Fuel for OpenStack

version 3.0

July 22, 2013

Contents

Fuel for OpenStack: User Guide

Table of contents

Preface

OpenStack is an extensible, versatile, and flexible cloud management platform. By exposing its portfolio of cloud infrastructure services – compute, storage, networking and other core resources — through ReST APIs, OpenStack enables a wide range of control over these services, both from the perspective of an integrated Infrastructure as a Service (IaaS) controlled by applications, as well as automated manipulation of the infrastructure itself.

This architectural flexibility doesn’t set itself up magically. It asks you, the user and cloud administrator, to organize and manage an extensive array of configuration options. Consequently, getting the most out of your OpenStack cloud over time – in terms of flexibility, scalability, and manageability – requires a thoughtful combination of complex configuration choices. This can be very time consuming and requires a significant amount of studious documentation to comprehend.

Mirantis Fuel for OpenStack was created to eliminate exactly these problems. This step-by-step guide takes you through this process of:

- Configuring OpenStack and its supporting components into a robust cloud architecture
- Deploying that architecture through an effective, well-integrated automation package that sets up and maintains the components and their configurations
- Providing access to a well-integrated, up-to-date set of components known to work together

Introduction

This document explains how to use Fuel to easily create and maintain an OpenStack cloud infrastructure.

Fuel can be used to create virtually any OpenStack configuration. To make things easier, the installation includes several pre-defined architectures. For the sake of simplicity, this guide emphasises a single, common reference architecture; the multi-node, high-availability configuration. We begin with an explanation of this architecture, then move on to the details of creating the configuration in a test environment using VirtualBox. Finally, we give you the information you need to know to create this and other OpenStack architectures in a production environment.

This document assumes that you are familiar with general Linux commands and administration concepts, as well as general networking concepts. You should have some familiarity with grid or virtualization systems such as Amazon Web Services or VMware, as well as OpenStack itself, but you don't need to be an expert.

The Fuel User's Guide is organized as follows:

- Section 1, *Introduction* (this section), gives you an overview of Fuel and gives you a general idea of how it works.
- Section 2, *Reference Architecture*, provides a general look at the components that make up OpenStack, and describes the reference architecture to be instantiated in Section 3.
- Section 3.1, *Create a multi-node OpenStack cluster using Fuel Web*, takes you step-by-step through the process of creating a high-availability OpenStack cluster using Fuel Web.
- Section 3.2, *Create a multi-node OpenStack cluster using Fuel*, takes you step-by-step through the more advanced process of creating a high-availability OpenStack cluster using the standard Fuel tools.

- Section 4, *Production Considerations*, looks at the real-world questions and problems involved in creating an OpenStack cluster for production use. We discuss issues such as network layout and hardware requirements, and provide tips and tricks for creating a cluster of up to 100 nodes.
- With the 3.1 release of Fuel, Fuel Web has been integrated. We encourage all users to use the Fuel Web process for installation and configuration. However, the standard Fuel installation process is still available for those of you who prefer a more detailed approach to deployment. Even with a utility as powerful as Fuel, creating an OpenStack cluster can be complex, and Section 5, *Frequently Asked Questions*, covers many of the issues that tend to arise during that process.
- Finally, the User's Guide assumes that you are taking advantage of certain shortcuts, such as using a pre-built Puppet master; if you prefer not to go that route, Appendix A, *Creating the Puppet master*, will give you details regarding the configuration of a Puppet Master.

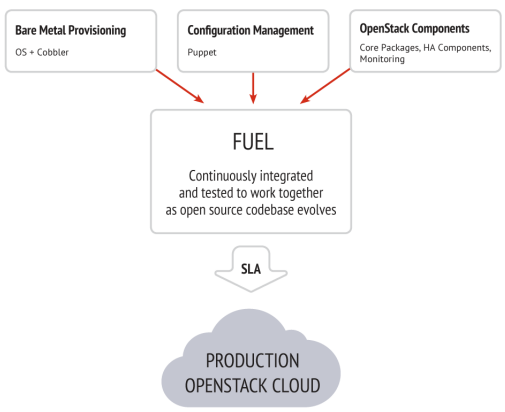
Lets start off by taking a look at Fuel itself. We'll start by explaining what it is and how it works, and then move to the process of installation itself.

What is Fuel?

Fuel is a ready-to-install collection of the packages and scripts you need to create a robust, configurable, vendor-independant OpenStack cloud in your own environment.

A single OpenStack cloud consists of packages from many different open source projects, each with its own requirements, installation procedures, and configuration management. Fuel brings all of these projects together into a single open source distribution, with components that have been tested and are guaranteed to work together, and all wrapped up using scripts to help you work through a single installation.

Simply put, Fuel is a way for you to easily configure and install an OpenStack-based infrastructure in your own environment.

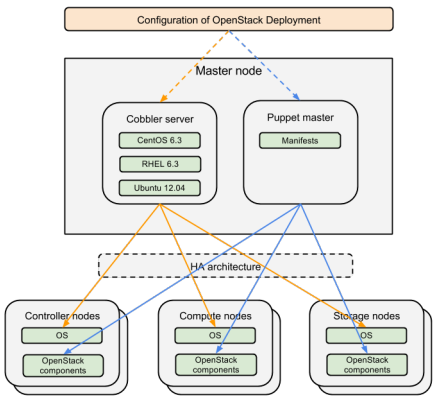


How Fuel Works

Fuel works on a simple premise. Rather than installing each of the myriad components that make up OpenStack directly, you instead use a configuration management system like Puppet to create scripts that can provide a configurable, reproducible, sharable installation process.

In practice, that means that the process of using Fuel Library looks like this:

1. First, use Fuel's automation tools and instructions to set up a master node with Puppet Master and Cobbler. This process only needs to be completed once per installation.
2. Next, use Fuel's snippets, kickstart files, and preseed files for Cobbler to boot the appropriate servers from bare metal and automatically install the appropriate operating systems. These virtual or physical servers boot up already prepared to call on the Puppet Master to receive their respective OpenStack components.
3. Finally, to complete the basic OpenStack install, use Fuel's puppet manifests to install OpenStack on the newly created servers. These manifests are completely customizable, enabling you to start with one of the included OpenStack architectures and adapt to your own situation as necessary.



Fuel comes with several pre-defined deployment configurations, some of which include additional options from which you can choose.

As of the 3.1 release of Fuel for OpenStack, Fuel Web is included as part of the package. Fuel Web is a simplified way to deploy production-grade OpenStack clouds. Fuel Web provides a streamlined, graphical console experience using the underlying scripts from Fuel Library, including proven deployment configurations and a well-organized workflow for deploying and managing OpenStack environments.

Fuel Web integrates all of the components of Fuel Library into a unified, web-based graphical user interface that walks administrators through the process of installing and configuring a fully functional OpenStack environment.

Deployment Configurations Provided By Fuel

One of the advantages of Fuel is that it comes with a number of pre-built deployment configurations that you can use to quickly build your own OpenStack cloud infrastructure. These are well-specified configurations of OpenStack and its constituent components are tailored to one or more common cloud use cases. Fuel provides the ability to create the following cluster types without requiring extensive customization:

Single node: Perfect for getting a feel for how OpenStack works, the Single-node installation is the simplest way to get OpenStack up and running. The Single-node installation provides an easy way to install an entire OpenStack cluster on a single physical server system or in a virtual machine environment.

Multi-node (non-HA): The Multi-node (non-HA) installation enables you to try out additional OpenStack services like Cinder, Neutron (formerly Quantum), and Swift without requiring the degree of increased hardware involved in ensuring high availability. In addition to the ability to independently specify which services to activate, you also have the following options:

> **Compact Swift**: When you choose this option, Swift will be installed on your controllers, reducing your hardware requirements by eliminating the need for additional Swift servers.
>
> **Standalone Swift**: This option enables you to install independant Swift nodes, so that you can separate their operation from your controller nodes.

Multi-node (HA): When you're ready to begin your move to production, the Multi-node (HA) configuration is a straightforward way to create an OpenStack cluster that provides high availability. With three controller nodes and the ability to individually specify services such as Cinder, Neutron, and Swift, Fuel provides the following variations of the Multi-node (HA) configuration:

> **Compact Swift**: When you choose this option, Swift will be installed on your controllers, reducing your hardware requirements by eliminating the need for additional Swift servers while still addressing high availability requirements.
>
> **Standalone Swift**: This option enables you to install independant Swift nodes, so that you can separate their operation from your controller nodes.
>
> **Compact Neutron**: If you don't need the flexibility of a separate Neutron node, Fuel provides the option to combine your Neutron node with one of your controllers.

In addition to these configurations, Fuel is designed to be completely customizable. For assistance on deeper customization options based on the included configurations you can contact Mirantis for further assistance.

Supported Software

Fuel has been tested and is guaranteed to work with the following software components:

- **Operating Systems**
  - CentOS 6.4 (x86_64 architecture only)
  - RHEL 6.4 (x86_64 architecture only)
- **Puppet (IT automation tool)**
  - 2.7.19

- **MCollective**
    - 2.2.4
- **Cobbler (bare-metal provisioning tool)**
    - 2.2.3
- **OpenStack**
    - Grizzly release 2013.1
- **Hypervisor**
    - KVM
- **Open vSwitch**
    - 1.10.0
- **HA Proxy**
    - 1.4.19
- **Galera**
    - 23.2.2
- **RabbitMQ**
    - 2.8.7
- **Pacemaker**
    - 1.1.8
- **Corosync**
    - 1.4.3
- **Keepalived**
    - 1.2.4
- **Nagios**
    - 3.4.4

Download Fuel

The first step in installing Fuel is to download the version appropriate for your environment.

Fuel is available for Essex, Folsom and Grizzly OpenStack installations, and will be available for Havana shortly after Havana's release.

To make your installation easier, we also offer a pre-built ISO for installing the master node with Puppet Master and Cobbler. You can mount this ISO on a physical machine or in VirtualBox in order to easily create your master node. (Instructions for performing this step without the ISO are given in *Appendix A*.)

The master node ISO, along with other Fuel releases, is available in the Downloads section of the Fuel portal.

Release Notes

v3.1-grizzly

New Features in Fuel and Fuel Web 3.1

PLACEHOLDER

v3.0-grizzly

New Features in Fuel and Fuel Web 3.0

- Support for OpenStack Grizzly
- Support for CentOS 6.4
- Deployment improvements
    - Deployment of Cinder as a standalone node
    - Users may now choose where to store Cinder volumes

    - User defined disk space allocation for the base OS, Cinder and Virtual Machines
    - Ability to add new compute nodes without redeployment of the whole environment
    - Swift installation occurs in a single pass instead of multiple passes
- Network configuration enhancements
    - Support for NIC bonding
    - Ability to map logical networks to physical interfaces
    - Improved firewall module

Support for OpenStack Grizzly

OpenStack Grizzly is the seventh release of the open source software for building public, private, and hybrid clouds. Fuel now supports deploying the Grizzly version of OpenStack in a variety of configurations including High Availability (HA). For a list of known limitations, please refer to the Known Issues section below.

Support for CentOS 6.4

CentOS 6.4 is now the base operating system for the Fuel master node, as well as the deployed slave nodes.

Deployment Improvements

- Deployment of Cinder as a standalone node / User choice

  Previously, Cinder could only be deployed onto a compute node. Now, you may choose to deploy Cinder as a standalone node separate from a compute node. Both options – either deployed with a compute node or standalone – are available.

v2.1-folsom

- Features
    - Support deploying Neutron (formerly Quantum) on controller nodes, as well as on a dedicated networking node
    - Active/Standby HA for Neutron with Pacemaker when Neutron is deployed on controller nodes
    - Logging: an option to send OpenStack logs to local and remote locations through syslog
    - Monitoring: deployment of Nagios, health checks for infrastructure components (OpenStack API, MySQL, RabbitMQ)
    - Installation of Puppet Master & Cobbler Server node from ISO
    - Deployment orchestration based on mcollective eliminates the need to run Puppet manually on each node
    - Recommended master node setup for mid-scale deployments, tested up to 100 nodes
- Improvements
    - Support for multiple environments from a single Fuel master node
    - RabbitMQ service moved behind HAProxy to make controller failures transparent to the clients
    - Updated RabbitMQ to 2.8.7 to improve handling on expired HA queues under Ubuntu
    - Changed RabbitMQ init script to automatically reassemble RabbitMQ cluster after failures
    - Configurable HTTP vs. HTTPS for Horizon
    - Changed mirror type option to either be 'default' (installation from the internet) or 'custom' (installation from a local mirror containing packages)
    - Option to allow cinder-volume deployment on controller nodes as well as compute nodes

v2.0-folsom

- Features:
    - Puppet manifests for deploying OpenStack Folsom in HA mode
    - Active/Active HA architecture for Folsom, based on RabbitMQ / MySQL Galera / HAProxy / keepalived
    - Added support for Ubuntu 12.04 in addition to CentOS 6.3 and RHEL 6.3 (includes bare metal provisioning, Puppet manifests, and OpenStack packages)
    - Supports deploying Folsom with Neutron (formerly Quantum)/OVS
    - Supports deploying Folsom with Cinder
    - Supports Puppet 2.7 and 3.0

v1.0-essex

- Features:
  - Puppet manifests for deploying OpenStack Essex in HA mode
  - Active/Active HA architecture for Essex, based on RabbitMQ / MySQL Galera / HAProxy / keepalived
  - Cobbler-based bare-metal provisioning for CentOS 6.3 and RHEL 6.3
  - Access to the mirror with OpenStack packages
  - Configuration templates for different OpenStack cluster setups
  - User Guide

Reference Architecture

Overview

Before you install any hardware or software, you must know what it is you're trying to achieve. This section looks at the basic components of an OpenStack infrastructure and organizes them into one of the more common reference architectures. You'll then use that architecture as a basis for installing OpenStack in the next section.

As you know, OpenStack provides the following basic services:

- **Compute**: Compute servers are the workhorses of your installation; they're the servers on which your users' virtual machines are created. *Nova-scheduler* controls the life-cycle of these VMs.
- **Networking**: Because an OpenStack cluster (virtually) always includes multiple servers, the ability for them to communicate with each other and with the outside world is crucial. Networking was originally handled by the *Nova-network* service, but it has given way to the newer Neutron (formerly Quantum) networking service. Authentication and authorization for these transactions are handled by *Keystone*.
- **Storage**: OpenStack provides for two different types of storage: block storage and object storage. Block storage is traditional data storage, with small, fixed-size blocks that are mapped to locations on storage media. At its simplest level, OpenStack provides block storage using *nova-volume*, but it is common to use *Cinder*.

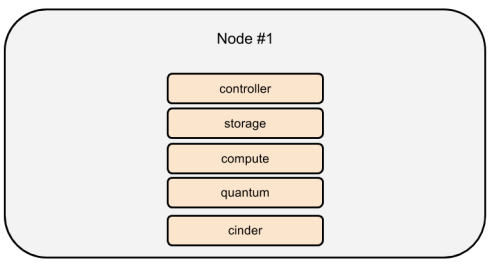
  Object storage, on the other hand, consists of single variable-size objects that are described by system-level metadata, and you can access this capability using *Swift*.

OpenStack storage is used for your users' objects, but it is also used for storing the images used to create new VMs. This capability is handled by *Glance*.

These services can be combined in many different ways. Out of the box, Fuel supports the following deployment configurations:

Single node deployment

In a production environment, you will never have a single-node deployment of OpenStack, partly because it forces you to make a number of compromises as to the number and types of services that you can deploy. It is, however, extremely useful if you just want to see how OpenStack works from a user's point of view. In this case, all of the essential services run out of a single server:

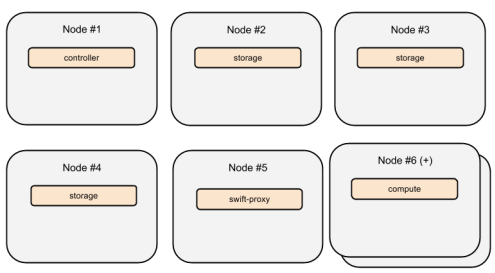


Multi-node (non-HA) deployment (compact Swift)

More commonly, your OpenStack installation will consist of multiple servers. Exactly how many is up to you, of course, but the main idea is that your controller(s) are separate from your compute servers, on which your users' VMs will actually run. One arrangement that will enable you to achieve this separation while still keeping your hardware investment relatively modest is to house your storage on your controller nodes.

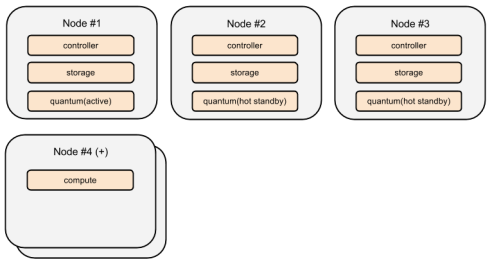
Multi-node (non-HA) deployment (standalone Swift)

A more common arrangement is to provide separate servers for storage. This has the advantage of reducing the number of controllers you must provide; because Swift runs on its own servers, you can reduce the number of controllers from three (or five, for a full Swift implementation) to one, if desired:



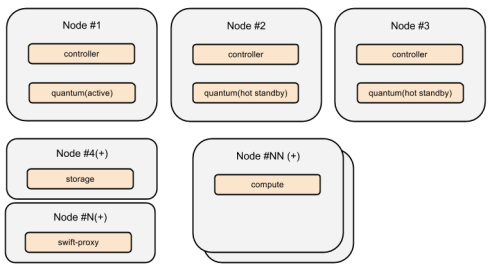
Multi-node (HA) deployment (Compact)

Production environments typically require high availability, which involves several architectural requirements. Specifically, you will need at least three controllers, and certain components will be deployed in multiple locations to prevent single points of failure. That's not to say, however, that you can't reduce hardware requirements by combining your storage, network, and controller nodes:



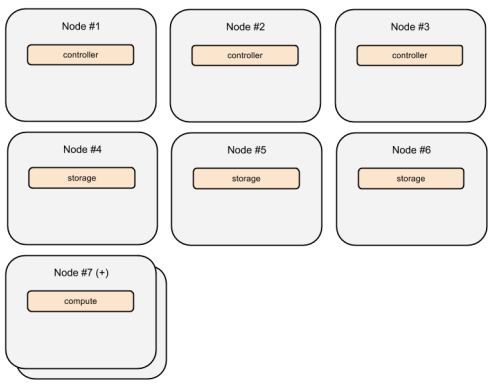
Multi-node (HA) deployment (Compact Neutron)

Another way you can add functionality to your cluster without increasing hardware requirements is to install Quantum on your controller nodes. This architecture still provides high availability, but avoids the need for a separate Neutron node:



Multi-node (HA) deployment (Standalone)

For larger production deployments, its more common to provide dedicated hardware for storage and networking. This architecture still gives you the advantages of high availability, but this clean separation makes your cluster more maintainable by separating storage, networking, and controller functionality:

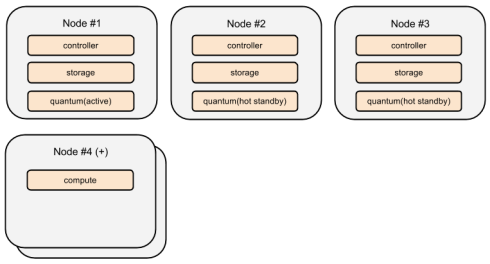


Where Fuel really shines is in the creation of more complex architectures, so in this document you'll learn how to use Fuel to easily create a multi-node HA OpenStack cluster. To reduce the amount of hardware you'll need to follow the installation in section 3, however, the guide focuses on the Multi-node HA Compact architecture.

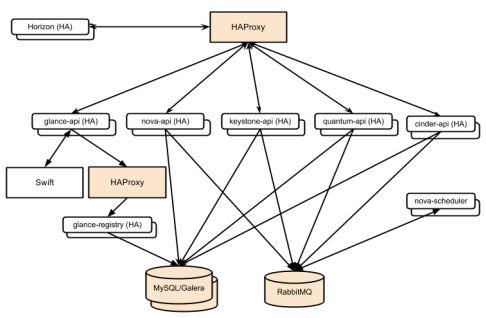
Lets take a closer look at the details of this deployment configuration.

A closer look at the Multi-node (HA) Compact deployment

In this section, you'll learn more about the Multi-node (HA) Compact deployment configuration and how it achieves high availability in preparation for installing this cluster in section 3. As you may recall, this configuration looks something like this:



OpenStack services are interconnected by RESTful HTTP-based APIs and AMQP-based RPC messages. So redundancy for stateless OpenStack API services is implemented through the combination of Virtual IP (VIP) management using keepalived and load balancing using HAProxy. Stateful OpenStack components, such as the state database and messaging server, rely on their respective active/active modes for high availability. For example, RabbitMQ uses built-in clustering capabilities, while the database uses MySQL/Galera replication.



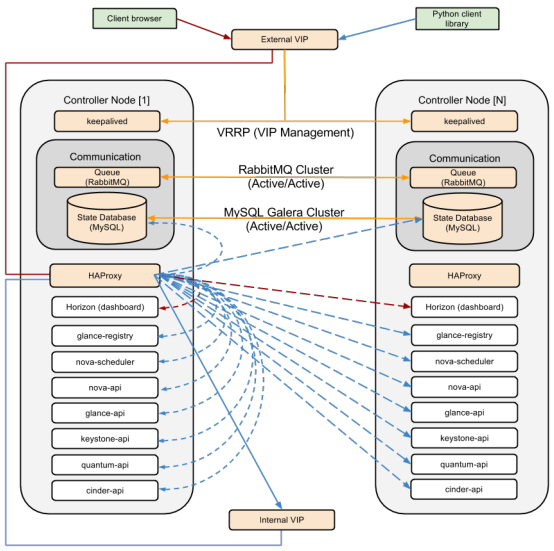
Lets take a closer look at what an OpenStack deployment looks like, and what it will take to achieve high availability for an OpenStack deployment.

Logical Setup

An OpenStack HA cluster involves, at a minimum, three types of nodes: controller nodes, compute nodes, and storage nodes.

Controller Nodes

The first order of business in achieving high availability (HA) is redundancy, so the first step is to provide multiple controller nodes. You must keep in mind, however, that the database uses Galera to achieve HA, and Galera is a quorum-based system. That means that you must provide at least 3 controller nodes.



Every OpenStack controller runs keepalived, which manages a single Virtual IP (VIP) for all controller nodes, and HAProxy, which manages HTTP and TCP load balancing of requests going to OpenStack API services, RabbitMQ, and MySQL.

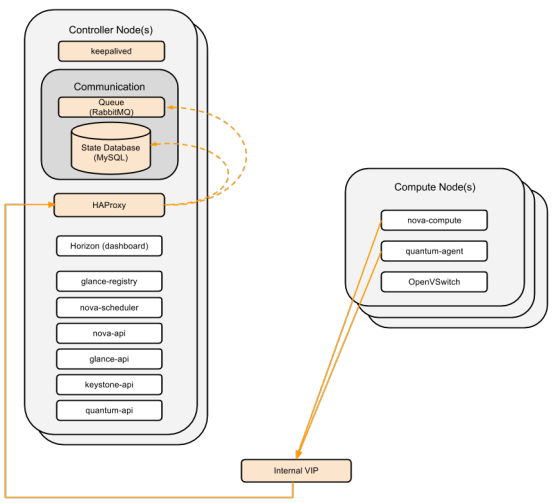
When an end user accesses the OpenStack cloud using Horizon or makes a request to the REST API for services such as nova-api, glance-api, keystone-api, quantum-api, nova-scheduler, MySQL or RabbitMQ, the request goes to the live controller node currently holding the VIP, and the connection gets terminated by HAProxy. When the next request comes in, HAProxy handles it, and may send it to the original controller or another in the cluster, depending on load conditions.

Each of the services housed on the controller nodes has its own mechanism for achieving HA:

- nova-api, glance-api, keystone-api, quantum-api and nova-scheduler are stateless services that do not require any special attention besides load balancing.
- Horizon, as a typical web application, requires sticky sessions to be enabled at the load balancer.
- RabbitMQ provides active/active high availability using mirrored queues.
- MySQL high availability is achieved through Galera active/active multi-master deployment.

Compute Nodes

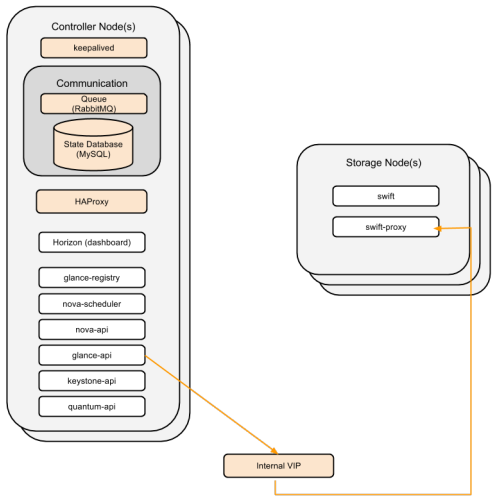
OpenStack compute nodes are, in many ways, the foundation of your cluster; they are the servers on which your users will create their Virtual Machines (VMs) and host their applications. Compute nodes need to talk to controller nodes and reach out to essential services such as RabbitMQ and MySQL. They use the same approach that provides redundancy to the end-users of Horizon and REST APIs, reaching out to controller nodes using the VIP and going through HAProxy.



Storage Nodes

In this OpenStack cluster reference architecture, shared storage acts as a backend for Glance, so that multiple Glance instances running on controller nodes can store images and retrieve images from it. To achieve this, you

are going to deploy Swift. This enables you to use it not only for storing VM images, but also for any other objects such as user files.

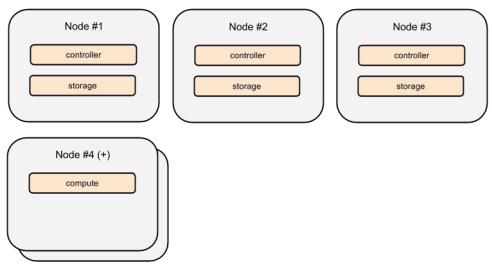


Cluster Sizing

This reference architecture is well suited for production-grade OpenStack deployments on a medium and large scale when you can afford allocating several servers for your OpenStack controller nodes in order to build a fully redundant and highly available environment.

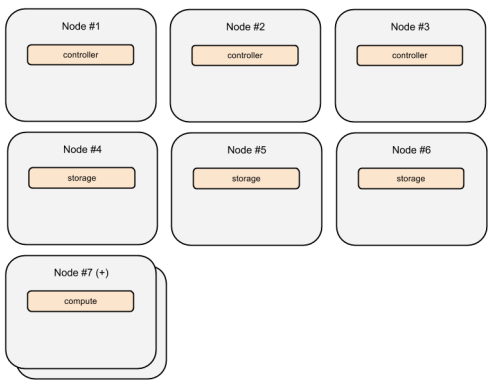
The absolute minimum requirement for a highly-available OpenStack deployment is to allocate 4 nodes:

- 3 controller nodes, combined with storage
- 1 compute node



If you want to run storage separately from the controllers, you can do that as well by raising the bar to 7 nodes:

- 3 controller nodes
- 3 storage nodes
- 1 compute node



Of course, you are free to choose how to deploy OpenStack based on the amount of available hardware and on your goals (such as whether you want a compute-oriented or storage-oriented cluster).

For a typical OpenStack compute deployment, you can use this table as high-level guidance to determine the number of controllers, compute, and storage nodes you should have:

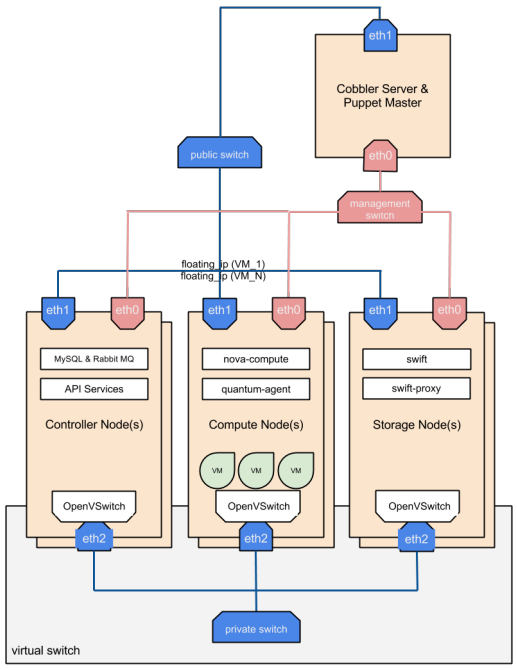
# of Machines	Controllers	Compute	Storage
4-10	3	1-7	on controllers
11-40	3	5-34	3 (separate)
41-100	4	31-90	6 (separate)
>100	5	>86	9 (separate)

Network Architecture

The current architecture assumes the presence of 3 NIC cards in hardware, but can be customized two or more NICs. Most servers come with at least two network interfaces. In this case, let's consider a typical example of three NIC cards. They're utilized as follows:

- **eth0**: the internal management network, used for communication with Puppet & Cobbler
- **eth1**: the public network, and floating IPs assigned to VMs
- **eth2**: the private network, for communication between OpenStack VMs, and the bridge interface (VLANs)

In the multi-host networking mode, you can choose between the FlatDHCPManager and VlanManager network managers in OpenStack. The figure below illustrates the relevant nodes and networks.



Lets take a closer look at each network and how its used within the cluster.

Public Network

This network allows inbound connections to VMs from the outside world (allowing users to connect to VMs from the Internet). It also allows outbound connections from VMs to the outside world.

For security reasons, the public network is usually isolated from the private network and internal (management) network. Typically, it's a single C class network from your globally routed or private network range.

To enable Internet access to VMs, the public network provides the address space for the floating IPs assigned to individual VM instances by the project administrator. Nova-network or Neutron (formerly Quantum) services can then configure this address on the public network interface of the Network controller node. If the cluster uses nova-network, nova- network uses iptables to create a Destination NAT from this address to the fixed IP of the corresponding VM instance through the appropriate virtual bridge interface on the Network controller node.

In the other direction, the public network provides connectivity to the globally routed address space for VMs. The IP address from the public network that has been assigned to a compute node is used as the source for the Source NAT performed for traffic going from VM instances on the compute node to Internet.

The public network also provides VIPs for Endpoint nodes, which are used to connect to OpenStack services APIs.

Internal (Management) Network

The internal network connects all OpenStack nodes in the cluster. All components of an OpenStack cluster communicate with each other using this network. This network must be isolated from both the private and public networks for security reasons.

The internal network can also be used for serving iSCSI protocol exchanges between Compute and Storage nodes.

This network usually is a single C class network from your private, non-globally routed IP address range.

Private Network

The private network facilitates communication between each tenant's VMs. Private network address spaces are part of the enterprise network address space. Fixed IPs of virtual instances are directly accessible from the rest of Enterprise network.

The private network can be segmented into separate isolated VLANs, which are managed by nova-network or Neutron (formerly Quantum) services.

Technical Considerations

Before performing any installations, you'll need to make a number of decisions about which services to deploy, but from a general architectural perspective, it's important to think about how you want to handle both networking and block storage.

Neutron vs. nova-network

Neutron (formerly Quantum) is a service which provides networking-as-a-service functionality in OpenStack. It has a rich tenant-facing API for defining network connectivity and addressing in the cloud, and gives operators the ability to leverage different networking technologies to power their cloud networking.

There are various deployment use cases for Neutron. Fuel supports the most common of them, called Provider Router with Private Networks. It provides each tenant with one or more private networks, which can communicate with the outside world via a Neutron router.

Neutron is not, however, required in order to run an OpenStack cluster; if you don't need (or want) this added functionality, it's perfectly acceptable to continue using nova-network.

In order to deploy Neutron, you need to enable it in the Fuel configuration. Fuel will then set up an additional node in the OpenStack installation to act as an L3 router, or, depending on the configuration options you've chosen, install Neutron on the controllers.

Cinder vs. nova-volume

Cinder is a persistent storage management service, also known as block-storage-as-a-service. It was created to replace nova-volume, and provides persistent storage for VMs.

If you decide use Cinder for persistent storage, you will need to both enable Cinder and create the block devices on which it will store data. You will then provide information about those blocks devices during the Fuel install. (You'll see an example how to do this in section 3.)

Cinder block devices can be:

- created by Cobbler during the initial node installation, or
- attached manually (e.g. as additional virtual disks if you are using VirtualBox, or as additional physical RAID, SAN volumes)

Swift (object storage) notes

FUEL currently supports several ways to deploy the swift service:

- Swift absent

  By default, Glance uses the filesystem backend to store virtual machine images. In this case, you can use any of shared file systems Glance supports.

- Swift compact

  In this mode the role of swift-storage and swift-proxy are combined with a nova-controller. Use it only for testing in order to save nodes; it's not suitable for production.

- Swift standalone

  In this case the Proxy service and Storage (account/container/object) services reside on separate nodes, with one proxy node and a minimum of three storage nodes. (For a production cluster, a minimum of five nodes is recommended.)

Now let's look at performing an actual OpenStack installation using Fuel.

Create a multi-node OpenStack cluster using Fuel Web

User Guide

Deveoper Guide

Create a multi-node OpenStack cluster using Fuel

In this section, you'll learn how to install OpenStack using Fuel and Fuel Web. In addition to getting a feel for the steps involved, you'll also gain valuable familiarity with some of the customization options. While Fuel provides different deployment configuration templates in the box, it is common for administrators to modify the architecture to meet enterprise requirements. Working hands on with Fuel for OpenStack will help you see how to move certain features around from the standard installation.

The first step, however, is to commit to a deployment template. A balanced, compact, and full-featured deployment is the Multi-node (HA) Compact deployment, so that's what we'll be using through the rest of this guide.

Production installations require a physical hardware infrastructure, but you can easily deploy a small simulation cloud on a single physical machine using VirtualBox. You can follow these instructions to install an OpenStack cloud into a test environment using VirtualBox, or to get a production-grade installation using physical hardware.

How installation works

While version 2.0 of Fuel provided the ability to simplify installation of OpenStack, versions 2.1 and above include orchestration capabilities that simplify deployment of OpenStack clusters. The deployment process follows this general procedure:

1. Design your architecture.
2. Install Fuel onto the fuel-pm machine.
3. Configure Fuel.
4. Create the basic configuration and load it into Cobbler.
5. PXE-boot the servers so Cobbler can install the operating system and prepare them for orchestration.
6. Use one of the templates included in Fuel and the configuration that populates Puppet's site.pp file.
7. Customize the site.pp file as needed.
8. Use the orchestrator to coordinate the installation of the appropriate OpenStack components on each node.

Start by designing your architecture, details about which you can find in the next section, *Before You Start*

Before You Start

Before you begin your installation, you will need to make a number of important decisions:

- **OpenStack features.** Your first decision is to decide which of the optional OpenStack features you will need. For example, you must decide whether you want to install Swift, whether you want Glance to use Swift for image storage, whether you want Cinder for block storage, and whether you want nova-network or Neutron (formerly Quantum) to handle your network connectivity. In our example installation we will be installing Glance with Swift support and Cinder for block storage. Also, due to the fact that it can be easily installed using orchestration, we will be using Neutron.
- **Deployment configuration.** Next, you need to decide whether your deployment requires high availability (HA). If you need HA for your deployment, you have a choice regarding the number of controllers you want to include. Following the recommendations in the previous section for a typical HA deployment configuration, we will use three OpenStack controllers.
- **Cobbler server and Puppet Master.** The heart of any Fuel install is the combination of Puppet Master and Cobbler used to create your resources. Although Cobbler and Puppet Master can be installed on separate machines, it is common practice to install both on a single machine for small to medium size clouds, and that's what we'll be doing in this example. By default, the Fuel ISO creates a single server with both services.
- **Domain name.** Puppet clients generate a Certificate Signing Request (CSR), which is then signed by the Puppet Master. The signed certificate can then be used to authenticate clients during provisioning. Certificate generation requires a fully qualified hostname, so you must choose a domain name to be used in your installation. Future versions of Fuel will enable you to choose this domain name on your own; by default, Fuel 3.1 uses `localdomain`.
- **Network addresses.** OpenStack requires a minimum of three networks. If you are deploying on physical hardware, two of them -- the public network and the internal, or management network -- must be routable in your networking infrastructure. The third network is used by the nodes for inter-node communications. Also, if you intend for your cluster to be accessible from the Internet, you'll want the public network to be on the proper network segment. For simplicity in this case, this example assumes an Internet router at 192.168.0.1. Additionally, a set of private network addresses should be selected for automatic assignment to guest VMs. (These are fixed IPs for the private network). In our case, we are allocating network addresses as follows:
    - Public network: 192.168.0.0/24
    - Internal network: 10.0.0.0/24
    - Private network: 10.0.1.0/24
- **Network interfaces.** All of those networks need to be assigned to the available NIC cards on the allocated machines. Additionally, if a fourth NIC is available, Cinder or block storage traffic can be separated and delegated to the fourth NIC. In our case, we're assigning networks as follows:
    - Public network: eth1
    - Internal network: eth0
    - Private network: eth2

Infrastructure Allocation and Installation

The next step is to make sure that you have all of the required hardware and software in place.

Software

You can download the latest release of the Fuel ISO from http://fuel.mirantis.com/your-downloads/.

Alternatively, if you can't use the pre-built ISO, Mirantis offers the Fuel Library as a tar.gz file downloadable from the Downloads section of the Fuel portal. Using this file requires a bit more effort, but will yield the same results as using the ISO.

Network setup

OpenStack requires a minimum of three distinct networks: internal (or management), public, and private. The simplest and best methodology to map NICs is to assign each network to a different physical interface. However, not all machines have three NICs, and OpenStack can be configured and deployed with only two physical NICs, combining the internal and public traffic onto a single NIC.

If you are building a simulation environment, you are not limited to the availability of physical NICs. Allocate three NICs to each VM in your OpenStack infrastructure, one each for the internal, public, and private networks.

Finally, we assign network ranges to the internal, public, and private networks, and IP addresses to fuel-pm, fuel-controllers, and fuel-compute nodes. For deployment to a physical infrastructure you must work with your IT department to determine which IPs to use, but for the purposes of this exercise we will assume the below network and IP assignments:

1. 10.0.0.0/24: management or internal network, for communication between Puppet master and Puppet clients, as well as PXE/TFTP/DHCP for Cobbler.
2. 192.168.0.0/24: public network, for the High Availability (HA) Virtual IP (VIP), as well as floating IPs assigned to OpenStack guest VMs
3. 10.0.1.0/24: private network, fixed IPs automatically assigned to guest VMs by OpenStack upon their creation

Next we need to allocate a static IP address from the internal network to eth0 for fuel-pm, and eth1 for the controller, compute, and, if necessary, quantum nodes. For High Availability (HA) we must choose and assign an IP address from the public network to HAProxy running on the controllers. You can configure network addresses/network mask according to your needs, but our instructions assume the following network settings on the interfaces:

1. eth0: internal management network, where each machine is allocated a static IP address from a the defined pool of available addresses:
    - 10.0.0.100 for Puppet Master
    - 10.0.0.101-10.0.0.103 for the controller nodes
    - 10.0.0.110-10.0.0.126 for the compute nodes
    - 10.0.0.10 internal Virtual IP for component access
    - 255.255.255.0 network mask
2. eth1: public network
    - 192.168.0.10 public Virtual IP for access to the Horizon GUI (OpenStack management interface)
3. eth2: for communication between OpenStack VMs without IP address with promiscuous mode enabled.

Physical installation infrastructure

The hardware necessary for an installation depends on the choices you have made above. This sample installation requires the following hardware:

- 1 server to host both the Puppet Master and Cobbler. The minimum configuration for this server is:
    - 32-bit or 64-bit architecture (64-bit recommended)
    - 1+ CPU or vCPU for up to 10 nodes (2 vCPU for up to 20 nodes, 4 vCPU for up to 100 nodes)
    - 1024+ MB of RAM for up to 10 nodes (4096+ MB for up to 20 nodes, 8192+ MB for up to 100 nodes)
    - 16+ GB of HDD for OS, and Linux distro storage
- 3 servers to act as OpenStack controllers (called fuel-controller-01, fuel-controller-02, and fuel-controller-03 for our sample deployment). The minimum configuration for a controller in Compact mode is:
    - 64-bit architecture
    - 1+ CPU (2 or more CPUs or vCPUs recommended)
    - 1024+ MB of RAM (2048+ MB recommended)
    - 400+ GB of HDD

- 1 server to act as the OpenStack compute node (called fuel-compute-01). The minimum configuration for a compute node with Cinder installed is:
    - 64-bit architecture
    - 2+ CPU, with Intel VTx or AMDV virtualization technology enabled
    - 2048+ MB of RAM
    - 1+ TB of HDD

If you choose to deploy Neutron (formerly Quantum) on a separate node, you will need an additional server with specifications comparable to the controller nodes.

Make sure your hardware is capable of PXE booting over the network from Cobbler. You'll also need each server's mac addresses.

For a list of certified hardware configurations, please contact the Mirantis Services team.

Virtual installation infrastructure

For a virtual installation, you need only a single machine. You can get by on 8GB of RAM, but 16GB or more is recommended.

To perform the installation, you need a way to create Virtual Machines. This guide assumes that you are using version 4.2.12 or later of VirtualBox, which you can download from the VirtualBox site. It is also required that you have the Extension Pack installed to enable features that are needed for a virtualized OpenStack test environment to work correctly.

You'll need to run VirtualBox on a stable host system. Mac OS 10.7.x, CentOS 6.3+, or Ubuntu 12.04 are preferred; results in other operating systems are unpredictable. It's also important to remember that Windows is incapable of running the installation scripts for Fuel so we cannot recommend Windows as a test platform.

Configuring VirtualBox

If you are on VirtualBox, please create the following host-only adapters and that they are configured correctly:

- VirtualBox -> File -> Preferences...
- Network -> Add HostOnly Adapter (vboxnet0)
    - IPv4 Address: 10.0.0.1
    - IPv4 Network Mask: 255.255.255.0
    - DHCP server: disabled
- Network -> Add HostOnly Adapter (vboxnet1)
    - IPv4 Address: 10.0.1.1
    - IPv4 Network Mask: 255.255.255.0
    - DHCP server: disabled
- Network -> Add HostOnly Adapter (vboxnet2)
    - IPv4 Address: 0.0.0.0
    - IPv4 Network Mask: 255.255.255.0
    - DHCP server: disabled

In this example, only the first two adapters will be used. If necessasry, though, you can choose to use the third adapter to handle your storage network traffic.

After creating these interfaces, reboot the host machine to make sure that DHCP isn't running in the background.

As stated before, installing on Windows isn't recommended, but if you're attempting to do so you will also need to set up the IP address & network mask under Control Panel > Network and Internet > Network and Sharing Center for the Virtual HostOnly Network adapter.

Creating fuel-pm

The process of creating a virtual machine to host Fuel in VirtualBox depends on whether your deployment is purely virtual or consists of a physical or virtual fuel-pm controlling physical hardware. If your deployment is purely virtual then Adapter 1 may be a Hostonly adapter attached to vboxnet0, but if your deployment infrastructure consists of a virtual fuel-pm controlling physical machines Adapter 1 must be a Bridged Adapter and connected to whatever network interface of the host machine is connected to your physical machines.

To create fuel-pm, start up VirtualBox and create a new machine as follows:

- Machine -> New...
    - Name: fuel-pm
    - Type: Linux
    - Version: Red Hat (64 Bit)
    - Memory: 2048 MB
    - Drive space: 16 GB HDD
- Machine -> Settings... -> Network
    - Adapter 1
        - **Physical network**
            - Enable Network Adapter
            - Attached to: Bridged Adapter
            - Name: The host machine's network with access to the network on which the physical machines reside
        - **VirtualBox installation**
            - Enable Network Adapter
            - Attached to: Hostonly Adapter
            - Name: vboxnet0
    - Adapter 2
        - Enable Network Adapter
        - Attached to: Bridged Adapter
        - Name: eth0 (or whichever physical network is attached to the Internet)
- Machine -> Storage
    - Attach the downloaded ISO as a drive

If you cannot or prefer not to install from the ISO, you can find instructions for installing from the Fuel Library in *Appendix A*.

Creating the OpenStack nodes

If you're using VirtualBox, you will need to create the necessary virtual machines for your OpenStack nodes. Follow these instructions to create machines named fuel-controller-01, fuel-controller-02, fuel- controller-03, and fuel-compute-01. Please, do NOT start these virtual machines until instructed.

As you create each network adapter, click Advanced to expose and record the corresponding mac address.

- Machine -> New...
    - Name: fuel-controller-01 (you will repeat these steps to create fuel-controller-02, fuel-controller-03, and fuel-compute-01)
    - Type: Linux
    - Version: Red Hat (64 Bit)
    - Memory: 2048MB
    - Drive space: 8GB
- Machine -> Settings -> System
    - Check Network in Boot sequence
- Machine -> Settings -> Storage
    - Controller: SATA
        - Click the Add icon at the bottom of the Storage Tree pane and choose Add Disk
        - Add a second VDI disk of 10GB for storage

- Machine -> Settings -> Network
    - Adapter 1
        - Enable Network Adapter
        - Attached to: Hostonly Adapter
        - Name: vboxnet0
    - Adapter 2
        - Enable Network Adapter
        - Attached to: Bridged Adapter
        - Name: eth0 (physical network attached to the Internet. You may also use a gateway if necessary.)
    - Adapter 3
        - Enable Network Adapter
        - Attached to: Hostonly Adapter
        - Name: vboxnet1
        - Advanced -> Promiscuous mode: Allow All

It is important that Adapter 1 is configured to load first as Cobbler will use vboxnet0 for PXE and VirtualBox boots from the LAN using the first available network adapter.

The additional drive volume will be used as storage space by Cinder and will be configured automatically by Fuel.

Installing & Configuring Fuel

Having planned your test or production deployment and have determined the resources you will be using, it's time to begin putting the pieces together. To do that, you'll need to create the Puppet master and Cobbler servers, which will actually provision and configure your OpenStack nodes.

Installing the Puppet Master is a one-time procedure for the entire infrastructure. Once done, Puppet Master will act as a single point of management for all of your servers, and you will never have to return to these installation steps again.

The deployment of the Puppet Master server, named fuel-pm in these instructions, varies slightly between the physical and simulation environments. In a physical infrastructure, fuel-pm must have a network presence on the same network as the physical machines in order to faciliate PXE booting. In a simulation environment fuel-pm only needs host-only virtual network connectivity.

At this point, you should have either a physical or virtual machine that can be booted from the Mirantis ISO, downloaded from http://fuel.mirantis.com/your-downloads/.

This ISO can be used to create fuel-pm on a physical or virtual machine based on CentOS 6.4. If for some reason you cannot use the ISO, follow the instructions in *Creating the Puppet master* to create your own fuel-pm, then skip ahead to *Configuring fuel-pm*.

Installing Fuel from the ISO

Start the new machine to install the ISO. The only real installation decision you will need to make is to specify the interface through which the installer can access the Internet. Select the eth1 interface you created earlier which should be configured for access to the public network.

Configuring fuel-pm from the ISO installation

Once fuel-pm finishes installing, you'll be presented with a basic menu. You can use this menu to set the basic information Fuel will need to configure your installation. You can customize these steps to suit your own need, of course, but here are the steps to take for the example installation:

1. Your admin node must be called `fuel-pm`, and your domain name must be `localdomain`.
2. To configure the management interface, choose 2.
    - The example specifies eth0 as the internal, or management interface, so select that interface.
    - The management network in the example is using static IP addresses, so specify NO for for using DHCP.
    - Enter the IP address of 10.0.0.100 for the Puppet Master with a netmask of 255.255.255.0.
    - Set the gateway and DNS servers if desired. In this example, we'll use the router at 192.168.0.1 as the gateway.

3. To configure the external interface that VMs will use to send traffic to and from the internet, choose 3. Set the interface to eth1. By default, this interface uses DHCP, which is what the example calls for.
4. To choose the start and end addresses to be used during PXE boot, choose 4. In the case of this example, the start address is 10.0.0.201 and the end address is 10.0.0.254. Later, these nodes will receive IP addresses from Cobbler.
5. Future versions of Fuel will enable you to choose a custom set of repositories.
6. If you need to specify a proxy through which fuel-pm will access the Internet, press 6.
7. Once you've finished editing, choose 9 to save your changes and exit the menu.

Please note: Even though defaults are shown, you must set actual values; if you simply press "enter" you will wind up with empty values.

To re-enter the menu at any time, type:

bootstrap_admin_node.sh

Installing the OS using Fuel

The first step to creating OpenStack nodes is to let Fuel's Cobbler kickstart and preseed files assist in the installation of operating systems on the target servers.

Configuring Cobbler with config.yaml

Fuel uses the `config.yaml` file to configure Cobbler and assist in the configuration of the `site.pp` file. This file appears in the `/root` directory when the master node (fuel-pm) is provisioned and configured.

You'll want to configure this example to meet your own needs, but the example looks like this:

```
common:
  orchestrator_common:
    attributes:
      deployment_mode: ha_compact
      deployment_engine: simplepuppet
    task_uuid: deployment_task
```

Possible values for `deployment_mode` are `singlenode_compute`, `multinode_compute`, `ha_compute`, `ha_compact`, `ha_full`, and `ha_minimal`. For this example, we will set the `deployment_mode` to `ha_compact` to tell Fuel to use HA architecture. Specifying the `simplepuppet` deployment engine means that the orchestrator will be calling Puppet on each of the nodes.

Next you'll need to set OpenStack's networking information:

```
openstack_common:
 internal_virtual_ip: 10.0.0.10
 public_virtual_ip: 192.168.0.10
 create_networks: true
 fixed_range: 172.16.0.0/16
 floating_range: 192.168.0.0/24
```

Change the virtual IPs to match the target networks, and set the fixed and floating ranges.

```
swift_loopback: loopback
nv_physical_volumes:
 - /dev/sdb
```

By setting the `nv_physical_volumes` value, you are not only telling OpenStack to use this value for Cinder (you'll see more about that in the `site.pp` file), but also where Cinder should store its data.

Later, we'll set up a new partition for Cinder, so tell Cobbler to create it here.

```
external_ip_info:
  public_net_router: 192.168.0.1
  ext_bridge: 0.0.0.0
  pool_start: 192.168.0.110
  pool_end: 192.168.0.126
```

Set the `public_net_router` to point to the real router at the public network. The `ext_bridge` is the IP of the Neutron (formerly Quantum) bridge. It should assigned to any available free IP on the public network that's outside

the floating range. You also have the option to simply set it to `0.0.0.0`. The `pool_start` and `pool_end` values represent the public addresses of your nodes, and should be within the `floating_range`.

```
segment_range: 900:999
network_manager: nova.network.manager.FlatDHCPManager
auto_assign_floating_ip: true
quantum_netnode_on_cnt: true
```

Fuel provides two choices for your network manager: FlatDHCPManager, and VlanManager. By default, the system uses FlatDHCPManager. Here you can see that we're also telling OpenStack to automatically assing a floating IP to an instance when it's created, and to put the Neutron services on the controllers rather than a sepearate node.You can also choose `tenant_network_type` for network segmentation type and segmentation range `segment_range` for network (consult Neutron documentation for details).

```
use_syslog: false
syslog_server: 127.0.0.1
mirror_type: default
```

THIS SETTING IS CRUCIAL: The `mirror_type` **must** to be set to `default` unless you have your own repositories set up, or OpenStack will not install properly.

```
quantum: true
internal_interface: eth0
public_interface: eth1
private_interface: eth2
public_netmask: 255.255.255.0
internal_netmask: 255.255.255.0
```

Earlier, you decided which interfaces to use for which networks; note that here.

```
default_gateway: 192.168.0.1
```

Depending on how you've set up your network, you can either set the `default_gateway` to the master node (fuel-pm) or to the `public_net_router`.

```
nagios_master: fuel-controller-01.localdomain
loopback: loopback
cinder: true
cinder_nodes:
- controller
swift: true
```

The loopback setting determines how Swift stores data. If you set the value to `loopback`, Swift will use 1gb files as storage devices. If you tuned Cobbler to create a partition for Swift and mounted it to `/srv/nodes/`, then you should set `loopback` to `false`.

In this example, you're using Cinder and including it on the compute nodes, so note that appropriately. Also, you're using Swift, so turn that on here.

```
repo_proxy: http://10.0.0.100:3128
```

One improvement in Fuel 2.1 was the ability for the master node to cache downloads in order to speed up installs; by default the `repo_proxy` is set to point to fuel-pm in order to let that happen. One consequence of that is that your deployment will actually go faster if you let one install complete, then do all the others, rather than running all of them concurrently.

```
deployment_id: '53'
```

Fuel enables you to manage multiple clusters; setting the `deployment_id` will let Fuel know which deployment you're working with.

```
dns_nameservers:
- 10.0.0.100
- 8.8.8.8
```

The slave nodes should first look to the master node for DNS, so mark that as your first nameserver.

The next step is to define the nodes themselves. To do that, you'll list each node once for each role that needs to be installed. Note that by default the first node is called `fuel-cobbler`; change it to `fuel-pm`.

```
nodes:
- name: fuel-pm
  role: cobbler
  internal_address: 10.0.0.100
  public_address: 192.168.0.100
- name: fuel-controller-01
  role: controller
  internal_address: 10.0.0.101
  public_address: 192.168.0.101
  swift_zone: 1
- name: fuel-controller-02
  role: controller
  internal_address: 10.0.0.102
  public_address: 192.168.0.102
  swift_zone: 2
- name: fuel-controller-03
  role: controller
  internal_address: 10.0.0.103
  public_address: 192.168.0.103
  swift_zone: 3
- name: fuel-controller-01
  role: quantum
  internal_address: 10.0.0.101
  public_address: 192.168.0.101
- name: fuel-compute-01
  role: compute
  internal_address: 10.0.0.110
  public_address: 192.168.0.110
```

Notice that each node can be listed multiple times; this is because each node fulfills multiple roles. Notice also that the IP address for fuel-compute-01 is *.110, not *.105.

The `cobbler_common` section applies to all machines:

```
cobbler_common:
  # for Centos
  profile: "centos64_x86_64"
  # for Ubuntu
  # profile: "ubuntu_1204_x86_64"
```

Fuel can install CentOS or Ubuntu on your servers, or you can add a profile of your own. By default, `config.yaml` uses CentOS.

```
netboot-enabled: "1"
# for Ubuntu
# ksmeta: "puppet_version=2.7.19-1puppetlabs2 \
# for Centos
name-servers: "10.0.0.100"
name-servers-search: "localdomain"
gateway: 192.168.0.1
```

Set the default nameserver to be fuel-pm, and change the domain name to your own domain name. Set the `gateway` to the public network's default gateway. Alternatively, if you don't plan to use your public networks actual gateway, you can set this value to be the IP address of the master node.

Please note: You must specify a working gateway (or proxy) in order to install OpenStack, because the system will need to communicate with public repositories.

```
ksmeta: "puppet_version=2.7.19-1puppetlabs2 \
  puppet_auto_setup=1 \
  puppet_master=fuel-pm.localdomain \
```

Change the fully-qualified domain name for the Puppet Master to reflect your own domain name.

```
puppet_enable=0 \
ntp_enable=1 \
mco_auto_setup=1 \
mco_pskey=un0aez2ei9eiGaequaey4loocohjuch4Ievu3shaeweeg5Uthi \
mco_stomphost=10.0.0.100 \
```

Make sure the `mco_stomphost` is set for the master node so that the orchestrator can find the nodes.

```
mco_stompport=61613 \
mco_stompuser=mcollective \
mco_stomppassword=AeN5mi5thahz2Aiveexo \
mco_enable=1"
```

This section sets the system up for orchestration; you shouldn't have to touch it.

Next you'll define the actual servers.

```
fuel-controller-01:
  hostname: "fuel-controller-01"
  role: controller
  interfaces:
    eth0:
      mac: "08:00:27:BD:3A:7D"
      static: "1"
      ip-address: "10.0.0.101"
      netmask: "255.255.255.0"
      dns-name: "fuel-controller-01.localdomain"
      management: "1"
    eth1:
      mac: "08:00:27:ED:9C:3C"
      static: "0"
    eth2:
      mac: "08:00:27:B0:EB:2C"
      static: "1"
  interfaces_extra:
    eth0:
      peerdns: "no"
    eth1:
      peerdns: "no"
    eth2:
      promisc: "yes"
      userctl: "yes"
      peerdns: "no"
```

For a VirtualBox installation, you can retrieve the MAC ids for your network adapters by expanding "Advanced" for the adapater in VirtualBox, or by executing ifconfig on the server itself.

For a physical installation, the MAC address of the server is often printed on the sticker attached to the server for the LOM interfaces, or is available from the BIOS screen. You may also be able to find the MAC address in the hardware inventory BMC/DRAC/ILO, though this may be server-dependent.

Also, make sure the `ip-address` is correct, and that the `dns-name` has your own domain name in it.

In this example, IP addresses should be assigned as follows:

```
fuel-controller-01:  10.0.0.101
fuel-controller-02:  10.0.0.102
fuel-controller-03:  10.0.0.103
fuel-compute-01:     10.0.0.110
```

Repeat this step for each of the other controllers, and for the compute node. Note that the compute node has its own role:

```
fuel-compute-01:
  hostname: "fuel-compute-01"
  role: compute
  interfaces:
```

```
    eth0:
      mac: "08:00:27:AE:A9:6E"
      static: "1"
      ip-address: "10.0.0.110"
      netmask: "255.255.255.0"
      dns-name: "fuel-compute-01.localdomain"
      management: "1"
    eth1:
      mac: "08:00:27:B7:F9:CD"
      static: "0"
    eth2:
      mac: "08:00:27:8B:A6:B7"
      static: "1"
  interfaces_extra:
    eth0:
      peerdns: "no"
    eth1:
      peerdns: "no"
    eth2:
      promisc: "yes"
      userctl: "yes"
      peerdns: "no"
```

Loading the configuration

Once you've completed the changes to `config.yaml`, you need to load the information into Cobbler. To do that, use the `cobbler_system` script:

```
cobbler_system -f config.yaml
```

Now you're ready to start spinning up the controllers and compute nodes.

Installing the operating system

Now that Cobbler has the correct configuration, the only thing you need to do is to PXE-boot your nodes. This means that they will boot over the network, with DHCP/TFTP provided by Cobbler, and will be provisioned accordingly, with the specified operating system and configuration.

If you installed Fuel from the ISO, start fuel-controller-01 first and let the installation finish before starting the other nodes; Fuel will cache the downloads so subsequent installs will go faster.

The process for each node looks like this:

1. Start the VM.
2. Press F12 immediately and select l (LAN) as a bootable media.
3. Wait for the installation to complete.
4. Log into the new machine using root/r00tme.
5. **Change the root password.**
6. Check that networking is set up correctly and the machine can reach the Internet:

   ```
   ping fuel-pm.localdomain
   ping www.mirantis.com
   ```

If you're unable to ping outside addresses, add the fuel-pm server as a default gateway:

```
route add default gw 10.0.0.100
```

It is important to note that if you use VLANs in your network configuration, you always have to keep in mind the fact that PXE booting does not work on tagged interfaces. Therefore, all your nodes, including the one where the Cobbler service resides, must share one untagged VLAN (also called native VLAN). If necessary, you can use the `dhcp_interface` parameter of the `cobbler::server` class to bind the DHCP service to the appropriate interface.

Generating the Puppet Manifest

Before you can deploy OpenStack, you will need to configure the site.pp file. Previous versions of Fuel required you to manually configure `site.pp`. Version 3.1 includes the `openstack_system` script, which uses the `config.yaml`

file and reference architecture templates to create the appropriate Puppet manifest. To create `site.pp`, execute this command:

```
openstack_system -c config.yaml \
  -t /etc/puppet/modules/openstack/examples/site_openstack_ha_compact.pp \
  -o /etc/puppet/manifests/site.pp \
  -a astute.yaml
```

The four parameters in the command above are:

- `-c`: The absolute or relative path to the `config.yaml` file you customized earlier.
- `-t`: The template file to serve as a basis for `site.pp`. Possible templates include `site_openstack_ha_compact.pp`, `site_openstack_ha_minimal.pp`, `site_openstack_ha_full.pp`, `site_openstack_single.pp`, and `site_openstack_simple.pp`.
- `-o`: The output file. This should always be `/etc/puppet/manifests/site.pp`.
- `-a`: The orchestration configuration file, to be output for use in the next step.

From there you're ready to install your OpenStack components. Before that, however, let's look at what is actually in the new `site.pp` manifest, so that you can understand how to customize it if necessary. (Similarly, if you are installing Fuel Library without the ISO, you will need to make these customizations yourself.)

Understanding the Puppet Manifest

At this point you should have functioning servers that are ready to take an OpenStack installation. If you're using VirtualBox, save the current state of every virtual machine by taking a snapshot using `File->Take Snapshot`. Snapshots are a useful tool when you make a mistake, encounter an issue, or just want to try different configurations, all without having to start from scratch.

Next, go through the `/etc/puppet/manifests/site.pp` file and make any necessary customizations. If you have run `openstack_system`, there shouldn't be anything to change (with one small exception) but if you are installing Fuel manually, you will need to make these changes yourself.

Let's start with the basic network customization:

```
### GENERAL CONFIG ###
# This section sets main parameters such as hostnames and IP addresses of different nodes

# This is the name of the public interface. The public network provides address space for Floating IPs, as
$public_interface = 'eth1'
$public_br = 'br-ex'

# This is the name of the internal interface. It will be attached to the management network, where data ex
$internal_interface = 'eth0'
$internal_br = 'br-mgmt'

# This is the name of the private interface. All traffic within OpenStack tenants' networks will go throug
$private_interface = 'eth2'
```

In this case, we don't need to make any changes to the interface settings, because they match what we've already set up.

```
# Public and Internal VIPs. These virtual addresses are required by HA topology and will be managed by kee
$internal_virtual_ip = '10.0.0.10'

# Change this IP to IP routable from your 'public' network,
# e. g. Internet or your office LAN, in which your public
# interface resides
$public_virtual_ip = '192.168.0.10'
```

Make sure the virtual IPs you see here don't conflict with your network configuration. The IPs you use should be routeable, but not within the range of a DHCP scope. These are the IPs through which your services will be accessed.

The following section configures the servers themselves. If you are setting up Fuel manually, make sure to add each server with the appropriate IP addresses; if you ran `openstack_system`, the values will be overridden by the next section, and you can ignore this array.

```
$nodes_harr = [
  {
    'name' => 'fuel-pm',
    'role' => 'cobbler',
    'internal_address' => '10.0.0.100',
    'public_address'   => '192.168.0.100',
    'mountpoints'=> "1 1\n2 1",
    'storage_local_net_ip' => '10.0.0.100',
  },
  {
    'name' => 'fuel-controller-01',
    'role' => 'primary-controller',
    'internal_address' => '10.0.0.101',
    'public_address'   => '192.168.0.101',
    'mountpoints'=> "1 1\n2 1",
    'storage_local_net_ip' => '10.0.0.101',
  },
  {
    'name' => 'fuel-controller-02',
    'role' => 'controller',
    'internal_address' => '10.0.0.102',
    'public_address'   => '192.168.0.102',
    'mountpoints'=> "1 1\n2 1",
    'storage_local_net_ip' => '10.0.0.102',
  },
  {
    'name' => 'fuel-controller-03',
    'role' => 'controller',
    'internal_address' => '10.0.0.105',
    'public_address'   => '192.168.0.105',
    'mountpoints'=> "1 1\n2 1",
    'storage_local_net_ip' => '10.0.0.105',
  },
  {
    'name' => 'fuel-compute-01',
    'role' => 'compute',
    'internal_address' => '10.0.0.106',
    'public_address'   => '192.168.0.106',
    'mountpoints'=> "1 1\n2 1",
    'storage_local_net_ip' => '10.0.0.106',
  }
]
```

Because this section comes from a template, it will likely include a number of servers you're not using; feel free to leave them or take them out.

Next, the `site.pp` file lists all of the nodes and roles you defined in the `config.yaml` file:

```
$nodes = [{'public_address' => '192.168.0.101','name' => 'fuel-controller-01','role' =>
           'primary-controller','internal_address' => '10.0.0.101',
           'storage_local_net_ip' => '10.0.0.101', 'mountpoints' => '1 2\n2 1',
           'swift-zone' => 1 },
          {'public_address' => '192.168.0.102','name' => 'fuel-controller-02','role' =>
           'controller','internal_address' => '10.0.0.102',
           'storage_local_net_ip' => '10.0.0.102', 'mountpoints' => '1 2\n2 1',
           'swift-zone' => 2},
          {'public_address' => '192.168.0.103','name' => 'fuel-controller-03','role' =>
           'storage','internal_address' => '10.0.0.103',
           'storage_local_net_ip' => '10.0.0.103', 'mountpoints' => '1 2\n2 1',
           'swift-zone' => 3},
          {'public_address' => '192.168.0.110','name' => 'fuel-compute-01','role' =>
           'compute','internal_address' => '10.0.0.110'}]
```

Possible roles include ‘compute’, ‘controller’, ‘primary-controller’, ‘storage’, ‘swift-proxy’, ‘quantum’, ‘master’, and ‘cobbler’. Check the IP addresses for each node and make sure that they match the contents of this array.

The file also specifies the default gateway to be the fuel-pm machine:

```
$default_gateway = '192.168.0.1'
```

Next `site.pp` defines DNS servers and provides netmasks:

```
# Specify nameservers here.
# You can point this to the cobbler node IP, or to specially prepared nameservers as needed.
$dns_nameservers = ['10.0.0.100','8.8.8.8']

# Specify netmasks for internal and external networks.
$internal_netmask = '255.255.255.0'
$public_netmask = '255.255.255.0'
...
# Set this to anything other than pacemaker if you do not want Neutron HA (formerly Quantum HA)
# Also, if you do not want Neutron HA, you MUST enable $quantum_network_node
# only on the controller
$ha_provider = 'pacemaker'
$use_unicast_corosync = false
```

Next specify the main controller as the Nagios master.

```
# Set nagios master fqdn
$nagios_master = 'fuel-controller-01.localdomain'
## proj_name  name of environment nagios configuration
$proj_name            = 'test'
```

Here again we have a parameter that looks ahead to things to come; OpenStack supports monitoring via Nagios. In this section, you can choose the Nagios master server as well as setting a project name.

```
#Specify if your installation contains multiple Nova controllers. Defaults to true as it is the most commo
$multi_host             = true
```

A single host cloud isn't especially useful, but if you really want to, you can specify that here.

Finally, you can define the various usernames and passwords for OpenStack services.

```
# Specify different DB credentials for various services
$mysql_root_password     = 'nova'
$admin_email             = 'openstack@openstack.org'
$admin_password          = 'nova'

$keystone_db_password    = 'nova'
$keystone_admin_token    = 'nova'

$glance_db_password      = 'nova'
$glance_user_password    = 'nova'

$nova_db_password        = 'nova'
$nova_user_password      = 'nova'

$rabbit_password         = 'nova'
$rabbit_user             = 'nova'

$swift_user_password     = 'swift_pass'
$swift_shared_secret     = 'changeme'

$quantum_user_password   = 'quantum_pass'
$quantum_db_password     = 'quantum_pass'
$quantum_db_user         = 'quantum'
$quantum_db_dbname       = 'quantum'

# End DB credentials section
```

Now that the network is configured for the servers, let's look at the various OpenStack services.

Enabling Neutron

In order to deploy OpenStack with Neutron you need to set up an additional node that will act as an L3 router, or run Neutron out of one of the existing nodes.

```
### NETWORK/QUANTUM ###
# Specify network/quantum specific settings

# Should we use quantum or nova-network (deprecated).
# Consult OpenStack documentation for differences between them.
$quantum = true
$quantum_netnode_on_cnt  = true
```

In this case, we're using a "compact" architecture, so we want to install Neutron on the controllers:

```
# Specify network creation criteria:
# Should puppet automatically create networks?
$create_networks = true

# Fixed IP addresses are typically used for communication between VM instances.
$fixed_range = '172.16.0.0/16'

# Floating IP addresses are used for communication of VM instances with the outside world (e.g. Internet).
$floating_range = '192.168.0.0/24'
```

OpenStack uses two ranges of IP addresses for virtual machines: fixed IPs, which are used for communication between VMs, and thus are part of the private network, and floating IPs, which are assigned to VMs for the purpose of communicating to and from the Internet.

```
# These parameters are passed to the previously specified network manager , e.g. nova-manage network creat
# Not used in Neutron.
$num_networks    = 1
$network_size    = 31
$vlan_start      = 300
```

These values don't actually relate to Neutron; they are used by nova-network. IDs for the VLANs OpenStack will create for tenants run from `vlan_start` to (`vlan_start + num_networks - 1`), and are generated automatically.

```
# Neutron

# Segmentation type for isolating traffic between tenants
# Consult Openstack Neutron docs
$tenant_network_type     = 'gre'

# Which IP address will be used for creating GRE tunnels.
$quantum_gre_bind_addr = $internal_address
```

If you are installing Neutron in non-HA mode, you will need to specify which single controller controls Neutron.

```
# If $external_ipinfo option is not defined, the addresses will be allocated automatically from $floating_
# the first address will be defined as an external default router,
# the second address will be attached to an uplink bridge interface,
# the remaining addresses will be utilized for the floating IP address pool.
$external_ipinfo = {
   'pool_start' => '192.168.0.115',
       'public_net_router' => '192.168.0.1',
       'pool_end' => '192.168.0.126',
       'ext_bridge' => '0.0.0.0'
}

# Neutron segmentation range.
# For VLAN networks: valid VLAN VIDs can be 1 through 4094.
# For GRE networks: Valid tunnel IDs can be any 32-bit unsigned integer.
$segment_range = '900:999'

# Set up OpenStack network manager. It is used ONLY in nova-network.
# Consult Openstack nova-network docs for possible values.
$network_manager = 'nova.network.manager.FlatDHCPManager'
```

```
# Assign floating IPs to VMs on startup automatically?
$auto_assign_floating_ip = false

# Database connection for Neutron configuration (quantum.conf)
$quantum_sql_connection  = "mysql://${quantum_db_user}:${quantum_db_password}@${$internal_virtual_ip}/{qua

if $quantum {
  $public_int   = $public_br
  $internal_int = $internal_br
} else {
  $public_int   = $public_interface
  $internal_int = $internal_interface
}
```

If the system is set up to use Neutron, the public and internal interfaces are set to use the appropriate bridges, rather than the defined interfaces.

The remaining configuration is used to define classes that will be added to each Neutron node:

```
#Network configuration
stage {'netconfig':
      before  => Stage['main'],
}
class {'l23network': use_ovs => $quantum, stage=> 'netconfig'}
class node_netconfig (
  $mgmt_ipaddr,
  $mgmt_netmask  = '255.255.255.0',
  $public_ipaddr = undef,
  $public_netmask= '255.255.255.0',
  $save_default_gateway=true,
  $quantum = $quantum,
) {
  if $quantum {
    l23network::l3::create_br_iface {'mgmt':
      interface => $internal_interface, # !!! NO $internal_int /sv !!!
      bridge    => $internal_br,
      ipaddr    => $mgmt_ipaddr,
      netmask   => $mgmt_netmask,
      dns_nameservers      => $dns_nameservers,
      save_default_gateway => $save_default_gateway,
    } ->
    l23network::l3::create_br_iface {'ex':
      interface => $public_interface, # !! NO $public_int /sv !!!
      bridge    => $public_br,
      ipaddr    => $public_ipaddr,
      netmask   => $public_netmask,
      gateway   => $default_gateway,
    }
  } else {
    # nova-network mode
    l23network::l3::ifconfig {$public_int:
      ipaddr  => $public_ipaddr,
      netmask => $public_netmask,
      gateway => $default_gateway,
    }
    l23network::l3::ifconfig {$internal_int:
      ipaddr  => $mgmt_ipaddr,
      netmask => $mgmt_netmask,
      dns_nameservers      => $dns_nameservers,
    }
  }
  l23network::l3::ifconfig {$private_interface: ipaddr=>'none' }
}
### NETWORK/QUANTUM END ###
```

All of this assumes, of course, that you're using Neutron; if you're using nova-network instead, only these values apply.

Defining the current cluster

Fuel enables you to control multiple deployments simultaneously by setting an individual deployment ID:

```
# This parameter specifies the the identifier of the current cluster. This is required for environments wh
# installation. Each cluster requires a unique integer value.
# Valid identifier range is 0 to 254
$deployment_id = '79'
```

Enabling Cinder

Our example uses Cinder, and with some very specific variations from the default. Specifically, as we said before, while the Cinder scheduler will continue to run on the controllers, the actual storage takes place on the compute nodes, specifically the `/dev/sdb1` partition you created earlier. Cinder will be activated on any node that contains the specified block devices -- unless specified otherwise -- so let's look at what all of that means for the configuration.

```
# Choose which nodes to install cinder onto
# 'compute'            -> compute nodes will run cinder
# 'controller'         -> controller nodes will run cinder
# 'storage'            -> storage nodes will run cinder
# 'fuel-controller-XX' -> specify particular host(s) by hostname
# 'XXX.XXX.XXX.XXX'    -> specify particular host(s) by IP address
# 'all'                -> compute, controller, and storage nodes will run cinder (excluding swift and prox
$cinder_nodes          = ['controller']
```

We want Cinder to be on the controller nodes, so set this value to `['controller']`.

```
# Set this option to true if cinder-volume has been installed to the host
# otherwise it will install api and scheduler services
$manage_volumes = true

# Setup network interface, which Cinder uses to export iSCSI targets.
$cinder_iscsi_bind_addr = $internal_address
```

Here you have the opportunity to specify which network interface Cinder uses for its own traffic. For example, you could set up a fourth NIC at `eth3` and specify that rather than `$internal_int`.

```
# Below you can add physical volumes to cinder. Please replace values with the actual names of devices.
# This parameter defines which partitions to aggregate into cinder-volumes or nova-volumes LVM VG
# !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
# USE EXTREME CAUTION WITH THIS SETTING! IF THIS PARAMETER IS DEFINED,
# IT WILL AGGREGATE THE VOLUMES INTO AN LVM VOLUME GROUP
# AND ALL THE DATA THAT RESIDES ON THESE VOLUMES WILL BE LOST!
# !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
# Leave this parameter empty if you want to create [cinder|nova]-volumes VG by yourself
$nv_physical_volume = ['/dev/sdb']

#Evaluate cinder node selection
if ($cinder) {
  if (member($cinder_nodes,'all')) {
     $is_cinder_node = true
  } elsif (member($cinder_nodes,$::hostname)) {
     $is_cinder_node = true
  } elsif (member($cinder_nodes,$internal_address)) {
     $is_cinder_node = true
  } elsif ($node[0]['role'] =~ /controller/)) {
     $is_cinder_node = member($cinder_nodes, 'controller')
  } else {
     $is_cinder_node = member($cinder_nodes, $node[0]['role'])
  }
} else {
  $is_cinder_node = false
}
```

```
### CINDER/VOLUME END ###
```

We only want to allocate the `/dev/sdb` volume to Cinder, so adjust `$nv_physical_volume` accordingly. Note, however, that this is a global value; it will apply to all servers, including the controllers -- unless we specify otherwise, which we illustrate below.

Be careful to not add block devices to the list which contain useful data (e.g. block devices on which your OS resides), as they will be destroyed after you allocate them for Cinder. It is always a good rule of thumb to deploy OpenStack on blank storage and move content to those volumes later instead of try to retain existing data.

Now lets look at Swift, the other storage-based service option.

Enabling Glance and Swift

There aren't many changes that you will need to make to the default configuration in order to enable Swift to work properly in Swift Compact mode, but you will need to adjust if you want to run Swift on physical partitions

```
...
### GLANCE and SWIFT ###

# Which backend to use for glance
# Supported backends are 'swift' and 'file'
$glance_backend = 'swift'

# Use loopback device for swift:
# options are 'loopback' or 'false'
# This parameter controls where swift partitions are located:
# on physical partitions or inside loopback devices.
$swift_loopback = loopback
```

The default value is `loopback`, which tells Swift to use a loopback storage device, which is basically a file that acts like a drive, rather than a physical drive. You can also set this value to `false`, which tells OpenStack to use a physical drive (or drives) instead.

```
# Which IP address to bind swift components to: e.g., which IP swift-proxy should listen on
$swift_local_net_ip = $internal_address

# IP node of controller used during swift installation
# and put into swift configs
$controller_node_public = $internal_virtual_ip

# Hash of proxies hostname|fqdn => ip mappings.
# This is used by controller_ha.pp manifests for haproxy setup
# of swift_proxy backends
$swift_proxies = $controller_internal_addresses
```

Next, you're specifying the `swift-master`:

```
# Set hostname of swift_master.
# It tells on which swift proxy node to build
# *ring.gz files. Other swift proxies/storages
# will rsync them.
if $node[0]['role'] == 'primary-controller' {
  $primary_proxy = true
} else {
  $primary_proxy = false
}
if $node[0]['role'] == 'primary-controller' {
  $primary_controller = true
} else {
  $primary_controller = false
}
$master_swift_proxy_nodes = filter_nodes($nodes,'role','primary-controller')
$master_swift_proxy_ip = $master_swift_proxy_nodes[0]['internal_address']
```

In this case, there's no separate `fuel-swiftproxy-01`, so the master controller will be the primary Swift controller.

Configuring OpenStack to use syslog

To use the syslog server, adjust the corresponding variables in the `if $use_syslog` clause:

```
$use_syslog = true
if $use_syslog {
    class { "::rsyslog::client":
        log_local => true,
        log_auth_local => true,
        server => '127.0.0.1',
        port => '514'
    }
}
```

For remote logging, use the IP or hostname of the server for the `server` value and set the `port` appropriately. For local logging, set `log_local` and `log_auth_local` to `true`.

Setting the version and mirror type

You can customize the various versions of OpenStack's components, though it's typical to use the latest versions:

```
### Syslog END ###
case $::osfamily {
    "Debian":  {
       $rabbitmq_version_string = '2.8.7-1'
    }
    "RedHat": {
       $rabbitmq_version_string = '2.8.7-2.el6'
    }
}
# OpenStack packages and customized component versions to be installed.
# Use 'latest' to get the most recent ones or specify exact version if you need to install custom version.
$openstack_version = {
  'keystone'         => 'latest',
  'glance'           => 'latest',
  'horizon'          => 'latest',
  'nova'             => 'latest',
  'novncproxy'       => 'latest',
  'cinder'           => 'latest',
  'rabbitmq_version' => $rabbitmq_version_string,
}
```

To tell Fuel to download packages from external repos provided by Mirantis and your distribution vendors, make sure the `$mirror_type` variable is set to `default`:

```
# If you want to set up a local repository, you will need to manually adjust mirantis_repos.pp,
# though it is NOT recommended.
$mirror_type = 'default'
$enable_test_repo = false
$repo_proxy = 'http://10.0.0.100:3128'
```

Once again, the `$mirror_type` **must** be set to `default`. If you set it correctly in `config.yaml` and ran `openstack_system` this will already be taken care of. Otherwise, **make sure** to set this value manually.

Future versions of Fuel will enable you to use your own internal repositories.

Setting verbosity

You also have the option to determine how much information OpenStack provides when performing configuration:

```
# This parameter specifies the verbosity level of log messages
# in openstack components config. Currently, it disables or enables debugging.
$verbose = true
```

Configuring Rate-Limits

Openstack has predefined limits on different HTTP queries for nova-compute and cinder services. Sometimes (e.g. for big clouds or test scenarios) these limits are too strict. (See

http://docs.openstack.org/folsom/openstack-compute/admin/content/configuring-compute-API.html.) In this case you can change them to more appropriate values.

There are two hashes describing these limits: `$nova_rate_limits` and `$cinder_rate_limits`.

```
#Rate Limits for cinder and Nova
#Cinder and Nova can rate-limit your requests to API services.
#These limits can be reduced for your installation or usage scenario.
#Change the following variables if you want. They are measured in requests per minute.
$nova_rate_limits = {
  'POST' => 1000,
  'POST_SERVERS' => 1000,
  'PUT' => 1000, 'GET' => 1000,
  'DELETE' => 1000
}
$cinder_rate_limits = {
  'POST' => 1000,
  'POST_SERVERS' => 1000,
  'PUT' => 1000, 'GET' => 1000,
  'DELETE' => 1000
}
...
```

Enabling Horizon HTTPS/SSL mode

Using the `$horizon_use_ssl` variable, you have the option to decide whether the OpenStack dashboard (Horizon) uses HTTP or HTTPS:

```
...
#  'custom': require fileserver static mount point [ssl_certs] and hostname based certificate existence
$horizon_use_ssl = false
```

This variable accepts the following values:

- `false`: In this mode, the dashboard uses HTTP with no encryption.
- `default`: In this mode, the dashboard uses keys supplied with the standard Apache SSL module package.
- `exist`: In this case, the dashboard assumes that the domain name-based certificate, or keys, are provisioned in advance. This can be a certificate signed by any authorized provider, such as Symantec/Verisign, Comodo, GoDaddy, and so on. The system looks for the keys in these locations:

  for Debian/Ubuntu:

  - public `/etc/ssl/certs/domain-name.pem`
  - private `/etc/ssl/private/domain-name.key`

  for Centos/RedHat:

  - public `/etc/pki/tls/certs/domain-name.crt`
  - private `/etc/pki/tls/private/domain-name.key`
- `custom`: This mode requires a static mount point on the fileserver for `[ssl_certs]` and certificate pre-existence. To enable this mode, configure the puppet fileserver by editing `/etc/puppet/fileserver.conf` to add:

  ```
  [ssl_certs]
    path /etc/puppet/templates/ssl
    allow *
  ```

  From there, create the appropriate directory:

  ```
  mkdir -p /etc/puppet/templates/ssl
  ```

  Add the certificates to this directory. (Reload the puppetmaster service for these changes to take effect.)

Now we just need to make sure that all of our nodes get the proper values.

Defining the node configurations

Now that we've set all of the global values, its time to make sure that the actual node definitions are correct. For example, by default all nodes will enable Cinder on `/dev/sdb`. If you don't want to enable Cinder on all controllers set `nv_physical_volume` to `null` for a specific node or nodes.

```
...
class compact_controller (
  $quantum_network_node = $quantum_netnode_on_cnt
) {
  class { 'openstack::controller_ha':
    controller_public_addresses   => $controller_public_addresses,
    controller_internal_addresses => $controller_internal_addresses,
    internal_address        => $internal_address,
    public_interface        => $public_int,
    internal_interface      => $internal_int,
 ...
    use_unicast_corosync    => $use_unicast_corosync,
    ha_provider             => $ha_provider
  }
  class { 'swift::keystone::auth':
    password         => $swift_user_password,
    public_address   => $public_virtual_ip,
    internal_address => $internal_virtual_ip,
    admin_address    => $internal_virtual_ip,
  }
}
...
```

To reduce repeated manual configuration, Fuel includes a class for the controllers. This eliminates the need to make global changes for each individual controller. You will note that lower down in this configuration segment that this class also lets you specify the individual controllers and compute nodes:

```
...
    node /fuel-controller-[\d+]/ {
      include stdlib
      class { 'operatingsystem::checksupported':
          stage => 'setup'
      }

      class {'::node_netconfig':
          mgmt_ipaddr    => $::internal_address,
          mgmt_netmask   => $::internal_netmask,
          public_ipaddr  => $::public_address,
          public_netmask => $::public_netmask,
          stage          => 'netconfig',
      }

      class {'nagios':
        proj_name       => $proj_name,
        services        => [
          'host-alive','nova-novncproxy','keystone', 'nova-scheduler',
          'nova-consoleauth', 'nova-cert', 'haproxy', 'nova-api', 'glance-api',
          'glance-registry','horizon', 'rabbitmq', 'mysql', 'swift-proxy',
          'swift-account', 'swift-container', 'swift-object',
        ],
        whitelist       => ['127.0.0.1', $nagios_master],
        hostgroup       => 'controller',
      }

      class { compact_controller: }
      $swift_zone = $node[0]['swift_zone']

      class { 'openstack::swift::storage_node':
        storage_type       => $swift_loopback,
        swift_zone         => $swift_zone,
        swift_local_net_ip => $internal_address,
      }
```

```
      class { 'openstack::swift::proxy':
        swift_user_password     => $swift_user_password,
        swift_proxies           => $swift_proxies,
        ...
        rabbit_ha_virtual_ip      => $internal_virtual_ip,
      }
    }
```

Note that each controller has the swift_zone specified, so each of the three controllers can represent each of the three Swift zones. Similarly, site.pp defines a class for the compute nodes.

Installing Nagios Monitoring using Puppet

Fuel provides a way to deploy Nagios for monitoring your OpenStack cluster. Nagios is an open source distributed management and monitoring infrastructure that is commonly used in data centers to keep an eye on thousands of servers. Nagios requires the installation of a software agent on all nodes, as well as having a master server for Nagios which will collect and display all the results. The agent, the Nagios NRPE addon, allows OpenStack to execute Nagios plugins on remote Linux/Unix machines. The main reason for doing this is to monitor key resources (such as CPU load, memory usage, etc.), as well as provide more advanced metrics and performance data on local and remote machines.

Nagios Agent

In order to install Nagios NRPE on a compute or controller node, a node should have the following settings:

```
class {'nagios':
  proj_name       => 'test',
  services        => ['nova-compute','nova-network','libvirt'],
  whitelist       => ['127.0.0.1', $nagios_master],
  hostgroup       => 'compute',
}
```

- `proj_name`: An environment for nagios commands and the directory (`/etc/nagios/test/`).
- `services`: All services to be monitored by nagios.
- `whitelist`: The array of IP addreses trusted by NRPE.
- `hostgroup`: The group to be used in the nagios master (do not forget create the group in the nagios master).

Nagios Server

In order to install Nagios Master on any convenient node, a node should have the following applied:

```
class {'nagios::master':
  proj_name       => 'test',
  templatehost    => {'name' => 'default-host','check_interval' => '10'},
  templateservice => {'name' => 'default-service' ,'check_interval'=>'10'},
  hostgroups      => ['compute','controller'],
  contactgroups   => {'group' => 'admins', 'alias' => 'Admins'},
  contacts        => {'user' => 'hotkey', 'alias' => 'Dennis Hoppe',
              'email' => 'nagios@%{domain}',
              'group' => 'admins'},
}
```

- `proj_name`: The environment for nagios commands and the directory (`/etc/nagios/test/`).
- `templatehost`: The group of checks and intervals parameters for hosts (as a Hash).
- `templateservice`: The group of checks and intervals parameters for services (as a Hash).
- `hostgroups`: All groups which on NRPE nodes (as an Array).
- `contactgroups`: The group of contacts (as a Hash).
- `contacts`: Contacts to receive error reports (as a Hash)

Health Checks

You can see the complete definition of the available services to monitor and their health checks at `deployment/puppet/nagios/manifests/params.pp`.

Here is the list:

```
$services_list = {
  'nova-compute' => 'check_nrpe_1arg!check_nova_compute',
  'nova-network' => 'check_nrpe_1arg!check_nova_network',
  'libvirt' => 'check_nrpe_1arg!check_libvirt',
  'swift-proxy' => 'check_nrpe_1arg!check_swift_proxy',
  'swift-account' => 'check_nrpe_1arg!check_swift_account',
  'swift-container' => 'check_nrpe_1arg!check_swift_container',
  'swift-object' => 'check_nrpe_1arg!check_swift_object',
  'swift-ring' => 'check_nrpe_1arg!check_swift_ring',
  'keystone' => 'check_http_api!5000',
  'nova-novncproxy' => 'check_nrpe_1arg!check_nova_novncproxy',
  'nova-scheduler' => 'check_nrpe_1arg!check_nova_scheduler',
  'nova-consoleauth' => 'check_nrpe_1arg!check_nova_consoleauth',
  'nova-cert' => 'check_nrpe_1arg!check_nova_cert',
  'cinder-scheduler' => 'check_nrpe_1arg!check_cinder_scheduler',
  'cinder-volume' => 'check_nrpe_1arg!check_cinder_volume',
  'haproxy' => 'check_nrpe_1arg!check_haproxy',
  'memcached' => 'check_nrpe_1arg!check_memcached',
  'nova-api' => 'check_http_api!8774',
  'cinder-api' => 'check_http_api!8776',
  'glance-api' => 'check_http_api!9292',
  'glance-registry' => 'check_nrpe_1arg!check_glance_registry',
  'horizon' => 'check_http_api!80',
  'rabbitmq' => 'check_rabbitmq',
  'mysql' => 'check_galera_mysql',
  'apt' => 'nrpe_check_apt',
  'kernel' => 'nrpe_check_kernel',
  'libs' => 'nrpe_check_libs',
  'load' => 'nrpe_check_load!5.0!4.0!3.0!10.0!6.0!4.0',
  'procs' => 'nrpe_check_procs!250!400',
  'zombie' => 'nrpe_check_procs_zombie!5!10',
  'swap' => 'nrpe_check_swap!20%!10%',
  'user' => 'nrpe_check_users!5!10',
  'host-alive' => 'check-host-alive',
}
```

Node definitions

The following is a list of the node definitions generated for a Compact HA deployment. Other deployment configurations generate other definitions. For example, the `openstack/examples/site_openstack_full.pp` template specifies the following nodes:

- fuel-controller-01
- fuel-controller-02
- fuel-controller-03
- fuel-compute-[d+]
- fuel-swift-01
- fuel-swift-02
- fuel-swift-03
- fuel-swiftproxy-[d+]
- fuel-quantum

Using this architecture, the system includes three stand-alone swift-storage servers, and one or more swift-proxy servers.

With `site.pp` prepared, you're ready to perform the actual installation.

Deploying OpenStack

You have two options for deploying OpenStack. The best method is to use orchestration, but you can also deploy your nodes manually.

Deploying via orchestration

Performing a small series of manual installs many be an acceptable approach in some circumstances, but if you plan on deploying to a large number of servers then we strongly suggest that you consider orchestration. You can perform a deployment using orchestration with Fuel using the "astute" script. This script is configured using the "astute.yaml" file that was created when you ran "openstack_system" earlier in this process.

To confirm that your servers are ready for orchestration, execute the following command:

```
mco ping
```

You should see all three controllers, plus the compute node, in the response to the command:

```
fuel-compute-01                                  time=107.26 ms
fuel-controller-01                               time=120.14 ms
fuel-controller-02                               time=135.94 ms
fuel-controller-03                               time=139.33 ms
```

To run the orchestrator, log in to `fuel-pm` and execute:

```
astute -f astute.yaml
```

You will see a message on `fuel-pm` stating that the installation has started on fuel-controller-01. To see what's going on on the target node on Ubuntu-based OS, enter this command:

```
tail -f /var/log/syslog
```

for CentOS or RedHat use this command:

```
tail -f /var/log/messages
```

Note that Puppet will require several runs to install all the different roles. The first time it runs, the orchestrator will show an error. This error means that the installation isn't complete, but will be rectified after the various rounds of installation are completed. Also, after the first run on each server, the orchestrator doesn't output messages on fuel-pm; when it's finished running, it will return you to the command prompt. In the meantime, you can see what's going on by watching the logs on each individual machine.

Installing OpenStack using Puppet directly

If, for some reason, you choose not to use orchestration -- one common example it when adding a single node to an existing (non-HA) cluster -- you have the option to install on one or more nodes using Puppet.

Start by logging in to the target server -- fuel-controller-01 to start, if you're starting from scratch -- and running the Puppet agent.

One optional step would be to use the script command to log all of your output so you can check for errors:

```
script agent-01.log
puppet agent --test
```

You will to see a great number of messages scroll by, and the installation will take a significant amount of time. When the process has completed, press CTRL-D to stop logging and grep for errors:

```
grep err: agent-01.log
```

If you find any errors relating to other nodes you may safely ignore them for now.

Now you can run the same installation procedure on fuel-controller-02 and fuel-controller-03, as well as fuel-compute-01.

Note that the controllers must be installed sequentially due to the nature of assembling a MySQL cluster based on Galera. This means that one server must complete its installation before the next installation is started. However, compute nodes can be installed concurrently once the controllers are in place.

In some cases, you may find errors related to resources that are not yet available when the installation takes place. To solve that problem, simply re-run the puppet agent on the affected node after running the other controllers, and again grep for error messages. When you see no errors on any of your nodes, your OpenStack cluster is ready to go.

Examples of OpenStack installation sequences

When running Puppet manually, the exact sequence depends on the configuration goals you're trying to achieve. In most cases, you'll need to run Puppet more than once; with every pass Puppet collects and adds necessary absent information to the OpenStack configuration, stores it to PuppedDB and applies necessary changes.

Note: *Sequentially run* means you don't start the next node deployment until previous one is finished.

Example 1: Full OpenStack deployment with standalone storage nodes

- Create necessary volumes on storage nodes as described in *create-the-XFS-partition*.
- Sequentially run a deployment pass on every SwiftProxy node (`fuel-swiftproxy-01 ... fuel-swiftproxy-xx`), starting with the `primary-swift-proxy` node. Node names are set by the `$swift_proxies` variable in `site.pp`. There are 2 Swift Proxies by default.
- Sequentially run a deployment pass on every storage node (`fuel-swift-01` ... `fuel-swift-xx`).
- Sequentially run a deployment pass on the controller nodes (`fuel-controller-01 ... fuel-controller-xx`). starting with the `primary-controller` node.
- Run a deployment pass on the Neutron (formerly Quantum) node (`fuel-quantum`) to install the Neutron router.
- Run a deployment pass on every compute node (`fuel-compute-01 ... fuel-compute-xx`) - unlike the controllers, these nodes may be deployed in parallel.
- Run an additional deployment pass on Controller 1 only (`fuel-controller-01`) to finalize the Galera cluster configuration.

Example 2: Compact OpenStack deployment with storage and swift-proxy combined with nova-controller on the same nodes

- Create the necessary volumes on controller nodes as described in *create-the-XFS-partition*
- Sequentially run a deployment pass on the controller nodes (`fuel-controller-01 ... fuel-controller-xx`), starting with the `primary-controller` node. Errors in Swift storage such as */Stage[main]/Swift::Storage::Container/Ring_container_device[<device address>]: Could not evaluate: Device not found check device on <device address>* are expected during the deployment passes until the very final pass.
- Run an additional deployment pass on Controller 1 only (`fuel-controller-01`) to finalize the Galera cluster configuration.
- Run a deployment pass on the Neutron node (`fuel-quantum`) to install the Neutron router.
- Run a deployment pass on every compute node (`fuel-compute-01 ... fuel-compute-xx`) - unlike the controllers these nodes may be deployed in parallel.

Example 3: OpenStack HA installation without Swift

- Sequentially run a deployment pass on the controller nodes (`fuel-controller-01 ... fuel-controller-xx`), starting with the primary controller. No errors should appear during this deployment pass.
- Run an additional deployment pass on the primary controller only (`fuel-controller-01`) to finalize the Galera cluster configuration.
- Run a deployment pass on the Neutron node (`fuel-quantum`) to install the Neutron router.
- Run a deployment pass on every compute node (`fuel-compute-01 ... fuel-compute-xx`) - unlike the controllers these nodes may be deployed in parallel.

Example 4: The simplest OpenStack installation: Controller + Compute on the same node

- Set the `node /fuel-controller-[\d+]/` variable in `site.pp` to match the hostname of the node on which you are going to deploy OpenStack. Set the `node /fuel-compute-[\d+]/` variable to **mismatch** the node name. Run a deployment pass on this node. No errors should appear during this deployment pass.
- Set the `node /fuel-compute-[\d+]/` variable in `site.pp` to match the hostname of the node on which you are going to deploy OpenStack. Set the `node /fuel-controller-[\d+]/` variable to **mismatch** the node name. Run a deployment pass on this node. No errors should appear during this deployment pass.

Testing OpenStack

Now that you've installed OpenStack, its time to take your new openstack cloud for a drive around the block. Follow these steps:

1. On the host machine, open your browser to

http://192.168.0.10/ (Adjust this value to your own `public_virtual_ip`.)

and login as nova/nova (unless you changed this information in `site.pp`)

1. Click the Project tab in the left-hand column.
2. Under Manage Compute, choose Access & Security to set security settings:
    1. Click Create Keypair and enter a name for the new keypair. The private key should download automatically; make sure to keep it safe.
    2. Click Access & Security again and click Edit Rules for the default Security Group. Add a new rule allowing TCP connections from port 22 to port 22 for all IP addresses using a CIDR of 0.0.0.0/0. (You can also customize this setting as necessary.) Click Add Rule to save the new rule.
    3. Add a second new rule allowing ICMP connections with a type and code of -1 to the default Security Group and click Add Rule to save.
1. Click Allocate IP To Project and add two new floating ips. Notice that they come from the pool specified in `config.yaml` and `site.pp`.
2. Click Images & Snapshots, then Create Image. Enter a name and specify the Image Location as https://launchpad.net/cirros/trunk/0.3.0/+download/cirros-0.3.0-x86_64-disk.img, with a Format of QCOW2. Check the Public checkbox.
3. The next step is to upload an image to use for creating VMs, but an OpenStack bug prevents you from doing this in the browser. Instead, log in to any of the controllers as root and execute the following commands:

   cd ~ source openrc glance image-create --name cirros --container-format bare --disk-format qcow2 --is-public yes --location https://launchpad.net/cirros/trunk/0.3.0/+download/cirros-0.3.0-x86_64-disk.img

1. Go back to the browser and refresh the page. Launch a new instance of this image using the tiny flavor. Click the Networking tab and choose the default `net04_ext` network, then click the Launch button.
2. On the instances page:
    1. Click the new instance and look at the settings.
    2. Click the Logs tab to look at the logs.
    3. Click the VNC tab to log in. If you see just a big black rectangle, the machine is in screensaver mode; click the grey area and press the space bar to wake it up, then login as `cirros/cubswin:)`.
    4. At the command line, enter `ifconfig -a | more` and see the assigned ip address.
    5. Enter `sudo fdisk -l` to see that no volume has yet been assigned to this VM.
1. On the Instances page, click Assign Floating IP and assign an IP address to your instance. You can either choose from one of the existing created IPs by using the pulldown menu or click the plus sign (+) to choose a network and allocate a new IP address.
    1. From your host machine, ping the floating ip assigned to this VM.
    2. If that works, try to `ssh cirros@floating-ip` from the host machine.
1. Back in the browser, click Volumes and Create Volume. Create the new volume, and attach it to the instance.
2. Go back to the VNC tab and repeat `fdisk -l` to see the new unpartitioned disk attached.

Now your new VM is ready to be used.

Production Considerations

Fuel simplifies the set up of an OpenStack cluster, affording you the ability to dig in and fully understand how OpenStack works. You can deploy on test hardware or in a virtualized environment and root around all you like, but when it comes time to deploy to production there are a few things to take into consideration.

In this section we will talk about such things including how to size your hardware and how to handle large-scale deployments.

Sizing Hardware

One of the first questions people ask when planning an OpenStack deployment is "what kind of hardware do I need?" There is no such thing as a one-size-fits-all answer, but there are straightforward rules to selecting appropriate hardware that will suit your needs. The Golden Rule, however, is to always accomodate for growth. With the potential for growth accounted for, you can move on to the actual hardware needs.

Many factors contribute to selecting hardware for an OpenStack cluster -- contact Mirantis for information on your specific requirements -- but in general, you will want to consider the following factors:

- Processing
- Memory
- Storage
- Networking

Your needs in each of these areas are going to determine your overall hardware requirements.

Processing

In order to calculate how much processing power you will need to aquire you will need to determine the number of VMs your cloud will support. You must also consider the average and maximum processor resources you will allocate to each VM. In the vast majority of deployments, the allocated resources will be the same for all of your VMs. However, if you are planning to create groups of VMs that have different requirements, you will need to calculate for all of them in aggregate. Consider this example:

- 100 VMs
- 2 EC2 compute units (2 GHz) average
- 16 EC2 compute units (16 GHz) max

To make it possible to provide the maximum CPU in this example you will need at least 5 cores (16 GHz/(2.4 GHz per core * 1.3 to adjust for hyperthreading)) per machine, and at least 84 cores ((100 VMs * 2 GHz per VM)/2.4 GHz per core) in total. If you were to select the Intel E5 2650-70 8 core CPU, that means you need 11 sockets (84 cores / 8 cores per socket). This breaks down to six dual core servers (12 sockets / 2 sockets per server), for a "packing density" of 17 VMs per server (102 VMs / 6 servers).

This process also accomodates growth since you now know what a single server using this CPU configuration can support. You can add new servers accounting for 17 VMs each as needed without having to re-calculate.

You will also need to take into account the following:

- This model assumes you are not oversubscribing your CPU.
- If you are considering Hyperthreading, count each core as 1.3, not 2.
- Choose a good value CPU that supports the technologies you require.

Memory

Continuing to use the example from the previous section, we need to determine how much RAM will be required to support 17 VMs per server. Let's assume that you need an average of 4GBs of RAM per VM with dynamic allocation for up to 12GBs for each VM. Calculating that all VMs will be using 12GBs of RAM requires that each server have 204GBs of available RAM.

You must also consider that the node itself needs sufficient RAM to accomodate core OS operations as well as RAM for each VM container (not the RAM allocated to each VM, but the memory the core OS uses to run the VM). The node's OS must run it's own operations, schedule proccesses, allocate dynamic resources, and handle network operations, so giving the node itself at least 16GBs or more RAM is not unreasonable.

Considering that the RAM we would consider for servers comes in 4GB, 8GB, 16GB, and 32GB sticks, we would need a total of 265GBs of RAM installed per server. For an average 2-CPU server board you get 16-24 RAM slots. To have 256GBs installed you would need sixteen 16GB sticks of RAM to satisfy your RAM needs for up to 17 VMs requiring dynamic allocation up to 12GBs and to support all core OS requirements.

You can adjust this calculation based on your needs.

Storage Space

When it comes to disk space there are several types that you need to consider:

- Ephemeral (the local drive space for a VM)
- Persistent (the remote volumes that can be attached to a VM)
- Object Storage (such as images or other objects)

As far as local drive space that must reside on the compute nodes, in our example of 100 VMs we make the following assumptions:

- 150 GB local storage per VM
- 5 TB total of local storage (100 VMs * 50 GB per VM)
- 500 GB of persistent volume storage per VM
- 50 TB total persistent storage

Returning to our already established example, we need to figure out how much storage to install per server. This storage will service the 17 VMs per server. If we are assuming 50GBs of storage for each VMs drive container, then we would need to install 2.5TBs of storage on the server. Since most servers have anywhere from 4 to 32 2.5" drive slots or 2 to 12 3.5" drive slots, depending on server form factor (i.e., 2U vs. 4U), you will need to consider how the storage will be impacted by the intended use.

If storage impact is not expected to be significant, then you may consider using unified storage. For this example a single 3TB drive would provide more than enough storage for 17 150GB VMs. If speed is really not an issue, you might even consider installing two or three 3TB drives and configure a RAID-1 or RAID-5 for redundancy. If speed is critical, however, you will likely want to have a single hardware drive for each VM. In this case you would likely look at a 3U form factor with 24-slots.

Don't forget that you will also need drive space for the node itself, and don't forget to order the correct backplane that supports the drive configuration that meets your needs. Using our example specifications and assuming that speed it critical, a single server would need 18 drives, most likely 2.5" 15,000RPM 146GB SAS drives.

Throughput

As far as throughput, that's going to depend on what kind of storage you choose. In general, you calculate IOPS based on the packing density (drive IOPS * drives in the server / VMs per server), but the actual drive IOPS will depend on the drive technology you choose. For example:

- 3.5" slow and cheap (100 IOPS per drive, with 2 mirrored drives)
    - 100 IOPS * 2 drives / 17 VMs per server = 12 Read IOPS, 6 Write IOPS
- 2.5" 15K (200 IOPS, 4 600 GB drive, RAID 10)
    - 200 IOPS * 4 drives / 17 VMs per server = 48 Read IOPS, 24 Write IOPS
- SSD (40K IOPS, 8 300 GB drive, RAID 10)
    - 40K * 8 drives / 17 VMs per server = 19K Read IOPS, 9.5K Write IOPS

Clearly, SSD gives you the best performance, but the difference in cost between SSDs and the less costly platter-based solutions is going to be signficant, to say the least. The acceptable cost burden is determined by the balance between your budget and your performance and redundancy needs. It is also important to note that the rules for redundancy in a cloud environment are different than a traditional server installation in that entire servers provide redundancy as opposed to making a single server instance redundant.

In other words, the weight for redundant components shifts from individual OS installation to server redundancy. It is far more critical to have redundant power supplies and hot-swappable CPUs and RAM than to have redundant compute node storage. If, for example, you have 18 drives installed on a server and have 17 drives directly allocated to each VM installed and one fails, you simply replace the drive and push a new node copy. The remaining VMs carry whatever additional load is present due to the temporary loss of one node.

Remote storage

IOPS will also be a factor in determining how you plan to handle persistent storage. For example, consider these options for laying out your 50 TB of remote volume space:

- 12 drive storage frame using 3 TB 3.5" drives mirrored
    - 36 TB raw, or 18 TB usable space per 2U frame
    - 3 frames (50 TB / 18 TB per server)
    - 12 slots x 100 IOPS per drive = 1200 Read IOPS, 600 Write IOPS per frame
    - 3 frames x 1200 IOPS per frame / 100 VMs = 36 Read IOPS, 18 Write IOPS per VM
- 24 drive storage frame using 1TB 7200 RPM 2.5" drives
    - 24 TB raw, or 12 TB usable space per 2U frame
    - 5 frames (50 TB / 12 TB per server)
    - 24 slots x 100 IOPS per drive = 2400 Read IOPS, 1200 Write IOPS per frame
    - 5 frames x 2400 IOPS per frame / 100 VMs = 120 Read IOPS, 60 Write IOPS per frame

You can accomplish the same thing with a single 36 drive frame using 3 TB drives, but this becomes a single point of failure in your cluster.

Object storage

When it comes to object storage, you will find that you need more space than you think. For example, this example specifies 50 TB of object storage. Easy right? Not really. Object storage uses a default of 3 times the required space for replication, which means you will need 150 TB. However, to accommodate two hands-off zones, you will need 5 times the required space, which actually means 250 TB. The calculations don't end there. You don't ever want to run out of space, so "full" should really be more like 75% of capacity, which means you will need a total of 333 TB, or a multiplication factor of 6.66.

Of course, that might be a bit much to start with; you might want to start with a happy medium of a multiplier of 4, then acquire more hardware as your drives begin to fill up. That calculates to 200 TB in our example. So how do you put that together? If you were to use 3 TB 3.5" drives, you could use a 12 drive storage frame, with 6 servers hosting 36 TB each (for a total of 216 TB). You could also use a 36 drive storage frame, with just 2 servers hosting 108 TB each, but its not recommended due to the high cost of failure to replication and capacity issues.

Networking

Perhaps the most complex part of designing an OpenStack cluster is the networking. An OpenStack cluster can involve multiple networks even beyond the Public, Private, and Internal networks. Your cluster may involve tenant networks, storage networks, multiple tenant private networks, and so on. Many of these will be VLANs, and all of them will need to be planned out in advance to avoid configuration issues.

In terms of the example network, consider these assumptions:

- 100 Mbits/second per VM
- HA architecture
- Network Storage is not latency sensitive

In order to achieve this, you can use two 1Gb links per server (2 x 1000 Mbits/second / 17 VMs = 118 Mbits/second). Using two links also helps with HA. You can also increase throughput and decrease latency by using 2 10 Gb links, bringing the bandwidth per VM to 1 Gb/second, but if you're going to do that, you've got one more factor to consider.

Scalability and oversubscription

It is one of the ironies of networking that 1Gb Ethernet generally scales better than 10Gb Ethernet -- at least until 100Gb switches are more commonly available. It's possible to aggregate the 1Gb links in a 48 port switch, so that you have 48 1Gb links down, but 4 10GB links up. Do the same thing with a 10Gb switch, however, and you have 48 10Gb links down and 4 100Gb links up, resulting in oversubscription.

Like many other issues in OpenStack, you can avoid this problem to a great extent with careful planning. Problems only arise when you are moving between racks, so plan to create "pods", each of which includes both storage and compute nodes. Generally, a pod is the size of a non-oversubscribed L2 domain.

Hardware for this example

In this example, you are looking at:

- 2 data switches (for HA), each with a minimum of 12 ports for data (2 x 1Gb links per server x 6 servers)
- 1 1Gb switch for IPMI (1 port per server x 6 servers)

- Optional Cluster Management switch, plus a second for HA

Because your network will in all likelihood grow, it's best to choose 48 port switches. Also, as your network grows, you will need to consider uplinks and aggregation switches.

Summary

In general, your best bet is to choose a 2 socket server with a balance in I/O, CPU, Memory, and Disk that meets your project requirements. Look for a 1U R-class or 2U high density C-class server. Some good options from Dell for compute nodes include:

- Dell PowerEdge R620
- Dell PowerEdge C6220 Rack Server
- Dell PowerEdge R720XD (for high disk or IOPS requirements)

You may also want to consider systems from HP (http://www.hp.com/servers) or from a smaller systems builder like Aberdeen, a manufacturer that specializes in powerful, low-cost systems and storage servers (http://www.aberdeeninc.com).

Redeploying An Environment

Because Puppet is additive only, there is no ability to revert changes as you would in a typical application deployment. If a change needs to be backed out, you must explicitly add a configuration to reverse it, check the configuration in, and promote it to production using the pipeline. This means that if a breaking change does get deployed into production, typically a manual fix is applied, with the proper fix subsequently checked into version control.

Fuel offers the ability to isolate code changes while developing a deployment and minimizes the headaches associated with maintaining multiple configurations through a single Puppet Master by creating what are called environments.

Environments

Puppet supports assigning nodes 'environments'. These environments can be mapped directly to your development, QA and production life cycles, so it’s a way to distribute code to nodes that are assigned to those environments.

- On the Master/Server Node

  The Puppet Master tries to find modules using its `modulepath` setting, which by default is `/etc/puppet/modules`. It is common practice to set this value once in your `/etc/puppet/puppet.conf`. Environments expand on this idea and give you the ability to use different settings for different configurations.

  For example, you can specify several search paths. The following example dynamically sets the `modulepath` so Puppet will check a per-environment folder for a module before serving it from the main set:

```
[master]
  modulepath = $confdir/$environment/modules:$confdir/modules

[production]
  manifest   = $confdir/manifests/site.pp

[development]
  manifest   = $confdir/$environment/manifests/site.pp
```

- On the Agent Node

  Once the agent node makes a request, the Puppet Master gets informed of its environment. If you don’t specify an environment, the agent uses the default `production` environment.

  To set an environment agent-side, just specify the environment setting in the `[agent]` block of `puppet.conf`:

```
[agent]
  environment = development
```

Deployment pipeline

- Deploy

  In order to deploy multiple environments that don't interfere with each other, you should specify the `$deployment_id` option in `/etc/puppet/manifests/site.pp`. It should be an even integer value in the range of 2-254.

This value is used in dynamic environment-based tag generation. Fuel applies that tag globally to all resources on each node. It is also used for the keepalived daemon, which evaluates a unique `virtual_router_id`.

- Clean/Revert

  At this stage you just need to make sure the environment has the original/virgin state.

- Puppet node deactivate

This will ensure that any resources exported by that node will stop appearing in the catalogs served to the agent nodes:

```
puppet node deactivate <node>
```

where <node> is the fully qualified domain name as seen in `puppet cert list --all`.

You can deactivate nodes manually one by one, or execute the following command to automatically deactivate all nodes:

```
cert list --all | awk '! /DNS:puppet/ { gsub(/", "", $2); print $2}' | xargs puppet node deactivate
```

- Redeploy

  Start the puppet agent again to apply a desired node configuration.

Links

- http://puppetlabs.com/blog/a-deployment-pipeline-for-infrastructure/
- http://docs.puppetlabs.com/guides/environment.html

Large Scale Deployments

When deploying large clusters -- those of 100 nodes or more -- there are two basic bottlenecks:

- Certificate signing requests and Puppet Master/Cobbler capacity
- Downloading of operating systems and other software

Careful planning is key to eliminating these potential problem areas, but there's another way. If you are deploying Fuel 3.1 from the ISO, Fuel takes care of these problems through caching and orchestration. We feel, however, that it's always good to have a sense of how to solve these problems should they appear.

Certificate signing requests and Puppet Master/Cobbler capacity

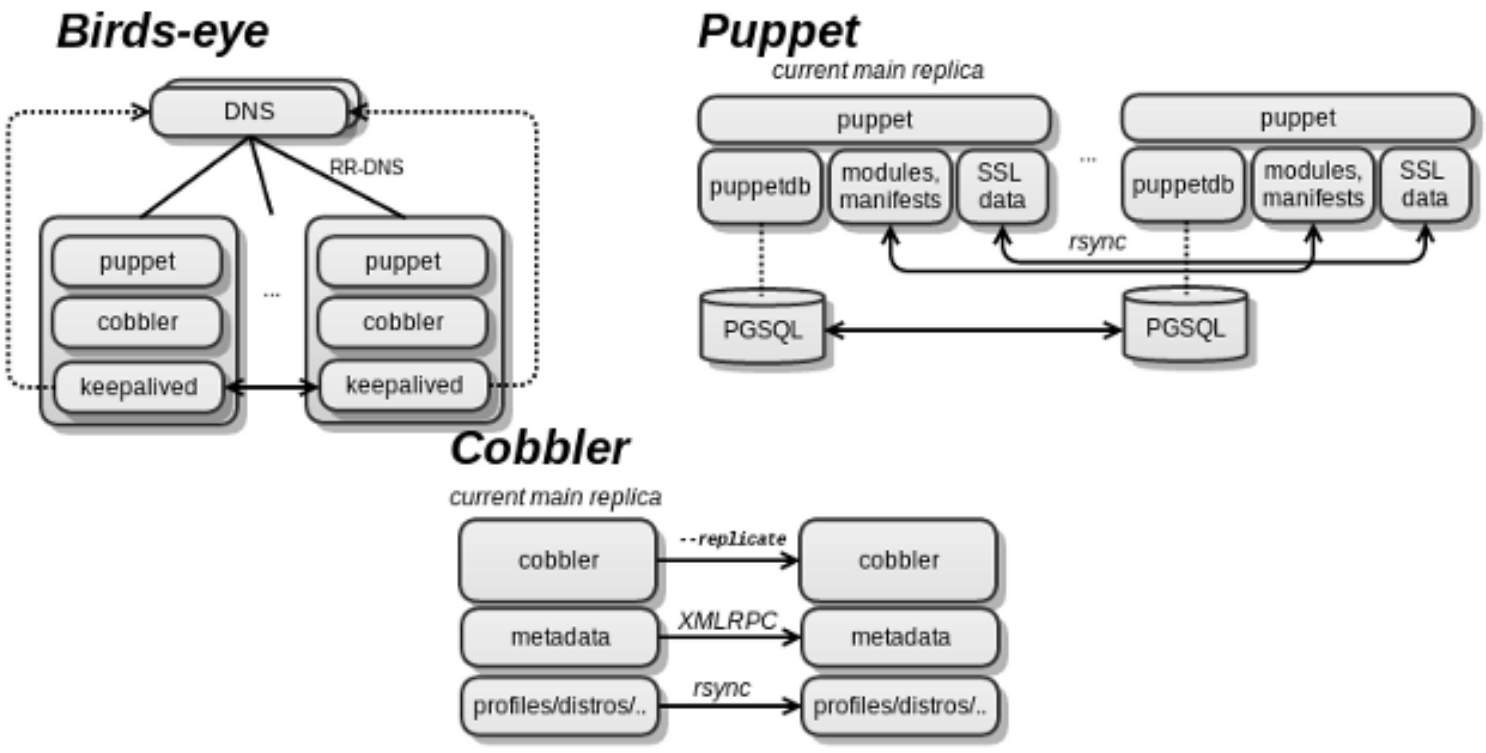
When deploying a large cluster, you may find that Puppet Master begins to have difficulty when you start exceeding 20 or more simultaneous requests. Part of this problem is because the initial process of requesting and signing certificates involves *.tmp files that can create conflicts. To solve this problem, you have two options: reduce the number of simultaneous requests, or increase the number of Puppet Master/Cobbler servers.

The number of simultaneous certificate requests that are active can be controlled by staggering the Puppet agent run schedule. This can be accomplished through orchestration. You don't need extreme staggering -- 1 to 5 seconds will do -- but if this method isn't practical, you can increase the number of Puppet Master/Cobbler servers.

If you're simply overwhelming the Puppet Master process and not running into file conflicts, one way to get around this problem is to use Puppet Master with Thin as the backend component and nginx as a front end component. This configuration dynamically scales the number of Puppet Master processes to better accommodate changing load.

You can find sample configuration files for nginx and puppetmasterd at [CONTENT NEEDED HERE].

You can also increase the number of servers by creating a cluster that utilizes a round robin DNS configuration through a service like HAProxy. You will need to ensure that these nodes are kept in sync. For Cobbler, that means a combination of the --replicate switch, XMLRPC for metadata, rsync for profiles and distributions. Similarly, Puppet Master and PuppetDB can be kept in sync with a combination of rsync (for modules, manifests, and SSL data) and database replication.



Downloading of operating systems and other software

Large deployments can also suffer from a bottleneck in terms of the additional traffic created by downloading software from external sources. One way to avoid this problem is by increasing LAN bandwidth through bonding multiple gigabit interfaces. You might also want to consider 10G Ethernet trunking between infrastructure switches using CAT-6a or fiber to improve backend speeds to reduce latency and provide more overall pipe. (See "Sizing Hardware" for more information on choosing networking equipment.)

Another option is to prevent the need to download so much data in the first place using either apt-cacher to cache frequently downloaded packages or to set up a private repository. The downside of using your own repository, however, is that you have to spend more time manually updating it. Apt-cacher automates this process. To use apt-cacher, the kickstart that Cobbler sends to each node should specify Cobbler's IP address and the apt-cacher port as the proxy server. This will prevent all of the nodes from having to download the software individually.

Contact Mirantis for information on creating a private repository.

FAQ (Frequently Asked Questions)

Known Issues and Workarounds

RabbitMQ Cluster Restart Issues Following A Systemwide Failure

Issue: As a rule of thumb, all RabbitMQ nodes must not be shut down simultaneously. RabbitMQ requires that after a full shutdown of the cluster, the first node brought up should be the last one to shut down, but it's not always possible to know which node that is in the event of a power outage or similar event. Versions 2.1 and later of Fuel solve this problem by managing the restart of available nodes, so you should not experience difficulty with this issue.

If you are still using previous versions of Fuel, the following describes how Fuel 2.1 works around this problem which, in turn, you can use to perform the steps manually.

Workaround: There are 2 possible scenarios, depending on the results of the shutdown:

1. The RabbitMQ master node is alive and can be started.
2. It's impossible to start the RabbitMQ master node due to a hardware or system failure

Fuel 2.1 updates the `/etc/init.d/rabbitmq-server` init scripts for RHEL/Centos and Ubuntu to customized versions. These scripts attempt to start RabbitMQ twice, giving the RabbitMQ master node the necessary time to start after complete power loss. With the scripts in place, power up all nodes, then check to see whether the RabbitMQ server started on all nodes. All nodes should start automatically.

On the other hand, if the RabbitMQ master node has failed, the init script performs the following actions during the rabbitmq-server start. It moves the existing Mnesia database to a backup directory, and then makes the third and final attempt to start the RabbitMQ server. In this case, RabbitMQ starts with a clean database, and the live rabbit nodes assemble a new cluster. The script uses the current RabbitMQ settings to find the current Mnesia location and creates a backup directory in the same path as Mnesia, tagged with the current date.

So with the customized init scripts included in Fuel 2.1, in most cases RabbitMQ simply starts after complete power loss and automatically assembles the cluster, but you can manage the process yourself.

Background: See http://comments.gmane.org/gmane.comp.networking.rabbitmq.general/19792.

Galera cluster has no built-in restart or shutdown mechanism

Issue: A Galera cluster cannot be simply started or stopped. It is designed to work continuously.

Workaround: Galera, as high availability software, does not include any built-in full cluster shutdown or restart sequence. It is supposed to be running on a 24/7/365 basis. On the other hand, deploying, updating or restarting Galera may lead to different issues. This guide is intended to help avoid some of these issues. Regular Galera cluster startup includes a combination of the procedures described below. These procedures, with some differences, are performed by Fuel manifests.

Stopping a single Galera node There is no dedicated Galera process - Galera works inside the MySQL server process. The MySQL server should be patched with Galera WSREP patch to be able to work as Galera cluster.

All Galera stop steps listed below are automatically performed by the mysql init script supplied by Fuel installation manifests, so in most cases it should be enough to perform the first step only. In case even init script fails in some (rare, as we hope) circumstances, repeat step 2 manually.

1. **Run `service mysql stop`.**

   Wait 15-30 seconds to ensure all MySQL processes are shut down.

2. **Run `ps -ef | grep mysql` and stop ALL(!) mysqld and mysqld_safe processes.**

   - Wait 20 seconds and run `ps -ef | grep mysql` again to see if any mysqld processes have restarted.
   - Stop or kill any new mysqld or mysqld_safe processes.

   It is very important to stop all MySQL processes. Galera uses `mysqld_safe` and it may start additional MySQL processes. So even if you don't immediately see any running processes, additional processes may be already starting. That is why we check running processes twice. `mysqld_safe` has a default timeout 15 seconds before processes restart. If, after that time, `mysqld` processes are running, the node may be considered shut down.

If there was nothing to kill and all MySQL processes stopped after the `service mysql stop` command, the node may be considered shut down gracefully.

Stop the Galera cluster A Galera cluster is a master-master replication cluster. Therefore, it is always in the process of synchronization.

The recommended way to stop the cluster involves the following steps:

1. Stop all requests to the cluster from outside. Under heavy load, a default Galera non-synchronized cache may be up to 1 Gb; you may have to wait until every node is fully synced to shut the cluster down.
2. Select the first node to shut down. In general, it's better to start with the non-primary nodes. Connect to this node with the mysql console.
3. Run `show status like 'wsrep_local_state%';`

   If it is "Synced", then you may start the shutdown node procedure.

   If the node is non-synchronized, you may still shut it down, but make sure you don't start a new cluster operation from this node in the future.

4. In mysql console, run the following command:

```
SET GLOBAL wsrep_on='OFF';
```

Replication stops immediately after the `wsrep_on` variable is set to "OFF", so avoid making any changes to the node after this changing this setting.

5. Exit from the mysql console.
6. Follow the steps described in *Stopping a single Galera node* to stop the node altogether.

Repeat these instructions for each remaining node in the cluster.

Remember which node you are going to shut down last -- ideally, it should be the primary node in the synced state. This is the node you should start first when you decide to continue cluster operation.

Starting Galera and creating a new cluster Galera writes its state to file the file `grastate.dat`, residing in the location specified in the `wsrep_data_home_dir` variable. This variable defaults to `mysql_real_data_home`, and Fuel OpenStack deployment manifests use this default location, creating the file at `/var/lib/mysql/grastate.dat`.

In the case of an unexpected cluster shutdown, this file can be useful for finding the node with the most recent commit. Simply compare the "UUID" values of `grastat.dat` from every node. The greater "UUID" value indicates which node has the latest commit.

If the cluster was shut down gracefully and last shut down node is known, simply perform the steps below to start up the cluster. Alternatively, you can find the node with the most recent commit using the `grastat.dat` files and start the cluster operation from that node.

1. Ensure that all Galera nodes are shut down.

   Any running nodes will be outside the new cluster untill restart, which could affect data integrity.

2. Select the primary node.

   This node is supposed to start first. It creates a new cluster ID and a new last commit UUID (the `wsrep_cluster_state_uuid` variable represents this UUID inside the MySQL process). Fuel deployment manifests with default settings set up `fuel-controller-01` to be both the primary Galera cluster node and the first deployed OpenStack controller. * Open `/etc/mysql/conf.d/wsrep.cnf` * Set empty cluster address as follows (including quotation marks):

   ```
   wsrep_cluster_address="gcomm://"
   ```

   - Save changes to the config file.

3. Run the `service mysql start` command on the first primary node or restart MySQL if there were configuration changes to `wsrep.cnf`.

   - Connect to MySQL server.
   - Run the `SET GLOBAL wsrep_on='ON';` to start replication within the new cluster. This variable can also be set by editing the `wsrep.cnf` file.
   - Check the new cluster status by running the following command: `show status like 'wsrep%';`
     - `wsrep_local_state_comment` should be "Synced"
     - `wsrep_cluster_status` should be "Primary"
     - `wsrep_cluster_size` should be "1", as this is the only cluster that's been started so far.
     - `wsrep_incoming_addresses` should include only the address of the current node.

4. Select one of the secondary nodes.

   - Check its `/etc/mysql/conf.d/wsrep.cnf` file.
     - The `wsrep_cluster_address="gcomm://node1,node2"` variable should include the name or IP address of the already started primary node. Otherwise, this node will definitely fail to start.

       Note. *Due to a Galera bug, do not include a node's own name and address in the ``wsrep_cluster_address`` specified for that node; while each Galera node attempts to exclude its own address, sometimes it fails. In this case, the Galera node fails to start, with a "Cannot open channel..." error in* **/etc/log/mysqld.log**

       In the case of OpenStack deployed by Fuel manifests with default settings (2 controllers), Fuel automatically removes local names and IP addresses from gcomm strings on every node to prevent a node from attempting to connect to itself. This parameter should look like this:

       ```
       wsrep_cluster_address="gcomm://fuel-controller-01:4567"
       ```

   - If `wsrep_cluster_address` is set correctly, run `rm -f /var/lib/mysql/grastate.dat` and then `service mysql start` on this node.

5. Connect to any node with mysql and run `show status like 'wsrep%';` again.

   - `wsrep_local_state_comment` should finally change from "Donor/Synced" or other statuses to "Synced".

Time to sync may vary depending on the database size and connection speed.

- `wsrep_cluster_status` should be "Primary" on both nodes.

Galera is a master-master replication cluster and every node becomes primary by default (i.e. master). Galera also supports master-slave configuration for special purposes. Slave nodes have the "Non-Primary" value for `wsrep_cluster_status`.

- `wsrep_cluster_size` should be "2", since we have just added one more node to the cluster.
- `wsrep_incoming_addresses` should include the addresses of both started nodes.

Note: State transfer is a heavy operation not only on the joining node, but also on the donor. In particular, the state donor may be not able to serve client requests, or it just plain may be slow.

6. Repeat step 4 on all remaining controllers

   If all secondary controllers are started successfully and became synced and you do not plan to restart the cluster in the near future, it is strongly recommended that you change the `wsrep` configuration settings on the first controller.

   - Open file `/etc/mysql/conf.d/wsrep.cnf`.
   - Set `wsrep_cluster_address=` to the same value (node list) that is used for every secondary controller.

   In case of OpenStack deployed by Fuel manifests with default settings (2 controllers), on every operating controller this parameter should finally look like

   ```
   wsrep_cluster_address="gcomm://fuel-controller-01:4567,fuel-controller-02:4567"
   ```

   This step is important for future failures or maintenance procedures. If the Galera primary controller node is restarted for any reason, if it has the empty "gcomm" value (i.e. `wsrep_cluster_address="gcomm://"`), it creates a new cluster and exits the existing cluster. The existing cluster nodes may also stop receiving requests and the synchronization process to prevent data de-synchronization issues.

Note:

Starting wtih mysql version 5.5.28_wsrep23.7 (Galera version 2.2), Galera cluster supports an additional start mode. Instead of setting `wsrep_cluster_address="gcomm://"`, on the first node one can set the following URL for cluster address:

```
wsrep_cluster_address="gcomm://node1,node2:port2,node3?pc.wait_prim=yes"
```

where `nodeX` is the name or IP address of one of available nodes, with optional port.

Therefore, every Galera node may have the same configuration file with the list of all nodes. It is designed to eliminate all configuration file changes on the first node after the cluster is started.

After the nodes are started, with mysql one may set the `pc.bootstrap=1` flag to the node which should start the new cluster and become the primary node. All other nodes should automatically perform initial synchronization with this new primary node. This flag may be also provided for a single selected node via the `wsrep.cnf` configuration file as follows:

```
wsrep_cluster_address="gcomm://node1,node2:port2,node3?pc.wait_prim=yes&pc.bootstrap=1"
```

Unfortunately, due to a bug in the mysql init script (<https://bugs.launchpad.net/codership-mysql/+bug/1087368>), the bootstrap flag is completely ignored in Galera 2.2 (wsrep_2.7). So, to start a new cluster, one should use the old way with an empty `gcomm://` URL. All other nodes may have both the single node and multiple node list in the `gcomm` URL, the bug affects only the first node - the one that starts the new cluster. Please note also that nodes with non-empty `gcomm` URL may start only if at least one of the nodes listed in `gcomm://node1,node2:port2,node3` is already started and is available for initial synchronization. For every starting Galera node it is enough to have at least one working node name/address to get full information about the cluster structure and to perform initial synchronization. Fuel deployment manifests with default settings may or may not set:

```
wsrep_cluster_address="gcomm://"
```

on the primary node (first deployed OpenStack controller) and node list like:

```
wsrep_cluster_address="gcomm://fuel-controller-01:4567,fuel-controller-02:4567"
```

on every secondary controller. Therefore, it is a good idea to check these parameters after the deployment is finished.

Note:

A Galera cluster is a very democratic system. As it is a master-master cluster, every primary node equals to other primary nodes. Primary nodes with the same sync state (same `wsrep_cluster_state_uuid` value) form the so called quorum - the majority of primary nodes with the same `wsrep_cluster_state_uuid`. Normally, one of the controllers gets a new commit, increases its `wsrep_cluster_state_uuid` value and performs synchronization with other nodes. If one of primary controllers fails, the Galera cluster continues serving requests as long as the quorum exists. Exit of the primary controller from the cluster equals a failure, because after exit this controller has a new cluster ID and a `wsrep_cluster_state_uuid` value less than the same value on the working nodes. So 3 working primary controllers are the very minimal Galera cluster size. The recommended Galera cluster size is 6 controllers.

Fuel deployment manifests with default settings deploy a non-recommended Galera configuration with 2 controllers only. This is suitable for testing purposes, but not for production deployments.

Restarting an existing cluster after failure

Continuing a Galera cluster after a power failure or other types of breakdown basically consists of two steps:

- Backing up every node
- Finding the node with the most recent non-damaged replica.

- Helpful tip: add `wsrep_provider_options="wsrep_on = off;"` to the `/etc/mysql/conf.d/wsrep.cnf` configuration file.

After these steps simply perform the **Start Galera and create a new cluster** procedure, starting from the node with the most recent non-damaged replica.

Useful links

- Galera documentation from Galera authors:
  - http://www.codership.com/wiki/doku.php
- Actual Galera and WSREP patch bug list and official Galera/WSREP bug tracker:
  - https://launchpad.net/codership-mysql
  - https://launchpad.net/galera
- One of recommended Galera cluster robust configurations:
  - http://wiki.vps.net/vps-net-features/cloud-servers/template-information/galeramysql-recommended-cluster-configuratio
- Why we use Galera:
  - http://openlife.cc/blogs/2011/july/ultimate-mysql-high-availability-solution
- Other questions (seriously, sometimes there is not enough info about Galera available in the official Galera docs):
  - http://www.google.com

Common Technical Issues

Puppet fails with

```
err: Could not retrieve catalog from remote server: Error 400 on SERVER: undefined method 'fact_merge' for nil:NilCl
```

  - This is a Puppet bug. See: http://projects.puppetlabs.com/issues/3234
  - Workaround: `service puppetmaster restart`

2. Puppet client will never resend the certificate to Puppet Master. The certificate cannot be signed and verified.
   - This is a Puppet bug. See: http://projects.puppetlabs.com/issues/4680
   - **Workaround:**
     - On Puppet client:

```
rm -f /etc/puppet/ssl/certificate_requests/\*.pem
rm -f /etc/puppet/ssl/certs/\*.pem
```

     - On Puppet master:

```
rm -f /var/lib/puppet/ssl/ca/requests/\*.pem
```

3. The manifests are up-to-date under `/etc/puppet/manifests`, but Puppet master keeps serving the previous version of manifests to the clients. Manifests seem to be cached by the Puppet Master.

    - More information: https://groups.google.com/forum/?fromgroups=#!topic/puppet-users/OpCBjV1nR2M
    - Workaround: `service puppetmaster restart`

Timeout error for fuel-controller-XX when running `puppet-agent --test` to install OpenStack when using HDD instead of SSD

```
| Sep 26 17:56:15 fuel-controller-02 puppet-agent[1493]: Could not retrieve catalog from remote server: execution exp
| Sep 26 17:56:15 fuel-controller-02 puppet-agent[1493]: Not using cache on failed catalog
| Sep 26 17:56:15 fuel-controller-02 puppet-agent[1493]: Could not retrieve catalog; skipping run
```

- **Workaround: `vi /etc/puppet/puppet.conf`**
    - add: `configtimeout = 1200`

ent --test", the error messages below occur:

```
lib/puppet/lib]: Could not evaluate: Could not retrieve information from environment production source(s) puppet://fue
```

//projects.reductivelabs.com/issues/2244

```
t retrieve catalog from remote server: Error 400 on SERVER: stack level too deep
using cache on failed catalog
t retrieve catalog; skipping run
```

he second problem can be solved by rebooting Puppet master.

ge:

```
om remote server: Error 400 on SERVER: Failed to submit 'replace facts' command for fuel-pm to PuppetDB at fuel-pm:808
```

This message is often the result of one of the following:

- Firewall blocking the puppetdb port
- DNS issues with the hostname specified in your puppetdb.conf
- DNS issues with the ssl-host specified in your jetty.ini on the puppetdb server
- Workaround: If you are able to connect (e.g. via telnet) to port 8081 on the puppetdb machine, puppetdb is running. To try and isolate the problem, add the following to `/etc/puppetdb/conf.d/jetty.ini`:

```
certificate-whitelist = /etc/puppetdb/whitelist.txt
```

Be sure to list all aliases for the machine in that file.

Creating the XFS partition

In most cases, Fuel creates the XFS partition for you. If for some reason you need to create it yourself, use this procedure:

1. Create the partition itself:

```
fdisk /dev/sdb
n(for new)
p(for partition)
<enter> (to accept the defaults)
<enter> (to accept the defaults)
w(to save changes)
```

2. Initialize the XFS partition:

```
mkfs.xfs -i size=1024 -f /dev/sdb1
```

3. For a standard swift install, all data drives are mounted directly under /srv/node, so first create the mount point:

```
mkdir -p /srv/node/sdb1
```

4. Finally, add the new partition to fstab so it mounts automatically, then mount all current partitions:

```
echo "/dev/sdb1 /srv/node/sdb1 xfs
noatime,nodiratime,nobarrier,logbufs=8 0 0" >> /etc/fstab
mount -a
```

Redeploying a node from scratch

Compute and Cinder nodes in an HA configuration and controller in any configuration cannot be redeployed without completely redeploying the cluster. However, in a non-HA situation you can redeploy a compute or Cinder node. To do so, follow these steps:

1. Remove the certificate for the node by executing the command `puppet cert clean <hostname>` on fuel-pm.
2. Re-boot the node over the network so it can be picked up by cobbler.
3. Run the puppet agent on the target node using `puppet agent --test`.

Other Questions

1. **[Q]** Why did you decide to provide OpenStack packages through your own repository?

   [A] We are fully committed to providing our customers with working and stable bits and pieces in order to make successful OpenStack deployments. Please note that we do not distribute our own version of OpenStack; we rather provide a plain vanilla distribution. As such, there is no vendor lock-in. For convenience, our repository maintains the history of OpenStack packages certified to work with our Puppet manifests.

   The advantage of this approach is that you can install any OpenStack version you want. If you are running Essex, just use the Puppet manifests which reference OpenStack packages for Essex from our repository. With each new release we add new OpenStack packages to our repository and created a separate branch with the Puppet manifests (which, in turn, reference these packages) corresponding to each release. With EPEL this would not be possible, as that repository only keeps the latest version for OpenStack packages.