more sensible for clouds that have a unique architecture.

Platform requirements

There are some technical and logistical prerequisites that must be met for cloud-init support.

Technical requirements

A cloud needs to be able to identify itself to cloud-init at runtime, and that the cloud be able to provide configuration to the instance.

A mechanism for self-identification

Each cloud platform must positively identify itself to the guest. This allows the guest to make educated decisions based on the platform on which it is running. On the x86 and arm64 architectures, many clouds identify themselves through DMI data. For example, Oracle's public cloud provides the string `'OracleCloud.com'` in the DMI chassis-asset field.

`Cloud-init`-enabled images produce a log file with details about the platform. Reading through this log in `/run/cloud-init/ds-identify.log` may provide the information needed to uniquely identify the platform. If the log is not present, one can generate the log by running `ds-identify` manually.

The mechanism used to identify the platform will be required for `ds-identify` and the datasource module sections below.

A mechanism for cloud-init to retrieve configuration

There are different methods that cloud-init can use to retrieve cloud-configuration for configuring the instance. The most common method is a webserver providing configuration over a link-local network.

Logistical requirements

As with any open source project, multiple logistal requirements exist.

Testing access

A platform that isn't available for testing cannot be independently validated. You will need to provide some means for community members and upstream developers to test and verify this platform. Code that cannot be used cannot be supported.

Maintainer support

A point of contact is required who can answer questions and occasionally provide testing or maintenance support. Maintainership is relatively informal, but there is an expectation that from time to time upstream may need to contact a the maintainer with inquiries. Datasources that appear to be unmaintained and/or unused may be considered for eventual removal.

Adding cloud-init support

There are multiple ways to provide *user data*, *metadata*, and *vendor data*, and each cloud solution prefers its own way. A datasource abstract base class defines a single interface to interact with the different clouds. Each cloud implementation must inherit from this base class to use this shared functionality and interface. See `cloud-init/sources/__init__.py` to see this class.

If you are interested in adding a new datasource for your cloud platform you will need to do all of the following:

Add datasource module cloudinit/sources/DataSource<CloudPlatform>.py

We suggest you start by copying one of the simpler datasources such as `DataSourceHetzner`.

Re-run datasource detection

While developing a new datasource it may be helpful to manually run datasource detection without rebooting the system.

To re-run datasource detection, you must first force `ds-identify` to re-run, then clean up any logs, and finally, re-run `cloud-init`:

```
sudo DI_LOG=stderr /usr/lib/cloud-init/ds-identify --force
sudo cloud-init clean --logs
sudo cloud-init init --local
sudo cloud-init init
```

Add tests for datasource module

Add a new file with some tests for the module to `cloudinit/sources/test_<yourplatform>.py`. For example, see `cloudinit/sources/tests/test_oracle.py`

Update `ds-identify`

In `systemd` systems, `ds-identify` is used to detect which datasource should be enabled, or if `cloud-init` should run at all. You'll need to make changes to `tools/ds-identify`.

Add tests for `ds-identify`

Add relevant tests in a new class to `tests/unittests/test_ds_identify.py`. You can use `TestOracle` as an example.

Add your datasource name to the built-in list of datasources

Add your datasource module name to the end of the `datasource_list` entry in `cloudinit/settings.py`.

Add your cloud platform to apport collection prompts

Update the list of cloud platforms in `cloudinit/apport.py`. This list will be provided to the user who invokes **`ubuntu-bug cloud-init`**.

Enable datasource by default in Ubuntu packaging branches

Ubuntu packaging branches contain a template file, `config/cloud.cfg.tmpl`, which ultimately sets the default `datasource_list` that is installed by distros that use the upstream packaging configuration.

Add documentation for your datasource

You should add a new file in `doc/rtd/reference/datasources/<cloudplatform>.rst` and reference it in `doc/rtd/reference/datasources.rst`

Benefits of including your datasource in upstream cloud-init

Datasources included in upstream cloud-init benefit from ongoing maintenance, compatibility with the rest of the codebase, and security fixes by the upstream development team.

If this is not possible, one can add *custom out-of-tree datasources* to cloud-init.

The two most popular contributions we receive are new cloud config *modules* and new *datasources*; these pages will provide instructions on how to create them.

Note that any new modules should use underscores in any new config options and not hyphens (e.g. `new_option` and *not* `new-option`).

2.6.3 Code style and design

We generally adhere to PEP 8, and this is enforced by our use of `black`, `isort` and `ruff`.

Python support

Cloud-init upstream currently supports Python 3.6 and above.

Cloud-init upstream will stay compatible with a particular Python version for 6 years after release. After 6 years, we will stop testing upstream changes against the unsupported version of Python and may introduce breaking changes. This policy may change as needed.

The following table lists the cloud-init versions in which the minimum Python version changed:

Cloud-init version	Python version
22.1	3.6+
20.3	3.5+
19.4	2.7+

Type annotations

The cloud-init codebase uses Python's annotation support for storing type annotations in the style specified by PEP-484 and PEP-526. Their use in the codebase is encouraged.

2.6.4 Other resources

Directory layout

`/var/lib/cloud`

The main directory containing the cloud-init-specific subdirectories. It is typically located at `/var/lib` but there are certain configuration scenarios where this can be changed.

`.../data/`

This directory contains information about instance IDs, datasources and hostnames of the previous and current instance if they are different. These can be examined as needed to determine any information related to a previous boot (if applicable).

`.../handlers/`

Custom `part-handlers` code is written out here. Files that end up here are written out within the scheme of `part-handler-XYZ` where `XYZ` is the handler number (the first handler found starts at `0`).

`.../instance`

A symlink to the current `instances/` subdirectory, which points to the currently active instance. Note that the active instance depends on the loaded datasource.

`.../instances/`

All instances that were created using this image end up with instance identifier subdirectories (with corresponding data for each instance). The currently active instance will be symlinked to the `instance` symlink file defined previously.

`.../scripts/`

Scripts in one of these subdirectories are downloaded/created by the corresponding `part-handler`.

.../seed/

Contains seeded data files: `meta-data`, `network-config`, `user-data`, `vendor-data`.

.../sem/

Cloud-init has a concept of a module semaphore, which consists of the module name and its frequency. These files are used to ensure a module is only run "per-once", "per-instance", or "per-always". This folder contains semaphore `files` which are only supposed to run "per-once" (not tied to the instance ID).

- *Explanation of the directory structure*

Feature flags

Feature flags are used as a way to easily toggle configuration **at build time**. They are provided to accommodate feature deprecation and downstream configuration changes.

Currently used upstream values for feature flags are set in `cloudinit/features.py`. Overrides to these values should be patched directly (e.g., via quilt patch) by downstreams.

Each flag should include a short comment regarding the reason for the flag and intended lifetime.

Tests are required for new feature flags, and tests must verify all valid states of a flag, not just the default state.

`cloudinit.features.`**`ALLOW_EC2_MIRRORS_ON_NON_AWS_INSTANCE_TYPES = False`**

When configuring apt mirrors, if `ALLOW_EC2_MIRRORS_ON_NON_AWS_INSTANCE_TYPES` is `True` cloud-init will detect that a datasource's `availability_zone` property looks like an EC2 availability zone and set the `ec2_region` variable when generating mirror URLs; this can lead to incorrect mirrors being configured in clouds whose AZs follow EC2's naming pattern.

As of 20.3, `ALLOW_EC2_MIRRORS_ON_NON_AWS_INSTANCE_TYPES` is `False` so we no longer include `ec2_region` in mirror determination on non-AWS cloud platforms.

If the old behavior is desired, users can provide the appropriate mirrors via `apt:` directives in cloud-config.

`cloudinit.features.`**`APT_DEB822_SOURCE_LIST_FILE = True`**

On Debian and Ubuntu systems, cc_apt_configure will write a deb822 compatible /etc/apt/sources.list.d/(debian|ubuntu).sources file. When set False, continue to write /etc/apt/sources.list directly.

`cloudinit.features.`**`DEPRECATION_INFO_BOUNDARY = 'devel'`**

DEPRECATION_INFO_BOUNDARY is used by distros to configure at which upstream version to start logging deprecations at a level higher than INFO.

The default value "devel" tells cloud-init to log all deprecations higher than INFO. This value may be overriden by downstreams in order to maintain stable behavior across releases.

Jsonschema key deprecations and inline logger deprecations include a deprecated_version key. When the variable below is set to a version, cloud-init will use that version as a demarcation point. Deprecations which are added after this version will be logged as at an INFO level. Deprecations which predate this version will be logged at the higher DEPRECATED level. Downstreams that want stable log behavior may set the variable below to the first version released in their stable distro. By doing this, they can expect that newly added deprecations will be logged at INFO level. The implication of the different log levels is that logs at DEPRECATED level result in a return code of 2 from *cloud-init status*.

format:

<value> :: = <default> | <version> <default> ::= "devel" <version> ::= <major> "." <minor> ["." <patch>]

where <major>, <minor>, and <patch> are positive integers

cloudinit.features.**ERROR_ON_USER_DATA_FAILURE = True**

If there is a failure in obtaining user data (i.e., #include or decompress fails) and ERROR_ON_USER_DATA_FAILURE is False, cloud-init will log a warning and proceed. If it is True, cloud-init will instead raise an exception.

As of 20.3, ERROR_ON_USER_DATA_FAILURE is True.

(This flag can be removed after Focal is no longer supported.)

cloudinit.features.**EXPIRE_APPLIES_TO_HASHED_USERS = True**

If EXPIRE_APPLIES_TO_HASHED_USERS is True, then when expire is set true in cc_set_passwords, hashed passwords will be expired. Previous to 22.3, only non-hashed passwords were expired.

(This flag can be removed after Jammy is no longer supported.)

cloudinit.features.**NETPLAN_CONFIG_ROOT_READ_ONLY = True**

If NETPLAN_CONFIG_ROOT_READ_ONLY is True, then netplan configuration will be written as a single root read-only file /etc/netplan/50-cloud-init.yaml. This prevents wifi passwords in network v2 configuration from being world-readable. Prior to 23.1, netplan configuration is world-readable.

(This flag can be removed after Jammy is no longer supported.)

cloudinit.features.**NOCLOUD_SEED_URL_APPEND_FORWARD_SLASH = True**

Append a forward slash '/' if NoCloud seedurl does not end with either a querystring or forward slash. Prior to 23.1, nocloud seedurl would be used unaltered, appending meta-data, user-data and vendor-data to without URL path separators.

(This flag can be removed when Jammy is no longer supported.)

cloudinit.features.**get_features**() → Dict[str, bool]

Return a dict of applicable features/overrides and their values.

2.7 Contribute to our docs

2.7.1 Documentation style guide

Language

Where possible, text should be written in UK English. However, discretion and common sense can both be applied. For example, where text refers to code elements that exist in US English, the spelling of these elements should not be changed to UK English.

Try to be concise and to the point in your writing. It is acceptable to link to official documentation elsewhere rather than repeating content. It's also good practice not to assume that your reader has the same level of knowledge as you, so if you're covering a new or complicated topic, then providing contextual links to help the reader is encouraged.

Feel free to include a "Further reading" section at the end of a page if you have additional resources an interested reader might find helpful.

Headings

In reStructuredText, headings are denoted using symbols to underline the text. The headings used across the documentation use the following hierarchy, which is borrowed from the Python style guide:

- `#####`: Top level header (reserved for the main index page)
- `*****`: Title header (used once at the top of a new page)
- `=====`: Section headers
- `-----`: Subsection headers
- `^^^^^`: Sub-subsection headers
- `"""""`: Paragraphs

The length of the underline must be at least as long as the title itself.

Ensure that you do not skip header levels when creating your document structure, i.e., that a section is followed by a subsection, and not a sub-subsection.

Line length

Please keep the line lengths to a maximum of **79** characters. This ensures that the pages and tables do not get so wide that side scrolling is required.

Blank spaces at the ends of lines must also be removed, otherwise the *tox* build checks will fail (it will warn you about trailing whitespace).

Anchor labels

Adding an anchor label at the top of the page allows for the page to be referenced by other pages. For example for the FAQ page this would be:

```
.. _faq:

FAQ
***
```

When the reference is used in a document, the displayed text will be that of the next heading immediately following the label (so, FAQ in this example), unless specifically overridden.

If you use labels within a page to refer, for example, to a subsection, use a label that follows the format: `[pagelabel]-[Section]` e.g., for this "Anchor labels" section, something like `_docs-Anchor:` or `_docs-Label:`. Using a consistent style will aid greatly when referencing from other pages.

Links

To aid in documentation maintenance and keeping links up-to-date, links should be presented in a single block at the end of the page.

Where possible, use contextual text in your links to aid users with screen readers and other accessibility tools. For example, "check out our *documentation style guide*" is preferable to "click *here* for more".

Images

It is generally best to avoid screenshots where possible. If you need to refer to text output, you can use code blocks. For diagrams, we recommend the use of Mermaid.

Code blocks

Our documentation uses the Sphinx extension "sphinx-copybutton", which creates a small button on the right-hand side of code blocks for users to copy the code snippets we provide.

The copied code will strip out the prompt symbol ($) so that users can paste commands directly into their terminal. For user convenience, please ensure that code output is presented in a separate code block to the commands.

Vertical whitespace

One newline between each section helps ensure readability of the documentation source code.

Common words

There are some common words that should follow specific usage in text:

- **cloud-init**: Always hyphenated, and follows sentence case, so only capitalised at the start of a sentence.
- **metadata**, **datasource**: One word.
- **user data**, **vendor data**: Two words, not to be combined or hyphenated.

When referring to file names, which may be hyphenated, they should be decorated with backticks to ensure monospace font is used to distinguish them from regular text.

Acronyms

Acronyms are always capitalised (e.g., JSON, YAML, QEMU, LXD) in text.

The first time an acronym is used on a page, it is best practice to introduce it by showing the expanded name followed by the acronym in parentheses. E.g., Quick EMUlator (QEMU). If the acronym is very common, or you provide a link to a documentation page that provides such details, you will not need to do this.

2.7.2 Documentation directory layout

Cloud-init's documentation directory structure, with respect to the root directory:

```
/doc/
    - examples/
    - man/
    - module-docs/
    - rtd/
        - tutorial/
        - howto/
        - explanation/
        - reference/

        - development/
```

(continues on next page)

(continued from previous page)

```
      - static/
          - css/
          - js/
          - *logos*
      - *conf.py*
      - *index.rst*
      - *links.txt*
  - rtd_html/
  - sources/
```

examples/

man/

This subdirectory contains the Linux man pages for the binaries provided by cloud-init.

module-docs/

The documentation for modules is generated automatically using YAML files and templates. Each module has its own sub-directory, containing:

- `data.yaml` file: Contains the text and descriptions rendered on the *modules documentation* page.
- `example*.yaml` files: These examples stand alone as valid cloud-config. They always start with `#cloud-config`, and ideally, should also have some accompanying discussion or context in the `comment` field in the `data.yaml` file to explain what's happening.

Edit existing module docs

In the `data.yaml` file, the fields support reStructuredText markup in the `description` and `comment` fields. With the pipe character (`|`) preceding these fields, the text will be preserved so that using rST directives (such as notes or code blocks) will render correctly in the documentation. If you don't need to use directives, you can use the greater-than character (`>`), which will fold broken lines together into paragraphs (while respecting empty lines).

Create new module docs

Creating documentation for a **new** module involves a little more work, and the process for that is outlined in the *contributing modules* page.

rtd/

This subdirectory is of most interest to anyone who wants to create or update either the content of the documentation, or the styling of it.

- The content of the documentation is organised according to the Diataxis framework and can be found in the subdirectories: `tutorial/`, `howto/`, `explanation/`, and `reference/`.
- The `development/` subdirectory contains documentation for contributors.
- `static/` contains content that relates to the styling of the documentation in the form of custom CSS or javascript files found in `css/` and `js/` respectively. This is also where you can find the cloud-init logo.

- `conf.py` contains Sphinx configuration commands.
- `index.rst` is the front page of the documentation.
- `links.txt` contains common (and reusable) links so that you do not need to define the same URLs on every page and can use a more convenient shorthand when referencing often-used links.

`rtd_html/`

When the documentation is built locally using `tox -e doc`, the built pages can be found in this folder.

`sources/`

This subdirectory contains demos which can help the reader understand how parts of the product work.

The documentation for cloud-init is hosted in the cloud-init GitHub repository and rendered on Read the Docs. It is mostly written in reStructuredText.

The process for contributing to the docs is largely the same as for code, except that for cosmetic changes to the documentation (spelling, grammar, etc) you can also use the GitHub web interface to submit changes as quick PRs.

2.7.3 Previewing the docs

The documentation for submitted/active PRs is automatically built by Read the Docs and served from the PR's "conversation" tab as an automatic check.

However, while you are working on docs for a feature you are adding, you will most likely want to build the docs locally. There is a Makefile target to build the documentation for you:

```
$ tox -e doc
```

This will do two things:

- Build the documentation using Sphinx.
- Run doc8 against the documentation source code.

Once built, the HTML files will be viewable in *doc/rtd_html*. Use your web browser to open *index.html* to view and navigate the site.

2.7.4 How are the docs structured?

We use Diataxis to organise our documentation. There is more detail on the layout of the `doc` directory in the *Documentation directory layout* article.

We also have a *Documentation style guide* that will help you if you need to edit or write any content.

2.7.5 In your first PR

You will need to add your GitHub username (alphabetically) to the in-repository list that we use to track *CLA signatures*: tools/.github-cla-signers.

Please include this in the same PR alongside your first contribution. Do not create a separate PR to add your name to the CLA signatures.

If you need some help with your contribution, you can contact us on our IRC channel. If you have already submitted a work-in-progress PR, you can also ask for guidance from our technical author by tagging s-makin as a reviewer.

2.8 The cloud-init summit

One of the major highlights in our calendar is the cloud-init summit! The summit is an annual gathering of cloud-init contributors and community members held in Seattle, Washington.

At the summit, we enjoy meeting with our fellow contributors to cloud-init, demoing recent developments, collecting feedback, and holding workshops to discuss outstanding issues, bugs, and possible fixes.

After an unfortunate hiatus of a couple of years due to "global travel difficulties", we are pleased to announced that the next summit will be held in August 2023! More details to follow…

2.8.1 Previous summits

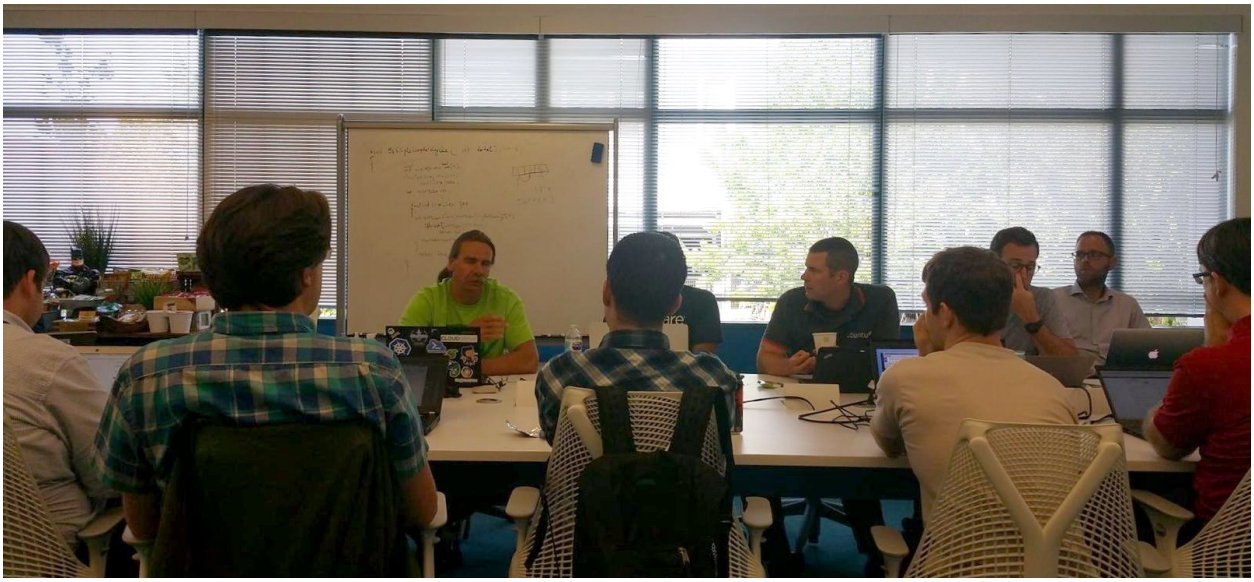
cloud-init: Summit in Seattle, Washington

Note: This article was written by Joshua Powers and originally published on 31 August 2017. It is shared here under license with no changes.

Last week the cloud-init development team from Canonical ran a two-day summit in Seattle, Washington. The purpose of the summit was to meet with contributors to cloud-init from cloud providers and OS vendors to demo recent devel-

opments in cloud-init, resolve outstanding issues, and collect feedback on development and test processes as well as future features.

Attendees included developers from Amazon, Microsoft, Google, VMWare, and IBM cloud teams, as well as the maintainers of cloud-init from Red Hat, SUSE, and of course, Ubuntu. Special thanks go to Google for hosting us and to Microsoft for buying everyone dinner!



Demos

The cloud-init development team came with a number of prepared demos and talks that they gave as a part of the summit:

- **cloud-init analyze**: Ryan demoed the recently added analyze feature to aid in doing boot time performance analysis. This tool parses the cloud-init log into formatted and sorted events to assist in determining long running steps during instance initialization.
- **cloud-config Schema Validation**: Chad demonstrated the early functionality to validate cloud-configs before launching instances. He demoed two modules that exist today, how to write the validation, and what positive and negative results look like.
- **Integration Testing and CI**: Josh demonstrated the integration test framework and shared plans on running tests on actual clouds. Then showed the merge request CI process and encouraged this as a way for other OSes to participate.
- **Using lxd for Rapid Development and Testing**: Scott demoed setting userdata when launching a lxd instance and how this can be used in the development process. He also discussed lxd image remotes and types of images.

Breakout Sessions

In addition to the prepared demos, the summit had numerous sessions that were requested by the attendees as additional topics for discussion:

- Netplan (v2 YAML) as primary format
- How to query metadata
- Version numbering
- Device hot-plug
- Python 3
- And more…

During the summit, we took time to have merge review and bug squashing time. During this time, attendees came with outstanding bugs to discuss possible fixes as well as go through outstanding merge requests and get live reviews.

Conclusions

A big thanks to the community for attending! The summit was a great time to meet many long time users and contributors face-to-face as well as collect feedback for cloud-init development.

Notes of both days can be found on the cloud-init mailing list. There you will find additional details about what I have described above and much more.

Finally, if you are interested in following or getting involved in cloud-init development check out #cloud-init on Freenode or subscribe to the cloud-init mailing list.

cloud-init: Summit 2018

Note: This article was written by Joshua Powers and originally published on 27 August 2018. It is shared here under license with no changes.

Last week the cloud-init development team from Canonical ran our second annual two-day summit. Attendees included cloud developers from Amazon, Microsoft, Google, VMWare, and Oracle, as well as the maintainer of cloud-init from Amazon Linux, SUSE, and Ubuntu.

The purpose of this two-day event is to meet with contributors, demo recent developments, present future plans, resolve outstanding issues, and collect additional feedback on the past year.

Like last year, the even was held in Seattle, Washington. A special thanks goes to Microsoft for providing breakfast and lunch while hosting us and to the Amazon Linux and AWS teams for buying everyone dinner!

Talks, Demos, and Discussions

The cloud-init development team came with a number of prepared demos and talks that they gave as a part of the summit:

- **Recent Features and Retrospective**: Ryan started the summit off with an overview of features landed in the past year as we all metrics since the start of faster releases with date-based versioning.
- **Community Checkpoint & Feedback**: Scott hosted a session where he explored the various avenues contributors have and received input and ideas for even better collaboration.
- **Roadmap**: Ryan presented the roadmap for upcoming releases and requested feedback from those in attendance.
- **Ending Python 2.6 Support**: Scott announced the end of Python 2.6 support and there was a discussion on ending Python 2.7 support as well. An announcement to the mailing list is coming soon.
- **Instance-data.json support and cloud-init cli**: Chad demoed a standard way of querying instance data keys to enable scripting, templating, and access across all clouds.
- **Multipass**: Alberto from the Canonical Multipass team joined us to demo the Multipass project. Multipass is the fastest way to get a virtual machine launched with the latest Ubuntu images.
- **Integration Testing and CI**: Josh gave an update on the new CI processes, auto-landing merge requests, and demoed the integration tests. He went through what it takes to add additional clouds and his wish-list for additional testing.

- **Pre-Network Detection for Clouds**: Chad ran a discussion on collecting pre-networking detection for clouds in order to speed up instance initialization and decrease boot time.

Breakout Sessions

In addition to the prepared demos and discussions, the summit had numerous sessions that were requested by the attendees as additional topics for discussion.

SUSE led at discussion around the sysconfig renderer and network rework, while the Amazon Linux team discussed some of their patches. Both distros are working to minimize the number of patches required.

During the summit, we took time to have merge review and bug squashing time. During this time, attendees came with outstanding bugs to discuss possible fixes as well as go through outstanding merge requests and get live reviews.

Conclusions

As always a huge thank you to the community for attending! The summit was a great time to see many contributors face-to-face as well as collect feedback for cloud-init development.

Notes of both days can be found on the cloud-init mailing list. There you will find additional details about what I have described above and much more.

Finally, if you are interested in following or getting involved in cloud-init development check out #cloud-init on Freenode or subscribe to the cloud-init mailing list.

cloud-init: Summit 2019

Note: This article was written by Joshua Powers and originally published on 21 October 2019. It is shared here under license with no changes.

Last month the cloud-init development team from Canonical ran our third annual two-day summit. Attendees included cloud developers from Amazon, Cisco, Microsoft, Google, and Oracle, as well as the maintainers of cloud-init from Amazon Linux, SUSE, Red Hat, and Ubuntu.

The purpose of this two-day event is to meet with contributors, demo recent developments, present future plans, resolve outstanding issues, and collect additional feedback on the past year.

Like last year, the even was held in Seattle, Washington. A special thanks goes to Amazon for providing breakfast and lunch while hosting us!

Topics and Decisions

Here are summary of some of the topics discussed during the sprint:

- **New Security Process**: I proposed a process by which security issues would be reported to the project, how they would be evaluated, fixed, and eventually disclosed. While this is not fully complete, the process has already been used once to evaluate what turned out to be non-security issues.
- **Boot Performance**: Ryan started the second day off talking about the boot performance analysis that he is conducted. He has proposed an initial branch with changes to help many clouds improve their time to SSH. While this work will involve effort across platforms, kernels, distros, and cloud-init, we can already start to make changes to cloud-init.
- **GitHub Transition**: We are moving the project to GitHub in an effort to continue to gather contributions and improve our merge proposal process. We have some early testing and CI branches ready to go. We are waiting on some open questions around the CLA and mirroring back to Launchpad to continue the move.
- **Python Support**: The last release of cloud-init in 2019 is the final version to support python 2.7. We will cut a branch for future bug fixes. After that master will now support Python 3.4 going forward. A future discussion around how to move the Python 3 version is needed. See the mailing list post for more details.
- **Red Hat Support**: Edwardo gave a presentation on Red Hat's process around cloud-init. He showed the versions they are on and what they do when a new release comes out.

Working Sessions

During the summit, we took time to have merge review and bug squashing time. During this time, attendees came with outstanding bugs to discuss possible fixes as well as go through outstanding merge requests and get live reviews.

Thank you

As always a huge thank you to the community for attending! The summit was a great time to see many contributors face-to-face as well as collect feedback for cloud-init development.

Notes of both days can be found on the cloud-init mailing list. There you will find additional details about what I have described above and much more.

Finally, if you are interested in following or getting involved in cloud-init development check out #cloud-init on Freenode or subscribe to the cloud-init mailing list.

cloud-init: Summit 2023

After a three-year hiatus the two-day cloud-init summit finally resumed in August, giving Canonical a chance to reconnect with the community in person, and to realign on the direction and goals of the project.

The event was generously hosted by Microsoft this year at their Redmond campus in Seattle, Washington, and we are grateful to the Microsoft community members "on the ground" who coordinated with Canonical's cloud-init development team to help organise and run the event. Big thanks go as well to the Canonical community team for helping us to set up the event site, as well as for their support and guidance with all the planning involved.

As in previous years, the summit was a great opportunity for cloud-init contributors to get together and discuss the most recent developments in the project, provide demos of new features, resolve outstanding issues, and shape the

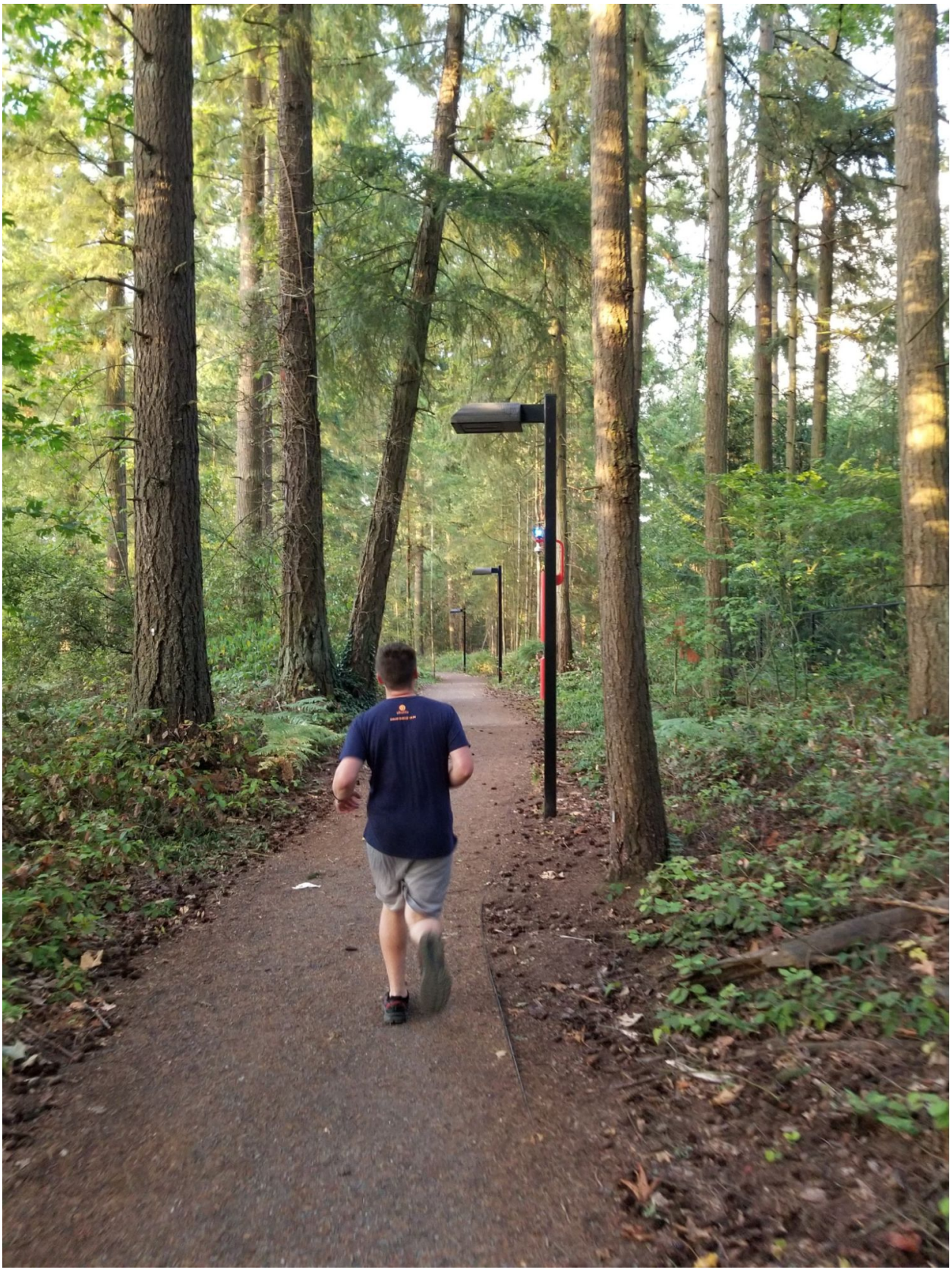


Fig. 1: Enjoying a jog through the beautiful forest around the Microsoft Redmond campus!

future development direction of the project. It was wonderful to see some "old" faces again after such a long time, as well as getting to meet some of our newer contributors in person.

The first hybrid summit

This summit was organised as a hybrid event for the first time, and despite some initial uncertainties about how to implement that, it worked very well. In-person attendees included developers and contributors from Microsoft, Google, Amazon, Oracle, openSUSE and we had remote presentations provided by FreeBSD and AlpineLinux maintainers.

In addition to our in-person gathering, we also had lively participation from our remote attendees from around the world. With this hybrid format allowing attendance from community members who might not otherwise have been able to take part, this is a format that we'll want to carry forward to next year to open the event to the widest possible audience.

Special thanks go to Canonical for sponsoring the dinner! It was a great chance to build community interactions "after hours", with topics ranging far and wide. Overall, it was a perfect opportunity to dig into industry dynamics that influence cloud-init engagement.

Highlights of the discussions

Thanks to all our presenters; Mina Galic (FreeBSD) and Dermot Bradley (AlpineLinux), Chris Patterson (Microsoft) James Falcon, Brett Holman, Catherine Redfield, Alberto Contreras, Sally Makin, John Chittum, Daniel Bungert and Chad Smith. You really helped to make this event a success.

Presentation takeaways

- **Integration-testing tour/demo**: James showed how Canonical uses our integration tests and pycloudlib during SRU verification, and demonstrated how we think other clouds should be involved in standard evaluation of cloud-init releases before publication.

  There was interest in leveraging this testing at openSUSE, Amazon Linux and possibly Microsoft's Mariner; they may be looking to extend our framework for their release testing. Our homework from this is improved developer docs on extending integration tests for other distributions.

  Azure are looking to invest in distribution-agnostic integration test frameworks and want to knowledge-share with the cloud-init community on that.

- **Security-policy for cloud-init CVE handling**: Our community would like Canonical to provide more context during the embargo period on CVEs about the mitigation steps required. This is especially the case in any downstream packaging, to allow downstream package maintainers more time to prepare.

- **Cloud platforms want/like strict schema validation** and errors on invalid user-data/config. They also requested more visibility into any warnings surfaced by cloud-init with simple tools so that they can avoid costly "log spelunking".

  This aligns well with Brett's ongoing roadmap work to raise warnings from the CLI and some of the strict JSON schema validation on network-config and user data/vendor data.

- Good lessons from both AlpineLinux (Dermot Bradley), who investigated SSH alternatives like dropbearSSH and tinySSH, and FreeBSD (Mina Galić), who reported on the development and publishing process and on finding better ways for FreeBSD and Alpine to engage with clouds, so they can get sponsorship of open source images with cloud-init hosted in AWS.

Round-table discussions

- **Boot-speed**: The discussion hosted by Catherine, Alberto and Chad confirmed that our ongoing boot speed work is critical to clouds and cloud-customers, who continue to gauge boot speed based on wall time to SSH into the instance.

  This is a more critical measurement than the time to all services being up. In our discussion, we received feedback that every millisecond counts. We also learned that there is hesitation about moving to precompiled languages such as Go, due to the potential image size impacts, or Rust, due to the somewhat limited platform support.

  Partners are also looking for *cloud-init analyze* to report on external systemd-related impacts (such as *NetworkManager-wait-online.service* or *systemd-network-wait-online.service* delays) due to external units/services that affect boot.

- **Review of our Python support matrix** for all downstreams, with the goal of Python 3.6 version deprecation. Due to ongoing downstream support needs, we are looking to retain Python 3.6 support until March 2024.

- **Shared test frameworks**: Azure intends to invest in integration testing with the cloud-init community, to develop distribution-agnostic best practices for verification of distribution releases, and boot-speed and image health analysis. If there are ways we want to collaborate on generalised testing and verification of images, they may provide some development toward this cause.

Breakout sessions

- **Private reviews of partner engagements** with Oracle and AWS, and Fabio Martins, Kyler Horner, and James to prioritise ongoing work and plan for the future development of IPv6-only datasource support - as well as other features.
- **Brett and openSUSE's Robert Schweikert** worked through downstream patch review with the intent of merging many openSUSE patches upstream. Amazon Linux has a couple of downstream patches that they may want to upstream as well.

Conclusions

This two-day event gave us a fantastic chance to take the pulse of the cloud-init project. It's given us a healthy understanding of areas in which we can better serve the community and how we can continue to build momentum.

Meeting face-to-face to reflect our cloud-init plans with the community helped confirm interest in some of the usability features we are developing, such as better error and warning visibility and improving boot speed in cloud-init. There is plenty of enthusiasm for continued collaboration on improved testing and verification that all distributions and clouds can leverage.

We also appreciated the opportunity to get valuable feedback on our documentation, our communication, and our security processes. We've also discussed and gained input into better practices we can adopt through GitHub automation, workflows that automate pull request digests, and upstream test matrix coverage for downstreams (beside Ubuntu). All of these things will help us to maintain the momentum of the cloud-init project and ensure that we are best serving the needs of our community.

Thank you!

This event could not have taken place without the hard work and preparation of all our presenters, organisers, and the voices of our community members in attendance. So, thank you again to everyone who participated, and we very much hope to see you again at the next cloud-init summit!

Notes of both days can be found on the cloud-init mailing list, and also are hosted in our GitHub repository. There you will find additional details about each topic and related discussions.

Finally, if you are interested in following or getting involved in cloud-init development check out #cloud-init on Libera.chat or subscribe to the cloud-init mailing list.

PYTHON MODULE INDEX

INDEX