kayobe Documentation

Release

OpenStack Foundation

Feb 27, 2018

Contents

CHAPTER 1

Kayobe

Deployment of Scientific OpenStack using OpenStack kolla.

Kayobe is an open source tool for automating deployment of Scientific OpenStack onto a set of bare metal servers. Kayobe is composed of Ansible playbooks, a python module, and makes heavy use of the OpenStack kolla project. Kayobe aims to complement the kolla-ansible project, providing an opinionated yet highly configurable OpenStack deployment and automation of many operational procedures.

- Documentation: https://kayobe.readthedocs.io/en/latest/
- Source: https://github.com/stackhpc/kayobe
- Bugs: https://github.com/stackhpc/kayobe/issues
- IRC: #openstack-kayobe

1.1 Features

- Heavily automated using Ansible
- *kayobe* Command Line Interface (CLI) for cloud operators
- Deployment of a *seed* VM used to manage the OpenStack control plane
- Configuration of physical network infrastructure
- Discovery, introspection and provisioning of control plane hardware using OpenStack bifrost
- Deployment of an OpenStack control plane using OpenStack kolla-ansible
- Discovery, introspection and provisioning of bare metal compute hosts using OpenStack ironic and ironic inspector
- Virtualised compute using OpenStack nova
- Containerised workloads on bare metal using OpenStack magnum
- Big data on bare metal using OpenStack sahara

In the near future we aim to add support for the following:

- Control plane and workload monitoring and log aggregation using OpenStack monasca

1.2 Documentation

Note: Kayobe and its documentation is currently under heavy development, and therefore may be incomplete or out of date. If in doubt, contact the project's maintainers.

1.2.1 Architecture

Hosts in the System

In a system deployed by Kayobe we define a number of classes of hosts.

Control host The control host is the host on which kayobe, kolla and kolla-ansible will be installed, and is typically where the cloud will be managed from.

Seed host The seed host runs the bifrost deploy container and is used to provision the cloud hosts. By default, container images are built on the seed. Typically the seed host is deployed as a VM but this is not mandatory.

Cloud hosts The cloud hosts run the OpenStack control plane, network, monitoring, storage, and virtualised compute services. Typically the cloud hosts run on bare metal but this is not mandatory.

Bare metal compute hosts In a cloud providing bare metal compute services to tenants via ironic, these hosts will run the bare metal tenant workloads. In a cloud with only virtualised compute this category of hosts does not exist.

Note: In many cases the control and seed host will be the same, although this is not mandatory.

Cloud Hosts

Cloud hosts can further be divided into subclasses.

Controllers Controller hosts run the OpenStack control plane services.

Network Network hosts run the neutron networking services and load balancers for the OpenStack API services.

Monitoring Monitoring host run the control plane and workload monitoring services. Currently, kayobe does not deploy any services onto monitoring hosts.

Virtualised compute hypervisors Virtualised compute hypervisors run the tenant Virtual Machines (VMs) and associated OpenStack services for compute, networking and storage.

Networks

Kayobe's network configuration is very flexible but does define a few default classes of networks. These are logical networks and may map to one or more physical networks in the system.

Overcloud out-of-band network Name of the network used by the seed to access the out-of-band management controllers of the bare metal overcloud hosts.

Overcloud provisioning network The overcloud provisioning network is used by the seed host to provision the cloud hosts.

Workload out-of-band network Name of the network used by the overcloud hosts to access the out-of-band management controllers of the bare metal workload hosts.

Workload provisioning network The workload provisioning network is used by the cloud hosts to provision the bare metal compute hosts.

Internal network The internal network hosts the internal and admin OpenStack API endpoints.

Public network The public network hosts the public OpenStack API endpoints.

External network The external network provides external network access for the hosts in the system.

1.2.2 Installation

Prerequisites

Currently Kayobe supports the following Operating Systems on the control host:

- CentOS 7.3
- Ubuntu 16.04

To avoid conflicts with python packages installed by the system package manager it is recommended to install Kayobe in a virtualenv. Ensure that the `virtualenv` python module is available on the control host. It is necessary to install the GCC compiler chain in order to build the extensions of some of kayobe's python dependencies. Finally, for cloning and working with the kayobe source code repository, Git is required.

On CentOS:

```
$ yum install -y python-devel python-virtualenv gcc git
```

On Ubuntu:

```
$ apt install -y python-dev python-virtualenv gcc git
```

Installation

This guide will describe how to install Kayobe from source in a virtualenv.

The directory structure for a kayobe control host environment is configurable, but the following is recommended, where `<base_path>` is the path to a top level directory:

```
<base_path>/
    src/
        kayobe/
        kayobe-config/
        kolla-ansible/
    venvs/
        kayobe/
        kolla-ansible/
```

First, change to the top level directory, and make the directories for source code repositories and python virtual environments:

```
$ cd <base_path>
$ mkdir -p src venvs
```

Next, obtain the Kayobe source code. For example:

```
$ cd <base_path>/src
$ git clone https://github.com/stackhpc/kayobe
```

Create a virtualenv for Kayobe:

```
$ virtualenv <base_path>/venvs/kayobe
```

Activate the virtualenv and update pip:

```
$ source <base_path>/venvs/kayobe/bin/activate
(kayobe) $ pip install -U pip
```

Install Kayobe and its dependencies using the source code checkout:

```
(kayobe) $ cd <base_path>/src/kayobe
(kayobe) $ pip install .
```

Finally, deactivate the virtualenv:

```
(kayobe) $ deactivate
```

Creation of a `kayobe-config` source code repository will be covered in the *configuration guide*. The kolla-ansible source code checkout and python virtual environment will be created automatically by kayobe.

1.2.3 Usage

Command Line Interface

Note: Where a prompt starts with `(kayobe)` it is implied that the user has activated the Kayobe virtualenv. This can be done as follows:

```
$ source kayobe/bin/activate
```

To deactivate the virtualenv:

```
(kayobe) $ deactivate
```

To see information on how to use the `kayobe` CLI and the commands it provides:

```
(kayobe) $ kayobe help
```

As the `kayobe` CLI is based on the `cliff` package (as used by the `openstack` client), it supports tab auto-completion of subcommands. This can be activated by generating and then sourcing the bash completion script:

```
(kayobe) $ kayobe complete > kayobe-complete
(kayobe) $ source kayobe-complete
```

Working with Ansible Vault

If Ansible vault has been used to encrypt Kayobe configuration files, it will be necessary to provide the `kayobe` command with access to vault password. There are three options for doing this:

Prompt Use `kayobe --ask-vault-pass` to prompt for the password.

File Use `kayobe --vault-password-file <file>` to read the password from a (plain text) file.

Environment variable Export the environment variable `KAYOBE_VAULT_PASSWORD` to read the password from the environment.

Limiting Hosts

Sometimes it may be necessary to limit execution of kayobe or kolla-ansible plays to a subset of the hosts. The `--limit <SUBSET>` argument allows the kayobe ansible hosts to be limited. The `--kolla-limit <SUBSET>` argument allows the kolla-ansible hosts to be limited. These two options may be combined in a single command. In both cases, the argument provided should be an Ansible host pattern, and will ultimately be passed to `ansible-playbook` as a `--limit` argument.

Tags

Ansible tags provide a useful mechanism for executing a subset of the plays or tasks in a playbook. The `--tags <TAGS>` argument allows execution of kayobe ansible playbooks to be limited to matching plays and tasks. The `--kolla-tags <TAGS>` argument allows execution of kolla-ansible ansible playbooks to be limited to matching plays and tasks. The `--skip-tags <TAGS>` and `--kolla-skip-tags <TAGS>` arguments allow for avoiding execution of matching plays and tasks.

1.2.4 Configuration Guide

Kayobe Configuration

This section covers configuration of Kayobe. As an Ansible-based project, Kayobe is for the most part configured using YAML files.

Configuration Location

Kayobe configuration is by default located in `/etc/kayobe` on the Ansible control host. This location can be overridden to a different location to avoid touching the system configuration directory by setting the environment variable `KAYOBE_CONFIG_PATH`. Similarly, kolla configuration on the Ansible control host will by default be located in `/etc/kolla` and can be overridden via `KOLLA_CONFIG_PATH`.

Configuration Directory Layout

The Kayobe configuration directory contains Ansible `extra-vars` files and the Ansible inventory. An example of the directory structure is as follows:

```
extra-vars1.yml
extra-vars2.yml
inventory/
```

```
    group_vars/
        group1-vars
        group2-vars
    groups
    host_vars/
        host1-vars
        host2-vars
    hosts
```

Configuration Patterns

Ansible's variable precedence rules are fairly well documented and provide a mechanism we can use for providing site localisation and customisation of OpenStack in combination with some reasonable default values. For global configuration options, Kayobe typically uses the following patterns:

- Playbook group variables for the *all* group in `<kayobe repo>/ansible/group_vars/all/*` set **global defaults**. These files should not be modified.
- Playbook group variables for other groups in `<kayobe repo>/ansible/group_vars/<group>/*` set **defaults for some subsets of hosts**. These files should not be modified.
- Extra-vars files in `${KAYOBE_CONFIG_PATH}/*.yml` set **custom values for global variables** and should be used to apply global site localisation and customisation. By default these variables are commented out.

Additionally, variables can be set on a per-host basis using inventory host variables files in `${KAYOBE_CONFIG_PATH}/inventory/host_vars/*`. It should be noted that variables set in extra-vars files take precedence over per-host variables.

Configuring Kayobe

The kayobe-config git repository contains a Kayobe configuration directory structure and unmodified configuration files. This repository can be used as a mechanism for version controlling Kayobe configuration. As Kayobe is updated, the configuration should be merged to incorporate any upstream changes with local modifications.

Alternatively, the baseline Kayobe configuration may be copied from a checkout of the Kayobe repository to the Kayobe configuration path:

```
$ cp -r etc/ ${KAYOBE_CONFIG_PATH:-/etc/kayobe}
```

Once in place, each of the YAML and inventory files should be manually inspected and configured as required.

Inventory

The inventory should contain the following hosts:

Control host This should be localhost.

Seed hypervisor If provisioning a seed VM, a host should exist for the hypervisor that will run the VM, and should be a member of the `seed-hypervisor` group.

Seed The seed host, whether provisioned as a VM by Kayobe or externally managed, should exist in the `seed` group.

Cloud hosts and bare metal compute hosts are not required to exist in the inventory if discovery of the control plane hardware is planned, although entries for groups may still be required.

Use of advanced control planes with multiple server roles and customised service placement across those servers is covered in *Control Plane Service Placement*.

Site Localisation and Customisation

Site localisation and customisation is applied using Ansible extra-vars files in `${KAYOBE_CONFIG_PATH}/*.yml`.

Encryption of Secrets

Kayobe supports the use of Ansible vault to encrypt sensitive information in its configuration. The `ansible-vault` tool should be used to manage individual files for which encryption is required. Any of the configuration files may be encrypted. Since encryption can make working with Kayobe difficult, it is recommended to follow best practice, adding a layer of indirection and using encryption only where necessary.

Remote Execution Environment

By default, ansible executes modules remotely using the system python interpreter, even if the ansible control process is executed from within a virtual environment (unless the `local` connection plugin is used). This is not ideal if there are python dependencies that must be installed without isolation from the system python packages. Ansible can be configured to use a virtualenv by setting the host variable `ansible_python_interpreter` to a path to a python interpreter in an existing virtual environment.

If kayobe detects that `ansible_python_interpreter` is set and references a virtual environment, it will create the virtual environment if it does not exist. Typically this variable should be set via a group variable for hosts in the `seed`, `seed-hypervisor`, and/or `overcloud` groups.

Network Configuration

Kayobe provides a flexible mechanism for configuring the networks in a system. Kayobe networks are assigned a name which is used as a prefix for variables that define the network's attributes. For example, to configure the `cidr` attribute of a network named `arpanet`, we would use a variable named `arpanet_cidr`.

Global Network Configuration

Global network configuration is stored in `${KAYOBE_CONFIG_PATH}/networks.yml`. The following attributes are supported:

`cidr` CIDR representation (<IP>/<prefix length>) of the network's IP subnet.

`allocation_pool_start` IP address of the start of Kayobe's allocation pool range.

`allocation_pool_end` IP address of the end of Kayobe's allocation pool range.

`inspection_allocation_pool_start` IP address of the start of ironic inspector's allocation pool range.

`inspection_allocation_pool_end` IP address of the end of ironic inspector's allocation pool range.

`neutron_allocation_pool_start` IP address of the start of neutron's allocation pool range.

`neutron_allocation_pool_end` IP address of the end of neutron's allocation pool range.

`gateway` IP address of the network's default gateway.

inspection_gateway IP address of the gateway for the hardware introspection network.

neutron_gateway IP address of the gateway for a neutron subnet based on this network.

vlan VLAN ID.

mtu Maximum Transmission Unit (MTU).

routes List of static IP routes. Each item should be a dict containing the item `cidr`, and optionally `gateway` and `table`. `cidr` is the CIDR representation of the route's destination. `gateway` is the IP address of the next hop. `table` is the name or ID of a routing table to which the route will be added.

rules List of IP routing rules. Each item should be an `iproute2` IP routing rule.

physical_network Name of the physical network on which this network exists. This aligns with the physical network concept in neutron.

libvirt_network_name A name to give to a Libvirt network representing this network on the seed hypervisor.

Configuring an IP Subnet

An IP subnet may be configured by setting the `cidr` attribute for a network to the CIDR representation of the subnet.

To configure a network called `example` with the `10.0.0.0/24` IP subnet:

Listing 1.1: `networks.yml`

```
example_cidr: 10.0.0.0/24
```

Configuring an IP Gateway

An IP gateway may be configured by setting the `gateway` attribute for a network to the IP address of the gateway.

To configure a network called `example` with a gateway at `10.0.0.1`:

Listing 1.2: `networks.yml`

```
example_gateway: 10.0.0.1
```

This gateway will be configured on all hosts to which the network is mapped. Note that configuring multiple IP gateways on a single host will lead to unpredictable results.

Configuring Static IP Routes

Static IP routes may be configured by setting the `routes` attribute for a network to a list of routes.

To configure a network called `example` with a single IP route to the `10.1.0.0/24` subnet via `10.0.0.1`:

Listing 1.3: `networks.yml`

```
example_routes:
  - cidr: 10.1.0.0/24
    gateway: 10.0.0.1
```

These routes will be configured on all hosts to which the network is mapped.

Configuring a VLAN

A VLAN network may be configured by setting the `vlan` attribute for a network to the ID of the VLAN.

To configure a network called `example` with VLAN ID `123`:

Listing 1.4: `networks.yml`

```
example_vlan: 123
```

IP Address Allocation

IP addresses are allocated automatically by Kayobe from the allocation pool defined by `allocation_pool_start` and `allocation_pool_end`. The allocated addresses are stored in `${KAYOBE_CONFIG_PATH}/network-allocation.yml` using the global per-network attribute `ips` which maps Ansible inventory hostnames to allocated IPs.

If static IP address allocation is required, the IP allocation file `network-allocation.yml` may be manually populated with the required addresses.

Configuring Dynamic IP Address Allocation

To configure a network called `example` with the `10.0.0.0/24` IP subnet and an allocation pool spanning from `10.0.0.4` to `10.0.0.254`:

Listing 1.5: `networks.yml`

```
example_cidr: 10.0.0.0/24
example_allocation_pool_start: 10.0.0.4
example_allocation_pool_end: 10.0.0.254
```

Note: This pool should not overlap with an inspection or neutron allocation pool on the same network.

Configuring Static IP Address Allocation

To configure a network called `example` with statically allocated IP addresses for hosts `host1` and `host2`:

Listing 1.6: `network-allocation.yml`

```
example_ips:
  host1: 10.0.0.1
  host2: 10.0.0.2
```

Advanced: Policy-Based Routing

Policy-based routing can be useful in complex networking environments, particularly where asymmetric routes exist, and strict reverse path filtering is enabled.

Configuring IP Routing Tables

Custom IP routing tables may be configured by setting the global variable `network_route_tables` in `${KAYOBE_CONFIG_PATH}/networks.yml` to a list of route tables. These route tables will be added to `/etc/iproute2/rt_tables`.

To configure a routing table called `exampleroutetable` with ID `1`:

Listing 1.7: `networks.yml`

```
network_route_tables:
  - name: exampleroutetable
    id: 1
```

To configure route tables on specific hosts, use a host or group variables file.

Configuring IP Routing Policy Rules

IP routing policy rules may be configured by setting the `rules` attribute for a network to a list of rules. The format of a rule is the string which would be appended to `ip rule <add|del>` to create or delete the rule.

To configure a network called `example` with an IP routing policy rule to handle traffic from the subnet `10.1.0.0/24` using the routing table `exampleroutetable`:

Listing 1.8: `networks.yml`

```
example_rules:
  - from 10.1.0.0/24 table exampleroutetable
```

These rules will be configured on all hosts to which the network is mapped.

Configuring IP Routes on Specific Tables

A route may be added to a specific routing table by adding the name or ID of the table to a `table` attribute of the route:

To configure a network called `example` with a default route and a 'connected' (local subnet) route to the subnet `10.1.0.0/24` on the table `exampleroutetable`:

Listing 1.9: `networks.yml`

```
example_routes:
  - cidr: 0.0.0.0/0
    gateway 10.1.0.1
    table: exampleroutetable
  - cidr: 10.1.0.0/24
    table: exampleroutetable
```

Per-host Network Configuration

Some network attributes are specific to a host's role in the system, and these are stored in `${KAYOBE_CONFIG_PATH}/inventory/group_vars/<group>/network-interfaces`. The following attributes are supported:

`interface` The name of the network interface attached to the network.

bridge_ports For bridge interfaces, a list of names of network interfaces to add to the bridge.

bond_mode For bond interfaces, the bond's mode, e.g. 802.3ad.

bond_slaves For bond interfaces, a list of names of network interfaces to act as slaves for the bond.

bond_miimon For bond interfaces, the time in milliseconds between MII link monitoring.

bond_updelay For bond interfaces, the time in milliseconds to wait before declaring an interface up (should be multiple of `bond_miimon`).

bond_downdelay For bond interfaces, the time in milliseconds to wait before declaring an interface down (should be multiple of `bond_miimon`).

bond_xmit_hash_policy For bond interfaces, the xmit_hash_policy to use for the bond.

bond_lacp_rate For bond interfaces, the lacp_rate to use for the bond.

IP Addresses

An interface will be assigned an IP address if the associated network has a `cidr` attribute. The IP address will be assigned from the range defined by the `allocation_pool_start` and `allocation_pool_end` attributes, if one has not been statically assigned in `network-allocation.yml`.

Configuring Ethernet Interfaces

An Ethernet interface may be configured by setting the `interface` attribute for a network to the name of the Ethernet interface.

To configure a network called `example` with an Ethernet interface on `eth0`:

Listing 1.10: `inventory/group_vars/<group>/network-interfaces`

```
example_interface: eth0
```

Configuring Bridge Interfaces

A Linux bridge interface may be configured by setting the `interface` attribute of a network to the name of the bridge interface, and the `bridge_ports` attribute to a list of interfaces which will be added as member ports on the bridge.

To configure a network called `example` with a bridge interface on `breth1`, and a single port `eth1`:

Listing 1.11: `inventory/group_vars/<group>/network-interfaces`

```
example_interface: breth1
example_bridge_ports:
  - eth1
```

Bridge member ports may be Ethernet interfaces, bond interfaces, or VLAN interfaces. In the case of bond interfaces, the bond must be configured separately in addition to the bridge, as a different named network. In the case of VLAN interfaces, the underlying Ethernet interface must be configured separately in addition to the bridge, as a different named network.

Configuring Bond Interfaces

A bonded interface may be configured by setting the `interface` attribute of a network to the name of the bond's master interface, and the `bond_slaves` attribute to a list of interfaces which will be added as slaves to the master.

To configure a network called `example` with a bond with master interface `bond0` and two slaves `eth0` and `eth1`:

Listing 1.12: `inventory/group_vars/<group>/network-interfaces`

```
example_interface: bond0
example_bond_slaves:
  - eth0
  - eth1
```

Optionally, the bond mode and MII monitoring interval may also be configured:

Listing 1.13: `inventory/group_vars/<group>/network-interfaces`

```
example_bond_mode: 802.3ad
example_bond_miimon: 100
```

Bond slaves may be Ethernet interfaces, or VLAN interfaces. In the case of VLAN interfaces, underlying Ethernet interface must be configured separately in addition to the bond, as a different named network.

Configuring VLAN Interfaces

A VLAN interface may be configured by setting the `interface` attribute of a network to the name of the VLAN interface. The interface name must be of the form `<parent interface>.<VLAN ID>`.

To configure a network called `example` with a VLAN interface with a parent interface of `eth2` for VLAN `123`:

Listing 1.14: `inventory/group_vars/<group>/network-interfaces`

```
example_interface: eth2.123
```

To keep the configuration DRY, reference the network's `vlan` attribute:

Listing 1.15: `inventory/group_vars/<group>/network-interfaces`

```
example_interface: "eth2.{{ example_vlan }}"
```

Ethernet interfaces, bridges, and bond master interfaces may all be parents to a VLAN interface.

Bridges and VLANs

Adding a VLAN interface to a bridge directly will allow tagged traffic for that VLAN to be forwarded by the bridge, whereas adding a VLAN interface to an Ethernet or bond interface that is a bridge member port will prevent tagged traffic for that VLAN being forwarded by the bridge.

Network Role Configuration

In order to provide flexibility in the system's network topology, Kayobe maps the named networks to logical network roles. A single named network may perform multiple roles, or even none at all. The available roles are:

Overcloud out-of-band network (`oob_oc_net_name`) Name of the network used by the seed to access the out-of-band management controllers of the bare metal overcloud hosts.

Overcloud provisioning network (`provision_oc_net_name`) Name of the network used by the seed to provision the bare metal overcloud hosts.

Workload out-of-band network (`oob_wl_net_name`) Name of the network used by the overcloud hosts to access the out-of-band management controllers of the bare metal workload hosts.

Workload provisioning network (`provision_wl_net_name`) Name of the network used by the overcloud hosts to provision the bare metal workload hosts.

Internal network (`internal_net_name`) Name of the network used to expose the internal OpenStack API endpoints.

Public network (`public_net_name`) Name of the network used to expose the public OpenStack API endpoints.

External networks (`external_net_names`, deprecated: `external_net_name`) List of names of networks used to provide external network access via Neutron. If `external_net_name` is defined, `external_net_names` defaults to a list containing only that network.

Storage network (`storage_net_name`) Name of the network used to carry storage data traffic.

Storage management network (`storage_mgmt_net_name`) Name of the network used to carry storage management traffic.

Workload inspection network (`inspection_net_name`) Name of the network used to perform hardware introspection on the bare metal workload hosts.

These roles are configured in `${KAYOBE_CONFIG_PATH}/networks.yml`.

Configuring Network Roles

To configure network roles in a system with two networks, `example1` and `example2`:

Listing 1.16: `networks.yml`

```
oob_oc_net_name: example1
provision_oc_net_name: example1
oob_wl_net_name: example1
provision_wl_net_name: example2
internal_net_name: example2
public_net_name: example2
external_net_name: example2
storage_net_name: example2
storage_mgmt_net_name: example2
inspection_net_name: example2
```

Overcloud Provisioning Network

If using a seed to inspect the bare metal overcloud hosts, it is necessary to define a DHCP allocation pool for the seed's ironic inspector DHCP server using the `inspection_allocation_pool_start` and `inspection_allocation_pool_end` attributes of the overcloud provisioning network.

Note: This example assumes that the `example` network is mapped to `provision_oc_net_name`.

To configure a network called `example` with an inspection allocation pool:

```
example_inspection_allocation_pool_start: 10.0.0.128
example_inspection_allocation_pool_end: 10.0.0.254
```

Note: This pool should not overlap with a kayobe allocation pool on the same network.

Workload Provisioning Network

If using the overcloud to provision bare metal workload (compute) hosts, it is necessary to define an IP allocation pool for the overcloud's neutron provisioning network using the `neutron_allocation_pool_start` and `neutron_allocation_pool_end` attributes of the workload provisioning network.

Note: This example assumes that the `example` network is mapped to `provision_wl_net_name`.

To configure a network called `example` with a neutron provisioning allocation pool:

```
example_neutron_allocation_pool_start: 10.0.1.128
example_neutron_allocation_pool_end: 10.0.1.195
```

Note: This pool should not overlap with a kayobe or inspection allocation pool on the same network.

Workload Inspection Network

If using the overcloud to inspect bare metal workload (compute) hosts, it is necessary to define a DHCP allocation pool for the overcloud's ironic inspector DHCP server using the `inspection_allocation_pool_start` and `inspection_allocation_pool_end` attributes of the workload provisioning network.

Note: This example assumes that the `example` network is mapped to `provision_wl_net_name`.

To configure a network called `example` with an inspection allocation pool:

```
example_inspection_allocation_pool_start: 10.0.1.196
example_inspection_allocation_pool_end: 10.0.1.254
```

Note: This pool should not overlap with a kayobe or neutron allocation pool on the same network.

Neutron Networking

Note: This assumes the use of the neutron `openvswitch` ML2 mechanism driver for control plane networking.

Certain modes of operation of neutron require layer 2 access to physical networks in the system. Hosts in the `network` group (by default, this is the same as the `controllers` group) run the neutron networking services (Open vSwitch agent, DHCP agent, L3 agent, metadata agent, etc.).

The kayobe network configuration must ensure that the neutron Open vSwitch bridges on the network hosts have access to the external network. If bare metal compute nodes are in use, then they must also have access to the workload provisioning network. This can be done by ensuring that the external and workload provisioning network interfaces are bridges. Kayobe will ensure connectivity between these Linux bridges and the neutron Open vSwitch bridges via a virtual Ethernet pair. See *Configuring Bridge Interfaces*.

Network to Host Mapping

Networks are mapped to hosts using the variable `network_interfaces`. Kayobe's playbook group variables define some sensible defaults for this variable for hosts in the top level standard groups. These defaults are set using the network roles typically required by the group.

Seed

By default, the seed is attached to the following networks:

- overcloud out-of-band network
- overcloud provisioning network

This list may be extended by setting `seed_extra_network_interfaces` to a list of names of additional networks to attach. Alternatively, the list may be completely overridden by setting `seed_network_interfaces`. These variables are found in `${KAYOBE_CONFIG_PATH}/seed.yml`.

Seed Hypervisor

By default, the seed hypervisor is attached to the same networks as the seed.

This list may be extended by setting `seed_hypervisor_extra_network_interfaces` to a list of names of additional networks to attach. Alternatively, the list may be completely overridden by setting `seed_hypervisor_network_interfaces`. These variables are found in `${KAYOBE_CONFIG_PATH}/seed-hypervisor.yml`.

Controllers

By default, controllers are attached to the following networks:

- overcloud provisioning network
- workload (compute) out-of-band network
- workload (compute) provisioning network
- internal network
- storage network
- storage management network

In addition, if the controllers are also in the `network` group, they are attached to the following networks:

- public network
- external network

This list may be extended by setting `controller_extra_network_interfaces` to a list of names of additional networks to attach. Alternatively, the list may be completely overridden by setting `controller_network_interfaces`. These variables are found in `${KAYOBE_CONFIG_PATH}/controllers.yml`.

Monitoring Hosts

By default, the monitoring hosts are attached to the same networks as the controllers when they are in the `controllers` group. If the monitoring hosts are not in the `controllers` group, they are attached to the following networks by default:

- overcloud provisioning network
- internal network
- public network

This list may be extended by setting `monitoring_extra_network_interfaces` to a list of names of additional networks to attach. Alternatively, the list may be completely overridden by setting `monitoring_network_interfaces`. These variables are found in `${KAYOBE_CONFIG_PATH}/monitoring.yml`.

Virtualised Compute Hosts

By default, virtualised compute hosts are attached to the following networks:

- overcloud provisioning network
- internal network
- storage network

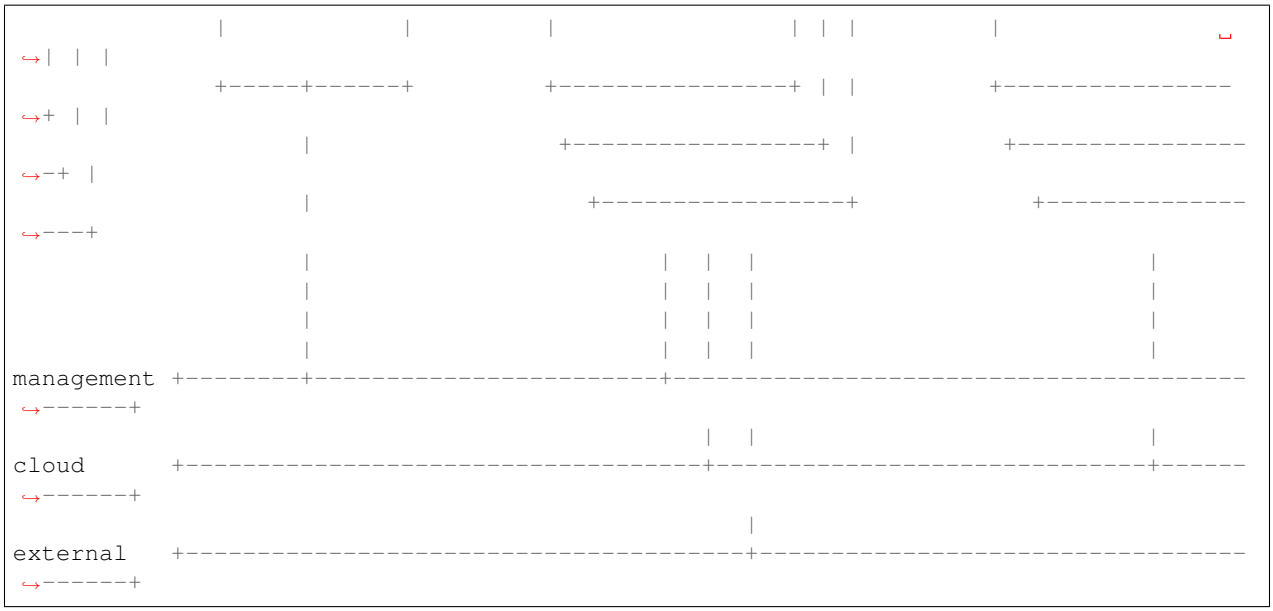
This list may be extended by setting `compute_extra_network_interfaces` to a list of names of additional networks to attach. Alternatively, the list may be completely overridden by setting `compute_network_interfaces`. These variables are found in `${KAYOBE_CONFIG_PATH}/compute.yml`.

Other Hosts

If additional hosts are managed by kayobe, the networks to which these hosts are attached may be defined in a host or group variables file. See *Control Plane Service Placement* for further details.

Complete Example

The following example combines the complete network configuration into a single system configuration. In our example cloud we have three networks: `management`, `cloud` and `external`:



The `management` network is used to access the servers' BMCs and by the seed to inspect and provision the cloud hosts. The `cloud` network carries all internal control plane and storage traffic, and is used by the control plane to provision the bare metal compute hosts. Finally, the `external` network links the cloud to the outside world.

We could describe such a network as follows:

Listing 1.17: `networks.yml`

```
---
# Network role mappings.
oob_oc_net_name: management
provision_oc_net_name: management
oob_wl_net_name: management
provision_wl_net_name: cloud
internal_net_name: cloud
public_net_name: external
external_net_name: external
storage_net_name: cloud
storage_mgmt_net_name: cloud
inspection_net_name: cloud

# management network definition.
management_cidr: 10.0.0.0/24
management_allocation_pool_start: 10.0.0.1
management_allocation_pool_end: 10.0.0.127
management_inspection_allocation_pool_start: 10.0.0.128
management_inspection_allocation_pool_end: 10.0.0.254

# cloud network definition.
cloud_cidr: 10.0.1.0/24
cloud_allocation_pool_start: 10.0.1.1
cloud_allocation_pool_end: 10.0.1.127
cloud_inspection_allocation_pool_start: 10.0.1.128
cloud_inspection_allocation_pool_end: 10.0.1.195
cloud_neutron_allocation_pool_start: 10.0.1.196
cloud_neutron_allocation_pool_end: 10.0.1.254
```

```
# external network definition.
external_cidr: 10.0.3.0/24
external_allocation_pool_start: 10.0.3.1
external_allocation_pool_end: 10.0.3.127
external_neutron_allocation_pool_start: 10.0.3.128
external_neutron_allocation_pool_end: 10.0.3.254
external_routes:
  - cidr 10.0.4.0/24
    gateway: 10.0.3.1
```

We can map these networks to network interfaces on the seed and controller hosts:

Listing 1.18: `inventory/group_vars/seed/network-interfaces`

```
---
management_interface: eth0
```

Listing 1.19: `inventory/group_vars/controllers/network-interfaces`

```
---
management_interface: eth0
cloud_interface: breth1
cloud_bridge_ports:
  - eth1
external_interface: eth2
```

We have defined a bridge for the cloud network on the controllers as this will allow it to be plugged into a neutron Open vSwitch bridge.

Kayobe will allocate IP addresses for the hosts that it manages:

Listing 1.20: `network-allocation.yml`

```
---
management_ips:
  seed: 10.0.0.1
  control0: 10.0.0.2
  control1: 10.0.0.3
  control2: 10.0.0.4
cloud_ips:
  control0: 10.0.1.1
  control1: 10.0.1.2
  control2: 10.0.1.3
external_ips:
  control0: 10.0.3.1
  control1: 10.0.3.2
  control2: 10.0.3.3
```

Note that although this file does not need to be created manually, doing so allows for a predictable IP address mapping which may be desirable in some cases.

Kolla-ansible Configuration

Kayobe relies heavily on kolla-ansible for deployment of the OpenStack control plane. Kolla-ansible is installed locally on the ansible control host (the host from which kayobe commands are executed), and kolla-ansible commands are executed from there.

Local Environment

Environment variables are used to configure the environment in which kolla-ansible is installed and executed.

Table 1.1: Kolla-ansible environment variables

Variable	Purpose	Default
`$KOLLA_CONFIG_PATH`	Path on the ansible control host in which the kolla-ansible configuration will be generated. These files should not be manually edited.	`/etc/kolla`
`$KOLLA_SOURCE_PATH`	Path on the ansible control host in which the kolla-ansible source code will be cloned.	`$PWD/src/kolla-ansible`
`$KOLLA_VENV_PATH`	Path on the ansible control host in which the kolla-ansible virtualenv will be created.	`$PWD/venvs/kolla-ansible`

Remote Execution Environment

By default, ansible executes modules remotely using the system python interpreter, even if the ansible control process is executed from within a virtual environment (unless the `local` connection plugin is used). This is not ideal if there are python dependencies that must be installed without isolation from the system python packages. Ansible can be configured to use a virtualenv by setting the host variable `ansible_python_interpreter` to a path to a python interpreter in an existing virtual environment.

If the variable `kolla_ansible_target_venv` is set, kolla-ansible will be configured to create and use a virtual environment on the remote hosts. This variable is by default set to `{{ virtualenv_path }}/kolla-ansible`. The previous behaviour of installing python dependencies directly to the host can be used by setting `kolla_ansible_target_venv` to `None`.

Control Plane Services

Kolla-ansible provides a flexible mechanism for configuring the services that it deploys. Kayobe adds some commonly required configuration options to the defaults provided by kolla-ansible, but also allows for the free-form configuration supported by kolla-ansible. The kolla-ansible documentation should be used as a reference.

Global Variables

Kolla-ansible uses a single file for global variables, `globals.yml`. Kayobe provides configuration variables for all required variables and many of the most commonly used the variables in this file. Some of these are in `$KAYOBE_CONFIG_PATH/kolla.yml`, and others are determined from other sources such as the networking configuration in `$KAYOBE_CONFIG_PATH/networks.yml`.

Configuring Custom Global Variables

Additional global configuration may be provided by creating `$KAYOBE_CONFIG_PATH/kolla/globals.yml`. Variables in this file will be templated using Jinja2, and merged with the Kayobe `globals.yml` configuration.

Listing 1.21: `$KAYOBE_CONFIG_PATH/kolla/globals.yml`

```
---
# Use a custom tag for the nova-api container image.
nova_api_tag: v1.2.3
```

Passwords

Kolla-ansible auto-generates passwords to a file, `passwords.yml`. Kayobe handles the orchestration of this, as well as encryption of the file using an ansible vault password specified in the `KAYOBE_VAULT_PASSWORD` environment variable, if present. The file is generated to `$KAYOBE_CONFIG_PATH/kolla/passwords.yml`, and should be stored along with other kayobe configuration files. This file should not be manually modified.

Configuring Custom Passwords

In order to write additional passwords to `passwords.yml`, set the kayobe variable `kolla_ansible_custom_passwords` in `$KAYOBE_CONFIG_PATH/kolla.yml`.

Listing 1.22: `$KAYOBE_CONFIG_PATH/kolla.yml`

```
---
# Dictionary containing custom passwords to add or override in the Kolla
# passwords file.
kolla_ansible_custom_passwords: >
  {{ kolla_ansible_default_custom_passwords |
     combine({'my_custom_password': 'correcthorsebatterystaple'}) }}
```

Service Configuration

Kolla-ansible's flexible configuration is described in the kolla-ansible service configuration documentation. We won't duplicate that here, but essentially it involves creating files under a directory which for users of kayobe will be `$KOLLA_CONFIG_PATH/config`. In kayobe, files in this directory are auto-generated and managed by kayobe. Instead, users should create files under `$KAYOBE_CONFIG_PATH/kolla/config` with the same directory structure. These files will be templated using Jinja2, merged with kayobe's own configuration, and written out to `$KOLLA_CONFIG_PATH/config`.

The following files, if present, will be templated and provided to kolla-ansible. All paths are relative to `$KAYOBE_CONFIG_PATH/kolla/config`. Note that typically kolla-ansible does not use the same wildcard patterns, and has a more restricted set of files that it will process. In some cases, it may be necessary to inspect the kolla-ansible configuration tasks to determine which files are supported.

Configuring an OpenStack Component

To provide custom configuration to be applied to all glance services, create `$KAYOBE_CONFIG_PATH/kolla/config/glance.conf`. For example:

Listing 1.23: `$KAYOBE_CONFIG_PATH/kolla/config/glance.conf`

```
[DEFAULT]
api_limit_max = 500
```

Configuring an OpenStack Service

To provide custom configuration for the glance API service, create `$KAYOBE_CONFIG_PATH/kolla/config/glance/glance-api.conf`. For example:

Listing 1.24: $KAYOBE_CONFIG_PATH/kolla/config/glance/glance-api.conf

```
[DEFAULT]
api_limit_max = 500
```

1.2.5 Deployment

This section describes usage of Kayobe to install an OpenStack cloud onto a set of bare metal servers. We assume access is available to a node which will act as the hypervisor hosting the seed node in a VM. We also assume that this seed hypervisor has access to the bare metal nodes that will form the OpenStack control plane. Finally, we assume that the control plane nodes have access to the bare metal nodes that will form the workload node pool.

Ansible Control Host

Before starting deployment we must bootstrap the Ansible control host. Tasks performed here include:

- Install Ansible and role dependencies from Ansible Galaxy.
- Generate an SSH key if necessary and add it to the current user's authorised keys.

To bootstrap the Ansible control host:

```
(kayobe) $ kayobe control host bootstrap
```

Physical Network

The physical network can be managed by Kayobe, which uses Ansible's network modules. Currently Dell Network OS 6 and Dell Network OS 9 switches are supported but this could easily be extended. To provision the physical network:

```
(kayobe) $ kayobe physical network configure --group <group> [--enable-discovery]
```

The `--group` argument is used to specify an Ansible group containing the switches to be configured.

The `--enable-discovery` argument enables a one-time configuration of ports attached to baremetal compute nodes to support hardware discovery via ironic inspector.

It is possible to limit the switch interfaces that will be configured, either by interface name or interface description:

```
(kayobe) $ kayobe physical network configure --group <group> --interface-limit
↪<interface names>
(kayobe) $ kayobe physical network configure --group <group> --interface-description-
↪limit <interface descriptions>
```

The names or descriptions should be separated by commas. This may be useful when adding compute nodes to an existing deployment, in order to avoid changing the configuration interfaces in use by active nodes.

The `display` argument will display the candidate switch configuration, without actually applying it.

Seed Hypervisor

Note: It is not necessary to run the seed services in a VM. To use an existing bare metal host or a VM provisioned outside of Kayobe, this section may be skipped.

Host Configuration

To configure the seed hypervisor's host OS, and the Libvirt/KVM virtualisation support:

```
(kayobe) $ kayobe seed hypervisor host configure
```

Seed

VM Provisioning

Note: It is not necessary to run the seed services in a VM. To use an existing bare metal host or a VM provisioned outside of Kayobe, this step may be skipped. Ensure that the Ansible inventory contains a host for the seed.

The seed hypervisor should have CentOS and `libvirt` installed. It should have `libvirt` networks configured for all networks that the seed VM needs access to and a `libvirt` storage pool available for the seed VM's volumes. To provision the seed VM:

```
(kayobe) $ kayobe seed vm provision
```

When this command has completed the seed VM should be active and accessible via SSH. Kayobe will update the Ansible inventory with the IP address of the VM.

Host Configuration

To configure the seed host OS:

```
(kayobe) $ kayobe seed host configure
```

Note: If the seed host uses disks that have been in use in a previous installation, it may be necessary to wipe partition and LVM data from those disks. To wipe all disks that are not mounted during host configuration:

```
(kayobe) $ kayobe seed host configure --wipe-disks
```

Building Container Images

Note: It is possible to use prebuilt container images from an image registry such as Dockerhub. In this case, this step can be skipped.

It is possible to use prebuilt container images from an image registry such as Dockerhub. In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of kolla. Images are built by hosts in the `container-image-builders` group, which by default includes the `seed`.

To build container images:

```
(kayobe) $ kayobe seed container image build
```

It is possible to build a specific set of images by supplying one or more image name regular expressions:

```
(kayobe) $ kayobe seed container image build bifrost-deploy
```

In order to push images to a registry after they are built, add the `--push` argument.

Deploying Containerised Services

At this point the seed services need to be deployed on the seed VM. These services are deployed in the `bifrost_deploy` container. This command will also build the Operating System image that will be used to deploy the overcloud nodes using Disk Image Builder (DIB).

To deploy the seed services in containers:

```
(kayobe) $ kayobe seed service deploy
```

After this command has completed the seed services will be active.

Building Deployment Images

Note: It is possible to use prebuilt deployment images. In this case, this step can be skipped.

It is possible to use prebuilt deployment images from the OpenStack hosted tarballs or another source. In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of Ironic Python Agent (IPA). In order to build IPA images, the `ipa_build_images` variable should be set to `True`. To build images locally:

```
(kayobe) $ kayobe seed deployment image build
```

Accessing the Seed via SSH (Optional)

For SSH access to the seed, first determine the seed's IP address. We can use the `kayobe configuration dump` command to inspect the seed's IP address:

```
(kayobe) $ kayobe configuration dump --host seed --var-name ansible_host
```

The `kayobe_ansible_user` variable determines which user account will be used by Kayobe when accessing the machine via SSH. By default this is `stack`. Use this user to access the seed:

```
$ ssh <kayobe ansible user>@<seed VM IP>
```

To see the active Docker containers:

```
$ docker ps
```

Leave the seed VM and return to the shell on the control host:

```
$ exit
```

Overcloud

Discovery

Note: If discovery of the overcloud is not possible, a static inventory of servers using the bifrost `servers.yml` file format may be configured using the `kolla_bifrost_servers` variable in `${KAYOBE_CONFIG_PATH}/bifrost.yml`.

Discovery of the overcloud is supported by the ironic inspector service running in the `bifrost_deploy` container on the seed. The service is configured to PXE boot unrecognised MAC addresses with an IPA ramdisk for introspection. If an introspected node does not exist in the ironic inventory, ironic inspector will create a new entry for it.

Discovery of the overcloud is triggered by causing the nodes to PXE boot using a NIC attached to the overcloud provisioning network. For many servers this will be the factory default and can be performed by powering them on.

On completion of the discovery process, the overcloud nodes should be registered with the ironic service running in the seed host's `bifrost_deploy` container. The node inventory can be viewed by executing the following on the seed:

```
$ docker exec -it bifrost_deploy bash
(bifrost_deploy) $ source env-vars
(bifrost_deploy) $ ironic node-list
```

In order to interact with these nodes using Kayobe, run the following command to add them to the Kayobe and bifrost Ansible inventories:

```
(kayobe) $ kayobe overcloud inventory discover
```

Saving Hardware Introspection Data

If ironic inspector is in use on the seed host, introspection data will be stored in the local nginx service. This data may be saved to the control host:

```
(kayobe) $ kayobe overcloud introspection data save
```

`--output-dir` may be used to specify the directory in which introspection data files will be saved. `--output-format` may be used to set the format of the files.

BIOS and RAID Configuration

Note: BIOS and RAID configuration may require one or more power cycles of the hardware to complete the operation. These will be performed automatically.

Configuration of BIOS settings and RAID volumes is currently performed out of band as a separate task from hardware provisioning. To configure the BIOS and RAID:

```
(kayobe) $ kayobe overcloud bios raid configure
```

After configuring the nodes' RAID volumes it may be necessary to perform hardware inspection of the nodes to reconfigure the ironic nodes' scheduling properties and root device hints. To perform manual hardware inspection:

```
(kayobe) $ kayobe overcloud hardware inspect
```

Provisioning

Provisioning of the overcloud is performed by the ironic service running in the bifrost container on the seed. To provision the overcloud nodes:

```
(kayobe) $ kayobe overcloud provision
```

After this command has completed the overcloud nodes should have been provisioned with an OS image. The command will wait for the nodes to become `active` in ironic and accessible via SSH.

Host Configuration

To configure the overcloud hosts' OS:

```
(kayobe) $ kayobe overcloud host configure
```

Note: If the controller hosts use disks that have been in use in a previous installation, it may be necessary to wipe partition and LVM data from those disks. To wipe all disks that are not mounted during host configuration:

```
(kayobe) $ kayobe overcloud host configure --wipe-disks
```

Building Container Images

Note: It is possible to use prebuilt container images from an image registry such as Dockerhub. In this case, this step can be skipped.

In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of kolla. Images are built by hosts in the `container-image-builders` group, which by default includes the `seed`. If no seed host is in use, for example in an all-in-one controller development environment, this group may be modified to cause containers to be built on the controllers.

To build container images:

```
(kayobe) $ kayobe overcloud container image build
```

It is possible to build a specific set of images by supplying one or more image name regular expressions:

```
(kayobe) $ kayobe overcloud container image build ironic- nova-api
```

In order to push images to a registry after they are built, add the `--push` argument.

Pulling Container Images

Note: It is possible to build container images locally avoiding the need for an image registry such as Dockerhub. In this case, this step can be skipped.

In most cases suitable prebuilt kolla images will be available on Dockerhub. The stackhpc account provides image repositories suitable for use with kayobe and will be used by default. To pull images from the configured image registry:

```
(kayobe) $ kayobe overcloud container image pull
```

Building Deployment Images

Note: It is possible to use prebuilt deployment images. In this case, this step can be skipped.

It is possible to use prebuilt deployment images from the OpenStack hosted tarballs or another source. In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of Ironic Python Agent (IPA). In order to build IPA images, the `ipa_build_images` variable should be set to `True`. To build images locally:

```
(kayobe) $ kayobe overcloud deployment image build
```

Deploying Containerised Services

To deploy the overcloud services in containers:

```
(kayobe) $ kayobe overcloud service deploy
```

Once this command has completed the overcloud nodes should have OpenStack services running in Docker containers.

Interacting with the Control Plane

Kolla-ansible writes out an environment file that can be used to access the OpenStack admin endpoints as the admin user:

```
$ source ${KOLLA_CONFIG_PATH:-/etc/kolla}/admin-openrc.sh
```

Kayobe also generates an environment file that can be used to access the OpenStack public endpoints as the admin user which may be required if the admin endpoints are not available from the control host:

```
$ source ${KOLLA_CONFIG_PATH:-/etc/kolla}/public-openrc.sh
```

Performing Post-deployment Configuration

To perform post deployment configuration of the overcloud services:

```
(kayobe) $ source ${KOLLA_CONFIG_PATH:-/etc/kolla}/admin-openrc.sh
(kayobe) $ kayobe overcloud post configure
```

This will perform the following tasks:

- Register Ironic Python Agent (IPA) images with glance
- Register introspection rules with ironic inspector
- Register a provisioning network and subnet with neutron
- Configure Grafana organisations, dashboards and datasources

1.2.6 Upgrading

This section describes how to upgrade from one OpenStack release to another.

Preparation

Before you start, be sure to back up any local changes, configuration, and data.

Upgrading Kayobe

If a new, suitable version of kayobe is available, it should be installed. If using kayobe from a git checkout, this may be done by pulling down the new version from Github. Make sure that any local changes to kayobe are committed. For example, to pull version 1.0.0 from the `origin` remote:

```
$ git pull origin 1.0.0
```

If local changes were made to kayobe, these should now be reapplied.

The upgraded kayobe python module and dependencies should be installed:

```
(kayobe) $ pip install -U .
```

Migrating Kayobe Configuration

Kayobe configuration options may be changed between releases of kayobe. Ensure that all site local configuration is migrated to the target version format. If using the kayobe-config git repository to manage local configuration, this process can be managed via git. For example, to fetch version 1.0.0 of the configuration from the `origin` remote and merge it into the current branch:

```
$ git fetch origin 1.0.0
$ git merge 1.0.0
```

The configuration should be manually inspected after the merge to ensure that it is correct. Any new configuration options may be set at this point. In particular, the following options may need to be changed if not using their default values:

- `kolla_openstack_release`
- `kolla_sources`
- `kolla_build_blocks`

- `kolla_build_customizations`

Once the configuration has been migrated, it is possible to view the global variables for all hosts:

```
(kayobe) $ kayobe configuration dump
```

The output of this command is a JSON object mapping hosts to their configuration. The output of the command may be restricted using the `--host`, `--hosts`, `--var-name` and `--dump-facts` options.

If using the `kayobe-env` environment file in `kayobe-config`, this should also be inspected for changes and modified to suit the local ansible control host environment if necessary. When ready, source the environment file:

```
$ source kayobe-env
```

Upgrading the Control Host

Before starting the upgrade we must upgrade the Ansible control host. Tasks performed here include:

- Install updated Ansible role dependencies from Ansible Galaxy.
- Generate an SSH key if necessary and add it to the current user's authorised keys.

To upgrade the Ansible control host:

```
(kayobe) $ kayobe control host upgrade
```

Upgrading the Seed Hypervisor

Currently, upgrading the seed hypervisor services is not supported. It may however be necessary to upgrade some host services:

```
(kayobe) $ kayobe seed hypervisor host upgrade
```

Note that this will not perform full configuration of the host, and will instead perform a targeted upgrade of specific services where necessary.

Upgrading the Seed

Currently, upgrading the seed services is not supported. It may however be necessary to upgrade some host services:

```
(kayobe) $ kayobe seed host upgrade
```

Note that this will not perform full configuration of the host, and will instead perform a targeted upgrade of specific services where necessary.

Upgrading the Overcloud

The overcloud services are upgraded in two steps. First, new container images should be obtained either by building them locally or pulling them from an image registry. Second, the overcloud services should be replaced with new containers created from the new container images.

Upgrading Host Services

Prior to upgrading the OpenStack control plane, the overcloud host services should be upgraded:

```
(kayobe) $ kayobe overcloud host upgrade
```

Note that this will not perform full configuration of the host, and will instead perform a targeted upgrade of specific services where necessary.

Building Ironic Deployment Images

Note: It is possible to use prebuilt deployment images. In this case, this step can be skipped.

It is possible to use prebuilt deployment images from the OpenStack hosted tarballs or another source. In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of Ironic Python Agent (IPA). In order to build IPA images, the `ipa_build_images` variable should be set to `True`. To build images locally:

```
(kayobe) $ kayobe overcloud deployment image build
```

Upgrading Ironic Deployment Images

Prior to upgrading the OpenStack control plane, the baremetal compute nodes should be configured to use an updated deployment ramdisk. This procedure is not currently automated by kayobe, and should be performed manually.

Building Container Images

Note: It is possible to use prebuilt container images from an image registry such as Dockerhub. In this case, this step can be skipped.

In some cases it may be necessary to build images locally either to apply local image customisation or to use a downstream version of kolla. To build images locally:

```
(kayobe) $ kayobe overcloud container image build
```

It is possible to build a specific set of images by supplying one or more image name regular expressions:

```
(kayobe) $ kayobe overcloud container image build ironic- nova-api
```

In order to push images to a registry after they are built, add the `--push` argument.

Pulling Container Images

Note: It is possible to build container images locally avoiding the need for an image registry such as Dockerhub. In this case, this step can be skipped.

In most cases suitable prebuilt kolla images will be available on Dockerhub. The stackhpc account provides image repositories suitable for use with kayobe and will be used by default. To pull images from the configured image registry:

```
(kayobe) $ kayobe overcloud container image pull
```

Saving Overcloud Service Configuration

It is often useful to be able to save the configuration of the control plane services for inspection or comparison with another configuration set prior to a reconfiguration or upgrade. This command will gather and save the control plane configuration for all hosts to the ansible control host:

```
(kayobe) $ kayobe overcloud service configuration save
```

The default location for the saved configuration is `$PWD/overcloud-config`, but this can be changed via the `output-dir` argument. To gather configuration from a directory other than the default `/etc/kolla`, use the `node-config-dir` argument.

Generating Overcloud Service Configuration

Prior to deploying, reconfiguring, or upgrading a control plane, it may be useful to generate the configuration that will be applied, without actually applying it to the running containers. The configuration should typically be generated in a directory other than the default configuration directory of `/etc/kolla`, to avoid overwriting the active configuration:

```
(kayobe) $ kayobe overcloud service configuration generate --node-config-dir /path/to/
↪generated/config
```

The configuration will be generated remotely on the overcloud hosts in the specified directory, with one subdirectory per container. This command may be followed by `kayobe ovecloud service configuration save` to gather the generated configuration to the ansible control host.

Upgrading Containerised Services

Containerised control plane services may be upgraded by replacing existing containers with new containers using updated images which have been pulled from a registry or built locally.

To upgrade the containerised control plane services:

```
(kayobe) $ kayobe overcloud service upgrade
```

It is possible to specify tags for Kayobe and/or kolla-ansible to restrict the scope of the upgrade:

```
(kayobe) $ kayobe overcloud service upgrade --tags config --kolla-tags keystone
```

1.2.7 Administration

This section describes how to use kayobe to simplify post-deployment administrative tasks.

Reconfiguring Containerised Services

When configuration is changed, it is necessary to apply these changes across the system in an automated manner. To reconfigure the overcloud, first make any changes required to the configuration on the control host. Next, run the following command:

```
(kayobe) $ kayobe overcloud service reconfigure
```

In case not all services' configuration have been modified, performance can be improved by specifying Ansible tags to limit the tasks run in kayobe and/or kolla-ansible's playbooks. This may require knowledge of the inner workings of these tools but in general, kolla-ansible tags the play used to configure each service by the name of that service. For example: `nova`, `neutron` or `ironic`. Use `-t` or `--tags` to specify kayobe tags and `-kt` or `--kolla-tags` to specify kolla-ansible tags. For example:

```
(kayobe) $ kayobe overcloud service reconfigure --tags config --kolla-tags nova,ironic
```

Upgrading Containerised Services

Containerised control plane services may be upgraded by replacing existing containers with new containers using updated images which have been pulled from a registry or built locally. If using an updated version of Kayobe or upgrading from one release of OpenStack to another, be sure to follow the *kayobe upgrade guide*. It may be necessary to upgrade one or more services within a release, for example to apply a patch or minor release.

To upgrade the containerised control plane services:

```
(kayobe) $ kayobe overcloud service upgrade
```

As for the reconfiguration command, it is possible to specify tags for Kayobe and/or kolla-ansible:

```
(kayobe) $ kayobe overcloud service upgrade --tags config --kolla-tags keystone
```

Destroying the Overcloud Services

Note: This step will destroy all containers, container images, volumes and data on the overcloud hosts.

To destroy the overcloud services:

```
(kayobe) $ kayobe overcloud service destroy --yes-i-really-really-mean-it
```

Deprovisioning The Cloud

Note: This step will power down the overcloud hosts and delete their nodes' instance state from the seed's ironic service.

To deprovision the overcloud:

```
(kayobe) $ kayobe overcloud deprovision
```

Deprovisioning The Seed VM

Note: This step will destroy the seed VM and its data volumes.

To deprovision the seed VM:

```
(kayobe) $ kayobe seed vm deprovision
```

Saving Overcloud Service Configuration

It is often useful to be able to save the configuration of the control plane services for inspection or comparison with another configuration set prior to a reconfiguration or upgrade. This command will gather and save the control plane configuration for all hosts to the ansible control host:

```
(kayobe) $ kayobe overcloud service configuration save
```

The default location for the saved configuration is `$PWD/overcloud-config`, but this can be changed via the `output-dir` argument. To gather configuration from a directory other than the default `/etc/kolla`, use the `node-config-dir` argument.

Generating Overcloud Service Configuration

Prior to deploying, reconfiguring, or upgrading a control plane, it may be useful to generate the configuration that will be applied, without actually applying it to the running containers. The configuration should typically be generated in a directory other than the default configuration directory of `/etc/kolla`, to avoid overwriting the active configuration:

```
(kayobe) $ kayobe overcloud service configuration generate --node-config-dir /path/to/
↪generated/config
```

The configuration will be generated remotely on the overcloud hosts in the specified directory, with one subdirectory per container. This command may be followed by `kayobe ovecloud service configuration save` to gather the generated configuration to the ansible control host.

Checking Network Connectivity

In complex networking environments it can be useful to be able to automatically check network connectivity and diagnose networking issues. To perform some simple connectivity checks:

```
(kayobe) $ kayobe network connectivity check
```

Note that this will run on the seed, seed hypervisor, and overcloud hosts. If any of these hosts are not expected to be active (e.g. prior to overcloud deployment), the set of target hosts may be limited using the `--limit` argument.

Baremetal Compute Node Management

When enrolling new hardware or performing maintenance, it can be useful to be able to manage many bare metal compute nodes simulteneously.

In all cases, commands are delegated to one of the controller hosts, and executed concurrently. Note that ansible's `forks` configuration option, which defaults to 5, may limit the number of nodes configured concurrently.

By default these commands wait for the state transition to complete for each node. This behavior can be changed by overriding the variable `baremetal_compute_wait` via `-e baremetal_compute_wait=False`

Manage

A node may need to be set to the `manageable` provision state in order to perform certain management operations, or when an enrolled node is transitioned into service. In order to manage a node, it must be in one of these states: `enroll`, `available`, `cleaning`, `clean failed`, `adopt failed` or `inspect failed`. To move the baremetal compute nodes to the `manageable` provision state:

```
(kayobe) $ kayobe baremetal compute manage
```

Provide

In order for nodes to be scheduled by nova, they must be `available`. To move the baremetal compute nodes from the `manageable` state to the `available` provision state:

```
(kayobe) $ kayobe baremetal compute provide
```

Inspect

Nodes must be in one of the following states: `manageable`, `inspect failed`, or `available`. To trigger hardware inspection on the baremetal compute nodes:

```
(kayobe) $ kayobe baremetal compute inspect
```

Running Kayobe Playbooks on Demand

In some situations it may be necessary to run an individual Kayobe playbook. Playbooks are stored in `<kayobe repo>/ansible/*.yml`. To run an arbitrary Kayobe playbook:

```
(kayobe) $ kayobe playbook run <playbook> [<playbook>]
```

Running Kolla-ansible Commands

To execute a kolla-ansible command:

```
(kayobe) $ kayobe kolla ansible run <command>
```

Dumping Kayobe Configuration

The Ansible configuration space is quite large, and it can be hard to determine the final values of Ansible variables. We can use Kayobe's `configuration dump` command to view individual variables or the variables for one or more hosts. To dump Kayobe configuration for one or more hosts:

```
(kayobe) $ kayobe configuration dump
```

The output is a JSON-formatted object mapping hosts to their hostvars.

We can use the `--var-name` argument to inspect a particular variable or the `--host` or `--hosts` arguments to view a variable or variables for a specific host or set of hosts.

1.3 Advanced Documentation

1.3.1 Control Plane Service Placement

Note: This is an advanced topic and should only be attempted when familiar with kayobe and OpenStack.

The default configuration in kayobe places all control plane services on a single set of servers described as 'controllers'. In some cases it may be necessary to introduce more than one server role into the control plane, and control which services are placed onto the different server roles.

Configuration

Overcloud Inventory Discovery

If using a seed host to enable discovery of the control plane services, it is necessary to configure how the discovered hosts map into kayobe groups. This is done using the `overcloud_group_hosts_map` variable, which maps names of kayobe groups to a list of the hosts to be added to that group.

This variable will be used during the command `kayobe overcloud inventory discover`. An inventory file will be generated in `${KAYOBE_CONFIG_PATH}/inventory/overcloud` with discovered hosts added to appropriate kayobe groups based on `overcloud_group_hosts_map`.

Kolla-ansible Inventory Mapping

Once hosts have been discovered and enrolled into the kayobe inventory, they must be added to the kolla-ansible inventory. This is done by mapping from top level kayobe groups to top level kolla-ansible groups using the `kolla_overcloud_inventory_top_level_group_map` variable. This variable maps from kolla-ansible groups to lists of kayobe groups, and variables to define for those groups in the kolla-ansible inventory.

Variables For Custom Server Roles

Certain variables must be defined for hosts in the `overcloud` group. For hosts in the `controllers` group, many variables are mapped to other variables with a `controller_` prefix in files under `ansible/group_vars/controllers/`. This is done in order that they may be set in a global extra variables file, typically `controllers.yml`, with defaults set in `ansible/group_vars/all/controllers`. A similar scheme is used for hosts in the `monitoring` group.

Table 1.2: Overcloud host variables

Variable	Purpose
`ansible_user`	Username with which to access the host via SSH.
`bootstrap_user`	Username with which to access the host before `ansible_user` is configured.
`lvm_groups`	List of LVM volume groups to configure. See mrlesmithjr.manage-lvm role for format.
`network_interfaces`	List of names of networks to which the host is connected.
`sysctl_parameters`	Dict of sysctl parameters to set.
`users`	List of users to create. See singleplatform-eng.users role

If configuring BIOS and RAID via `kayobe overcloud bios raid configure`, the following variables should also be defined:

Table 1.3: Overcloud BIOS & RAID host variables

Variable	Purpose
`bios_config`	Dict mapping BIOS configuration options to their required values. See stackhpc.drac role for format.
`raid_config`	List of RAID virtual disks to configure. See stackhpc.drac role for format.

These variables can be defined in inventory host or group variables files, under `${KAYOBE_CONFIG_PATH}/inventory/host_vars/<host>` or `${KAYOBE_CONFIG_PATH}/inventory/group_vars/<group>` respectively.

Custom Kolla-ansible Inventories

As an advanced option, it is possible to fully customise the content of the kolla-ansible inventory, at various levels. To facilitate this, kayobe breaks the kolla-ansible inventory into three separate sections.

Top level groups define the roles of hosts, e.g. `controller` or `compute`, and it is to these groups that hosts are mapped directly.

Components define groups of services, e.g. `nova` or `ironic`, which are mapped to top level groups.

Services define single containers, e.g. `nova-compute` or `ironic-api`, which are mapped to components.

The default top level inventory is generated from `kolla_overcloud_inventory_top_level_group_map`. Kayobe's component- and service-level inventory for kolla-ansible is static, and taken from the kolla-ansible example `multinode` inventory. The complete inventory is generated by concatenating these inventories.

Each level may be separately overridden by setting the following variables:

Table 1.4: Custom kolla-ansible inventory variables

Variable	Purpose
`kolla_overcloud_inventory_custom_top_level`	Overcloud inventory containing a mapping from top level groups to hosts.
`kolla_overcloud_inventory_custom_components`	Overcloud inventory containing a mapping from components to top level groups.
`kolla_overcloud_inventory_custom_services`	Overcloud inventory containing a mapping from services to components.
`kolla_overcloud_inventory_custom`	Full overcloud inventory contents.

Examples

Example 1: Adding Network Hosts

This example walks through the configuration that could be applied to enable the use of separate hosts for neutron network services and load balancing. The control plane consists of three controllers, `controller-[0-2]`, and two network hosts, `network-[0-1]`. All file paths are relative to `${KAYOBE_CONFIG_PATH}`.

First, we must map the hosts to kayobe groups.

Listing 1.25: `overcloud.yml`

```
overcloud_group_hosts_map:
  controllers:
    - controller-0
    - controller-1
    - controller-2
  network:
    - network-0
    - network-1
```

Next, we must map these groups to kolla-ansible groups.

Listing 1.26: `kolla.yml`

```
kolla_overcloud_inventory_top_level_group_map:
  control:
    groups:
      - controllers
  network:
    groups:
      - network
```

Finally, we create a group variables file for hosts in the network group, providing the necessary variables for a control plane host.

Listing 1.27: `inventory/group_vars/network`

```
ansible_user: "{{ kayobe_ansible_user }}"
bootstrap_user: "{{ controller_bootstrap_user }}"
lvm_groups: "{{ controller_lvm_groups }}"
network_interfaces: "{{ controller_network_host_network_interfaces }}"
sysctl_parameters: "{{ controller_sysctl_parameters }}"
users: "{{ controller_users }}"
```

Here we are using the controller-specific values for some of these variables, but they could equally be different.

Example 2: Overriding the Kolla-ansible Inventory

This example shows how to override one or more sections of the kolla-ansible inventory. All file paths are relative to `${KAYOBE_CONFIG_PATH}`.

First, create a file containing the customised inventory section. We'll use the **components** section in this example.

Listing 1.28: `kolla/inventory/overcloud-components.j2`

```
[nova]
control

[ironic]
{% if kolla_enable_ironic | bool %}
control
{% endif %}

...
```

Next, we must configure kayobe to use this inventory template.

Listing 1.29: `kolla.yml`

```
kolla_overcloud_inventory_custom_components: "{{ lookup('template', kayobe_config_
↪path ~ '/kolla/inventory/overcloud-components.j2') }}"
```

Here we use the `template` lookup plugin to render the Jinja2-formatted inventory template.

1.4 Developer Documentation

1.4.1 Kayobe Development Guide

Vagrant

Kayobe provides a Vagrantfile that can be used to bring up a virtual machine for use as a development environment. The VM is based on the stackhpc/centos-7 CentOS 7 image, and supports the following providers:

- VirtualBox
- VMWare Fusion

The VM is configured with 4GB RAM. It has a single private network in addition to the standard Vagrant NAT network.

Preparation

First, ensure that Vagrant is installed and correctly configured to use the required provider. Also install the following vagrant plugin:

```
vagrant plugin install vagrant-reload
```

If using the VirtualBox provider, install the following vagrant plugin:

```
vagrant plugin install vagrant-vbguest
```

Note: if using Ubuntu 16.04 LTS, you may be unable to install any plugins. To work around this install the upstream version from www.virtualbox.org.

Usage

Later sections in the development guide cover in more detail how to use the development VM in different configurations. These steps cover bringing up and accessing the VM.

Clone the kayobe repository:

```
git clone https://github.com/stackhpc/kayobe
```

Change the current directory to the kayobe repository:

```
cd kayobe
```

Inspect kayobe's `Vagrantfile`, noting the provisioning steps:

```
less Vagrantfile
```

Bring up a virtual machine:

```
vagrant up
```

Wait for the VM to boot, then SSH in:

```
vagrant ssh
```

Manual Setup

This section provides a set of manual steps to set up a development environment for an OpenStack controller in a virtual machine using Vagrant and Kayobe.

For a more automated and flexible procedure, see *Automated Setup*.

Preparation

Follow the steps in *Vagrant* to prepare your environment for use with Vagrant and bring up a Vagrant VM.

Manual Installation

Sometimes the best way to learn a tool is to ditch the scripts and perform a manual installation.

SSH into the controller VM:

```
vagrant ssh
```

Source the kayobe virtualenv activation script:

```
source kayobe-venv/bin/activate
```

Change the current directory to the Vagrant shared directory:

```
cd /vagrant
```

Source the kayobe environment file:

```
source kayobe-env
```

Bootstrap the kayobe control host:

```
kayobe control host bootstrap
```

Configure the controller host:

```
kayobe overcloud host configure
```

At this point, container images must be acquired. They can either be built locally or pulled from an image repository if appropriate images are available.

Either build container images:

```
kayobe overcloud container image build
```

Or pull container images:

```
kayobe overcloud container image pull
```

Deploy the control plane services:

```
kayobe overcloud service deploy
```

Source the OpenStack environment file:

```
source ${KOLLA_CONFIG_PATH:-/etc/kolla}/admin-openrc.sh
```

Perform post-deployment configuration:

```
kayobe overcloud post configure
```

Next Steps

The OpenStack control plane should now be active. Try out the following:

- register a user
- create an image
- upload an SSH keypair
- access the horizon dashboard

The cloud is your oyster!

To Do

Create virtual baremetal nodes to be managed by the OpenStack control plane.

Automated Setup

This section provides information on the development tools provided by kayobe to automate the deployment of various development environments.

For a manual procedure, see *Manual Setup*.

Overview

The kayobe development environment automation tooling is built using simple shell scripts. Some minimal configuration can be applied by setting the environment variables in *dev/config.sh*. Control plane configuration is typically provided via the dev-kayobe-config repository, although it is also possible to use your own kayobe configuration. This allows us to build a development environment that is as close to production as possible.

Environments

The following development environments are supported:

- Overcloud (single OpenStack controller)
- Seed hypervisor
- Seed VM

The seed VM environment may be used in an environment already deployed as a seed hypervisor.

Overcloud

Preparation

Clone the kayobe repository:

```
git clone https://github.com/stackhpc/kayobe
```

Change the current directory to the kayobe repository:

```
cd kayobe
```

Clone the `dev-kayobe-config` repository to `config/src/kayobe-config`:

```
mkdir -p config/src
git clone https://github.com/stackhpc/dev-kayobe-config config/src/kayobe-config
```

Inspect the kayobe configuration and make any changes necessary for your environment.

If using Vagrant, follow the steps in *Vagrant* to prepare your environment for use with Vagrant and bring up a Vagrant VM.

If not using Vagrant, the default development configuration expects the presence of a bridge interface on the OpenStack controller host to carry control plane traffic. The bridge should be named `breth1` with a single port `eth1`, and an IP address of `192.168.33.3/24`. This can be modified by editing `config/src/kayobe-config/etc/kayobe/inventory/group_vars/controllers/network-interfaces`. Alternatively, this can be added using the following commands:

```
sudo ip l add breth1 type bridge
sudo ip l set breth1 up
sudo ip a add 192.168.33.3/24 dev breth1
sudo ip l add eth1 type dummy
sudo ip l set eth1 up
sudo ip l set eth1 master breth1
```

Usage

If using Vagrant, SSH into the Vagrant VM and change to the shared directory:

```
vagrant ssh
cd /vagrant
```

If not using Vagrant, run the `dev/install.sh` script to install kayobe and its dependencies in a virtual environment:

```
./dev/install.sh
```

Run the `dev/overcloud-deploy.sh` script to deploy the OpenStack control plane:

```
./dev/overcloud-deploy.sh
```

Upon successful completion of this script, the control plane will be active.

Seed Hypervisor

The seed hypervisor development environment is supported for CentOS 7. The system must be either bare metal, or a VM on a system with nested virtualisation enabled.

Preparation

The following commands should be executed on the seed hypervisor.

Clone the kayobe repository:

```
git clone https://github.com/stackhpc/kayobe
```

Change the current directory to the kayobe repository:

```
cd kayobe
```

Clone the `add-seed-and-hv` branch of the `dev-kayobe-config` repository to `config/src/kayobe-config`:

```
mkdir -p config/src
git clone https://github.com/stackhpc/dev-kayobe-config -b add-seed-and-hv config/src/
↪kayobe-config
```

Inspect the kayobe configuration and make any changes necessary for your environment.

Usage

Run the `dev/install.sh` script to install kayobe and its dependencies in a virtual environment:

```
./dev/install.sh
```

Run the `dev/seed-hypervisor-deploy.sh` script to deploy the seed hypervisor:

```
./dev/seed-hypervisor-deploy.sh
```

Upon successful completion of this script, the seed hypervisor will be active.

Seed VM

The seed VM should be deployed on a system configured as a libvirt/KVM hypervisor, using the kayobe seed hypervisor support or otherwise.

Preparation

The following commands should be executed on the seed hypervisor.

Change the current directory to the kayobe repository:

```
git clone https://github.com/stackhpc/kayobe
```

Change to the `kayobe` directory:

```
cd kayobe
```

Clone the `add-seed-and-hv` branch of the `dev-kayobe-config` repository to `config/src/kayobe-config`:

```
mkdir -p config/src
git clone https://github.com/stackhpc/dev-kayobe-config -b add-seed-and-hv config/src/
↪kayobe-config
```

Inspect the kayobe configuration and make any changes necessary for your environment.

Usage

Run the `dev/install.sh` script to install kayobe and its dependencies in a virtual environment:

```
./dev/install.sh
```

Run the `dev/seed-deploy.sh` script to deploy the seed VM:

```
./dev/seed-deploy.sh
```

Upon successful completion of this script, the seed VM will be active. The seed VM may be accessed via SSH as the `stack` user:

```
ssh stack@192.168.33.5
```

Testing

Kayobe has a number of test suites covering different areas of code. Many tests are run in virtual environments using `tox`.

Preparation

System Prerequisites

The following packages should be installed on the development system prior to running kayobe's tests.

- Ubuntu/Debian:

```
sudo apt-get install build-essential python-dev libssl-dev python-pip git
```

- Fedora 21/RHEL7/CentOS7:

```
sudo yum install python-devel openssl-devel python-pip git gcc
```

- Fedora 22 or higher:

```
sudo dnf install python-devel openssl-devel python-pip git gcc
```

- OpenSUSE/SLE 12:

```
sudo zypper install python-devel python-pip libopenssl-devel git
```

Python Prerequisites

If your distro has at least `tox 1.8`, use your system package manager to install the `python-tox` package. Otherwise install this on all distros:

```
sudo pip install -U tox
```

You may need to explicitly upgrade `virtualenv` if you've installed the one from your OS distribution and it is too old (tox will complain). You can upgrade it individually, if you need to:

```
sudo pip install -U virtualenv
```

Running Unit Tests Locally

If you haven't already, the kayobe source code should be pulled directly from git:

```
# from your home or source directory
cd ~
git clone https://github.com/stackhpc/kayobe
cd kayobe
```

Running Unit and Style Tests

Kayobe defines a number of different tox environments in `tox.ini`. The default environments may be displayed:

```
tox -list
```

To run all default environments:

```
tox
```

To run one or more specific environments, including any of the non-default environments:

```
tox -e <environment>[,<environment>]
```

Environments

The following tox environments are provided:

alint Run Ansible linter.

ansible Run Ansible tests for some ansible roles using Ansible playbooks.

ansible-syntax Run a syntax check for all Ansible files.

docs Build Sphinx documentation.

molecule Run Ansible tests for some Ansible roles using the molecule test framework.

pep8 Run style checks for all shell, python and documentation files.

py27,py34 Run python unit tests for kayobe python module.

Writing Tests

Unit Tests

Unit tests follow the lead of OpenStack, and use `unittest`. One difference is that tests are run using the discovery functionality built into `unittest`, rather than `ostestr`/`stestr`. Unit tests are found in `kayobe/tests/unit/`, and should be added to cover all new python code.

Ansible Role Tests

Two types of test exist for Ansible roles - pure Ansible and molecule tests.

Pure Ansible Role Tests

These tests exist for the `kolla-ansible` role, and are found in `ansible/<role>/tests/*.yml`. The role is exercised using an ansible playbook.

Molecule Role Tests

Molecule is an Ansible role testing framework that allows roles to be tested in isolation, in a stable environment, under multiple scenarios. Kayobe uses Docker engine to provide the test environment, so this must be installed and running on the development system.

Molecule scenarios are found in `ansible/<role>/molecule/<scenario>`, and defined by the config file `ansible/<role>/molecule/<scenario>/molecule.yml` Tests are written in python using the pytest framework, and are found in `ansible/<role>/molecule/<scenario>/tests/test_*.py`.

Molecule tests currently exist for the `kolla-openstack` role, and should be added for all new roles where practical.

How to Contribute

Kayobe does not currently follow the upstream OpenStack development process, but we will still be incredibly grateful for any contributions.

Please raise issues and submit pull requests via Github.

For team discussion we use the #openstack-kayobe IRC channel.

Thanks in advance!

1.5 Release Notes

1.5.1 Release Notes

In Development

Features

Upgrade Notes

Kayobe 3.1.0

Kayobe 3.1.0 was released on 22nd February 2018 and is based on the Pike release of OpenStack.

Features

- Adds `--interface-limit` and `--interface-description-limit` arguments to the `kayobe physical network configure` command. These arguments allow configuration to be limited to a subset of switch interfaces.
- Adds a `--display` argument to `kayobe physical network configure` command. This will output the candidate switch configuration without applying it.
- Adds support for configuration of custom fluentd filters, and additional config file templates for heat, ironic, keystone, magnum, murano, sahara, and swift in `$KAYOBE_CONFIG_PATH/kolla/config/<component>/`.
- Adds support for specifying a local Yum mirror for package installation.
- Adds the command `kayobe network connectivity check` which can be used to verify network connectivity in the cloud hosts.
- Adds a variable `kolla_nova_compute_ironic_host` which may be used to set which hosts run the nova compute service for ironic. This may be used to avoid the experimental HA nova compute service for ironic, by specifying a single host.
- Adds support for deployment of virtualised compute hosts. These hosts should be added to the `[compute]` group.
- Adds support for multiple external networks. `external_net_names` should be a list of names of networks.
- Adds commands for management of baremetal compute nodes - `kayobe baremetal compute inspect`, `kayobe baremetal compute manage`, and `kayobe baremetal compute provide`.
- Adds support for installation and use of a python virtual environment for remote execution of ansible modules, providing isolation from the system's python packages. This is enabled by setting a host variable, `ansible_python_interpreter`, to the path to a python interpreter in a virtualenv, noting that Jinja2 templating is not supported for this variable.

- Adds support for configuration of a python virtual environment for remote execution of ansible modules in kolla-ansible, providing isolation from the system's python packages. This is enabled by setting the variable `kolla_ansible_target_venv` to a path to the virtualenv. The default for this variable is `{{ virtualenv_path }}/kolla-ansible`.
- Adds tags to plays to support more fine grained configuration using the `--tags` argument.
- Adds support for deployment of storage hosts. These hosts should be added to the `[storage]` group.
- Adds support for the tagging of ceph disks.
- Adds support for post-deployment configuration of Grafana data sources and dashboards.

Upgrade Notes

- Modifies the default value for `inspector_manage_firewall` from `False` to `True`. Management of the firewall by ironic inspector is important to ensure that DHCP offers are not made to nodes during provisioning by inspector's DHCP server.
- Disables swift by default. The default value of `kolla_enable_swift` is now `no`.
- The default list of neutron ML2 mechanism drivers (`kolla_neutron_ml2_mechanism_drivers`) has been removed in favour of using the defaults provided by kolla-ansible. Users relying on the default list of `openvswitch` and `genericswitch` should set the value explicitly.
- Adds a variable `config_path`, used to set the base path to configuration on remote hosts. The default value is `/opt/kayobe/etc`.
- Modifies the variable used to configure the kolla build configuration path from `kolla_config_path` to `kolla_build_config_path`. This provides a cleaner separation of kolla and kolla-ansible configuration options. The default value is `{{ config_path }}/kolla`.
- Adds a group `container-image-builders`, which defaults to containing the seed. Hosts in this group will build container images. Previously, container images for the seed were built on the seed, and container images for the overcloud were built on the controllers. The new design is intended to encourage a build, push, pull workflow.
- It is now possible to configure kayobe to use a virtual environment for remote execution of ansible modules. If this is required, the following commands should be run in order to ensure that the virtual environments exist on the remote hosts:

```
(kayobe) $ kayobe seed hypervisor host upgrade
(kayobe) $ kayobe seed host upgrade
(kayobe) $ kayobe overcloud host upgrade
```

- The default behaviour is now to configure kolla-ansible to use a virtual environment for remote execution of ansible modules. In order to ensure the virtual environment exists on the remote hosts, run the following commands:

```
(kayobe) $ kayobe seed hypervisor host upgrade
(kayobe) $ kayobe seed host upgrade
(kayobe) $ kayobe overcloud host upgrade
```

  The previous behaviour of installing python dependencies directly to the host can be used by setting `kolla_ansible_target_venv` to `None`.

- Adds a workaround for an issue with CentOS cloud images 7.2 (1511) onwards, which have a bogus name server entry in /etc/resolv.conf, 10.0.2.3. Cloud-init only appends name server entries to this file, and will not remove this bogus entry. Typically this leads to a delay of around 30 seconds when connecting via SSH, due to a timeout

in NSS. The workaround employed here is to remove this bogus entry from the image using virt-customize, if it exists. See https://bugs.centos.org/view.php?id=14369.

- Adds a group `storage`, which used for deploy node with cinder-volume, LVM or ceph-osd. If you want to add these services to compute or control group, you need to override `kolla_overcloud_inventory_storage_groups`.

Kayobe 3.0.0

Kayobe 3.0.0 was released on 20th September 2017.

Features

- Adds support for the OpenStack Pike release.
- Adds support for saving overcloud service configuration to the ansible control host.
- Adds support for generating overcloud service configuration, without applying it to the running system.

Upgrade Notes

See the upgrade notes for the pike release of the OpenStack services in use.

Kayobe 2.0.0

Kayobe 2.0.0 was released on 15th September 2017.

Features

- Adds support for configuration of networks for out-of-band management for the overcloud and control plane hosts via the `oob_oc_net_name` and `oob_wl_net_name` variables respectively.
- Adds support for configuration of a *seed hypervisor* host. This host runs the *seed VM*. Currently, configuration of host networking, NTP, and libvirt storage pools and networks is supported.
- Adds a `base_path` variable to simplify configuration of paths. This is used to set the default value of `image_cache_path` and `source_checkout_path`. The default value of the base path may be set by the `$KAYOBE_BASE_PATH` environment variable.
- Adds a `virtualenv_path` variable to configure the path on which to create virtual environments.
- Uses the CentOS 7 cloud image for the seed VM by default.
- Adds a command to deprovision the seed VM, `kayobe seed vm deprovision`.
- Adds support for configuration of Juniper switches.
- Adds support for bonded (LAG) host network interfaces.
- Adds support for the overlay docker storage driver on the seed and overcloud hosts.
- Improves the Vagrant development environment, and provides configuration for a single controller with a single network.

- Adds support for building customised Ironic Python Agent (IPA) deployment images using Diskimage Builder (DIB). These can be built using the commands `kayobe seed deployment image build` and `kayobe overcloud deployment image build`.
- Adds a command to save overcloud introspection data, `kayobe overcloud introspection data save`.
- Separates the external network into external and public networks. The public network carries public API traffic, and is configured via `public_net_name`.
- Adds a `network` group, with networking and load balancing services moved to it. The group is a subgroup of the `controllers` group by default.
- Decomposes the overcloud inventory into top level, components, and services. This allows a deployer to customise their inventory at various levels, by providing a custom inventory template for one or more sections of the inventory.
- Adds support for configuration of sysctl parameters on the seed, seed hypervisor and overcloud hosts.
- Adds an **inspection-store** container for storage of workload hardware inspection data in environments without Swift.
- Adds configuration of gatewys in provisioning and inspection networks.
- Adds support for free-form configuration of Glance.
- Adds support for Ubuntu control hosts.
- Adds support for passing through host variables from kayobe to kolla-ansible. By default `ansible_host`, `ansible_port`, and `ansible_ssh_private_key_file`.

Upgrade Notes

- It is no longer necessary to set the `seed_vm_interfaces` variable, as the seed VM's network interfaces are now determined by the standard `seed_network_interfaces` variable.
- If using a CentOS 7 cloud image for the seed VM, it is no longer necessary to set the `seed_vm_root_image` variable.
- The default value of `kolla_enable_haproxy` has been changed to `True`.
- If using a custom inventory, a `network` group should be added to it. If the control hosts are providing networking services, then the `network` group should be a subgroup of the `controllers` group.
- The `overcloud_groups` variable is now determined more intelligently, and it is generally no longer necessary to set it manually.
- The provisioning network is now used to access the TFTP server during workload hardware inspection.
- A default gateway may be advertised to compute nodes during workload inspection, allowing access to an ironic inspector API on the internal API network.

Kayobe 1.1.0

Kayobe 1.1.0 was released on 17th July 2017.

Features

- Support static routes on control plane networks

- Improve documentation
- Initial support for in-development Pike release
- Upgrade kayobe control host & control plane
- Support overcloud service destroy command
- Support fluentd custom output configuration

Kayobe 1.0.0

1.0.0 is the first 'official' release of the Kayobe OpenStack deployment tool. It was released on 29th June 2017.

Features

This release includes the following features:

- Heavily automated using Ansible
- `kayobe` Command Line Interface (CLI) for cloud operators
- Deployment of a seed VM used to manage the OpenStack control plane
- Configuration of physical network infrastructure
- Discovery, introspection and provisioning of control plane hardware using OpenStack bifrost
- Deployment of an OpenStack control plane using OpenStack kolla-ansible
- Discovery, introspection and provisioning of bare metal compute hosts using OpenStack ironic and ironic inspector
- Containerised workloads on bare metal using OpenStack magnum
- Big data on bare metal using OpenStack sahara