CCP Documentation

Release

OpenStack Foundation

Nov 15, 2017

Contents

CHAPTER 1

User docs

1.1 Quick Start

This guide provides a step by step instruction of how to deploy CCP on bare metal or a virtual machine.

1.1.1 Recommended Environment

CCP was tested on Ubuntu 16.04 x64. It will probably work on different OSes, but it's not officialy supported.

CCP was tested on the environment created by Kargo, via fuel-ccp-installer, which manages k8s, calico, docker and many other things. It will probably work on different setup, but it's not officialy supported.

Current tested version of different components are:

Component	Min Version	Max Version	Comment
Kubernetes	1.5.1	1.5.x	
Docker	1.10.0	1.13.x	
Calico-node	0.20.0	1.0.x	

Additionaly, you will need to have working kube-proxy, kube-dns and docker registry.

If you don't have a running k8s environment, please check out this guide

> **Warning:** All further steps assume that you already have a working k8s installation.

1.1.2 Deploy CCP

Install CCP CLI

Note: Some commands below may require root permissions and require a few packages to be installed by the provisioning underlay:

- python-pip
- python-dev
- python3-dev
- python-netaddr
- software-properties-common
- python-setuptools
- gcc

If you're deploying CCP from non-root user, make sure your user are in the `docker` group. Check if user is added to docker group

```
id -Gn | grep docker
```

If not added you can add your user to docker group via:

```
sudo usermod -a -G docker your_user_name
```

To clone the CCP CLI repo:

```
git clone https://git.openstack.org/openstack/fuel-ccp
```

To install CCP CLI and Python dependencies use:

```
sudo pip install fuel-ccp/
```

Create a local registry service (optional):

```
bash fuel-ccp/tools/registry/deploy-registry.sh
```

When you deploy a local registry using that script, the registry address is 127.0.0.1:31500.

Create CCP CLI configuration file:

```
cat > ~/.ccp.yaml << EOF
builder:
  push: True
registry:
  address: "127.0.0.1:31500"
repositories:
  skip_empty: True
EOF
```

If you're using some other registry, please use its address instead.

Append default topology and edit it, if needed:

```
cat fuel-ccp/etc/topology-example.yaml >> ~/.ccp.yaml
```

For example, you may want to install Stacklight to collect Openstack logs. See *Monitoring and Logging with StackLight* for the deployment of monitoring and logging services.

Append global CCP configuration:

```
cat >> ~/.ccp.yaml << EOF
configs:
    private_interface: eth0
    public_interface: eth1
    neutron:
      physnets:
        - name: "physnet1"
          bridge_name: "br-ex"
          interface: "ens8"
          flat: true
          vlan_range: "1001:1030"
          dpdk: false
EOF
```

Make sure to adjust it to your environment, since the network configuration of your environment may be different.

- `private_interface` - should point to eth with private ip address.
- `public_interface` - should point to eth with public ip address (you can use private iface here, if you want to bind all services to internal network)
- `neutron.physnets` - should contain description of Neutron physical networks. If only internal networking with VXLAN segmentation required, this option can be empty. `name` is name of physnet in Neutron. `bridge_name` is name of OVS bridge. `interface` should point to eth without ip addr. `flat` allow to use this network as flat, without segmentation. `vlan_range` is range of allowed VLANs, should be false if VLAN segmenantion is not allowed. `dpdk` if enabled for particular network, OVS will handle it via userspace DPDK

For the additional info about bootstrapping configuration please read the *Resource Bootstrapping*.

Append replicas configuration:

```
cat >> ~/.ccp.yaml << EOF
replicas:
  database: 3
  rpc: 3
  notifications: 1
EOF
```

This will sets the number of replicas to create for each service. We need 3 replicas for galera and rabbitmq cluster.

Fetch CCP components repos:

```
ccp fetch
```

Build CCP components and push them into the Docker Registry:

```
ccp build
```

Deploy OpenStack:

```
ccp deploy
```

If you want to deploy only specific components use:

```
ccp deploy -c COMPONENT_NAME1 COMPONENT_NAME2
```

For example:

```
ccp deploy -c etcd galera keystone memcached
```

Check deploy status

By default, CCP deploying all components into "ccp" k8s namespace. You could set context for all kubectl commands to use this namespace:

```
kubectl config set-context ccp --namespace ccp
kubectl config use-context ccp
```

Get all running pods:

```
kubectl get pod -o wide
```

Get all running jobs:

```
kubectl get job -o wide
```

Note: Deployment is successful when all jobs have "1" (Successful) state.

Deploying test OpenStack environment

Install openstack-client:

```
pip install python-openstackclient
```

openrc file for current deployment was created in the current working directory. To use it run:

```
source openrc-ccp
```

Run test environment deploy script:

```
bash fuel-ccp/tools/deploy-test-vms.sh -a create -n NUMBER_OF_VMS
```

This script will create flavor, upload cirrios image to glance, create network and subnet and launch bunch of cirrios based VMs.

Accessing horizon and nova-vnc

Currently, we don't have any external proxy (like Ingress), so, for now, we have to use k8s service "nodePort" feature to be able to access internal services.

Get nodePort of horizon service:

```
kubectl get service horizon -o yaml | awk '/nodePort: / {print $NF}'
```

Use external ip of any node in cluster plus this port to access horizon.

Get nodePort of nova-novncproxy service:

```
kubectl get service nova-novncproxy -o yaml | awk '/nodePort: / {print $NF}'
```

Take the url from Horizon console and replace "nova-novncproxy" string with an external IP of any node in cluster plus nodeport from the service.

Cleanup deployment

To cleanup your environment run:

```
ccp cleanup
```

This will delete all VMs created by OpenStack and destroy all neutron networks. After it's done it will delete all k8s pods in this deployment.

1.2 Monitoring and Logging with StackLight

This section provides information on deploying StackLight, the monitoring and logging system for CCP.

Warning: StackLight requires Kubernetes 1.4 or higher, and its deployment will fail with Kubernetes 1.3 and lower. So before deploying StackLight make sure you use an appropriate version of Kubernetes.

1.2.1 Overview

StackLight is composed of several components. Some components are related to logging, and others are related to monitoring.

The "logging" components:

- `heka` – for collecting logs
- `elasticsearch` – for storing/indexing logs
- `kibana` – for exploring and visualizing logs

The "monitoring" components:

- `stacklight-collector` – composed of Snap and Hindsight for collecting and processing metrics
- `influxdb` – for storing metrics as time-series
- `grafana` – for visualizing time-series

For fetching the StackLight repo (`fuel-ccp-stacklight`) and building the StackLight Docker images please refer to the *Quick Start* section as StackLight is not different from other CCP components for that matter. If you followed the *Quick Start* the StackLight images may be built already.

The StackLight Docker images are the following:

- `ccp/cron`
- `ccp/elasticsearch`
- `ccp/grafana`
- `ccp/heka`
- `ccp/hindsight`
- `ccp/influxdb`

- ccp/kibana

1.2.2 Deploy StackLight

The StackLight components are regular CCP components, so the deployment of StackLight is done through the CCP CLI like any other CCP component. Please read the *Quick Start* section and make sure the CCP CLI is installed and you know how to use it.

StackLight may be deployed together with other CCP components, or independently as a separate deployment process. You may also want to deploy just the "logging" components of StackLight, or just the "monitoring" components. Or you may want to deploy all the StackLight components at once.

In any case you will need to create StackLight-related roles in your CCP configuration file (e.g. `/etc/ccp/ccp.yaml`) and you will need to assign these roles to nodes.

For example:

```
nodes:
  node1:
    roles:
      - stacklight-backend
      - stacklight-collector
  node[2-3]:
    roles:
      - stacklight-collector
roles:
  stacklight-backend:
    - influxdb
    - grafana
  stacklight-collector:
    - stacklight-collector
```

In this example we define two roles: `stacklight-backend` and `stacklight-collector`. The role `stacklight-backend` is assigned to `node1`, and it defines where `influxdb` and `grafana` will run. The role `stacklight-collector` is assigned to all the nodes (`node1`, `node2` and `node3`), and it defines where `stacklight-collector` will run. In most cases you will want `stacklight-collector` to run on every cluster node, for node-level metrics to be collected for every node.

With this, you can now deploy `influxdb`, `grafana` and `stacklight-collector` with the following CCP command:

```
ccp deploy -c influxdb grafana stacklight-collector
```

Here is another example, in which both the "monitoring" and "logging" components will be deployed:

```
nodes:
  node1:
    roles:
      - stacklight-backend
      - stacklight-collector
  node[2-3]:
    roles:
      - stacklight-collector
roles:
  stacklight-backend:
    - influxdb
    - grafana
    - elasticsearch
```

```
    - kibana
  stacklight-collector:
    - stacklight-collector
    - heka
    - cron
```

And this is the command to use to deploy all the StackLight services:

```
ccp deploy -c influxdb grafana elasticsearch kibana stacklight-collector heka cron
```

To check the deployment status you can run:

```
kubectl --namespace ccp get pod -o wide
```

and check that all the StackLight-related pods have the `RUNNING` status.

1.2.3 Accessing the Grafana and Kibana interfaces

As already explained in *Quick Start* CCP does not currently include an external proxy (such as Ingress), so for now the Kubernetes `nodePort` feature is used to be albe to access services such as Grafana and Kibana from outside the Kubernetes cluster.

This is how you can get the node port for Grafana:

```
$ kubectl get service grafana -o yaml | awk '/nodePort: / {print $NF}'
31124
```

And for Kibana:

```
$ kubectl get service kibana -o yaml | awk '/nodePort: / {print $NF}'
31426
```

1.2.4 ElasticSearch cluster

Documentation above describes using elasticsearch as one node service without ability to scale — stacklight doesn't require elasticsearch cluster. This one node elasticsearch is master-eligible, so could be scaled with any another master, data or client node.

For more details about master, data and client node types please read elasticsearch node documentation.

CCP implementation of elasticsearch cluster contains three available services:

- `elasticsearch` — master-eligible service, represents master node;
- `elasticsearch-data` — data (non-master) service, represents data node, contains *elasticsearch-data* volume for storing data;
- `elasticsearch-client` — special type of coordinating only node that can connect to multiple clusters and perform search and other operations across all connected clusters. Represents tribe node type.

All these services can be scaled and deployed on several nodes with replicas - they will form cluster. It can be checked with command:

```
$ curl -X GET http://elasticsearch.ccp.svc.cluster.local:9200/_cluster/health?pretty
```

which will print total number of cluster nodes and number of data nodes. More detailed info about each cluster node called with command:

```
$ curl -X GET http://elasticsearch.ccp.svc.cluster.local:9200/_cluster/state?pretty
```

For example, we need elasticsearch cluster with 2 data nodes. Then, topology will be look like:

::

replicas: elasticsearch-data: 2 ...

nodes:

node1:

roles:

- controller

...

node[2-3]:

roles:

- es-data

roles:

es-data:

- elasticsearch-data

controller:

- elasticsearch
- elasticsearch-client

...

1.3 Configuration files

This section will describe configuration format used in CCP.

1.3.1 Understanding global and default configs

There are three config locations, which the CCP CLI uses:

1. `Global defaults` - fuel_ccp/resources/defaults.yaml in `fuel-ccp` repo.
2. `Component defaults` - service/files/defaults.yaml in each component repo.
3. `Global config` - Optional. For more information read the global_config.

Before deployment, CCP will merge all these files into one dict, using the order above, so "component defaults" will override "global defaults" and "global config" will override everything.

For example, one of common situations is to specify custom options for networking. To achieve user may overwrite options defined in `Global defaults` and `Component defaults` by setting new values in `Global config`.

File `fuel_ccp/resources/defaults.yaml` has follow settings:

```
configs:
  private_interface: eth0
  public_interface: eth1
  ...
```

And part of the `fuel-ccp-neutron/service/files/defaults.yaml` looks like:

```
configs:
  neutron:
    ...
    bootstrap:
      internal:
        net_name: int-net
        subnet_name: int-subnet
        network: 10.0.1.0/24
        gateway: 10.0.1.1
    ...
```

User may overwrite these sections by defining the following content in the ~/.ccp.yaml:

```
debug: true
configs:
  private_interface: ens10
  neutron:
    bootstrap:
      internal:
        network: 22.0.1.0/24
        gateway: 22.0.1.1
```

To validate these changes user needs to execute command `ccp config dump`. It will return final config file with changes, which user did. So output should contain the following changes:

```
debug: true
...
configs:
  private_interface: ens10     <----- it was changed
  public_interface: eth1       <----- it wasn't changed
  neutron:
    bootstrap:
      internal:
        net_name: int-net        <--- it wasn't changed
        subnet_name: int-subnet  <--- it wasn't changed
        network: 22.0.1.0/24   <----- it was changed
        gateway: 22.0.1.1      <----- it was changed
```

Global defaults

This is project wide defaults, CCP keeps it inside fuel-ccp repository in `fuel_ccp/resources/defaults.yaml` file. This file defines global variables, that is variables that are not specific to any component, like interface names.

Component defaults

Each component repository could contain a `service/files/defaults.yaml` file with default config for this component only.

Global config

See description in global_config.

1.4 Configuration key types

1.4.1 Overview

Each config could contain several keys. Each key has its own purpose and isolation, so you have to add your variable to the right key to make it work. For optimization description all keys will be splitted on several groups based on purpose.

1.4.2 CCP specific

Current list contains keys for configuration logging in the CCP CLI.

- *debug*
- *default_log_levels*
- *log_file*
- *verbose_level*

1.4.3 Build options

The biggest group of keys configures build process, i.e. *how to build*, *which sources and images to use*.

- *builder*
- *versions*
- *repositories*
- *sources*
- *url*
- *images*

1.4.4 Deployment Configuration

This group is dedicated to describe topology of deployment, configuration of the microservices and credentials for connecting to Kubernetes cluster.

- *configs*
- *secret_configs*
- *files*
- *kubernetes*
- *services*
- *nodes*
- *roles*

- *replicas*

1.4.5 Other specific variables

The last group includes keys, which should be described, but could not be a part of groups mentioned erlier.

- *registry*
- *action*
- *network_topology*
- *node_name*
- *pod_name*
- *address*

1.4.6 List of keys

debug

Isolation:

- Not used in any template file, only used by the CCP CLI.

Allowed content:

- Boolean value (default: False).

Option enable debug messages and tracebacks during **ccp** commands execution

default_log_levels

Isolation:

- Not used in any template file, only used by the CCP CLI.

Allowed content:

- Array of string values. Default value:

```
[
 'glanceclient=INFO',
 'keystoneauth=INFO',
 'neutronclient=INFO',
 'novaclient=INFO',
 'requests=WARN',
 'stevedore=INFO',
 'urllib3=WARN'
]
```

This array describes log levels for different components used by the CCP. Messages from these componenets will be written to **ccp** debug logs.

log_file

Isolation:

- Not used in any template file, only used by the CCP CLI.

Allowed content:

- String value (default: None).

Full path with file name for storing **ccp** execution logs. If only file name is specified, then CCP will try to find this file in the current directory.

verbose_level

Isolation:

- Not used in any template file, only used by the CCP CLI.

Allowed content:

- Integer value. (default: 1)

This option allows to specify verbose level for **ccp** debug logging.

builder

Isolation:

- Not used in any template file, only used by the CCP CLI for building images.

Allowed content:

- This key has the following list of sub-keys:

Name	Description	Schema	Default
workers	Number of the workers, which will be used during building component images.	integer	number of CPU in the system
keep_image_tree_consistent	Rebuild dependent images, if base image was rebuilt.	boolean	True
build_base_images_if_not_exist	Force base image building.	boolean	True
push	Push images to docker registry.	boolean	False
no_cache	Do not use docker caching during building images.	boolean	False

versions

Isolation:

- Used in Dockerfile.j2.
- Used in *Global Config* file.

Allowed content:

- Only versions of different software should be kept here.

For example:

```
versions:
 influxdb_version: "0.13.0"
```

So you could add this to influxdb Dockerfile.j2:

```
curl https://dl.influxdata.com/influxdb/releases/influxdb_{{ influxdb_version }}_
↪amd64.deb
```

repositories

Isolation:

- Not used in any template file, only used by the CCP CLI to fetch service repositories, e.g. fuel-ccp-* (nova, cinder and etc).

Detailed explanation can be found in repositories.

sources

Isolation:

- Used in Dockerfile.j2.
- Used in *Global Config* file.

Allowed content:

- This key has a restricted format, examples below.

Remote git repository example:

```
sources:
  openstack/keystone:
    git_url: https://github.com/openstack/keystone.git
    git_ref: master
```

Local git repository example:

```
sources:
  openstack/keystone:
    source_dir: /tmp/keystone
```

So you could add this to Dockerfile.j2:

```
{{ copy_sources("openstack/keystone", "/keystone") }}
```

CCP will use the chosen configuration, to copy git repository into Docker container, so you could use it later.

url

Isolation:

- Used in Dockerfile.j2.
- Used in *Global Config* file.

Allowed content:

- Only repos for artifacts (e.g. Deb, Pypi repos). Can be specific for different components.

Data which will be used by **ccp** during docker image building. For example for mariadb:

```
url:
  mariadb:
    debian:
      repo: "http://lon1.mirrors.digitalocean.com/mariadb/repo/10.1/debian"
      keyserver: "hkp://keyserver.ubuntu.com:80"
      keyid: "0xcbcb082a1bb943db"
```

images

Isolation:

- Not used in any template file, only used by the CCP CLI to build base images.

Allowed content:

- This key has the following list of sub-keys:

Name	Description	Schema	Default
namespace	Namespace which should be used for **ccp** related images.	string	ccp
tag	Tag for **ccp** related images.	string	latest
base_distro	Base image for building **ccp** images.	string	debian
base_tag	Tag of the base image for bulding **ccp** images.	string	jessie
base_images	Names of base images.	array of strings	['base']
maintainer	Maintainer of **ccp** images.	string	MOS Microservices <mos-microservices@mirantis.com>
image_specs	Extra keys for building images.	json	–

configs

Isolation:

- Used in service templates files (service/files/).
- Used in application definition file service/component_name.yaml.
- Used in *Global Config* file.

Allowed content:

- Any types of variables are allowed.

Example:

```
configs:
  keystone_debug: false
```

So you could add "{{ keystone_debug }}" variable to you templates, which will be rendered into "false" in this case.

secret_configs

Same as *configs*, but will be stored inside of k8s Secret instead of ConfigMap.

files

- Used in *Global Config* file.

> **Note:** This section is used in component repositories for configuration files references. In case *Global Config* usage is tricky for you, custom config files for a particular service can be set in ~/.ccp.yaml.

> **Warning:** This section has the different format from same section used in component defitinions (i.e. in fuel-ccp-* repositories).

Allowed content:

- Strict format mentioned below:

```
files:
 file_name: /path
```

kubernetes

Isolation:

- Not used in any template file, only used by the CCP CLI to operate with Kubernetes cluster.

Allowed content:

- This key has the following list of sub-keys:

Name	Description	Schema	Default
server	URL for accessing of Kubernetes API.	string	http://localhost:8080
namespace	Namespace which will be created and used for deploying Openstack.	string	ccp
ca_cert	Path of CA TLS certificate(s) used to verify the Kubernetes server's certificate.	string	–
key_file	Path of client key to use in SSL connection.	string	–
cert_file	Path of certificate file to use in SSL connection.	string	–
insecure	Explicitly allow **ccp** to perform "insecure SSL" (https) requests.	boolean	False
cluster_domain	Name of the cluster domain.	string	cluster.local

replicas

Isolation:

- Not used in any template file, only used by the CCP CLI to create a cluster topology.

Allowed content:

- JSON object where keys are service names with value equal number of replicas which should be run after deploy.

Note: For services defined with kind: DaemonSet replicas number can't be specified and will be always equal to number of nodes this service assigned to.

For example:

```
replicas:
  heat-engine: 3
```

services

Isolation:

- Not used in any template file, only used by the CCP CLI to create new services and connect them between each other.

Allowed content:

- This is a dict that contains definitions for dedicated services. Its keys are service names, values are dicts with the following keys:

Name	Description	Schema	Default
service_def	Name of the service definition associated with that service.	string	–
mapping	Dict to map service abstractions to defined services.	dict	–
configs	Config overrides for this particular service.	dict	–

You can find more information and examples in services page.

nodes

Isolation:

- Not used in any template file, only used by the CCP CLI to create a cluster topology.

Allowed content:

- This key contains a regular expression to match one or several nodes at once, example can be found in `fuel-ccp` git repository in `etc/topology-example.yaml` file. If your environment contains more than 9 nodes, you must explicitly specify the "end-of-line", because expression like node([1-5]|10|11) will also match node12, node13 etc. Example can be found in `fuel-ccp` git repository in `etc/topology-with-large-number-of-nodes.yaml` file. This key includes next two sub-keys:
  - *roles* sub-key, which contains a list of roles names. Example of such definition can be found in topology example file.
  - *configs* key, which defines dict of configs, specific for particular node and service. Configs serve to override global config defaults, for example, for variables, dependent on node hardware configuration. Example:

```
nodes:
  node[2-3]:
    roles:
      - openstack
    configs:
      nova:
        logging_debug: true
```

Note: It's very important: global configs merged with specific nodes

configs in lexicographic order, i.e. if you have override key *test* with value *2* for *node[1-3]* and with value *4 node[2-4]*, then *node2* will have key-value pair *(test, 4)* in configs.

roles

Isolation:

- Not used in any template file, only used by the CCP CLI to create a cluster topology.

Allowed content:

- The roles specified in the 'roles' key for node will apply to all matched nodes. If a node matches several 'nodes' keys, each with different roles, then roles from all keys will be added to node. Example can be found in the topology example file.

registry

Isolation:

- Not used in any template file, only used by the CCP CLI to configure docker registry, which will be used for deployment.

Allowed content:

- This key has the following list of sub-keys:

Name	Description	Schema	Default
address	Address of registry service.	string	–
insecure	Use insecure connection or not.	boolean	False
username	Username to access docker registry.	string	–
password	Password to access docker registry.	string	–
timeout	Value, which specifies how long the CCP waits response from registry.	integer	300

This is used to pass information for accessing docker registry. Example can be found in quickstart.

action

Warning: This option was deprecated in favor of CLI parameters, so please don't use it, because it will be removed in future.

network_topology

Isolation:

- Used in service templates files (service/files/).

Allowed content:

- This key is auto-created by entrypoint script and populated with container network topology, based on the following variables: `private_interface` and `public_interface`.

You could use it to get the private and public eth IP address. For example:

```
bind = "{{ network_topology["private"]["address"] }}"
listen = "{{ network_topology["public"]["address"] }}"
```

node_name

Isolation:

- Used in service templates files (service/files/).

Allowed content:

- This key is auto-created by entrypoint script based on kubernetes downward api.

You could use it to get the name of the node on which container is deployed. For example:

```
my_node = "{{ node_name }}"
```

pod_name

Isolation:

- Used in service templates files (service/files/).

Allowed content:

- This key is auto-created by entrypoint script based on kubernetes downward api.

You could use it to get the name of the pod on which container is deployed. For example:

```
my_pod = "{{ pod_name }}"
```

address

Isolation:

- Used in service templates files (service/files/).
- Used in application definition file service/component_name.yaml.

Allowed content:

- This is a function with the following params:

Parameter	Description	Required	Default
service	Name of the service.	True	–
port	Add port to the url. Port config section should be specified.	False	–
external	Use external url instead of internal.	False	False
with_scheme	Add scheme to the url.	False	False

You could use it to get address of the service. For example:

```
service_address = "{{ address('keystone', keystone.public_port, external=True, with_
↪scheme=True) }}"
```

1.5 Resource Bootstrapping

Current section describes what and how can be bootstrapped in the CCP. There are several services, which have bootstrapping. It's:

- *Network bootstrapping*

- *images*
- *Flavor bootstrapping*

1.5.1 Network bootstrapping

This section allows to configure internal and external networking in neutron. Snippet below demonstrates all available options:

```
configs:
  neutron:
    bootstrap:
      internal:
        enable: true
        net_name: int-net
        subnet_name: int-subnet
        network: 10.0.1.0/24
        gateway: 10.0.1.1
      external:
        enable: false
        net_name: ext-net
        subnet_name: ext-subnet
        physnet: changeme
        network: changeme
        gateway: changeme
        nameserver: changeme
        pool:
          start: changeme
          end: changeme
      router:
        name: ext-router
```

First part configures internal network. All options have default values:

Table 1.1: **Internal network configuration options**

Name	Description	Default
enable	boolean flag, which turns on/off bootstrap.	true
net_name	Name of the internal network, which will be created in neutron.	int-net
subnet_name	Name of the subnet in internal network, which will be created in neutron.	int-subnet
network	CIDR of the internal network for allocating internal IP addresses.	10.0.1.0/24
gateway	Gateway for subnet in the internal network.	10.0.1.1

Second part describes external network configuration. Bootstrapping for external network is disabled by default and user should specify all options after turning it on, because most of them don't have default values.

Table 1.2: **External network configuration options**

Name	Description	Default
enable	boolean flag, which turns on/off bootstrap.	false
net_name	Name of the external network, which will be created in neutron. Default value can be used.	ext-net
subnet_name	Name of the subnet in external network, which will be created in neutron. Default value can be used.	ext-subnet
physnet	Name of the physnet, which was defined in **physnets** section.	–
network	CIDR of the external network for allocating external IP addresses.	–
gateway	Gateway for subnet in the external network.	–
nameserver	DNS server for subnet in external network.	–
pool	Pool of the addresses from external network, which can be used for association with Openstack VMs. Should be specified by using nested keys: **start** and **end**, which requires corresponding IP addresses.	–

The last section is a router configuration. It allows to specify name of the router, which will be created in neutron. Both networks will be connected with this router by default (except situation, when bootstrapping only for internal network is enabled). If bootstrapping is enabled at least for one network, router will be automatically created. In case, when user does not want to change default router name (**ext-router**) this section can be skipped in config.

Creation of the networks is handled by neutron post deployment jobs **neutron-bootstrap-***, which call openstackclient with specified parameters.

Example

As a simple example let's use snippet below:

```
configs:
  neutron:
    physnets:
      - name: ext-physnet
        bridge_name: br-ex
        interface: ens5
        flat: true
        vlan_range: false
    bootstrap:
      # external network parameters
      external:
        enable: true
        physnet: ext-physnet
        network: 10.90.2.0/24
        gateway: 10.90.2.1
        nameserver: 8.8.8.8
        pool:
          start: 10.90.2.10
          end: 10.90.2.250
```

Now go through all options and comments, what and why was choosen. First of all need to note, that interface **ens5** and bridge **br-ex** are used for creation physnet. Then in bootstrap section name of created physnet is used for providing

references for external network. Google public DNS server (*8.8.8.8*) is used as a **nameserver**. The main tricky thing here is an IP range and a gateway. In the current example Host for Kubernetes cluster has interface with IP address equal to specified IP in the gateway field. It's usually necessary for providing access from Openstack VMs to service APIs. At the end don't forget to be careful with pool of available external addresses. It should not contain IPs outside of cluster.

1.5.2 Image bootstrapping

Bootstrap for image allows to create/upload one image after deploying glance services. To enable it, user needs to add lines mentioned below to ~/.ccp.yaml:

```
configs:
  glance:
    bootstrap:
      enable: true
      image:
        url: http://download.cirros-cloud.net/0.3.4/cirros-0.3.4-x86_64-disk.img
        disk_format: qcow2
        name: cirros
```

This snippet adds **bootstrap** section for glance service and enables it. Note, that by default **enable** option is False. So if user wants to use bootstrapping he should explicitly set it to True.

The last part of the snippet describes image specific options. All options should be specified, otherwise it will cause an error during job execution:

Table 1.3: **Glance image bootstrapping default configuration options**

Name	Description	Default
url	url, which will be used for downloading image.	http://download.cirros-cloud.net/0.3.4/cirros-0.3.4-x86_64-disk.img
disk_format	format of the image which will be used during image creation in the glance.	qcow2
name	name of the image, which will be created in the glance.	cirros

Creation of the image is handled by glance post deployment job **glance-cirros-image-upload**, which uses Bash script from fuel-ccp-glance repository: `service/files/glance-cirros-image-upload.sh.j2`.

1.5.3 Flavor bootstrapping

The CCP automatically creates list of the default flavors, which are mentioned in the table below:

Name	ID	RAM	Disk	VCPUs
m1.test	0	128	1	1
m1.tiny	1	512	1	1
m1.small	2	2048	20	1
m1.medium	3	4096	40	2
m1.large	4	8192	80	4
m1.xlarge	5	16384	160	8

The corresponding section in the config file looks like:

```
configs:
  nova:
```

```
    bootstrap:
      enable: true
      flavors:
        - name: m1.test
          id: 0
          ram: 128
          disk: 1
          vcpus: 1
        - name: m1.tiny
          id: 1
          ram: 512
          disk: 1
          vcpus: 1
        - name: m1.small
          id: 2
          ram: 2048
          disk: 20
          vcpus: 1
        - name: m1.medium
          id: 3
          ram: 4096
          disk: 40
          vcpus: 2
        - name: m1.large
          id: 4
          ram: 8192
          disk: 80
          vcpus: 4
        - name: m1.xlarge
          id: 5
          ram: 16384
          disk: 160
          vcpus: 8
```

Creation of the flavors is handled by nova post deployment job **nova-bootstrap-flavors**, which uses Bash script from fuel-ccp-nova repository: `service/files/create-flavors.sh.j2`.

User also can specify to bootstrap custom flavors. Similar to previous sections it can be done by adding the following snippet to ~/.ccp.yaml:

```
configs:
  nova:
    bootstrap:
      enable: true
      flavors:
        - name: custom_name1
          id: 42
          ram: 1024
          disk: 1
          vcpus: 1
        - name: custom_name2
          id: 43
          ram: 2024
          disk: 2
          vcpus: 2
```

Warning: New list of custom flavors will overwrite default flavors and they will not be created. To avoid it, just copy paste definition of default flavors to your config and then extend it by new custom flavors.

This snippet adds **bootstrap** section for nova service. Note, that by default **enable** option is True. So if user doesn't want to use bootstrapping he should explicitly set it to False.

The last part of the snippet describes list of flavors with related options. All options should be specified, otherwise it will cause an error during job execution:

Table 1.4: **Nova flavor bootstrapping configuration options**

Name	Description	Default
name	Name of the custom flavor.	–
id	Indentifier of the flavor.	–
ram	Vaqlue of the RAM memory for the current flavor.	–
disk	Disk size for the curretn flavor.	–
vcpus	Number of the vcpus for the current flavor.	–

1.6 Ingress

One of the ways to make services in Kubernetes externally-reachable is to use Ingress. This page describes how it can be enabled and used in CCP.

1.6.1 Ingress controller

In order to make Ingress work, the cluster should have an Ingress controller. You can use any implementation of it, the only requirement is that it should be configured to use TLS.

There is a script `deploy-ingress-controller.sh` in `fuel-ccp/tools/ingress` directory for testing purposes that can do it for you. It will deploy traefik ingress controller and expose it as a k8s service. The only required parameter is one of the k8s nodes IP which need to be specified with *-i*. Ingress controller will be configured to use TLS. If certificate and key were not provided with script parameters, the will be generated automatically.

1.6.2 Enable Ingress in CCP

The following parameters are responsible for Ingress configuration:

```
configs:
  ingress:
    enabled: False
    domain: external
    port: 8443
```

Ingress is disabled by default. To enable it, *enabled* config option should be set to *True*. Optionally domain and port can be changed.

Note: There's no option to run Ingress without TLS.

Note: *port* parameter should match HTTPS port of Ingress controller.

Note: For multiple OpenStack deployments highly recommended to use different ‘domain‘s or run multiple Ingress controllers with configured namespace isolation.

To get all Ingress domains of the current deployment you can run **`ccp domains list`** command:

```
+------------------------------+
| Ingress Domain               |
+------------------------------+
| application-catalog.external |
| identity.external            |
| orchestration.external       |
| image.external               |
| object-store.external        |
| network.external             |
| ironic.external              |
| volume.external              |
| console.external             |
| data-processing.external     |
| horizon.external             |
| compute.external             |
| search.external              |
+------------------------------+
```

All of them should be resolved to the exposed IP of the Ingress controller. It could be done with DNS or /etc/hosts.

The following command will prepare /etc/hosts for you. Only IP of the Ingress controller (and configuration file if needed) should be specified:

```
echo INGRESS_CONTROLLER_IP $(ccp domains list -q -f value) | sudo tee -a /etc/hosts
```

1.6.3 Expose a service with Ingress

To expose one of the ports of a service with Ingress, *ingress* parameter with subdomain should be specified in the config section associated with that port:

```
configs:
  public_port:
    cont: 5000
    ingress: identity
```

During the **`ccp deploy`** command execution Ingress objects will be created and all *address* occurrences with enabled *external* flag will be substituted with proper Ingress domains.

CHAPTER 2

Advanced topics

2.1 Deploying Mutliple Parallel Environments

This guide describes how to deploy and run in parallel more than one OpenStack environment on a single Kubernetes cluster.

Warning: This functionality may not work correctly until this Calico bug is fixed: https://github.com/projectcalico/libcalico/issues/148

2.1.1 Introduction

From the Kubernetes (K8s) perspective, CCP is just another application, therefore it should be possible to run multiple CCP deployments within a single K8s cluster. This also promotes flexibility as there is no need to deploy separate K8s clusters to run parallel but isolated OpenStack clouds. A sample use-case may include 3 clouds: development, staging and production - all run on a single K8s cluster and managed from one place.

How deployments are isolated:

- logically by K8s namespaces (including individual FQDNs for each CCP service in each namespace)
- on a Docker level for services that can share hosts (e.g. `keystone`)
- on a host level for services that can be run 1 per host only (e.g. `nova-libvirt`)

Warning: Network isolation for parallel deployments depends on networking solution deployed in the K8s cluster. E.g. in case of Calico it offers tenant isolation but it may not be yet available for particular K8s deployment methods (e.g. Kargo). Please be aware that if that is the case, pods in different CCP deployments can access networks of each other.

What is needed to deploy mutliple CCPs in parallel:

- runnning K8s environment (for a tested, recommended setup please check out this guide)
- CCP installed on a machine with access to `kube-apiserver` (e.g. K8s master node)
- CCP CLI config file for each deployment
- CCP topology YAML file for each deployment

2.1.2 Quick start

To quickly deploy 2 parallel OpenStack environments, run these commands on your K8s master node:

```
git clone https://git.openstack.org/openstack/fuel-ccp
cd fuel-ccp
tox -e multi-deploy -- --number-of-envs 2
```

2.1.3 Sample deployment model

Following is an example of 3 parallel CCP deployments. Here is breakdown of services assignment to nodes (please note this isn't yet CCP topology file):

```
node1:
  - openvswitch[1]
  - controller-net-host[1]
  - controller-net-bridge[.*]
node[2-3]
  - openvswitch[1]
  - compute[1]
  - controller-net-bridge[.*]
node4:
  - openvswitch[2]
  - controller-net-host[2]
  - controller-net-bridge[.*]
node[5-6]
  - openvswitch[2]
  - compute[2]
  - controller-net-bridge[.*]
node7:
  - openvswitch[3]
  - controller-net-host[3]
  - controller-net-bridge[.*]
node[8-9]
  - openvswitch[3]
  - compute[3]
  - controller-net-bridge[.*]
```

Deployments 1-3 are marked by numbers in brackets ([]). For each deployment we dedicate:

- 1 node for net-host Controller services + Open vSwitch (e.g. node1 in deployment #1, node4 in deployment #2, node7 in deployment #3)
- 2 nodes for Computes + Open vSwich (e.g. node2 and node3 in deployment #1, node5 and node6 in deployment #2, etc.)

2.1.4 Sample CCP configuration

Let's now write the deployment model described in previous section into specific CCP configuration files. For each of 3 deployments we need 2 separate config files (1 for CLI configuration and 1 with topology) + 2 shared config files for common configuration options and roles definitions.

```
cat > ccp-cli-config-1.yaml << EOF
!include
- ccp-configs-common.yaml
- ccp-roles.yaml
- ccp-topology-1.yaml
---
kubernetes:
  namespace: "ccp-1"
EOF
```

```
cat > ccp-cli-config-2.yaml << EOF
!include
- ccp-configs-common.yaml
- ccp-roles.yaml
- ccp-topology-2.yaml
---
kubernetes:
  namespace: "ccp-2"
EOF
```

```
cat > ccp-cli-config-3.yaml << EOF
!include
- ccp-configs-common.yaml
- ccp-roles.yaml
- ccp-topology-3.yaml
---
kubernetes:
  namespace: "ccp-3"
EOF
```

```
cat > ccp-configs-common.yaml << EOF
---
builder:
  push: True
registry:
  address: "127.0.0.1:31500"
repositories:
  path: /tmp/ccp-repos
  skip_empty: True
configs:
    private_interface: eth0
    public_interface: eth1
    neutron_external_interface: eth2
EOF
```

```
cat > ccp-roles.yaml << EOF
---
roles:
  controller-net-host:
    - neutron-dhcp-agent
    - neutron-l3-agent
```

```
    - neutron-metadata-agent
  controller-net-bridge:
    - etcd
    - glance-api
    - glance-registry
    - heat-api-cfn
    - heat-api
    - heat-engine
    - horizon
    - keystone
    - mariadb
    - memcached
    - neutron-server
    - nova-api
    - nova-conductor
    - nova-consoleauth
    - nova-novncproxy
    - nova-scheduler
    - rabbitmq
  compute:
    - nova-compute
    - nova-libvirt
  openvswitch:
    - neutron-openvswitch-agent
    - openvswitch-db
    - openvswitch-vswitchdvv
EOF
```

```
cat > ccp-topology-1.yaml << EOF
---
nodes:
  node[1,2-3,4,5-6,7,8-9]:
    roles:
      - controller-net-bridge
  node1:
    roles:
      - openvswitch
      - controller-net-host
  node[2-3]:
    roles:
      - openvswitch
      - compute
EOF
```

```
cat > ccp-topology-2.yaml << EOF
---
nodes:
  node[1,2-3,4,5-6,7,8-9]:
    roles:
      - controller-net-bridge
  node4:
    roles:
      - openvswitch
      - controller-net-host
  node[5-6]:
    roles:
      - openvswitch
```

```
      - compute
EOF
```

```
cat > ccp-topology-3.yaml << EOF
---
nodes:
  node[1,2-3,4,5-6,7,8-9]:
    roles:
      - controller-net-bridge
  node7:
    roles:
      - openvswitch
      - controller-net-host
  node[8-9]:
    roles:
      - openvswitch
      - compute
EOF
```

Since we will use the same Docker OpenStack images for all 3 deployments it is sufficient to build them (and push to local registry) only once:

```
ccp --config-file ccp-cli-config-1.yaml build
```

We can now deploy CCP as usually:

```
ccp --config-file ccp-cli-config-1.yaml deploy
ccp --config-file ccp-cli-config-2.yaml deploy
ccp --config-file ccp-cli-config-3.yaml deploy
```

CCP will create 3 K8s namespaces (ccp-1, ccp-2 and ccp-3) and corresponding jobs, pods and services in each namespace. Finally, it will create openrc files in current working directory for each deployment, named `openrc-ccp-1`, `openrc-ccp-2` and `openrc-ccp-3`. These files (or nodePort of horizon K8s service in each namespace) can be used to access each OpenStack cloud separately. To know when each deployment is ready to be accessed `kubectl get jobs` command can be used (all jobs should finish):

```
kubectl --namespace ccp-1 get jobs
kubectl --namespace ccp-2 get jobs
kubectl --namespace ccp-3 get jobs
```

To destroy selected deployment environments `ccp cleanup` command can be used, e.g. to destroy deployment #2:

```
ccp --config-file ccp-cli-config-2.yaml cleanup
```

2.2 Mysql Galera Guide

This guide provides an overview of Galera implementation in CCP.

2.2.1 Overview

Galera Cluster is a synchronous multi-master database cluster, based on synchronous replication and MySQL/InnoDB. When Galera Cluster is in use, you can direct reads and writes to any node, and you can lose any individual node without interruption in operations and without the need to handle complex failover procedures.

2.2.2 CCP implementaion details

Entrypoint script

To handle all required logic, CCP has a dedicated entrypoint script for Galera and its side-containers. Because of that, Galera pods are slightly different from the rest of CCP pods. For example, Galera container still uses CCP global entrypoint, but it executes Galera entrypoint, which is executing MySQL and handles all required logic, like bootstrapping, fail detection, etc.

Galera pod

Each Galera pod consists of 3 containers:

- galera
- galera-checker
- galera-haproxy

galera - a container which runs Galera itself.

galera-checker - a container with galera-checker script. It is used to check readiness and liveness of the Galera node.

galera-haproxy - a container with a haproxy instance.

Note: More info about each container is available in the "Galera containers" section.

Etcd usage

The current implementation uses etcd to store cluster state. The default etcd root the directory will be `/galera/k8scluster`.

Additional keys and directories are:

- **leader** - key with the IP address of the current leader. Leader - is just a single, random Galera node, which haproxy will be used as a backend.
- **nodes/** - directory with current Galera nodes. Each node key will be named as an IP address of the node and value will be a Unix time of the key creation.
- **queue/** - directory with current Galera nodes waiting in the recovery queue. This is needed to ensure that all nodes are ready, before looking for the node with the highest seqno. Each node key will be named as an IP addr of the node and value will be a Unix time of the key creation.
- **seqno/** - directory with current Galera nodes seqno's. Each node key will be named as an IP address of the node and its value will be a seqno of the node's data.
- **state** - key with current cluster state. Can be "STEADY", "BUILDING" or "RECOVERY"
- **uuid** - key with current uuid of the Galera cluster. If a new node will have a different uuid, this will indicate that we have a split brain situation. Nodes with the wrong uuid will be destroyed.

2.2.3 Galera containers

galera

This container runs Galera daemon, plus handles all the bootstrapping, reconnecting and recovery logic.

At the start of the container, it checks for the `init.ok` file in the Galera data directory. If this file doesn't exist, it removes all files from the data directory, running Mysql init, to create base mysql data files, after we're starting mysqld daemon without networking and setting needed permissions for expected users.

If `init.ok` file is found, it runs the `mysqld_safe --wsrep-recover` to recover Galera related information and write it to the `grastate.dat` file.

After that, it checks the cluster state and depending on the current state it chose required scenario.

galera-checker

This container is used for liveness and readiness checks of Galera pod.

To check if this Galera pod is ready it checks for the following things:

1. wsrep_local_state_comment = "Synced"
2. wsrep_evs_state = "OPERATIONAL"
3. wsrep_connected = "ON"
4. wsrep_ready = "ON"
5. wsrep_cluster_state_uuid = uuid in the etcd

To check if this Galera pod is alive we checking the following things:

1. If current cluster state is not "STEADY" - it skips liveness check.
2. If it detects that SST sync is in progress - it skips liveness check.
3. If it detects that there is no Mysql pid file yet - it skips liveness check.
4. If node "wsrep_cluster_state_uuid" differs from the etcd one - it kills Galera container, since it's a "split brain" situation.
5. If "wsrep_local_state_comment" is "Joined", and the previous state was "Joined" too - it kills Galera container since it can't finish joining to the cluster for some reason.
6. If it caught any exception during the checks - it kills Galera container.

If all checks passed - we're deciding that Galera pod is alive.

galera-haproxy

This container is used to run haproxy daemon, which is used to send all traffic to a single Galera pod.

This is needed to avoid deadlocks and stale reads. It chooses the "leader" out of all available Galera pods and once leader is chosen, all haproxy instances update their configuration with the new leader.

2.2.4 Supported scenarios

Initial bootstrap

In this scenario, there is no working Galera cluster yet. Each node trying to get the lock in etcd, first one which can start cluster bootstrapping. After it's done, next node gets the lock and connects to the existing cluster.

Note: During the bootstrap state of the cluster will be "BUILDING". It will be changed to "STEADY" after last node connection.

Re-connecting to the existing cluster

In this scenario, Galera cluster is already available. In most case it will be a node re-connection after some failures, such as node reboot. Each node tries to get the lock in etcd, once lock acquiring node connects to the existing cluster.

Note: During this scenario state of the cluster will be "STEADY".

Recovery

This scenario could be triggered by two possible options:

- Operator manually sets cluster state in etcd to the "RECOVERY"
- New node does a few checks before bootstrapping, if it finds that cluster state is "STEADY", but there is zero nodes in the cluster - it assumes that cluster has been destroyed somehow and we need to run recovery. In that case, it sets the state to the "RECOVERY" and starts recovery scenario.

During the recovery scenario cluster bootstrapping is different from the "Initial bootstrap". In this scenario, each node looks for its "seqno", which is basically the registered number of the transactions. A node with the highest seqno will bootstrap cluster and other nodes will join it, so in the end, we will have the latest data available before the cluster destruction.

Note: During the bootstrap state of the cluster will be "RECOVERY". It will be changed to "STEADY" after last node connection.

There is an option to manually choose the node to recover data from. For details please see the "force bootstrap" section in the "Advanced features" .

2.2.5 Advanced features

Cluster size

By default, galera cluster size will be 3 nodes. This is optimal for the most cases. If you want to change it to some custom number, you need to override **cluster_size** variable in the **percona** tree, for example:

```
configs:
  percona:
    cluster_size: 5
```

Note: Cluster size should be an odd number. Cluster size with more that 5 nodes will lead to big latency for write operations.

Force bootstrap

Sometimes operators may want to manually specify Galera node which recovery should be done from. In that case, you need to override **force_bootstrap** variable in the **percona** tree, for example:

```
configs:
  percona:
    force_bootstrap:
      enabled: true
      node: NODE_NAME
```

NODE_NAME should be the name of the k8s node, which will run Galera node with required data.

2.2.6 Troubleshooting

Galera operation requires some advanced knowledge in Mysql and in some general clustering conceptions. In most cases, we expect that Galera will "self-heal" itself, in the worst case via restart, full resync and reconnection to the cluster.

Our readiness and liveness scripts should cover this, and not allow misconfigured or non-operational node receive production traffic.

Yet it's possible that some failure scenarios is not covered and to fix them some manual actions could be required.

Check the logs

Each container of the Galera pod writes detailed logs to the stdout. You could read them via `kubectl logs POD_NAME -c CONT_NAME`. Make sure you check the `galera` container logs and `galera-checker` ones.

Additionally you should check the Mysql logs in the `/var/log/ccp/mysql/mysql.log`

Check the etcd state

Galera keeps its state in the etcd and it could be useful to check what is going on in the etcd right now. Assuming that you're using the **ccp** namespace, you could check etcd state using this command:

```
etcdctl --endpoints http://etcd.ccp.svc.cluster.local:2379 ls -r -p --sort /galera
etcdctl --endpoints http://etcd.ccp.svc.cluster.local:2379 get /galera/k8scluster/
↪state
etcdctl --endpoints http://etcd.ccp.svc.cluster.local:2379 get /galera/k8scluster/
↪leader
etcdctl --endpoints http://etcd.ccp.svc.cluster.local:2379 get /galera/k8scluster/uuid
```

Node restart

In most cases, it should be safe to restart a single Galera node. If you need to do it for some reason, just delete the pod, via kubectl:

```
kubectl delete pod POD_NAME
```

Full cluster restart

In some cases, you may need to restart the whole cluster. Make sure you have a backup before doing this. To do this, set the cluster state to the "RECOVERY":

```
etcdctl --endpoints http://etcd.ccp.svc.cluster.local:2379 set /galera/k8scluster/
↪state RECOVERY
```

After that restart all Galera pods:

```
kubectl delete pod POD1_NAME POD2_NAME POD3_NAME
```

Once that done, Galera cluster will be rebuilt and should be operational.

Note: For more info about cluster recovery please refer to the "Supported scenarios" section.

2.3 Ceph and Swift guide

This guide provides instruction for adding Ceph and Swift support for CCP deployment.

Note: It's expected that an external Ceph cluster is already available and accessible from the all k8s nodes. If you don't have a Ceph cluster, but still want to try CCP with Ceph, you can use *Ceph cluster deployment* guide for deploying a simple 3 node Ceph cluster.

2.3.1 Ceph

Prerequirements

You need to ensure that these pools are created:

- images
- volumes
- vms

And that users "glance" and "cinder" are created and have these permissions:

```
client.cinder
      caps: [mon] allow r
      caps: [osd] allow rwx pool=volumes, allow rwx pool=vms, allow rx pool=images
client.glance
      caps: [mon] allow r
      caps: [osd] allow rwx pool=images, allow rwx pool=vms
```

Deploy CCP with Ceph

In order to deploy CCP with Ceph you have to edit the `ccp.yaml` the file:

```
configs:
  ceph:
    fsid: "FSID_OF_THE_CEPH_CLUSTER"
    mon_host: "CEPH_MON_HOSTNAME"
  cinder:
    ceph:
      enable: true
      key: "CINDER_CEPH_KEY"
      rbd_secret_uuid: "RANDOM_UUID"
  glance:
    ceph:
      enable: true
      key: "GLANCE_CEPH_KEY"
  nova:
    ceph:
      enable: true
```

Example:

```
configs:
  ceph:
    fsid: "afca8524-2c47-4b81-a0b7-2300e62212f9"
    mon_host: "10.90.0.5"
  cinder:
    ceph:
      enable: true
      key: "AQBShfJXID9pFRAAm4VLpbNXa4XJ9zgAh7dm2g=="
      rbd_secret_uuid: "b416770d-f3d4-4ac9-b6db-b6a7ac1c61c0"
  glance:
    ceph:
      enable: true
      key: "AQBShfJXzXyNBRAA5kqXzCKcFoPBn2r6VDYdag=="
  nova:
    ceph:
      enable: true
```

- `fsid` - Should be the same as `fsid` variable in the Ceph cluster `ceph.conf` file.
- `mon_host` - Should contain any Ceph mon node IP or hostname.
- `key` - Should be taken from the corresponding Ceph user. You can use the `ceph auth list` command on the Ceph node to fetch list of all users and their keys.
- `rbd_secret_uuid` - Should be randomly generated. You can use the `uuidgen` command for this.

Make sure that your deployment topology has a `cinder` service. You could use `etc/topology-with-ceph-example.yaml` as a reference.

Now you're ready to deploy CCP with Ceph support.

2.3.2 Swift

Prerequirements

Make sure that your deployment topology has a `radosgw` service. You could use `etc/topology-with-ceph-example.yaml` as a reference.

Deploy CCP with Swift

Note: Currently, in CCP, only Glance supports Swift as a backend.

In order to deploy CCP with Swift you have to edit `ccp.yaml` the file:

```
ceph:
  fsid: "FSID_OF_THE_CEPH_CLUSTER"
  mon_host: "CEPH_MON_HOSTNAME"
radosgw:
  key: "RADOSGW_CEPH_KEY"
glance:
  swift:
    enable: true
    store_create_container_on_put: true
```

Example:

```
ceph:
  fsid: "afca8524-2c47-4b81-a0b7-2300e62212f9"
  mon_host: "10.90.0.2,10.90.0.3,10.90.0.4"
radosgw:
  key: "AQBIGP5Xs6QFCRAAkCf5YWeBHBlaj6S1rkcCYA=="
glance:
  swift:
    enable: true
    store_create_container_on_put: true
```

Troubleshooting

If the Glance image upload failed, you should check few things:

- Glance-api pod logs
- Radosgw pod logs
- Keystone pod logs

2.4 Ceph cluster deployment

Warning: This setup is very simple, limited, and not suitable for real production use. Use it as an example only.

Using this guide you'll deploy a 3 nodes Ceph cluster with RadosGW.

2.4.1 Prerequirements

- Three nodes with at least one unused disk available.
- In this example we're going to use Ubuntu 16.04 OS, if you're using a different one, you have to edit the following configs and commands to suit your OS.

In this doc we refer to these nodes as

- ceph_node_hostname1
- ceph_node_hostname2
- ceph_node_hostname3

2.4.2 Installation

```
sudo apt install ansible
git clone https://github.com/ceph/ceph-ansible.git
```

2.4.3 Configuration

cd into ceph-ansible directory:

```
cd ceph-ansible
```

Create `group_vars/all` with:

```
ceph_origin: upstream
ceph_stable: true
ceph_stable_key: https://download.ceph.com/keys/release.asc
ceph_stable_release: jewel
ceph_stable_repo: "http://download.ceph.com/debian-{{ ceph_stable_release }}"
cephx: true
generate_fsid: false
# Pre-created static fsid
fsid: afca8524-2c47-4b81-a0b7-2300e62212f9
# interface which ceph should use
monitor_interface: NAME_OF_YOUR_INTERNAL_IFACE
monitor_address: 0.0.0.0
journal_size: 1024
# network which you want to use for ceph
public_network: 10.90.0.0/24
cluster_network: "{{ public_network }}"
```

Make sure you change the `NAME_OF_YOUR_INTERNAL_IFACE` placeholder to the actual interface name, like `eth0` or `ens*` in modern OSs.

Create `group_vars/osds` with:

```
fsid: afca8524-2c47-4b81-a0b7-2300e62212f9
# Devices to use in ceph on all osd nodes.
# Make sure the disk is empty and unused.
devices:
- /dev/sdb
# Journal placement option.
```

```
# This one means that journal will be on the same drive but another partition
journal_collocation: true
```

Create `group_vars/mons` with:

```
fsid: afca8524-2c47-4b81-a0b7-2300e62212f9
monitor_secret: AQAjn8tUwBpnCRAAU8X0Syf+U8gfBvnbUkDPyg==
```

Create inventory file with:

```
[mons]
ceph_node_hostname1
ceph_node_hostname2
ceph_node_hostname3
[osds]
ceph_node_hostname1
ceph_node_hostname2
ceph_node_hostname3
```

2.4.4 Deploy

Make sure you have passwordless ssh key access to each node and run:

```
ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook -i inventory_file site.yml.sample
```

2.4.5 Check Ceph deployment

Go to any ceph node and run with root permissions:

```
sudo ceph -s
```

`health` should be HEALTH_OK. HEALTH_WARN signify non-critical error, check the description of the error to get the idea of how to fix it. HEALTH_ERR signify critical error or a failed deployment.

2.4.6 Configure pools and users

On any Ceph node run:

```
sudo rados mkpool images
sudo rados mkpool volumes
sudo rados mkpool vms
sudo rados mkpool backups
```

```
sudo ceph auth get-or-create client.glance osd 'allow class-read object_prefix rbd_
↪children, allow rwx pool=images, allow rwx pool=vms' mon 'allow r' -o /etc/ceph/
↪ceph.client.glance.keyring
sudo ceph auth get-or-create client.cinder osd 'allow class-read object_prefix rbd_
↪children, allow rwx pool=volumes, allow rwx pool=backups, allow rwx pool=vms, allow␣
↪rwx pool=images' mon 'allow r' -o /etc/ceph/ceph.client.cinder.keyring
sudo ceph auth get-or-create client.radosgw.gateway osd 'allow rwx' mon 'allow rwx' -
↪o /etc/ceph/ceph.client.radosgw.keyring
```

To list all user with permission and keys, run:

```
sudo ceph auth list
```

Now you're ready to use this Ceph cluster with CCP.

2.5 SR-IOV guide

This guide provides an instruction for enabling SR-IOV functionality in Fuel CCP.

2.5.1 Introduction

The SR-IOV specification defines a standardized mechanism to virtualize PCIe devices. This mechanism can virtualize a single PCIe Ethernet controller to appear as multiple PCIe devices. Each device can be directly assigned to an instance, bypassing the hypervisor and virtual switch layer. As a result, users are able to achieve low latency and near-line wire speed.

The following terms are used throughout this document:

Term	Definition
PF	Physical Function. The physical Ethernet controller that supports SR-IOV.
VF	Virtual Function. The virtual PCIe device created from a physical Ethernet controller.

Prerequirements

1. Ensure that a host has a SR-IOV capable device. One way of identifying whether a device supports SR-IOV is to check for an SR-IOV capability in the device configuration. The device configuration also contains the number of VFs the device can support. The example below shows a simple test to determine if the device located at the bus, device, and function number 1:00.0 can support SR-IOV.

```
# lspci -vvv -s 02:00.0 | grep -A 9 SR-IOV
    Capabilities: [160 v1] Single Root I/O Virtualization (SR-IOV)
            IOVCap: Migration-, Interrupt Message Number: 000
            IOVCtl: Enable+ Migration- Interrupt- MSE+ ARIHierarchy+
            IOVSta: Migration-
            Initial VFs: 32, Total VFs: 32, Number of VFs: 7, Function Dependency␣
↪Link: 00
            VF offset: 16, stride: 1, Device ID: 154c
            Supported Page Size: 00000553, System Page Size: 00000001
            Region 0: Memory at 0000000090400000 (64-bit, prefetchable)
            Region 3: Memory at 0000000092c20000 (64-bit, prefetchable)
            VF Migration: offset: 00000000, BIR: 0
```

2. Enable IOMMU in Linux by adding *intel_iommu=on* to the kernel parameters, for example, using GRUB.

3. Bring up the PF.

```
# ip l set dev ens2f1 up
```

4. Allocate the VFs, for example via the PCI SYS interface:

```
# echo '7' > /sys/class/net/ens2f1/device/sriov_numvfs
```

5. Verify that the VFs have been created.

```
# ip l show ens2f1
5: ens2f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT␣
↪group default qlen 1000
link/ether 0c:c4:7a:bd:42:ac brd ff:ff:ff:ff:ff:ff
vf 0 MAC 00:00:00:00:00:00, spoof checking on, link-state auto
vf 1 MAC 00:00:00:00:00:00, spoof checking on, link-state auto
vf 2 MAC 00:00:00:00:00:00, spoof checking on, link-state auto
vf 3 MAC 00:00:00:00:00:00, spoof checking on, link-state auto
vf 4 MAC 00:00:00:00:00:00, spoof checking on, link-state auto
vf 5 MAC 00:00:00:00:00:00, spoof checking on, link-state auto
vf 6 MAC 00:00:00:00:00:00, spoof checking on, link-state auto
```

2.5.2 Deploy CCP with SR-IOV

Neutron

In OpenStack SR-IOV doesn't work with VxLAN tunneling, that is why it is required to enable either VLAN of flat tenant network type in the *configs.neutron* section of the CCP configuration file:

```
ml2:
  tenant_network_types:
    - "vlan"
```

All Neutron SR-IOV specific parameters are located in the *configs.neutron.sriov* section. Example configuration:

```
sriov:
  enabled: true
  devices:
    - name: "ens2f1"
      physnets:
        - "physnet1"
      exclude_vfs:
        - 0000:02:00.2
        - 0000:02:00.3
```

- *enabled* - Boolean. Enables and disables the SR-IOV in Neutron, *false* by default.
- ***devices* - List. A node-specific list of SR-IOV devices. Each element of the list has 2 mandatory fields: *name* and *physnets*.**
  - *name* is a name of the SR-IOV interface.
  - *physnets* is a list of of names of physical networks a given device maps to.
  - If *exclude_vfs* is omitted all the VFs associated with a given device may be configured by the agent. To exclude specific VFs, add them to the *exclude_devices* parameter as shown above.

A new role should be added to compute nodes: *neutron-sriov-nic-agent*.

Nova

All Nova SR-IOV specific parameters are located in the *configs.nova.sriov* section. Example configuration:

```
sriov:
  enabled: true
  pci_alias:
```

```
    - name: "82599ES"
      product_id: "10fb"
      vendor_id: "8086"
    - name: "X710"
      product_id: "1572"
      vendor_id: "8086"
  pci_passthrough_whitelist:
     - devname: "ens2f1"
       physical_network: "physnet1"
```

- *enabled* - Boolean. Enables and disables the SR-IOV in Nova, *false* by default.
- *pci_alias* - List, optional. An alias for a PCI passthrough device requirement. This allows users to specify the alias in the

extra_spec for a flavor, without needing to repeat all the PCI property requirements.

- ***pci_passthrough_whitelist* - List. White list of PCI devices available to VMs.**
  - *devname* is a name of the SR-IOV interface.
  - *physical_network* - name of a physical network to map a device to.

Additionally it is required to add *PciPassthroughFilter* to the list of enable filters in Nova scheduler:

```
scheduler:
  enabled_filters:
    - RetryFilter
    - AvailabilityZoneFilter
    - RamFilter
    - DiskFilter
    - ComputeFilter
    - ComputeCapabilitiesFilter
    - ImagePropertiesFilter
    - ServerGroupAntiAffinityFilter
    - ServerGroupAffinityFilter
    - SameHostFilter
    - DifferentHostFilter
    - PciPassthroughFilter
```

Sample CCP configuration

```
services:
  database:
    service_def: galera
  rpc:
    service_def: rabbitmq
  notifications:
    service_def: rabbitmq
nodes:
  node1:
    roles:
      - db
      - messaging
      - controller
      - openvswitch
  node[2-3]:
    roles:
```

```
      - db
      - messaging
      - compute
      - openvswitch
roles:
  db:
    - database
  messaging:
    - rpc
    - notifications
  controller:
    - etcd
    - glance-api
    - glance-registry
    - heat-api-cfn
    - heat-api
    - heat-engine
    - horizon
    - keystone
    - memcached
    - neutron-dhcp-agent
    - neutron-l3-agent
    - neutron-metadata-agent
    - neutron-server
    - nova-api
    - nova-conductor
    - nova-consoleauth
    - nova-novncproxy
    - nova-scheduler
  compute:
    - neutron-sriov-nic-agent
    - nova-compute
    - nova-libvirt
  openvswitch:
    - neutron-openvswitch-agent
    - openvswitch-db
    - openvswitch-vswitchd
configs:
  private_interface: ens1f0
  neutron:
    physnets:
      - name: "physnet1"
        bridge_name: "br-ex"
        interface: "ens1f1"
        flat: false
        vlan_range: "50:1030"
    ml2:
      tenant_network_types:
        - "vlan"
    sriov:
      enabled: true
      devices:
        - name: "ens2f1"
          physnets:
            - "physnet1"
          exclude_vfs:
            - 0000:02:00.2
            - 0000:02:00.3
```

```
nova:
  sriov:
    enabled: true
    pci_alias:
      - name: "82599ES"
        product_id: "10fb"
        vendor_id: "8086"
      - name: "X710"
        product_id: "1572"
        vendor_id: "8086"
    pci_passthrough_whitelist:
       - devname: "ens2f1"
         physical_network: "physnet1"
  scheduler:
    enabled_filters:
      - RetryFilter
      - AvailabilityZoneFilter
      - RamFilter
      - DiskFilter
      - ComputeFilter
      - ComputeCapabilitiesFilter
      - ImagePropertiesFilter
      - ServerGroupAntiAffinityFilter
      - ServerGroupAffinityFilter
      - SameHostFilter
      - DifferentHostFilter
      - PciPassthroughFilter
```

2.5.3 Known limitations

- When using Quality of Service (QoS), *max_burst_kbps* (burst over *max_kbps*) is not supported. In addition, *max_kbps* is rounded to Mbps.
- Security groups are not supported when using SR-IOV, thus, the firewall driver is disabled.
- SR-IOV is not integrated into the OpenStack Dashboard (horizon). Users must use the CLI or API to configure SR-IOV interfaces.
- Live migration is not supported for instances with SR-IOV ports.

2.6 Enable Distributed Virtual Routing in Neutron

This guide provides an instruction for enabling DVR support in a CCP deployment.

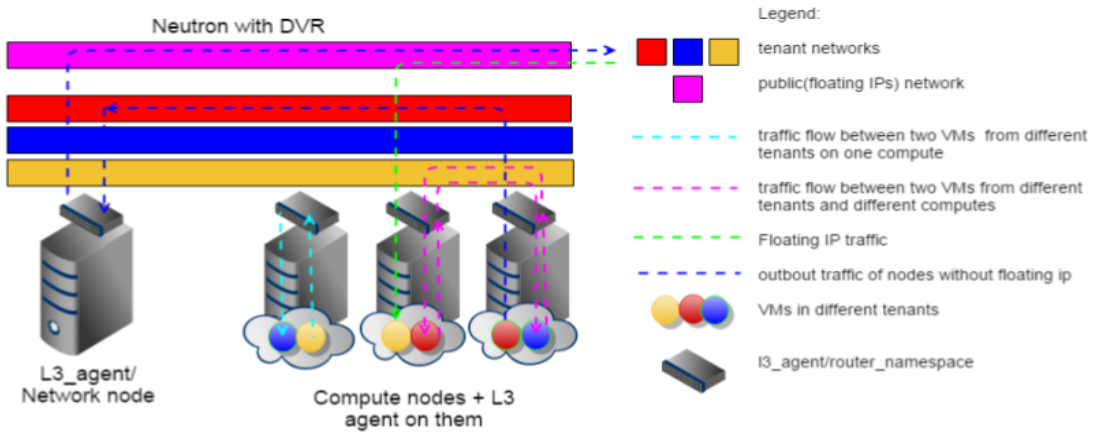
Note: DVR mode applies only for Neutron reference implementation with ML2/OpenVSwitch plugin. In order to determine distributed routing capabilities of other plugins/drivers please refer to their corresponding documentation.

2.6.1 Introduction

DVR

- Distributes L3 Routers across compute nodes when required by VMs

- L3 and Metadata Agents are running on each and every compute node
- Inter-subnets traffic is no longer affected by performance of one node with l3 agent
- Traffic for VMs with floating IP is no longer affected by performance of one node with l3 agent
- Removal of single l3 agent node as single-point-of-failure for all inter-tenant traffic and Floating IP traffic



2.6.2 Sample CCP configuration

ccp.yaml may look like:

```
builder:
  push: True
registry:
  address: "127.0.0.1:31500"
repositories:
  skip_empty: True
services:
  database:
    service_def: galera
  rpc:
    service_def: rabbitmq
  notifications:
    service_def: rabbitmq
nodes:
  node1:
    roles:
      - db
      - messaging
      - controller
      - openvswitch
  node[2-3]:
    roles:
      - db
      - messaging
      - compute
      - openvswitch
roles:
  db:
    - database
  messaging:
```

```
    - rpc
    - notifications
  controller:
    - etcd
    - glance-api
    - glance-registry
    - heat-api-cfn
    - heat-api
    - heat-engine
    - horizon
    - keystone
    - memcached
    - neutron-dhcp-agent
    - neutron-l3-agent
    - neutron-metadata-agent
    - neutron-server
    - nova-api
    - nova-conductor
    - nova-consoleauth
    - nova-novncproxy
    - nova-scheduler
  compute:
    - nova-compute
    - nova-libvirt
    - neutron-l3-agent-compute
    - neutron-metadata-agent
  openvswitch:
    - neutron-openvswitch-agent
    - openvswitch-db
    - openvswitch-vswitchd
configs:
    private_interface: ens7
    public_interface: ens7
    neutron:
      dvr: True
      physnets:
        - name: "physnet1"
          bridge_name: "br-ex"
          interface: "ens3"
          flat: true
          vlan_range: false
```

Compute node now has 2 additional roles: neutron-l3-agent-compute and neutron-metadata-agent.

Note: For Floating IPs to work properly, DVR requires each compute node to have access to the external net.

2.7 Using Calico instead of Open vSwitch

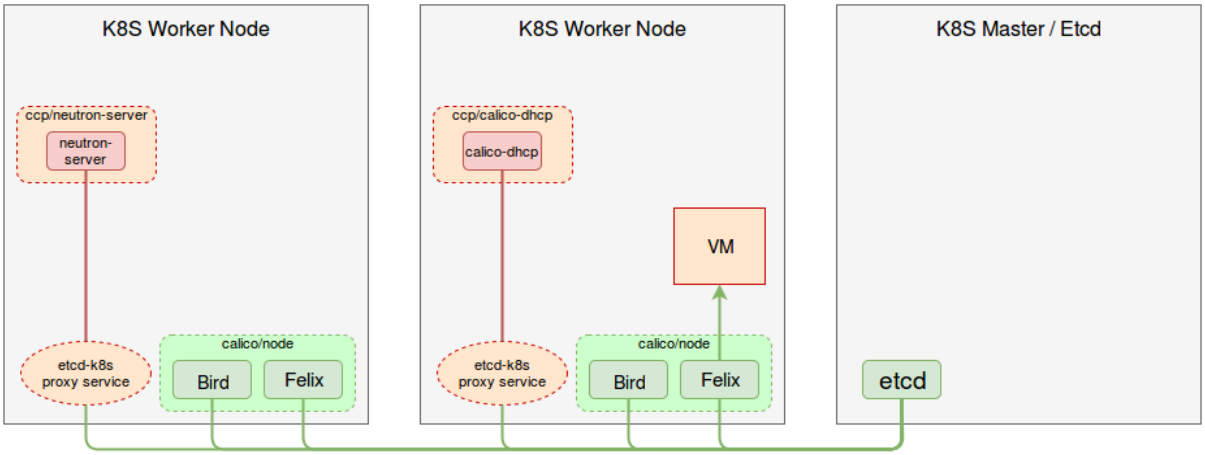
This guide describes how to deploy and run OpenStack environment with Calico ml2 Neutron plugin instead of OVS on top of Kubernetes cluster and how to integrate OpenStack and Kubernetes workloads.

2.7.1 Introduction

Calico's pure L3 approach to data center networking integrates seamlessly with cloud orchestration systems (such as OpenStack) to enable secure IP communication between virtual machines, containers, or bare metal workloads.

By using Calico network plugin for both Kubernetes and OpenStack Containerized Control Plane (CCP) we can provide pure L3 fabric and cross-workload security for mixed workloads.

Deployment diagram:



Deployment will look like this:

- Neutron is configured to use `networking-calico` ML2 plugin.
- Neutron DHCP agent is replaced with Calico DHCP agent.
- Open vSwitch pods are removed from the deployment topology.
- Additional Kubernetes proxy service is required to provide the connectivity from CCP pods to the main Etcd cluster (they cannot connect to etcd-proxy on a localhost since some containers are running in isolated network space, for example neutron-server).
- CCP Calico components are connected to the same Etcd DB as Calico services providing networking for Kubernetes.
- Calico/felix from `calico/node` container has reporting enabled.

What is needed to deploy CCP with Calico network plugin:

- Runnning K8s environment with Calico network plugin (for a tested, recommended setup please check out this guide).
- `calico/node` version 0.23.0 or higher (you can use `latest` image tag).
- CCP installed on a machine with access to `kube-apiserver` (e.g. K8s master node).
- CCP CLI config file with custom deployment topology.

2.7.2 Sample deployment

Sample deployment model

Following is an example of CCP deployment with Calico networking integrated with Kubernetes Calico components. Here is breakdown of services assignment to nodes (please note this isn't yet CCP topology file):

```
node1:
  - controller
  - neutron-server
  - neutron-metadata-agent
node[2-3]:
  - compute
  - calico-dhcp-agent
```

Configuring requirements in Kubernetes cluster

Before deploying CCP we should run etcd proxy service (please don't forget to replace IP addresses in this sample with your K8s cluster Etcd nodes' IPs):

```
cat > /var/tmp/etcd-k8s-svc.yaml << EOF
kind: "Endpoints"
apiVersion: "v1"
metadata:
  name: "etcd-k8s"
subsets:
  - addresses:
    - ip: "10.210.1.11"
    - ip: "10.210.1.12"
    - ip: "10.210.1.13"
    ports:
      - port: 2379
        name: "etcd-k8s"
---
apiVersion: "v1"
kind: "Service"
metadata:
  name: "etcd-k8s"
spec:
  ports:
  - name: "etcd-k8s"
    port: 2379
    protocol: TCP
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
EOF

kubectl --namespace=default create -f /var/tmp/etcd-k8s-svc.yaml
```

We also need to enable reporting in Felix:

```
etcdctl set /calico/v1/config/ReportingIntervalSecs 60
```

And add some custom export filters for BGP agent:

```
cat << EOF | etcdctl set /calico/bgp/v1/global/custom_filters/v4/tap_iface
  if ( ifname ~ "tap*" ) then {
    accept;
  }
EOF
```

Sample CCP configuration

Let's write CCP CLI configuration file now, make sure you have the following in your configuration file (let's say it's `ccp.yaml`):

```
kubernetes:
  namespace: "ccp"

configs:
  neutron:
    plugin_agent: "calico"
    calico:
      etcd_host: "etcd-k8s"
      etcd_port: "2379"

nodes:
  node1:
    roles:
      - controller
      - neutron-agents
  node[2-3]:
    roles:
      - compute
      - calico

roles:
  controller:
    - etcd
    - glance-api
    - glance-registry
    - heat-api-cfn
    - heat-api
    - heat-engine
    - horizon
    - keystone
    - mariadb
    - memcached
    - neutron-server
    - nova-api
    - nova-conductor
    - nova-consoleauth
    - nova-novncproxy
    - nova-scheduler
    - rabbitmq
  neutron-agents:
    - neutron-metadata-agent
  compute:
    - nova-compute
    - nova-libvirt
  calico:
    - calico-dhcp-agent
```

Now let's build images and push them to registry if you have not done this already:

```
ccp deploy --config-file ccp.yaml build
```

We can now deploy CCP as usually:

```
ccp deploy --config-file ccp.yaml deploy
```

CCP will create namespace named `ccp` and corresponding jobs, pods and services in it. To know when deployment is ready to be accessed `kubectl get jobs` command can be used (all jobs should finish):

```
kubectl --namespace ccp get jobs
```

Creating networks and instances in OpenStack

After CCP deployment is complete we can create Neutron networks and run VMs.

Install openstack-client:

```
pip install python-openstackclient
```

`openrc` file for current deployment was created in the current working directory. To use it run:

```
source openrc-ccp
```

Run test environment deploy script:

```
bash fuel-ccp/tools/deploy-test-vms.sh -a create -c -n NUMBER_OF_VMS
```

This script will create flavor, upload cirrios image to glance, create network and subnet and launch bunch of cirrios based VMs.

Uninstalling and undoing customizations

To destroy deployment environment `ccp cleanup` command can be used:

```
ccp --config-file ccp.yaml ccp cleanup
```

The following commands can be used to undo related customizations in Calico:

```
etcdctl rm /calico/bgp/v1/global/custom_filters/v4/tap_iface
etcdctl set /calico/v1/config/ReportingIntervalSecs 0
etcdctl ls /calico/felix/v1/host -r | grep status | xargs -n1 etcdctl rm
```

Remove Etcd proxy service:

```
kubectl --namespace=default delete -f /var/tmp/etcd-k8s-svc.yaml
```

2.8 Using OpenDaylight instead of Open vSwitch

This guide describes how to deploy and run OpenStack environment with OpenDaylight ML2 Neutron plugin instead of the reference OpenVSwitch ML2 on top of Kubernetes cluster using fuel-ccp.

2.8.1 Introduction

OpenDaylight (ODL) is a modular Open SDN platform for networks of any size and scale. OpenStack can use OpenDaylight as its network management provider through the Modular Layer 2 (ML2) north-bound plug-in. OpenDaylight manages the network flows for the OpenStack compute nodes via the OVSDB south-bound plug-in.

Deployment will look like this:

- new Docker container and service: opendaylight
- openvswitch service on nodes is configured to be managed by ODL
- neutron is configured to use `networking-odl` ML2 plugin.
- neutron openvswitch and l3 agent pods are removed from the deployment topology.

What is needed to deploy CCP with ODL network plugin:

- Runnning K8s environment with ODL network plugin (for a tested, recommended setup please check out the QuickStart Guide).
- CCP installed on a machine with access to `kube-apiserver` (e.g. K8s master node).
- CCP CLI config file with custom deployment topology.

2.8.2 Sample deployment

Sample CCP configuration

Let's write CCP CLI configuration file now, make sure you have the following in your configuration file (let's say it's `ccp.yaml`):

```
builder:
  push: True
registry:
  address: "127.0.0.1:31500"
repositories:
  skip_empty: True
nodes:
  node1:
    roles:
      - db
      - messaging
      - controller
      - openvswitch
      - opendaylight
  node[2-3]:
    roles:
      - db
      - messaging
      - compute
      - openvswitch
roles:
  db:
    - galera
  messaging:
    - rabbitmq
  controller:
    - etcd
    - glance-api
    - glance-registry
    - heat-api
    - heat-engine
    - horizon
    - keystone
    - memcached
```

```
    - neutron-dhcp-agent
    - neutron-metadata-agent
    - neutron-server
    - nova-api
    - nova-conductor
    - nova-consoleauth
    - nova-novncproxy
    - nova-scheduler
  compute:
    - nova-compute
    - nova-libvirt
  openvswitch:
    - openvswitch-db
    - openvswitch-vswitchd
  opendaylight:
    - opendaylight
configs:
    private_interface: eth1
    neutron:
      plugin_agent: "opendaylight"
versions:
    ovs_version: "2.5.1"
```

For the instructions for building images and deploying CCP refer to the QuickStart Guide.

To build only the opendaylight Docker image run:

```
ccp deploy --config-file ccp.yaml build -c opendaylight
```

To deploy only the opendaylight component run:

```
ccp deploy --config-file ccp.yaml deploy -c opendaylight
```

Check configuration

To check that neutron has been configured to work with OpenDaylight, attach to *neutron-server* container and run:

```
$ grep mechanism_drivers /etc/neutron/plugins/ml2/ml2_conf.ini
mechanism_drivers = opendaylight, logger
```

OpenDaylight creates only one bridge `br-int`, with all traffic being managed by OpenFlow, including routing and applying security group rules. To inspect flows, attach to an *openvswitch-vswitchd* container and exec:

```
ovs-ofctl -O OpenFlow13 dump-flows br-int
```

To connect to OpenDaylight console run the following command in *opendaylight* container:

```
/odl/bin/client
```

2.9 Ironic guide

This guide provides an instruction for adding Ironic support for CCP deployment.

2.9.1 Underlay

Note: That it's not the CCP responsibility to manage networking for baremetal servers. Ironic assumes that networking is properly configured in underlay.

Prerequirements

- Ironic conductor has access to IPMI of baremetal servers or to hypervisor when baremetal server is simulated by VM.
- Baremetal servers are attached to physical baremetal network.
- Swift, Ironic API endpoints, neutron-dhcp-agent, PXE/iPXE services are accessible from baremetal network.
- Swift and Ironic API endpoints has valid SSL certificate or Ironic deploy driver allows unverified connections.
- Baremetal network is accessible from Ironic conductor.

2.9.2 Neutron

Prerequirements

Ironic requires single flat network in Neutron which has L2 connectivity to physical baremetal network and appropriate L3 settings.

Example case when required access to Ironic services provided via Neutron external network:

```
# Create external network
neutron net-create ext --router:external true --shared --provider:network_type flat --
↪provider:physical_network physnet1

# Create subnet in external network, here 10.200.1.1 - is provider gateway
neutron subnet-create --name ext --gateway 10.200.1.1 --allocation-pool start=10.200.
↪1.10,end=10.200.1.200 ext 10.200.1.0/24

# Create internal network, here physnet2 is mapped to physical baremetal network
neutron net-create --shared --provider:network_type flat --provider:physical_network␣
↪physnet2 baremetal

# Create subnet in internal network, here 10.200.2.1 - is address of Neutron router,␣
↪10.11.0.174 - is address of DNS server which can resolve external endpoints
neutron subnet-create --name baremetal --gateway 10.200.2.1 --allocation-pool␣
↪start=10.200.2.10,end=10.200.2.200 --dns-nameserver 10.11.0.174 baremetal 10.200.2.
↪0/24

# Create router and connect networks
neutron router-create r1
neutron router-gateway-set r1 ext
neutron router-interface-add r1 baremetal
```

Example case when required access to Ironic services provided directly from baremetal network:

```
# Create internal network, here physnet2 is mapped to physical baremetal network
neutron net-create --shared --provider:network_type flat --provider:physical_network␣
↪physnet2 baremetal
```

```
# Create subnet in internal network, here 10.200.2.1 - is address Underlay router,␣
↪which provides required connectivity
neutron subnet-create --name baremetal --gateway 10.200.2.1 --allocation-pool␣
↪start=10.200.2.10,end=10.200.2.200 --dns-nameserver 10.11.0.174 baremetal 10.200.2.
↪0/24
```

2.9.3 Swift

Prerequirements

Make sure that Radosgw is deployed, available and configured in Glance as default Swift storage backend. Refer to *Ceph and Swift guide* guide for deploy Radosgw and configure Glance.

2.9.4 Ironic

Prerequirements

- Underlay networking
- Neutron networking
- Glance/Swift configuration

Deploy CCP with Ironic

In order to deploy CCP with Ironic you have to deploy following components: * ironic-api * ironic-conductor * nova-compute-ironic

---

Note: nova-compute-ironic is same as regular nova-compute service, but with special compute_driver required for integration Nova with Ironic. It requires neutron-openvswitch-agent running on same host. Is not possible to deploy nova-compute-ironic and regular nova-compute on same host. nova-compute-ironic has no significant load and can be deployed on controller node.

Ironic requires single endpoints for Swift and API accessible from remote baremetal network, Ingress should be configured.

Example of ccp.yaml:

```
roles:
  controller:
    [all default controller services]
    - ironic-api
    - ironic-conductor
    - nova-compute-ironic
configs:
  neutron:
    physnets:
      - name: "physnet1"
        bridge_name: "br-ex"
        interface: "ens8"
        flat: true
```

```
        vlan_range: "1001:1030"
      - name: "physnet2"
        bridge_name: "br-bm"
        interface: "ens9"
        flat: true
        vlan_range: "1001:1030"
  ceph:
    fsid: "a1adbec9-98cb-4d75-a236-2c595b73a8de"
    mon_host: "10.11.0.214"
  radosgw:
    key: "AQCDIStYGty1ERAALFeBif/6Y49s9S/hyVFXyw=="
  glance:
    swift:
      enable: true
  ingress:
    enabled: true
```

Now you're ready to deploy Ironic to existing CCP cluster.

```
ccp deploy -c ironic-api ironic-conductor nova-compute-ironic
```

Provision baremetal instance

Depends on selected deploy driver, provision procedure may differ. Basically provision require following steps: * Upload service and user's images to Glance * Create baremetal node in Ironic * Create node port in Ironic * Create appropriate flavor in Nova * Boot instance

Example with agent_ssh driver:

Note: Agent drivers will download images from Swift endpoint, in case you using self-signed certificates, make sure that agent allows unverified SSL connections.

Upload service kernel/ramdisk images, required for driver:

```
wget https://tarballs.openstack.org/ironic-python-agent/tinyipa/files/tinyipa-stable-
↪newton.vmlinuz
wget https://tarballs.openstack.org/ironic-python-agent/tinyipa/files/tinyipa-stable-
↪newton.gz

glance image-create --name kernel \
--visibility public \
--disk-format aki --container-format aki \
--file tinyipa-stable-newton.vmlinuz

glance image-create --name ramdisk \
--visibility public \
--disk-format ari --container-format ari \
--file tinyipa-stable-newton.gz
```

Upload user's image, which should be provisioned on baremetal node:

```
wget http://download.cirros-cloud.net/0.3.4/cirros-0.3.4-x86_64-disk.img

glance image-create --name cirros \
--visibility public \
```

```
--disk-format qcow2 \
--container-format bare \
--file cirros-0.3.4-x86_64-disk.img \
--property hypervisor_type='baremetal' \
--property cpu_arch='x86_64'
```

Create baremetal node with port in Ironic:

```
ironic node-create \
-n vm_node1 \
-d agent_ssh \
-i deploy_kernel=2fe932bf-a961-4d09-b0b0-72806edf05a4 \  # UUID of uploaded kernel␣
↪image
-i deploy_ramdisk=5546dead-e8a4-4ebd-93cf-a118580c33d5 \ # UUID of uploaded ramdisk␣
↪image
-i ssh_address=10.11.0.1 \ # address of hypervisor with VM (simulated baremetal␣
↪server)
-i ssh_username=user \ # credentials for ssh access to hypervisor
-i ssh_password=password \
-i ssh_virt_type=virsh \
-p cpus=1 \
-p memory_mb=3072 \
-p local_gb=150 \
-p cpu_arch=x86_64

ironic port-create -n vm_node1 -a 52:54:00:a4:eb:d5 # MAC address of baremetal server
```

Verify that node is available as Nova hypervisor:

```
ironic node-validate vm_node1 # Should has no errors in management, power interfaces
nova hypervisor-show 1 # Should output correct information about resources (cpu, mem,␣
↪disk)
```

Create nova flavor:

```
nova flavor-create bm_flavor auto 3072 150 1
nova flavor-key bm_flavor set cpu_arch=x86_64
```

Boot baremetal instance:

```
nova boot --flavor bm_flavor \
--image 11991c4e-95fd-4ad1-87a3-c67ec31c46f3 \ # Uploaded Cirros image
--nic net-id=0824d199-5c2a-4c25-be2c-14b5ab5a2838 \ # UUID of Neutron baremetal␣
↪network
bm_inst1
```

Troubleshooting

If something goes wrong, please ensure first: * You understand how Ironic works * Underlay networking is configured properly

For more information about issues, you may enable ironic.logging_debug and check logs of following pods: - nova-scheduler - nova-compute-ironic - ironic-api - ironic-conductor - neutron-server

2.10 ZeroMQ Guide

This guide provides information about how to enable zmq in the CCP.

To use zmq as an rpc backend the following steps are required:

1. *fuel-ccp-zmq* repository should be added to the repositories list:

```
repositories:
  repos:
    - git_url: https://git.openstack.org/openstack/fuel-ccp-zmq
      name: fuel-ccp-zmq
```

2. *zmq-proxy* and *redis* images should be built:

```
ccp build -c zmq-proxy redis
```

3. *rpc* service should be configured to use zmq:

```
services:
  rpc:
    service_def: zmq-proxy
```

4. *rpc* and *redis* services should be added to topology. Example of such topology provided in `fuel-ccp/etc/topology-with-zmq-example.yaml`

5. *configs* should be extended with the following values:

```
configs:
  messaging:
    backend:
      rpc: zmq
```

Pretty much the same steps required to enable zmq as a notifications backend:

```
services:
  notifications:
    service_def: zmq-proxy

configs:
  messaging:
    backend:
      notifications: zmq
```

2.11 Services Known Issues

This sections describe known issues and corresponding workarounds, if they are.

2.11.1 [Heat] WaitCondition and SoftwareDeployment resources

Problem description

CCP deploys Heat services with default configuration and changes endpoint_type from publicURL to internalURL. However such configuration in Kubernetes cluster is not enough for several type of resources like

OS::Heat::Waitcondition and OS::Heat::SoftwareDeployment, which require callback to Heat API or Heat API CFN. Due to Kubernetes architecture it's not possible to do such callback on the default port value (for heat-api it's - 8004 and 8000 for heat-api-cfn). Note, that exactly these ports are used in endpoints registred in keystone.

Also there is an issue with service domain name to ip resolving from VM booted in Openstack.

There are two ways to fix these issues, which will be described below:

- Out of the box, which requires just adding some data to `.ccp.yaml`.
- With manual actions.

Prerequisites for workarounds

Before applying workaround please make sure, that current ccp deployment satisfies the following prerequisites:

- VM booted in Openstack can be reached via ssh (don't forget to configure corresponding security group rules).
- IP address of Kubernetes node, where heat-api service is run, is accessible from VM booted in Openstack.

Workaround out of the box

This workaround is similar for both resources and it's related to kubernetes node external ip usage node with hardcoded node port in config.

1. Add the following lines in the config `.ccp.yaml`:

```
k8s_external_ip: x.x.x.x
heat:
  heat_endpoint_type: publicURL
  api_port:
    node: 31777
  api_cfn_port:
    node: 31778
```

   Where `x.x.x.x` is IP of kubernetes node, where Heat services are run. The second line explicitly sets publicURL in Heat config for initialisation of the heat client with public endpoint. Next lines set hardcoded ports for services: heat-api and heat-api-cfn. User may choose any free port from K8S range for these services.

   All these options should be used together, because external ip will be used by ccp only with node ports. Also combination of IP and port will be applied only for public enpoint.

2. After this change you may run `ccp deploy` command.

Warning: There are two potential risks here:

- Specified node port is in use by some other service, so user needs to change another free port.
- Using heatclient with enabled ingress can be broken. It was not tested fully yet.

Workaround after deploy

This workaround can be used, when Openstack is already deployed and cloud administrator can change only one component.

1. Need to gather information about Node Ports and IP of Kubernetes node with services. User may get `Node Ports` for all heat API services by using the following commands:

```
# get Node Port API
kubectl get service heat-api -o yaml | awk '/nodePort: / {print $NF}'

# get Node Port API CFN
kubectl get service heat-api-cfn -o yaml | awk '/nodePort: / {print $NF}'
```

Obtain `service IP` by executing `ping` command from Kubernetes host to domain names of services (e.g. `heat-api.ccp`).

2. Then these IP and Node ports should be used as internal endpoints for corresponding services in keystone, i.e. replace old internal endpoints with domain names to IP with Node Ports for heat-api and heat-api-cfn. It should look like:

```
# delete old endpoint
openstack endpoint delete <id of internal endpoints>

# create new endpoint for heat-api
openstack endpoint create --region RegionOne \
orchestration internal http://<service IP>:<Node Port API>/v1/%\(tenant_id\)s

# create new endpoint for heat-api-cfn
openstack endpoint create --region RegionOne \
cloudformation internal http://<service IP>:<Node Port API CFN>/v1/
```

Note: For Waitcondition resource validation simple heat template can be used.

1. The previous steps should be enough for fixing Waitcondition resource. However for SoftwareDeployment usage it is necessary to remove two options from `fuel-ccp-heat/service/files/heat.conf.j2 file`:

   - heat_waitcondition_server_url
   - heat_metadata_server_url

   It's necessary, because otherwise they will be used instead of internal endpoints. Such change requires partial redeploy, which can be done with commands:

```
ccp deploy -c heat-engine heat-api heat-api-cfn
```

To validate, that this change was applied just check, that new containers for these services were started.

2.12 Neutron Configuration

This guide provides instructions on configuring Neutron with Fuel-CCP.

2.12.1 Tenant network types

By default Neutron is configured to use VxLAN segmentation but it is possible to specify other network types like VLAN or flat.

To do so add the following lines to the `configs.neutron` section of the CCP configuration file:

```
ml2:
  tenant_network_types:
    - "vlan"
    - "vxlan"
```

Here `tenant_network_types` is an ordered list of network types to allocate as tenant networks. Enabling several network types allows creating networks with `--provider:network_type` equalling one of these types, if `--provider:network_type` is not specified then the first type from the `tenant_network_types` list will be used.

It is also possible to specify VxLAN VNI and VLAN ID ranges.

VxLAN VNI ranges are configured in `configs.neutron.ml2` section with default range being "1:1000".

```
ml2:
  tenant_network_types:
    - "vxlan"
  vni_ranges:
    - "1000:5000"
```

VLAN ranges are configured per each physical network in the `configs.neutron.physnets` section:

```
physnets:
  - name: "physnet1"
    bridge_name: "br-ex"
    interface: "eno2"
    flat: false
    vlan_range: "1050:2050"
    dpdk: false
```

For more information on configuring physical networks refer to the QuickStart Guide.

CHAPTER 3

Service plugins docs

3.1 Searchlight CCP plugin documentation

This is Fuel-CCP plugin for OpenStack Searchlight service.

Original searchlight service developer docs placed here.

3.1.1 Overview

The Searchlight project provides indexing and search capabilities across OpenStack resources. Its goal is to achieve high performance and flexible querying combined with near real-time indexing. It uses Elasticsearch, a real-time distributed indexing and search engine built on Apache Lucene, but adds OpenStack authentication and Role Based Access Control to provide appropriate protection of data.

CCP plugin has two components for searchlight service:

- `searchlight-api`
- `searchlight-listener`

So searchlight docker images are the following:

- `ccp/searchlight-api`
- `ccp/searchlight-listener`

You can deploy them with other components using *Quick Start*.

3.1.2 Dependencies

Searchlight depends on several services:

- Elasticsearch. Searchlight services depends on elasticsearch service, which should be deployed on env before searchlight installation. To deploy elasticsearch, it should be specified in CCP config file in components' list and (optionally, if you specified repositories manually) add next repo to repositories repos' list:

```
git_url: https://git.openstack.org/openstack/fuel-ccp-stacklight
name: fuel-ccp-stacklight
```

- Indexed services. Searchlight builds index on observed services, so should be deployed after them - index will be not complete with all resources from observed resources instead.

3.1.3 Configuration

Searchlight provides indexing and searching for several services, listed here. CCP plugin allows to specify, which services searchlight will handle. For enabling/disabling service, which you want to index and listen for updates, you need to change value *searchlight.services.<desirable service>* to *true* in `services/files/defaults.yaml` (and *false* to disable). After that you need to restart searchlight components and corresponding api component of service you enabled in config, if you already deployed components.

3.1.4 Installation

To install and configure searchlight service, you should follow next steps:

1. Ensure, that elasticsearch is ready to use. You can, for example, list all indices:

   curl -X GET elasticsearch.ccp:<elasticport>/_cat/indices?v

   where *elasticport* is elasticsearch port, which can be found with command:

   ```
   kubectl get svc elasticsearch -o yaml | awk '/port:/ {print $NF}'
   ```

   and it equals to 9200 by default.

   You'll get table with next header (if you don't use elasticsearch before, table will be empty):

   health status index pri rep docs.count docs.deleted store.size pri.store.size

2. Add *searchlight-api* and *searchlight-listener* services to your CCP configuration file (e.g. *.ccp.yaml*).

3. Deploy these components with command:

   ```
   ccp deploy -c searchlight-api searchlight-listener
   ```

   and wait until their won't be available.

4. Install *python-searchlightclient* and also install/update *python-openstackclient* with pip:

   ```
   pip install --user -U python-searchlightclient python-openstackclient
   ```

5. Check availability of searchlight with command **`openstack search resource type list`**, which will display all supported resource types to search.

3.1.5 Dashboard plugin

Searchlight has horizon dashboard plugin, which allows you to search and filter resources and get detailed information about it. It already available in horizon and activates, when searchlight is on board. Search panel places in `Projects` menu.

3.2 Designate CCP plugin documentation

This is Fuel-CCP plugin for OpenStack Designate service.

Original designate service developer docs are placed here.

3.2.1 Overview

Designate provides DNSaaS services for OpenStack. Designate architecture has next components:

- `designate-api` – provides the standard OpenStack style REST API service;
- `designate-central` – is the service that handles RPC requests via the MQ, it coordinates the persistent storage of data and applies business logic to data from the API;
- `designate-mdns` – is the service that sends DNS NOTIFY and answers zone transfer (AXFR) requests;
- `designate-pool-manager` – is a service that handles the states of the DNS servers Designate manages. Since mitaka replaced with `designate-worker` service;
- `designate-zone-manager` – is a service that handles all periodic tasks related to the zone shard it is responsible for;
- `designate-sink` – is an optional service which listens for event notifications, such as compute.instance.create.end. Currently supports Nova and Neutron;
- `designate-agent` – pool manager agent backend. This is an optional service. Agent uses an extension of the DNS protocol to send management requests to the remote agent processes, where the requests will be processed.

CCP components comprises next services:

- `designate-api`;
- `designate-central`;
- `designate-mdns`, which contains three containers: `designate-mdns` service, `designate-worker` and `designate-backend-bind9` - container, which implements bind9 backend for designate. All of them works in collaboration and provide ability to create and manage zones and records;
- `designate-agent`;
- `designate-sink`;
- `designate-pool-manager`;
- `designate-zone-manager`.

Three last services are optional and can't be omitted during deployment.

3.2.2 Configuration

Designate has configurable options for each component, which could be set for specific node with *nodes* configs section. These options are: *workers* and *threads*. They are placed in *designate.service.<service name>.<workers or threads>* configs path. Also, designate CCP plugin allows to configure defaults of domain purge: *interval*, *batch_size* and *time threshold*.

CCP designate plugin has bind9 backend implemented; it enabled by default with option *designate.backend*. If you want to turn off any backend, clear option's value - then fake backend, which has no effect for designate will be enabled.

3.2.3 Installation

Currently designate CCP plugin is not supported by default, so installation has next steps:

1. Add next item to `repositories.repos` list of CCP configuration file:

```
- git_url: https://git.openstack.org/openstack/fuel-ccp-designate
  name: fuel-ccp-designate
```

2. Add designate components to roles list. Next components are required:

```
- designate-api
- designate-central
- designate-mdns
```

   Components `designate-sink`, `designate-agent`, `designate-zone-manager` and `designate-pool-manager` are optional and could not be deployed.

3. Fetch, build, deploy components.
4. Install *python-designateclient* and also install/update *python-openstackclient* with pip:

```
pip install --user -U python-designateclient python-openstackclient
```

3.2.4 Dashboard plugin

Designate has horizon dashboard plugin, which allows to create and manage domains and records. It is already available in horizon and is activated when designate is on board. Domain panel is placed in `Projects` menu.

CHAPTER 4

Developer docs

4.1 How To Contribute

4.1.1 General info

1. Bugs should be filed on launchpad, not GitHub.
2. Please follow OpenStack Gerrit Workflow to contribute to CCP.
3. Since CCP has multiple Git repositories, make sure to use Depends-On Gerrit flag to create cross repository dependencies.

4.1.2 Useful documentation

- Please follow our *Quick Start* guide to deploy your environment and test your changes.
- Please refer to *CCP Docker images guide*, while making changes to Docker files.
- Please refer to *Application definition contribution guide*, while making changes to `service/*` files.

4.2 Application definition contribution guide

This document gives high overview of component repository structure.

4.2.1 Overview

CCP provides wide spectrum of operations for microservices manipulations on Kubernetes cluster. Each microservice is an independent component with common architecture. Whole data releated to component can be found in the corresponding repository. The full list of the related components can be found by link, where each repository has prefix `fuel-ccp-*`.

4.2.2 Structure

Component repositories have common structure:

1. Docker image related data

`docker` folder with Docker files, which will be used for building docker images. Each subfolder will be processed as a separate image for building. See detailed instructions are available in the *CCP Docker images guide*.

2. Application definition files

All application definition files should be located in the `service/` directory, as a `component_name.yaml` file, for example:

```
service/keystone.yaml
```

Please refer to *Application definition language* for detailed description of CCP DSL syntax.

3. Application related scripts and configs

All templates, such as configs, scripts, etc, which will be used for this service, should be located in `service/<component_name>/files`, for example:

```
service/files/keystone.conf.j2
```

All files inside this directory are Jinja2 templates, except the file with default variables. Default variables for these templates should be located in `service/files/defaults.yaml` inside the following section.

```
configs:
  <service_name>:
```

Description of available values can be found in the following guide *Configuration files*.

4. Shared configurations templates

You can export and share across all *fuel-ccp-x* repositories the most common parts of configs which are needed to use your service. In order to do this you should locate a jinja macros with a config template in `./exports/` directory:

```
./exports/your_jinja_template.j2
```

and then use it in a config file of any other repository:

```
file:nova.conf.j2
{{ your_jinja_template.your_macros() }}
```

Well known shared template is oslo_messaging

4.3 CCP Docker images guide

This guide covers CCP specific requirements for defining Docker images.

4.3.1 Docker files location

All docker files should be located in `docker/<component_name>` directory, for example:

```
docker/horizon
docker/keystone
```

The docker directory may contain multiple components.

4.3.2 Docker directory structure

Each docker directory should contain a `Dockerfile.j2` file. Dockerfile.j2 is a file which contains Docker build instructions in a Jinja2 template format. You can add additional files, which will be used in Dockerfile.j2, but only Dockerfile.j2 can be a Jinja2 template in this directory.

4.3.3 Dockerfile format

Please refer to the official Docker documentation which covers the Dockerfile format. CCP has some additional requirements, which is:

1. Use as few layers as possible. Each command in Dockerfile creates a layer, so make sure you're grouping multiple RUN commands into one.
2. If it's possible, please run container from the non-root user.
3. If you need to copy some scripts into the image, please place them into the `/opt/ccp/bin` directory.
4. Only one process should be started inside container. Do not use runit, supervisord or any other init systems, which will allow to spawn multiple processes in container.
5. Do not use CMD and ENTRYPOINT commands in Dockerfile.j2.
6. All OpenStack services should use `openstack-base` parent image in FROM section. All non-OpenStack services should use `base-tools` parent image in FROM section.

Here is an example of valid Dockerfile.j2: Keystone Dockerfile

Supported Jinja2 variables

Only specific variables can actually be used in Dockerfile.j2:

1. `namespace` - Used in the FROM section, renders into image namespace, by default into `ccp`.
2. `tag` - Used in the FROM section, renders into image tag, by default into `latest`.
3. `maintainer` - Used in the MAINTAINER section, renders into maintainer email, by default into "MOS Microservices <mos-microservices@mirantis.com>"
4. `copy_sources` - Used anywhere in the Dockerfile. please refer to corresponding documentation section below.

5. Additionaly, you could use variables with software versions, please refer to *Application definition contribution guide* for details.

copy_sources

The CCP CLI provides additional feature for Docker images creation, which will help to use git repositories inside Dockerfile, it's called `copy_sources`.

This feature uses configuration from `service/files/defaults.yaml` from the same repository or from global config, please refer to *Application definition contribution guide* for details.

4.3.4 Testing

After making any changes in docker directory, you should test it via build and deploy.

To test building, please run:

```
ccp build -c <component_name>
```

For example:

```
ccp build -c keystone
```

Make sure that image is built without errors.

To test the deployment, please build new images using the steps above and after run:

```
ccp deploy
```

Please refer to *Quick Start* for additional information.

4.4 Application definition language

There is a description of current syntax of application definition framework.

4.4.1 Application definition template

```
service:
    name: service-name
    kind: DaemonSet
    ports:
        - internal-port:external-port
    headless: true
    hostNetwork: true
    hostPID: true
    antiAffinity: local
    annotations:
        pod:
          description: frontend ports
        service:
          description: frontend service
    containers:
        - name: container-name
```

```
          image: container-image
          probes:
              readiness: readiness.sh
              liveness: liveness.sh
          volumes:
              - name: volume-name
                type: host
                path: /path
          pre:
              - name: service-bootstrap
                dependencies:
                    - some-service
                    - some-other-service
                type: single
                image: some_image
                command: /tmp/bootstrap.sh
                files:
                    - bootstrap.sh
                user: user
              - name: db-sync
                dependencies:
                    - some-dep
                command: some command
                user: user
          daemon:
              dependencies:
                  - demon-dep
              command: daemon.sh
              files:
                  - config.conf
              user: user
          post:
              - name: post-command
                dependencies:
                    - some-service
                    - some-other-service
                type: single
                command: post.sh
                files:
                    - config.conf

files:
    config.conf:
        path: /etc/service/config.conf
        content: config.conf.j2
        perm: "0600"
        user: user
    bootstrap.sh:
        path: /tmp/bootstrap.sh
        content: bootstrap.sh.j2
        perm: "0755"
```

4.4.2 Parameters description

service

Name	Description	Required	Schema	Default
name	Name of the service.	true	string	–
kind	Kind of k8s object to use for containers deployment.	false	one of: [”Deployment”, “DaemonSet”, “StatefulSet”]	Deployment
containers	List of containers under multi-container pod.	true	*container* array	–
ports	k8s Service will be created if specified (with NodePort type for now). Only internal or both internal:external ports can be specified.	false	internal-port: external-port array	–
hostNetwork	Use the host’s network namespace.	false	boolean	false
headless	Create headless service.	false	boolean	false
hostPID	Use the host’s pid namespace.	false	boolean	false
strategy	The strategy that should be used to replace old Pods by new ones.	false	one of: [”RollingUpdate”, “Recreate”]	RollingUpdate
antiAffinity	Restrict scheduling of pods on the same host: local - within namespace, global - within k8s cluster	false	one of: [null, “global”, “local”]	null
annotations	pod - annotations for pods, service - annotations for service.	false	string dict	null

container

Name	Description	Required	Schema	Default
name	Name of the container. It will be used to track status in etcd.	true	string	–
image	Name of the image. registry, namespace, tag will be added by framework.	true	string	–
probes	Readiness, liveness or both checks can be defined. Exec action will be used for both checks.	false	dict with two keys: **liveness:** cmd **readiness:** cmd	–
volumes	–	false	*volume* array	–
pre	List of commands that need to be executed before daemon process start.	false	*command* array	–
daemon	–	true	*command*	–
post	The same as for "pre" except that post commands will be executed after daemon process has been started.	false	*command* array	–
env	An array of environment variables defined in kubernetes way.	false	env array	–

volume

Name	Description	Required	Schema	Default
name	Name of the volume.	true	string	–
type	host and empty-dir type supported for now.	true	one of: ["host", "empty-dir"]	–
path	Host path that should be mounted (only if type = "host").	false	string	–
mountpath	Mount path in container.	false	string	path
readOnly	Mount mode of the volume.	false	bool	False

command

Name	Description	Required	Schema	Default
name	Name of the command. Required only for *pre* and *post* with type *single*.		string	
image	Image that will be used to run the command. Can be used only for *pre* and *post* with type *single*.	false	string	same as for daemon
command	–	true	string	–
dependencies	These keys will be polled from etcd before commands execution.	false	string array	–
type	type: single means that this command should be executed once per openstack deployment. For commands with type: single Job object will be created. type: local (or if type is not specified) means that command will be executed inside the same container as a daemon process.	false	one of: ["single", "local"]	local
files	List of the files that maps to the keys of files dict. It defines which files will be rendered inside a container.	false	*file* keys array	–
user	–	false	string	–

files

Name	Description	Required	Schema	Default
Name of the file to refer in files list of commands	–	false	*file* array	–

file

Name	Description	Required	Schema	Default
path	Destination path inside a container.	true	string	–
content	Name of the file under {{ service_repo }}/service/files directory. This file will be rendered inside a container and moved to the destination defined with path.	true	string	–
perm	–	false	string	–
user	–	false	string	–

4.4.3 DSL versioning

Some changes in CCP framework are backward compatible and some of them are not. To prevent situations when service definitions are being processed by incompatible version of CCP framework, DSL versioning has been implemented.

DSL versioning is based on Semantic Versioning model. Version has a format `MAJOR.MINOR.PATCH` and is being defined in `dsl_version` field of `fuel_ccp/__init__.py` module. Each service definition contains `dsl_version` field with the version of DSL it was implemented/updated for.

During the validation phase of **`ccp deploy`** those versions will be compared according to the following rules:

1. if DSL version of `fuel-ccp` is less than service's DSL version - they are incompatible - error will be printed, deployment will be aborted;
2. if `MAJOR` parts of these versions are different - they are incompatible - error will be printed, deployment will be aborted;
3. otherwise they are compatible and deployment can be continued.

For `dsl_version` in `fuel-ccp` repository you should increment:

1. MAJOR version when you make incompatible changes in DSL;
2. MINOR version when you make backward-compatible changes in DSL;
3. PATCH version when you make fixes that do not change DSL, but affect processing flow.

If you made a change in service definition that is not supposed to work with the current `` `dsl_version` ``, you should bump it to the minimal appropriate number.

4.5 Debugging microservice/application

This part of the documentation contains some practice recommendations, which can be used for debugging some issues in service code.

4.5.1 Problem description

Workable service is perfect, but sometimes user may be in situation, when application does not work as expected or fails with some unknown status. Obviously if service can not provide clear traceback or logs, there is no another option except debug this service. Let's take a look on some useful how to do it and use heat-engine service as example.

4.5.2 How to debug

1. Create a local copy of the source code related project.

```
cd /tmp
git clone http://github.com/openstack/heat
```

2. Do all necessary changes, i.e. add breakpoint and etc., in this source code.
3. Update global configuration file by using local source for heat service.

```
sources:
  openstack/heat:
    source_dir: /tmp/heat
```

4. Build new image and re-deploy heat service:

```
ccp build -c heat-engine
ccp deploy -c heat-engine
```

5. Login in container and enjoy debugging.

Note: This approach is really pure for understanding, but has one issue. If you want to change code again you need to repeat all operations from building image again.

4.5.3 Another way to debug

The idea is to run new process with necessary changes in code (breakpoints) inside already created container for current service. Execute follow commands to run bash in container with heat-engine service:

```
kubectl get pods | grep heat-engine
kubectl exec -it <id of pod from previous command> bash
```

So now bash is run in container. There are two new issues here:

1. It's not possible to change service source files, because we are logged as `heat` user.
2. If heat-engine process be killed, Kubernetes detect it and re-create container.

Both issues can be solved by changing Docker image and Service definition.

- First of all change user `heat` used in container to `root`. It should be done in file: `fuel-ccp-heat/docker/heat-engine/Dockerfile.j2`.

Note: There is also alternative way to obtain root access is container:

1. find node where pod with heat-engine was run

```
kubectl get pods -o wide | grep heat-engine
```

2. ssh to this node with heat-engine pod
3. find id of container with heat-engine and run bash as root user

```
docker ps | grep heat-engine
docker exec -it -u root <heat-container-id> bash
```

- The next step is to change run command in service definition: `fuel-ccp-heat/service/heat-engine.yaml`. Find key word `command`, comment it and write follow code:

```
command: sleep 1h
```

  It will allow to run container for 1 hour without real heat-engine process, it's usually enough for leisurely debugging.

To ship new container to the CCP is necessary to build new image and then re-deploy it:

```
ccp build -c heat-engine
ccp deploy -c heat-engine
```

When re-deploy is finished, run bash in new container again. The source code is placed in follow directory:

```
/var/lib/microservices/venv/lib/python2.7/site-packages/
```

Change it by adding necessary breakpoints.

Note: Text editor `vim` can work incorrect from container. For fixing it try to execute command: export TERM=xterm

The last step is to run updated service code. Execute command, which was commented in service definition file, in the current example it's:

```
heat-engine --config-file /etc/heat/heat.conf
```

Now patched service is active and can be used for debugging.

4.6 Diagnostic snapshot

In `fuel-ccp/tools` directory you can find tool called `diagnostic-snapshot.sh`. This tool helps to collect some debug data about your environment. You can run it with:

```
./tools/diagnostic_snapshot -n <namespace> -o <output_dir> -c <ccp_config>
```

4.6.1 parameters

Short option	Long option	Description
-n	–namespace	deployment namespace
-o	–output-dir	directory where diagnostic snapshot will be saved
-c	–config	should point to Fuel-ccp config file
-h	–help	print help

This tool collect some basic data about:

- k8s objects in kube-system and ccp namespaces:
  - pods
  - services
  - jobs

  - kubelet logs
- system:
  - diskspace
  - network configuration
  - cpu info/load
  - sysctl info
- docker:
  - logs
  - list of images
  - running containers
  - stats
- ccp:
  - status output

This script automatically create directory provided as parameter for -o option and archive file in it with all collected data. The name of this file is created with template: <datetime>-diagnostic.tar.gz

CHAPTER 5

Design docs

5.1 Clusters On Kubernetes

This document describes an architecture of Galera and RabbitMQ Clusters running in containers within Kubernetes pods and how to setup those in OpenStack on top of Kubernetes from deployment and networking standpoints. In addition to it, this document includes overview of alternative solutions for implementing database and message queue for OpenStack.

5.1.1 RabbitMQ Architecture with K8s

Clustering

The prerequisite for High Availability of queue server is the configured and working RabbitMQ cluster. All data/state required for the operation of a RabbitMQ cluster is replicated across all nodes. An exception to this are message queues, which by default reside on one node, though they are visible and reachable from all nodes. [1]

Cluster assembly requires installing and using a clustering plugin on all servers. The following choices are considered in this document:

- rabbitmq-autocluster
- rabbitmq-clusterer

rabbit-autocluster

Note that the plugin 'rabbitmq-autocluster' has unresolved issue that can cause split-brain condition to pass unnoticed by RabbitMQ cluster. This issue must be resolved before this plugin can be considered production ready.

The RabbitMQ cluster also needs proper fencing mechanism to exclude split brain conditions and preserve a quorum. Proposed solution for this problem is using 'pause_minority' partition mode with the rabbit-autocluster plugin, once the issue with silent split brain is resolved. See the following link for the proof of concept implementation of the K8s driven RabbitMQ cluster: https://review.openstack.org/#/c/345326/.

rabbit-clusterer

Plugin 'rabbitmq-clusterer' employs more opinionated and less generalized approach to the cluster assembly solution. It is also cannot be directly integrated with etcd and other K8s configuration management mechanisms because of static configuration. Additional engineering effort required to implement configuration middleware. Because of that it is considered a fallback solution.

Replication

Replication mechanism for RabbitMQ queues is known as 'mirroring'. By default, queues within a RabbitMQ cluster are located on a single node (the node on which they were first declared). This is in contrast to exchanges and bindings, which can always be considered to be on all nodes. Queues can optionally be made mirrored across multiple nodes. Each mirrored queue consists of one master and one or more slaves, with the oldest slave being promoted to the new master if the old master disappears for any reason. [2]

Messages published to the queue are replicated to all members of the cluster. Consumers are connected to the master regardless of which node they connect to, with slave nodes dropping messages that have been acknowledged at the master. Queue mirroring therefore aims to enhance availability, but does not distribute load across nodes (all participating nodes each do all the work). It is important to note that using mirroring in RabbitMQ actually reduces the availability of queues by dropping performance by about 2 times in performance tests. See below for the list of issues identified in the RabbitMQ mirroring implementation. [6-13]

There are two main types of messages in OpenStack:

- Remote Procedure Call messages carry commands and/or requests between microservices within a single component of OpenStack platform (e.g. nova-conductor to nova-compute).
- Notification messages are issued by a microservice upon specific events and are consumed by other components (e.g. Nova notifications about creating VMs are consumed by Ceilometer).

In proposed OpenStack architecture, only notification queues are mirrored as they require durability and should survive a failure of any single node in the cluster. All other queues are not, and if the instance of RabbitMQ server that hosts a particular queue fails after a message sent to that queue, but before it is read, that message is gone forever. This is a trade-off for significant (2 times) performance boost in potential bottleneck service. Potential drawbacks of this mode of operation are:

- Long-running tasks might stuck in transition states due to loss of messages. For example, Heat stacks might never leave spawning state. Most of the time, such conditions could be fixed by the user via API.

Data Persistence

OpenStack does not impose requirements for durable queues or messages. Thus, no durability required for RabbitMQ queues, and there is no 'disk' nodes in cluster. Restarting a RabbitMQ node then will cause all data of that node to be lost, both for RPC and Notification messages.

- RPC messages are not supposed to be guaranteed, thus no persistence is needed for them.
- Notifications will be preserved by mirroring if single RabbitMQ node fails (see above).

Networking Considerations

RabbitMQ nodes address each other using domain names, either short or fully-qualified (FQDNs). Therefore hostnames of all cluster members must be resolvable from all cluster nodes, as well as machines on which command line tools such as rabbitmqctl might be used.

RabbitMQ clustering has several modes of dealing with network partitions, primarily consistency oriented. Clustering is meant to be used across LAN. It is not recommended to run clusters that span WAN. The Shovel or Federation plugins are better solutions for connecting brokers across a WAN. Note that Shovel and Federation are not equivalent to clustering. [1]

Kubernetes Integration

Clustering plugins need configuration data about other nodes in the cluster. This data might be passed via etcd to RabbitMQ startup scripts. ConfigMaps are used to pass the data into containers by Kubernetes orchestration.

The RabbitMQ server pods shall be configured as a DaemonSet with corresponding service. Physical nodes shall be labelled so as to run the containers with RabbitMQ on dedicated nodes, one pod per node (as per DaemonSet), or co-located with other control plane services.

PetSets are not required to facilitate the RabbitMQ cluster as the servers are stateless, as described above.

Proposed solution for running RabbitMQ cluster under Kubernetes is a DaemonSet with node labels to specify which nodes will run RabbitMQ servers. This will allow to move the cluster onto a set of dedicated nodes, if necessary, or run them on the same nodes as the other control plane components.

Alternatives

ZeroMQ

This library provides direct exchange of messages between microservices. Its architecture may include simple brokers or proxies that just relay messages to endpoints, thus reducing the number of network connections.

ZeroMQ library support was present in OpenStack since early releases. However, the implementation assumed direct connections between services and thus a full mesh network between all nodes. This architecture doesn't scale well. More recent implementations introduce simple proxy services on every host that aggregate messages and relay them to a central proxy, which does host-based routing.

Benchmarks show that both direct and proxy-based ZeroMQ implementations are more efficient than RabbitMQ in terms of throughput and latency. However, in the direct implementation, quick exhaustion of network connections limit occurs at scale.

The major down side of the ZeroMQ-based solution is that the queues don't have any persistence. This is acceptable for RPC messaging, but Notifications require durable queues. Thus, if RPC is using ZeroMQ, the Telemetry will require a separate messaging transport (RabbitMQ or Kafka).

Demo Recording

The following recording demonstrates how RabbitMQ cluster works as a DaemonSet on K8s version 1.3 with rabbit-autocluster plugin.

5.1.2 Galera Architecture with K8s

Galera is synchronous multi-master database cluster, based on synchronous replication. At a high level, Galera Cluster consists on database server that uses Galera Replication plugin to manage replication. Through the wsrep API, Galera Cluster provides certification-based replication. A transaction for replication, the write-set, not only contains the database rows to replicate, but also includes information on all the locks that were held by the database during the transaction. Each node then certifies the replicated write-set against other write-sets in the applier queue. The write-set is then applied, if there are no conflicting locks. At this point, the transaction is considered committed, after which

each node continues to apply it to the tablespace. This approach is also called virtually synchronous replication, given that while it is logically synchronous, the actual writing and committing to the tablespace happens independently, and thus asynchronously on each node.

How Galera Cluster works

The primary focus is data consistency. The transactions are either applied to every node or not all. In a typical instance of a Galera Cluster, applications can write to any node in the cluster and transaction commits, (RBR events), are then applied to all the servers, through certification-based replication. Certification-based replication is an alternative approach to synchronous database replication, using group communication and transaction ordering techniques. In case of transaction collisions the application should be able to handle 'failed' transactions. Openstack Applications use oslo.db which has retry logic to rerun failed transaction.

Starting the cluster

By default, nodes do not start as part of the Primary Component (PC). Instead, they assume that the Primary Component exists already somewhere in the cluster.

When nodes start, they attempt to establish network connectivity with the other nodes in the cluster. For each node they find, they check whether or not it is a part of the Primary Component. When they find the Primary Component, they request a state transfer to bring the local database into sync with the cluster. If they cannot find the Primary Component, they remain in a nonoperational state.

There is no Primary Component when the cluster starts. In order to initialize it, you need to explicitly tell one node to do so with the –wsrep-new-cluster argument. By convention, the node you use to initialize the Primary Component is called the first node, given that it is the first that becomes operational.

When cluster is empty, any node can serve as the first node, since all databases are empty. In case of failure (power failure) the node with the most recent data should initialize Primary Component.

Node Provisioning

There are two methods available in Galera Cluster to provision nodes:

- State Snapshot Transfer (SST) where a snapshot of entire node state is transferred
- Incremental State Transfer (IST) where only missing data transactions are replayed

In SST, the cluster provisions nodes by transferring a full data copy from one node to another. When a new node joins or when it was offline (or left behind cluster) longer than IST buffer a new node (JOINER) initiates a SST to synchronize data.

In IST, the cluster provisions a node by identifying the missing transactions on the JOINER to send them only, instead of transferring entire state.

Networking Considerations

Load Balancing is a key element of networking configuration of the Galera cluster. Load balancer must be coordinated with the cluster, in terms that it redirect write requests to appropriate Galera Pod which has Sync state. Communication with Galera Pods that have any other state (OPEN, PRIMARY, JOINER, JOINED, DONOR) should be prohibited. Load Balancer also ensures failover to hot stand-by instances and fencing of failed active nodes.

The following options are considered for load balancer in K8s integration of Galera:

- Kubernetes Load Balancing

- HAProxy
- ProxySQL

Storage Considerations

Since every nodes in Galera Cluster has a copy of the data set at any time, there is no need to use networked storage (NFS, Ceph, GlasterFS). All Galera Pods can work with the local disk storage (Directory, LVM). From the Kubernetes standpoint, it means that local persistent volume must be mounted to Galera Pod on the same node. From the Kubernetes Scheduler standpoint, it means that Galera Pods should run on the nodes where Persistent Volume is created. At the same time, networking storage might be useful as in that case PV claimed on it can be assigned to any node eliminating bottleneck in Architecture. Using networking storage such as ceph might significantly improve SST operation though database write operations will be slower than local storage.

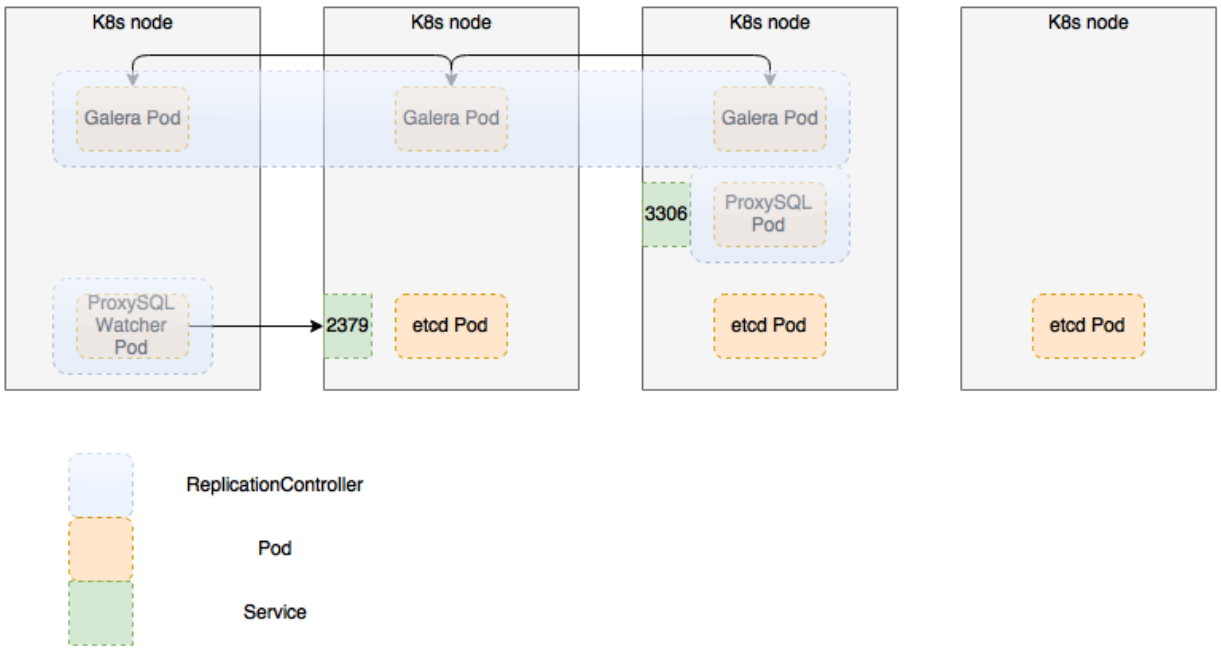
The following clustering solutions considered for Galera Cluster:

- Replication Controller with proxy (ProxySQL is used): An etcd cluster with startup scripts controlling assembly and liveliness of the cluster's nodes, for example:
  - https://github.com/percona/percona-docker/tree/master/pxc-56
  - https://github.com/Percona-Lab/percona-xtradb-cluster-docker
- PetSet:
  - https://github.com/kubernetes/contrib/blob/master/pets/mysql/galera/mysql-galera.yaml
- Replication controller with proxy and additional watcher
  - https://review.openstack.org/367650

Replication Controller Schema with additional proxy and watcher

The proposed solution is based on the native Kubernetes state management with etcd providing distributed monitoring and data exchange for the cluster. Cluster operations will be triggered by Kubernetes events and handled by custom scripts.

Failover and fencing of failed instances of Galera Cluster is provided by scripts triggered by Kubernetes upon the changes in state and availability of the members of Galera Cluster. State and configuration information is provided by etcd cluster.



- Proposed architecture allows to quickly replace failing instances of MySQL server without need to run full replication. It is still necessary to restore the pool of hot-stand-by instances whenever the failover event occurs.
- Additional proxy is stateless, e.g. it does not contain state and can be re-scheduled by k8s in case of failure
- Watcher is stateless as well, and is capable of populating the state from etcd to ProxySQL
- Additional proxy brings the benefit of more granular control over MySQL connections, which is not possible with k8s service:
  - Forward all writes to one node or special group of nodes (not implemented in current scheme, but can be easily added), and Reads to the rest of the group;
  - Central mysql cache;
  - Rate limits on per-user basis;
  - Hot-standby nodes can be added to the pool but not activated by default
- Storage considerations are the same as for PetSets, see below.

Future enhancements of this PoC may include:

- Rework custom bootstrap script and switch to election plugin for K8s.
- Integrate extended Galera checker that supports hostgroups (like this one)

Demo Recording

The following recording demonstrates how Galera cluster works as a Replication Controller on K8s 1.3 with ProxySQL middleware. It includes destructive test when one of instances of MySQL server in the cluster is shut off.

Open Questions

- ProxySQL requires management, and that is why watcher was written. Since it terminates queries, users/password should be managed in two places now: in MySQL and in ProxySQL itself.

- K8s does not have bare-metal storage provider (only cloud based ones), and it is crucial for any stateful application in self-hosted clouds. Until that is ready, no stateful application can actually go production.

PetSet Schema

Storage for database files shall be supported as one of the following options:

- a local file/LV configured as al HostPath volume and mounted to every pod in set;
- a remote SAN/NAS volume mounted to every pod;
- a volume or file on a shared storage (Ceph) configured as volume and mounted to every pod.

Persistent volumes for Galera PetSets must be created by the K8s installer, which is out of scope of this document.

Demo Recording

The following video *recording <https://asciinema.org/a/87411>* demonstrates how Galera MySQL cluster is installed and works as PetSet on Kubernetes with local volumes for persistent storage.

5.1.3 Galera Cluster Rebuild Problem

In case of general cluster failure or planned maintenance shutdown, all pods in Galera cluster are destroyed. When the new set of pods is started, they have to recover the cluster status, rebuild the cluster and continue from the last recorded point in time.

With local storage, Galera pods mount volumes created as a directory (default) or LVM volume (WIP). These volumes are used to store database files and replication logs. If cluster has to be rebuilt, all pods are assumed to be deleted, however, the volumes should stay and must be reused in the rebuild process. With local non-mobile volumes it means that new pods must be provisioned to the very same nodes they were running on originally.

Another problem is that during the rebuild process it is important to verify integrity and consistency of data on all static volumes before assembling the cluster and select a Primary Component. There are following criteria for this selection:

- The data must be readable, consistent and not corrupted.
- The most recent data set should be selected so the data loss is minimal and it could be used to incrementally update other nodes in the cluster via IST.

Currently, k8s scheduler does not allow for precise node-level placement of pods. It is also impossible to specify affinity of a pod to specific persistent local volume. Finally, k8s does not support LVM volumes out of the box.

5.1.4 References

This section contains references to external documents used in preparation of this document.

1. RabbitMQ Clustering
2. RabbitMQ High Availability
3. https://github.com/percona/percona-docker/tree/master/pxc-56
4. https://github.com/Percona-Lab/percona-xtradb-cluster-docker
5. http://docs.openstack.org/developer/performance-docs/test_results/mq/rabbitmq/index.html
6. https://github.com/rabbitmq/rabbitmq-server/issues/802

7. https://github.com/rabbitmq/rabbitmq-server/issues/803
8. https://github.com/rabbitmq/rabbitmq-server/pull/748
9. https://github.com/rabbitmq/rabbitmq-server/issues/616
10. https://github.com/rabbitmq/rabbitmq-server/pull/535
11. https://github.com/rabbitmq/rabbitmq-server/issues/368
12. https://github.com/rabbitmq/rabbitmq-server/pull/466
13. https://github.com/rabbitmq/rabbitmq-server/pull/431

5.2 OpenStack Compute node / VMs on K8s

This document describes approaches for implementing OpenStack Compute node and running VMs on top of K8s from the prospective of "Hypervisor Pod". It includes overview of currently selected approach, future steps and alternative solutions.

5.2.1 Potential solutions

This section consists of a list of potential solutions for implementing OpenStack VMs on topc K8s. This solutions the case when Neutron ML2 OVS used for OpenStack networking. Pros and Cons listed for each solution. Not all possible solutions (with all combinations of pod-container topologies) listed here, only part that makes sense or needed to show transition from bad to good options.

1. Everything in one pod

Pods and containers

Hypervisor / Compute / Networking Pod:

1. QEMU / KVM / Libvirt
2. OVS DB
3. OVS vswitchd
4. nova-compute
5. neutron-ovs-agent

Pros & cons

Pros:

1. One pod will represent the whole OpenStack compute node (very minor advantage)

Cons:

1. It's impossible to make upgrade of any service without killing virtual machines
2. All containers will have the same characteristics such as net=host, user, volumes and etc.

2. Libvirt and VMs baremetal, OpenStack part in one pod

Pods and containers

Baremetal (not in containers):

1. QEMU / KVM / Libvirt 4. OVS (DB and vswitchd)

Compute / Networking Pod:

1. nova-compute
2. neutron-ovs-agent

Pros & cons

Pros:

1. Restart of docker and docker containers will not affect running VMs as libvirt running on baremetal
2. Docker and docker containers downtime will not affect networking as OVS running on host, only new rules will not be passed to the host

Cons:

1. External orchestration required to for managing Libvirt on baremetal, especially for upgrades
2. It's impossible to update nova without neutron and vice versa.

3. Libvirt and VMs baremetal, pod per OpenStack process

Pods and containers

Baremetal (not in containers):

1. QEMU / KVM / Libvirt
2. OVS DB / vswitchd

Compute Pod:

1. nova-compute

Networking Pod:

1. neutron-ovs-agent

Pros & cons

Same as option number 3, but it's possible to upgrade nova and neutron separately.

4. Libvirt and VMs in one pod, pod per OpenStack service

Notes

It's a primary approach and it's currently implemented in Fuel CCP. Libvirt upgrade in such case could only be done by evacuating virtual machines from the host first, but, for example, nova-compute could be upgraded in place.

Pods and containers

Hypervisor pod:

1. QEMU / KVM / Libvirt

OVS DB pod:

1. OVS DB

OVS vswitchd pod:

1. OVS vswitchd

Compute Pod:

1. nova-compute

Networking pod:

1. neutron-ovs-agent

Pros & cons

Pros:

1. No external orchestration required for compute node provisioning
2. All OpenStack parts and dependencies are managed through K8s in such case, so it's possible to upgrade any service including libvirt, nova, neutron and ovs without external orchestration, just through the K8s API

Cons:

1. Docker or docker containers downtime will affect running VMs or networking

5. Libvirt in pod w/ host pid, pod per OpenStack service, VMs outside of containers

Notes

It's a "next step" approach based on Pros & Cond. It should be investigated in details and stability should be verified. If there will be no issues than it should become regerence approach of OPenStack VMs deployment on K8s. Potentially, another level of improvements needed to avoid affecting networking when docker or docker containers restarted.

Pods and containers

Hypervisor pod:

1. QEMU / KVM / Libvirt (using host pid)

OVS DB pod:

1. OVS DB

OVS vswitchd pod:

1. OVS vswitchd

Compute Pod:

1. nova-compute

Networking pod:

1. neutron-ovs-agent

Pros & cons

Same as option number 4, but improved to not affect virtual machines when docker or docker containers restart.

5.2.2 Conclusion

Option number 4 is currently selected as implementation design for Fuel CCP, while as end goal we'd like to achieve approach where restarting docker and docker containers will not affect running virtual machines. In future, we'll need to evaluate additional improvements to guarantee that K8s and docker downtime doesn't affect running VMs.

5.3 OpenStack Reference Architecture For 100, 300 and 500 Nodes

This document proposes a new Reference Architecture (RA) of OpenStack installation on top of Kubernetes that supports a number of 100, 300 and 500 compute nodes, using container technologies to improve scalability and high availability of OpenStack Control Plane services. Containerization of OpenStack components will also enable provisioning, patching and upgrading large numbers of nodes in parallel, with high reliability and minimal downtime.

5.3.1 Introduction/Executive Summary

This document contains recommendations for building specific clouds depending for different use cases. All recommendations are validated and tested on the described scale in both synthetic and real-world configurations.

The proposed Reference Architecture applies the following open source tools (among others):

- OpenStack Control Plane is a scalable, modular cloud controller with support for all aspects of virtualized infrastructure.
- Ceph is a distributed storage system that provides all the most popular types of storage to a virtualized infrastructure: object storage, virtual block storage and distributed file system.
- InfluxDB is a time-series database optimized for collecting metrics from multiple sources in nearly-real time and providing access to recorded metrics.
- Docker containers are used to isolate OpenStack services from the underlying operating system and control the state of every service more precisely.

Highlights

Highlights of this document include:

- Hardware and network configuration of the lab used to develop the Reference Architecture.
- OpenStack Control Plane overview - Details how the OpenStack Control Plane is organized, including placement of the services for scaling and high availability of the control plane.
- Data plane overview - Describes the approach to the data plane and technology stack used in the Reference Architecture.
- Granular update and upgrade overview - Describes how the proposed Reference Architecture supports updating and upgrading on all levels from individual services to the whole OpenStack application.

5.3.2 Overview

Hardware and network considerations

This section summarizes hardware considerations and network layouts for the proposed solution. It defines the basic requirements to server equipment hosting the cloud based on the CCP RA. Requirements to network infrastructure in terms of L2 and L3 topologies, services like DNS and NTP and external access provided in the network.

OpenStack Control Plane

The Control Plane consists of OpenStack component services, like Nova, Glance and Keystone, and supplementary services like MySQL database server and RabbitMQ server, all enveloped in Docker containers and managed by an orchestrator (e.g. Kubernetes).

OpenStack Data Plane

OpenStack data plane is constituted by backends to various drivers of different components of OpenStack. They all fall into 3 main categories:

- Hypervisor is data plane component backing OpenStack Compute (Nova), for example, libvirt or VMWare vSphere.
- Networking is multiple data plane components under management of OpenStack Networking, for example, OpenVSwitch.
- Storage has multiple components managed by OpenStack Storage and Images services. This category includes such systems as LVM, iSCSI, Ceph and others.

Granular Life Cycle Management, Updates and Upgrades

This document describes the strategy of updating the OpenStack cloud and its components to new version. The strategy of upgrade is based on containerization of all those components. Containers effectively split the state of the system into set of states of individual container. Every container's state is managed mostly independently.

5.3.3 Hardware Overview

Server Hardware Specifications

The following server hardware was used in a lab to install and test the proposed architecture solution. For Compute nodes, two configurations are used.

Configuration One

- Server model is Dell R630
- 2x12 Core CPUs E5-2680v3
- 256GB of RAM
- 2x800GB SSD Intel S3610
- 2x10GB Intel X710 dual-port NICs

Configuration Two

- Server model is Lenovo RD550-1U

- 2x12 Core CPUs E5-2680v3
- 256GB of RAM
- 2x800GB SSD Intel S3610
- 2x10GB Intel X710 dual-port NICs

For Storage nodes, the following configuration is used.

- Server model is Lenovo RD650
- 2x12 Core CPUs E5-2670v3
- 128GB RAM
- 2x480GB SSD Intel S3610
- 10x2TB HDD
- 2x10GB Intel X710 dual-port NICs

Resource Quantities

Compute/Controller Resources

The number of Compute/Controller nodes in the environment: 100 nodes

The number of CPU Cores available to hypervisors and control plane services: 2400 cores

The amount of RAM available to hypervisors and control plane services: 25,600 GB

Storage Resources

- The number of Storage nodes in the environment: 45 nodes
- The number of CPU Cores available to storage services: 1,080 cores
- The amount of RAM available to storage cache: 5,760 GB
- The total size of raw disk space available on storage nodes: 900 TB

Servers are installed in 9 racks connected by ToR switches to spine switches.

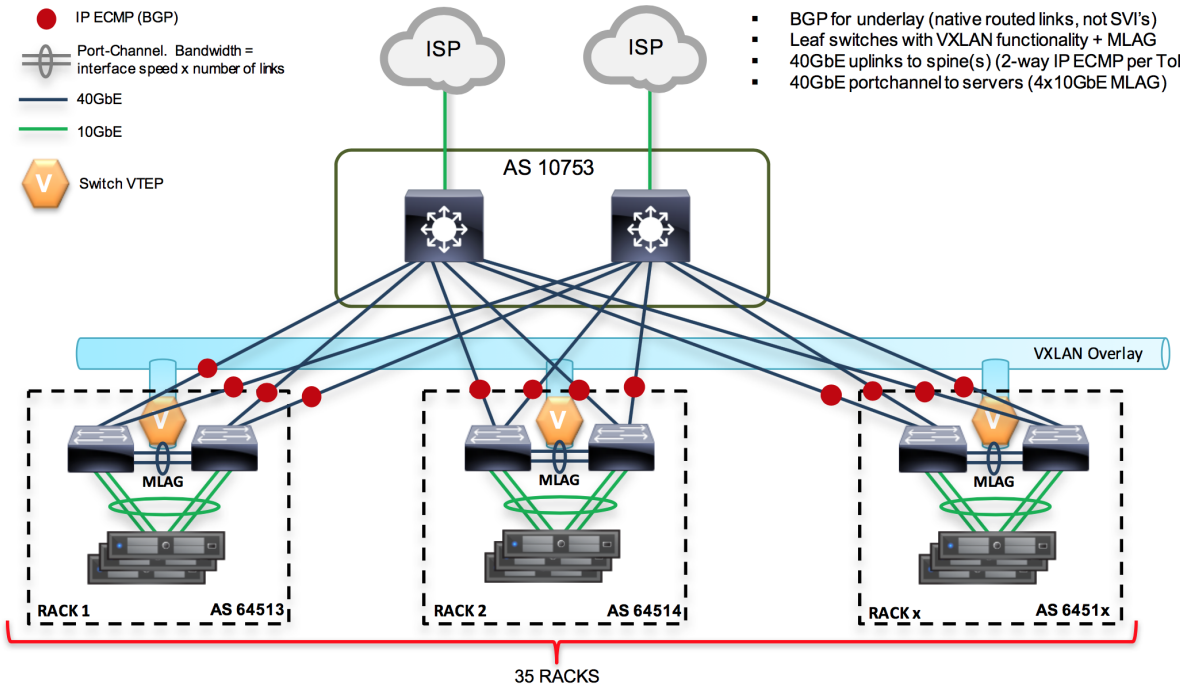
5.3.4 Network Schema

Underlay Network Topology

The environment employs leaf switches topology in the underlay network. BGP protocol used in the underlay network to ensure multipath links aggregation (IP ECMP) to leaf switches. ToR leaf switches are connected to spines with 40GbE uplinks.

The leaf switches use VXLANs to provide overlay network to servers, and MLAG aggregation to ensure availability and performance on the downstream links. Servers are connected to ToR switches with 40GbE port-channel links (4x10GbE with MLAG aggregation).

The following diagram depicts the network schema of the environment:



No specific QoS configuration was made in the underlay network. Assume that all services share the total bandwidth of network link without guarantees for individual processes or sockets.

The following models of switching hardware were used throughout testing effort in the schema described above:

- Spine switches: Arista 7508E (4x2900PS, 6xFabric-E modules, 1xSupervisorE module)
- ToR switches: Arista 7050X

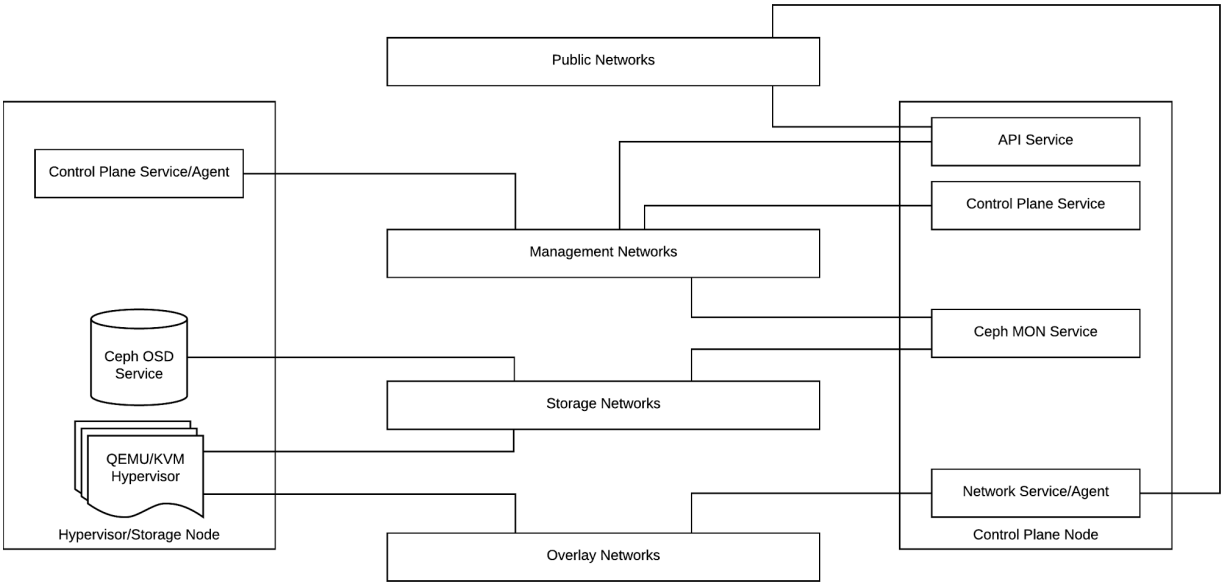
Network for OpenStack Platform

OpenStack platform uses underlay network to exchange data between its components, expose public API endpoints and transport the data of overlay or tenant networks. The following classes of networks are defined for OpenStack platform in proposed architecture.

- Management network. This is the network where sensitive data exchange happens. Sensitive data includes authentication and authorization credentials and tokens, database traffic, RPC messages and notifications. This network also provides access to administrative API endpoints exposed by components of the platform.
- **Public network. Network of this class provides access to API** endpoints exposed by various components of the platform. It also connects Floating IPs of virtual server instances to external network segments and Internet.
- Storage network. Specialized network to transport data of storage subsystem, i.e. Ceph cluster. It also connects to internal Ceph API endpoints.
- Overlay network. Networks of this class transport the payload of tenant networks, i.e. the private networks connecting virtual server instances. The current architecture employs VXLANs for L2 encapsulation.

There can be one or more networks of each class in a particular cloud, for example, Management network class can include separate segments for messaging, database traffic and Admin API endpoints, and so on.

In addition to role segmentation, networks in one class can be separated by scale. For example, Public network class might include multiple per-rack segments, all connected by an L3 router.



5.3.5 Control Plane

OpenStack Overview

OpenStack is a system that provides Infrastructure as a Service (IaaS). IaaS is essentially a set of APIs that allow for creation of elements of typical data center infrastructure, such as virtual servers, networks, virtual disks and so on. Every aspect of an infrastructure is controlled by a particular component of OpenStack:

- OpenStack Compute (Nova) controls virtual servers lifecycle, from creation through management to termination.
- OpenStack Networking (Neutron) provides connectivity between virtual servers and to the world outside.
- OpenStack Image (Glance) holds base disk images used to boot the instances of virtual servers with an operating system contained in the image.
- OpenStack Block Storage (Cinder) manages the virtual block storage and supplies it to virtual servers as block devices.
- OpenStack Identity (Keystone) provides authentication/authorization to other components and clients of all APIs.

Guidelines for OpenStack at Scale

Currently, OpenStack scalability is limited by several factors:

- Placement responsiveness desired
- MySQL scalability
- Messaging system scalability
- Scalability of the SDN
- Scalability of the SDS

Scheduler Scalability

In general, the scheduling strategy for OpenStack is a completely optimistic determination which means the state of all things is considered whenever a placement decision is made. As more factors are added to the consideration set (special hardware, more compute nodes, affinity, etc.) the time to place resources increases at a quadratic rate. There is work being done on splitting out the scheduling and placement parts of Nova and isolating them as a separate service, like Cinder or Neutron. For now, however, this is a work in progress and we should not rely on increased scheduler performance for another few releases.

OpenStack can seamlessly accommodate different server types based on CPU, memory and local disk without partitioning of the server pool. Several partitioning schemes exist that provide ability to specify pools of servers appropriate for a particular workload type. A commonly used scheme is server aggregates, which allows specific image sets to be scheduled to a set of servers based on server and image tags that the administrator has defined.

MySQL Scalability

As almost all OpenStack services use MySQL to track state, scalability of the database can become a problem. Mirantis OpenStack deploys with a MySQL + Galera cluster which allows for reads and writes to be scaled in a limited fashion.

However, careful tuning and care of the databases is a very important consideration. MySQL is recommended to run on dedicated nodes. The Control Plane DB should not be used by Tenant Applications or other solutions like Zabbix; those should provision another MySQL cluster on other physical nodes.

Messaging System Scalability

A majority of OpenStack services use AMQP implementations (RabbitMQ and Qpid) for message transport and RPC. As with MySQL, there are examples of installations in the wild which have successfully scaled this messaging infrastructure to tens of thousands of nodes.

Services Overview

The following paragraphs describe how the individual services and components should be placed and configured in the proposed Reference Architecture. All services are divided into two categories: stateless and stateful. Each category requires specific approach which is outlined below.

Stateless Services

Services of this type do not record their state in their environment. Those services can be killed and restarted without risk of losing data. Most of OpenStack component services are inherently stateless being either HTTP-based API servers or message queue processors.

General approach to stateless services is envelop the service into Docker container with minimal to no configuration embedded.

Stateful Services

Services that keep track of their state in some persistent storage and cannot be restarted in clean environment without losing that state are considered stateful. The main stateful component of the OpenStack platform is MySQL state database. Most of other services rely on the database to keep track of their own state.

Containers that run stateful services must have persistent storage attached to them. This storage must be made available in case if the container with stateful service should be move to another location. Such availability can be ensured on application level via replication of some sort, or on network level by redirecting connections from moving container to the same storage location.

5.3.6 OpenStack Identity (Keystone)

Keystone service provides identity to all other OpenStack services and external clients of OpenStack APIs. The main component of Identity service is an HTTP server that exposes an API of the service.

Keystone service is used by all other components of OpenStack for authorization of requests, including system requests between services and thus gets called a lot even if the platform is idle. This generates load on CPU resources of server where it runs. Thus, it is recommended to run Keystone service on dedicated node, althhough they could be co-located with other resource intensive components.

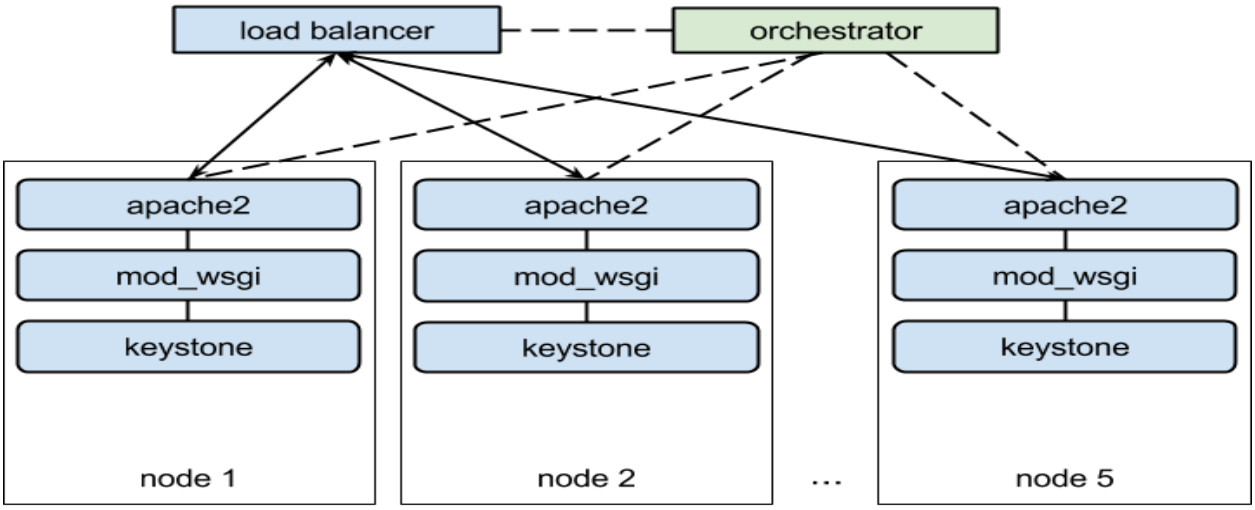
For availability and load distribution it is recommended to run at least 3 instances of Keystone at the scale of 100 and 300 nodes. For scale of 500 nodes, 5 instances of Keystone server are recommended.

Load balancer with a Virtual IP address should be placed in front of the Keystone services to ensure handling of failures and even distribution of requests.

Per these recommendations, the recommended number of dedicated physical nodes to run Keystone service is 3 or 5. However, instances of Keystone can share these nodes with any other services.

Apache2 Web Server

Apache web server wraps Keystone and Horizon services and exposes them as web services. However, since Horizon can generate significant load on resources, it is not recommended to co-locate it with Keystone, but it is possible at scale of 100 to 500 nodes. See below for recommendations on scaling Horizon.



5.3.7 OpenStack Compute (Nova)

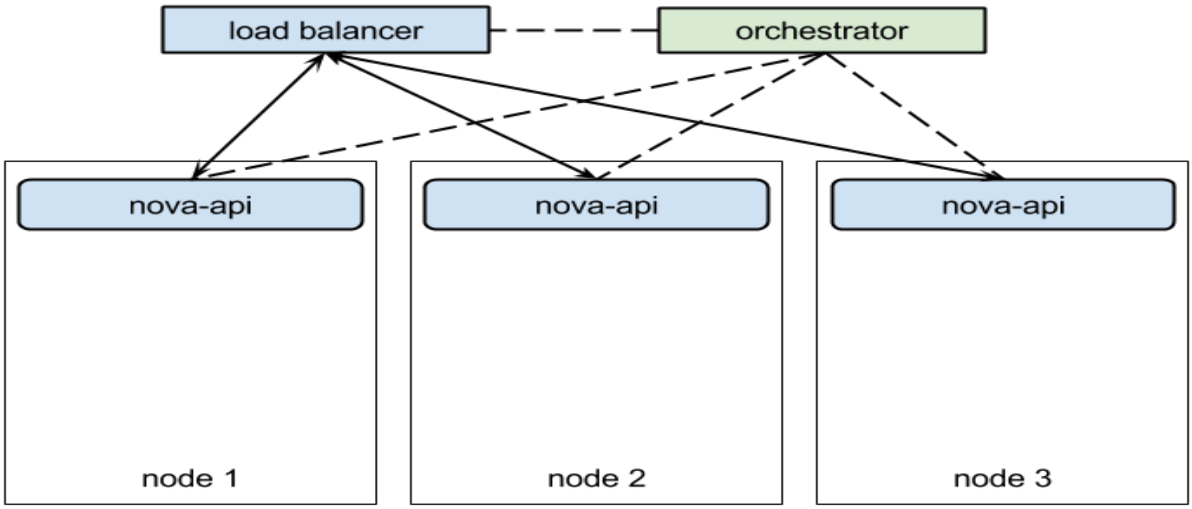
Nova has multiple services, not all of which are stateless. API server, scheduler and conductor in particular are stateless, while nova-compute service is not as it manages the state of data-plane components on a compute node.

Nova API

Web server that exposes Compute API of the OpenStack platform. This is a stateless service. Nova API server consumes significant resources at scale of hundreds of nodes.

Nova API server should run on a dedicated physical servers to ensure it does not share resources with other services with comparably big footprint. It can co-locate with more lightweight services.

For availability reasons, it is recommended to run at least 3 instances of API server at any time. Load balancer with Virtual IP address shall be placed in front of all API servers.



Nova Scheduler

Scheduling service of Nova is stateless and can be co-located with services that have larger footprint. Scheduler should be scaled by adding more service instances. Every scheduler instance must run with at least 3 worker threads.

Using multiple scheduler processes might lead to unexpected delays in provisioning due to scheduler retries if many simultaneous placement requests are being served.

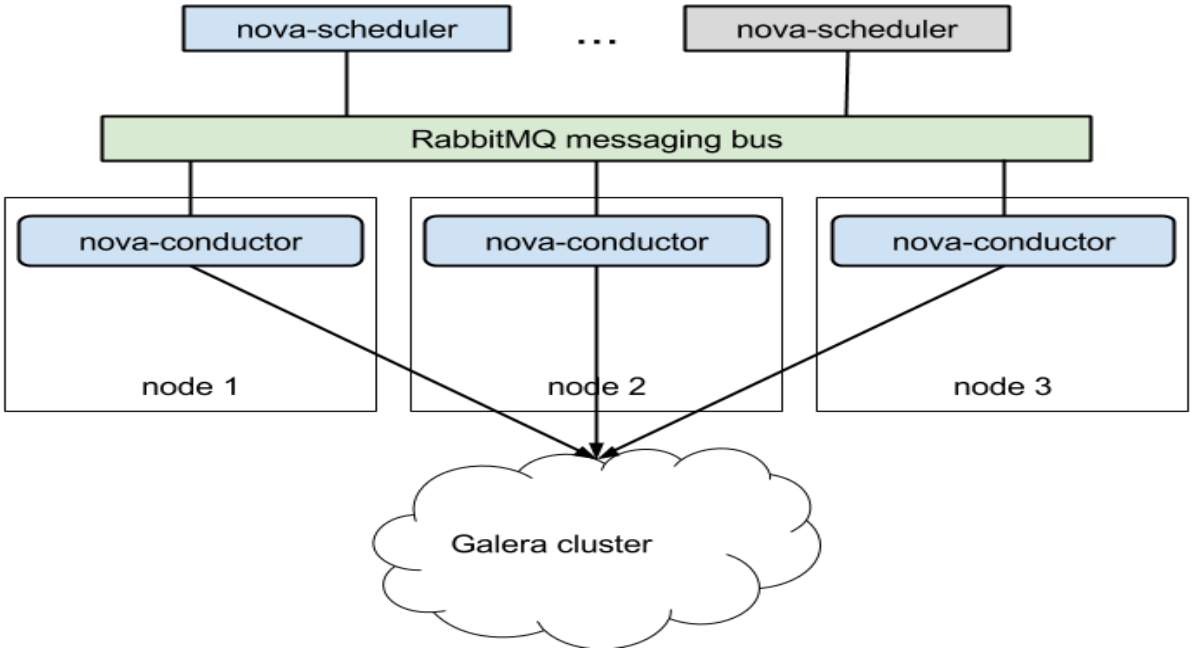
At the scale of 100 and 300 of nodes, 3 instances of Scheduler will suffice. For 500 nodes, 5 instances recommended.

Since they can be co-located with other services, it won't take any additional physical servers from the pool. Schedulers can be placed to nodes running other Nova control plane services (i.e. API and Conductor).

Nova Conductor

Centralized conductor service in Nova adds database interface layer for improved security and complex operations. It is a stateless service that is scaled horizontally by adding instances of it running on different physical servers.

To ensure that the Conductor service is highly available, run 3 instances of the service at any time. This will not require dedicated physical servers from the pool.



Nova Compute

Compute service is essentially an agent service of Nova. An instance of it runs on every hypervisor node in an OpenStack environment and controls the virtual machines and other connected resources on a node level.

Compute service in the proposed architecture is not containerized. It runs on the same node as a hypervisor and communicates to different local and remote APIs to get the virtual machines up and running. One instance of Compute service runs per hypervisor host.

Services Placement

Nova control plane services is spread across 3 nodes with the following services distributed evenly between those nodes:

- 3 instances of Conductor service
- 3 instances of API service
- 3 to 5 instances of Scheduler service

Load balancer is placed in front of the API services to ensure availability and load distribution for the Compute API of OpenStack platform.

Compute services run per-node on hypervisor hosts (compute nodes) in non-redundant fashion.

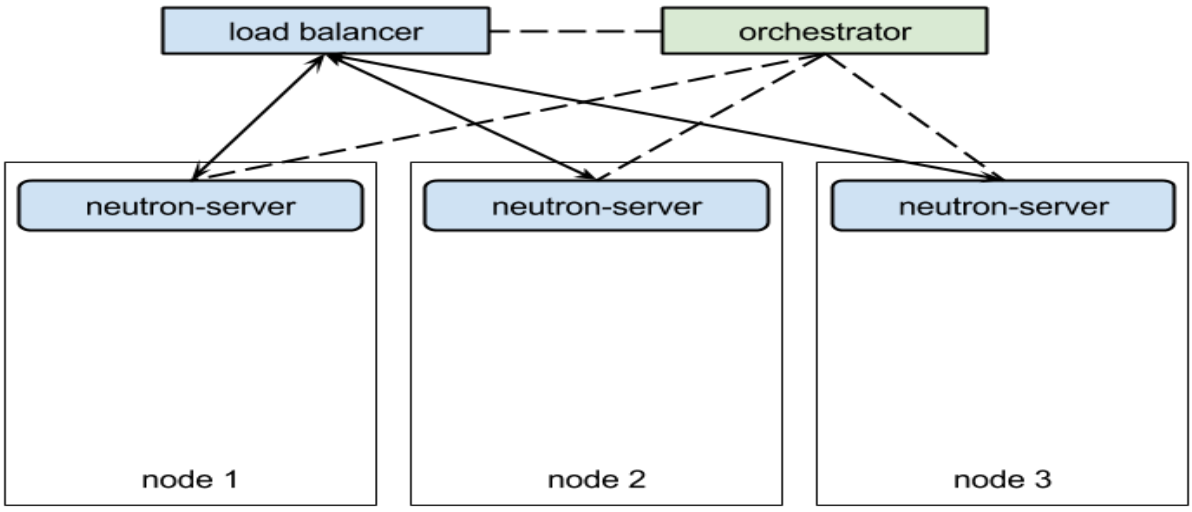
5.3.8 OpenStack Networking (Neutron)

This component includes API server that exposes HTTP-based API and a set of agents which actually manage data plane resources, such as IP addresses, firewall and virtual switches.

Neutron Server

An API server exposes Neutron API and passes all web service calls to the Neutron plugin for processing. This service generates moderate load on processors and memory of physical servers. In proposed architecture, it runs on the same nodes as other lightweight API servers. A minimal of 3 instances of the server is recommended for redundancy and load distribution.

Load balancer working in front of the API servers helps to ensure their availability and distribute the flow of requests.



Neutron DHCP agent

DHCP agent of Neutron is responsible for assigning IP addresses to VMs. The agent is horizontally scalable by adding new instances of it distributed between different nodes. DHCP agent can be co-located to any other services and can run on hypervisor hosts as well.

Neutron L3 agent

L3 agent controls routing in Public and Private networks by creating and managing namespaces, routers, floating IP addresses and network translations. The agent can be scaled by adding instances. High availability of the agent is ensured by running its instances in Corosync/Pacemaker cluster or using orchestrator-driven clustering and state tracking (e.g. Kubernetes events system with etcd).

The agent has low resources footprint and can be co-located with more resource-intensive services, for example, Neutron Server.

Neutron L2 agent

L2 agent manages data link layer configuration of the overlay network for Compute nodes. It also provides L2 functions to L3 agent. The agent is specific to Neutron plugin.

The L2 agent used with OpenVSwitch plugin generates high CPU load when creates and monitors the OVS configurations. It has to run on any host that runs nova-compute, neutron-l3-agent or neutron-dhcp-agent.

Neutron metadata agent

The metadata agent provides network metadata to virtual servers. For availability and redundancy, it is recommended to run at least 3 instances of the agent. They can share a node with other Neutron or control plane services.

Services Placement

Neutron control plane services run in batches 3 for redundancy of Neutron Server. All micro services of Neutron could be co-located with the Neutron Servers or with other components' services.

Load balancer should be placed in front of the Neutron Server to ensure availability and load distribution for the Network API of the OpenStack platform.

L3 and DHCP agents are co-located with instances of Neutron Server on the same physical nodes. L2 agent works on every Compute node and on every node that runs L3 and/or DHCP agent.

5.3.9 OpenStack Images (Glance)

Images service consists of API and indexing service. Both of them are stateless and can be containerized.

Glance API

This service exposes Images API of OpenStack platform. It is used to upload and download images and snapshot.

Glance API server has low resource consumption and can be co-located with other services. It does not require dedicated physical servers.

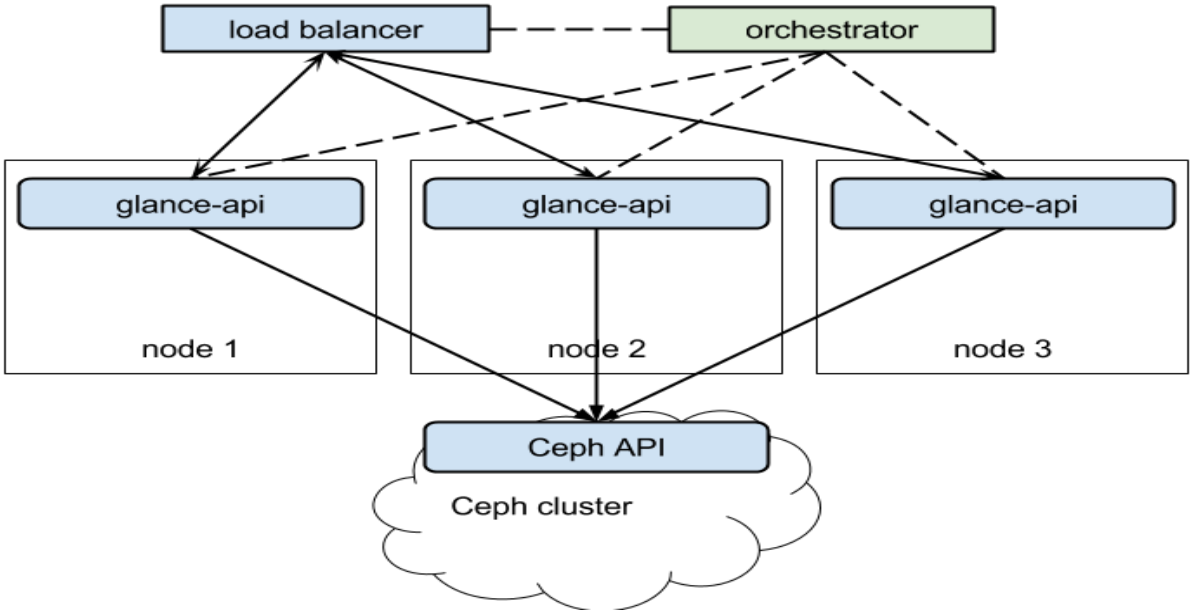
Glance API server scales by adding new instances. Load balancer is required to provide a single virtual address to access the API. The load can be evenly distributed between instances of Glance API.

Important parameter of the Image Service architecture is the storage backend. The following types of storage are proposed for Glance in this Reference Architecture:

- Local filesystem
- Ceph cluster

With local storage, the consistency of the store for all instances of Glance API must be ensured by an external component (e.g. replication daemon).

Ceph backend natively provides availability and replication of stored data. Multiple instances of Glance API service work with the same shared store in Ceph cluster.



Glance Registry

Registry serves images metadata part of the Images API. It stores and manages a catalog of images. It has low footprint and can share a physical node with other resource intensive services.

Services Placement

Micro services of Glance do not require dedicated physical servers. They can be co-located with other services.

For the purposes of high availability and redundancy, at least 3 instances of Glance API service should run at any time. Load balancer must be placed in front of those instances to provide single API endpoint and distribute the load.

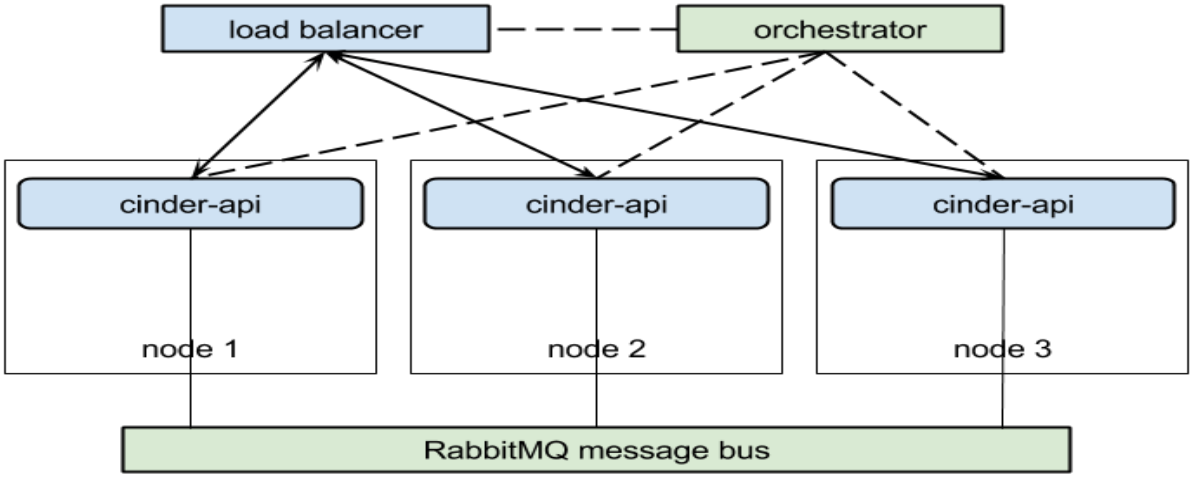
5.3.10 OpenStack Block Storage (Cinder)

Storage service manages and exposes a virtual block storage. The server that handles the management of low-level storage requires access to the disk subsystem.

Cinder API

This service exposes Volume API of the OpenStack platform. Volume API is not very often used by other components of OpenStack and doesn't consume too much resources. It could run on the same physical node with other resource non intensive services.

For availability and redundancy purposes, it is proposed to run at least 3 instances of Cinder API. Load balancer should be placed in front of these instances to provide distribution of load and failover capabilities.



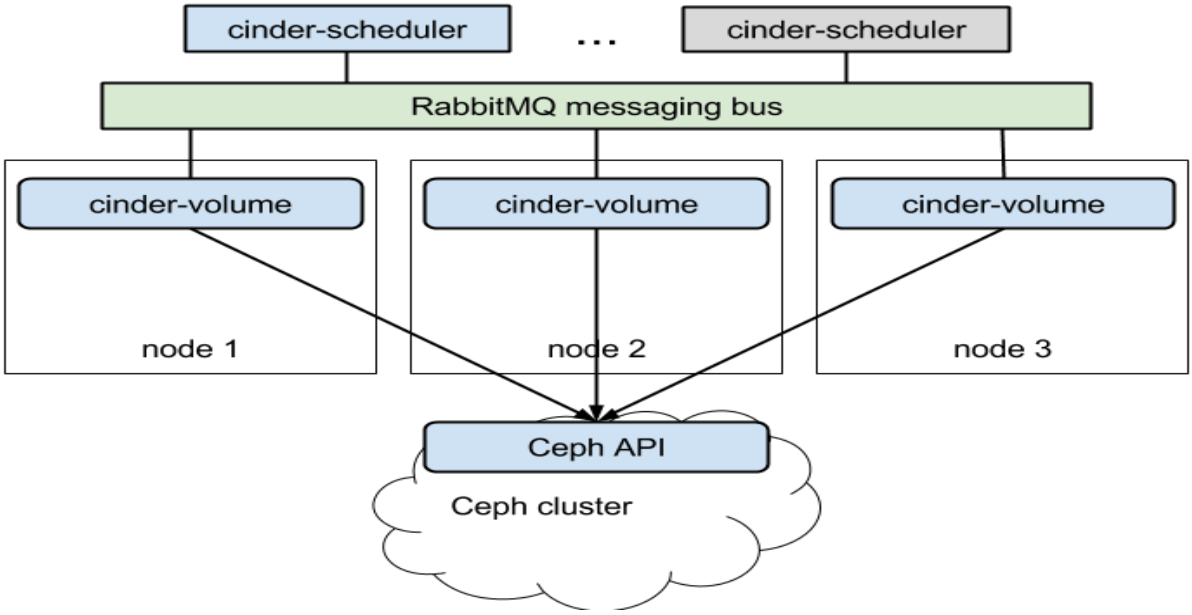
Cinder Scheduler

Scheduler selects an instance of the Volume micro service to call when a client requests creation of a virtual block volume. Scheduler is not resource intensive and can be co-located with other services. It scales horizontally by adding new instances of the service. For redundancy purposes, the instances of scheduler service should be distributed between different physical servers.

Cinder Volume

Volume service manages block storage via appropriate API which depends on the technology in use. With Ceph as a provider of virtual block storage, single instance of Cinder Volume service is required to manage the Ceph cluster via its API. If virtual block storage is provided by using LVM with local disks of Compute nodes, the Volume service must be running on every Compute node in the OpenStack environment.

In Ceph case, it is recommended to run at least 3 instances of the Volume service for the availability and redundancy of the service. Each virtual volume is managed by one instance Volume service at a time. However, if that instance is lost, another one takes over on its volumes.



Services Placement

Cinder services do not require dedicated physical nodes. They run on the same physical servers with other components of control plane of the OpenStack platform.

The instances of Cinder API service are placed behind a load balancer to ensure the distribution of load and availability of the service.

5.3.11 OpenStack Dashboard (Horizon)

Horizon dashboard provides user interface to the cloud's provisioning APIs. This is a web application running on top of Apache2 web server. For availability purposes, multiple instances of the server are started. Load balancer is placed in front of the instances to provide load distribution and failover. Horizon does not require dedicated physical node. It can be co-located with other services of other components of the OpenStack platform.

For security reasons, it is not recommended to use the same instances Apache2 server to wrap both Horizon and Keystone API services.

5.3.12 RabbitMQ

The messaging server allows all distributed components of an OpenStack service to communicate to each other. Services use internal RPC for communication. All OpenStack services also send broadcast notifications via messaging queue bus.

Clustering

The prerequisite for High Availability of queue server is the configured and working RabbitMQ cluster. All data/state required for the operation of a RabbitMQ cluster is replicated across all nodes. An exception to this are message queues, which by default reside on one node, though they are visible and reachable from all nodes.

Cluster assembly requires installing and using a clustering plugin on all servers. Proposed solution for RabbitMQ clustering is the rabbitmq-autocluster*plugin*.

The RabbitMQ cluster also needs proper fencing mechanism to exclude split brain conditions and preserve a quorum. Proposed solution for this problem is using 'pause_minority' partition mode with the rabbit-autocluster plugin.

Replication

Replication mechanism for RabbitMQ queues is known as 'mirroring'. By default, queues within a RabbitMQ cluster are located on a single node (the node on which they were first declared). This is unlike exchanges and bindings, which can always be considered to be on all nodes. Queues can optionally be made mirrored across multiple nodes. Each mirrored queue consists of one master and one or more slaves, with the oldest slave being promoted to the new master if the old master disappears for any reason.

Messages published to the queue are replicated to all members of the cluster. Consumers are connected to the master regardless of which node they connect to, with slave nodes dropping messages that have been acknowledged at the master.

Queue mirroring therefore aims to enhance availability, but not distribution of load across nodes (all participating nodes each do all the work). It is important to note that using mirroring in RabbitMQ actually reduces the availability of queues by dropping performance by about 2 times in tests, and eventually leads to failures of RabbitMQ because of extremely high rate of context switches at node's CPUs.

There are two main types of messages in OpenStack:

- **Remote Procedure Call (RPC) messages carry commands and/or requests** between microservices within a single component of OpenStack platform (e.g. nova-conductor to nova-compute).
- **Notification messages are issued by a microservice upon specific** events and are consumed by other components (e.g. Nova notifications about creating VMs are consumed by Ceilometer).

In proposed OpenStack architecture, only notification queues are mirrored. All other queues are not, and if the instance of RabbitMQ server dies after a message sent, but before it is read, that message is gone forever. This is a trade-off for significant (at least 2 times) performance and stability boost in potential bottleneck service. Drawbacks of this mode of operation are:

- **Long-running tasks might stuck in transition states due to loss of** messages. For example, Heat stacks might never leave spawning state. Most of the time, such conditions could be fixed by the user via API.

Data Persistence

OpenStack's RPC mechanism does not impose requirements for durable queues or messages. Thus, no durability required for RabbitMQ queues, and there are no 'disk' nodes in cluster. Restarting a RabbitMQ node then will cause all data of that node to be lost. Since OpenStack does not rely on RPC as a guaranteed transport, it doesn't break the control plane. Clients shall detect failure of a server they are talking to and connect to another server automatically.

RabbitMQ service considered stateless in terms defined in this document due to the reasons mentioned above.

Networking Considerations

RabbitMQ nodes address each other using domain names, either short or fully-qualified (FQDNs). Therefore hostnames of all cluster members must be resolvable from all cluster nodes, as well as machines on which command line tools such as rabbitmqctl might be used.

RabbitMQ clustering has several modes of dealing with network partitions, primarily consistency oriented. Clustering is meant to be used across LAN. It is not recommended to run clusters that span WAN. The Shovel or Federation

plugins are better solutions for connecting brokers across a WAN. Note that Shovel and Federation are not equivalent to clustering.

Services Placement

RabbitMQ servers are to be installed on the dedicated nodes. Co-locating RabbitMQ with other Control Plane services has negative impact on its performance and stability due to high resource consumption under the load. Other services that have different resource usage patterns can prevent RabbitMQ from allocating sufficient resources and thus make messaging unstable.

Based on that, RabbitMQ will require 3 dedicated nodes out of the pool of Compute/Controller servers. These nodes can also host lightweight services like Cinder, Keystone or Glance.

Alternatives

RabbitMQ is a server of choice for OpenStack messaging. Other alternatives include:

- **0MQ (ZeroMQ), a lightweight messaging library that integrates into** all components and provides serverless distributed message exchange.
- **Kafka, a distributed commit log type messaging system, supported by** oslo.messaging library in experimental mode.

ZeroMQ

This library provides direct exchange of messages between microservices. Its architecture may include simple brokers or proxies that just relay messages to endpoints, thus reducing the number of network connections.

ZeroMQ library support was present in OpenStack since early releases. However, the implementation assumed direct connections between services and thus a full mesh network between all nodes. This architecture doesn't scale well. More recent implementations introduce simple proxy services on every host that aggregate messages and relay them to a central proxy, which does host-based routing.

Benchmarks show that both direct and proxy-based ZeroMQ implementations are more efficient than RabbitMQ in terms of throughput and latency. However, in the direct implementation, quick exhaustion of network connections limit occurs at scale.

The major down side of the ZeroMQ-based solution is that the queues don't have any persistence. This is acceptable for RPC messaging, but Notifications may require durable queues. Thus, if RPC is using ZeroMQ, the Telemetry will require a separate messaging transport (RabbitMQ or Kafka).

Kafka

Distributed commit log based service Kafka is supported in OpenStack's oslo.messaging library as an experimental. This makes it unfeasible to include in the Reference Architecture..

5.3.13 MySQL/Galera

State database of OpenStack contains all data that describe the state of the cloud. All components of OpenStack use the database to read and store changes in their state and state of the data plane components.

Clustering

The proposed clustering solution is based on the native orchestrator-specific state management with etcd providing distributed monitoring and data exchange for the cluster. Cluster operations will be triggered by orchestrator events and handled by custom scripts.

Failover and fencing of failed instances of MySQL is provided by scripts triggered by the orchestrator upon changes in state and availability of the members of Galera cluster. State and configuration information is provided by etcd cluster.

Data Persistence

Galera implements replication mechanism to ensure that any data written to one of the MySQL servers is synchronously duplicated to other members of the cluster. When a new instance joins the cluster, one of the two replication methods is used to synchronize it: IST or SST. If the initial data set exists on the new node, incremental method (IST) is used. Otherwise, full replication will be performed (SST).

Since all nodes in the cluster have synchronous copies of the data set at any time, there is no need to use shared storage. All DB servers work with the local disk storage.

Replication

Incremental synchronous replication is used to keep MySQL databases of members in Galera cluster in sync. If a new member is added to the cluster, full replication (SST) will be performed.

Full SST replication can take indefinite time if the data set is big enough. To mitigate this risk, the proposed architecture includes a number of hot stand-by MySQL servers in addition to one Active server. The access to said servers is provided by an instance of a load balancer (see details in Networking Considerations section).

Proposed architecture allows to quickly replace failing instances of MySQL server without need to run full replication. It is still necessary to restore the pool of hot standby instances whenever the failover event occurs.

Networking Considerations

Load balancer is a key element of networking configuration of the Galera cluster. Load balancer must be coordinated with the cluster, in terms that it redirect write requests to appropriate instance of MySQL server. It also ensures failover to hot standby instances and fencing of failed active nodes.

Services Placement

MySQL servers forming the Galera cluster could run on the same physical servers as instances of RabbitMQ broker.

5.3.14 Ceph Distributed Storage

Summary

Ceph is a distributed storage system with built-in replication capability. Architecture of Ceph is designed for resiliency and durability.

Ceph provides few different APIs for different types of supported storage:

- Distributed file system
- Object storage

- Virtual block device

OpenStack platform generally uses Object API and Block Device API of Ceph for Image and Virtual Block storage correspondingly.

Main components of Ceph are as follows.

- A Ceph Monitor maintains a master copy of the cluster map. A cluster of Ceph monitors ensures high availability should a monitor daemon fail. Storage cluster clients retrieve a copy of the cluster map from the Ceph Monitor.
- A Ceph OSD Daemon checks its own state and the state of other OSDs and reports back to monitors.
- RADOS gateway exposes the Object API of the platform. This API is generally compatible with OpenStack Object Storage API, which allows to use it as a Glance back-end without additional modifications.

Ceph Monitor

For reliability purposes, Ceph Monitors should be placed to different physical nodes. Those nodes might be the Storage nodes themselves, albeit that is not generally recommended. Proper resilience of the cluster can be ensured by using 3 instances of Ceph Monitor, each running on different hosts.

Ceph OSD

Every storage device requires an instance of Ceph OSD daemon to run on the node. These daemons might be resource intensive under certain conditions. Since one node usually have multiple devices attached to it, there are usually more than one OSD process running on the node. Thus, it is recommended that no other services are placed to the OSD nodes.

RADOS Gateway

This service exposes an Object API via HTTP. It have low resource footprint and can be co-located with other services, for example, with a Ceph Monitor. Multiple radosgw daemons should be used to provide redundancy of the service. Load balancer should be placed in front of instances of radosgw for load distribution and failover.

Services Placement

Ceph scales very well by adding new OSD nodes when capacity increase is required. So the size of the Ceph cluster may vary for different clouds of the same scale. In this document, a cluster with 45 OSD nodes is described.

Instances of Monitor service require 1 dedicated node per instance, to the total of 3 nodes. RADOS gateway servers run on the same nodes as Monitors.

5.3.15 Control Plane Operations Monitoring

Summary

Monitoring the OpenStack Control Plane infrastructure is essential for operating the platform. The main goal of it is to ensure that an operator is alerted about failures and degradations in service level of the environment. Metering plays less important role in this type of monitoring.

Currently proposed solution for infrastructure monitoring is produced by Stacklight project.

Stacklight is a distributed framework for collecting, processing and analyzing metrics and logs from OpenStack components. It includes the following services:

- Stacklight Collector + Aggregator
- Elasticsearch + Kibana
- InfluxDB + Grafana
- Nagios

Stacklight Collector

Smart agent that runs on every node in the OpenStack environment. It collects logs, processes metrics and notifications, generates and sends alerts when needed. Specialized instance of Collector called Aggregator aggregates metrics on the cluster level and performs special correlation functions.

Elasticsearch + Kibana

These components are responsible for analyzing large amounts of textual data, particularly logs records and files coming from OpenStack platform nodes. Kibana provides graphical interface that allows to configure and view correlations in messages from multiple sources.

A typical setup at least requires a quad-core server with 8 GB of RAM and fast disks (ideally, SSDs). The actual disk space you need to run the subsystem on depends on several factors including the size of your OpenStack environment, the retention period, the logging level, and workload. The more of the above, the more disk space you need to run the Elaticsearch-Kibana solution. It is highly recommended to use dedicated disk(s) for your data storage.

InfluxDB + Grafana

InfluxDB is a time series database that provides high throughput and real-time queries for reduced support of standard SQL capabilities like efficiently updating records. Grafana exposes graphic interface to time series data, displays graphs of certain metrics in time and so on.

The hardware configuration (RAM, CPU, disk(s)) required by this subsystem depends on the size of your cloud environment and other factors like the retention policy. An average setup would require a quad-core server with 8 GB of RAM and access to a 500-1000 IOPS disk. For sizeable production deployments it is strongly recommended to use a disk capable of 1000+ IOPS like an SSD. See the *InfluxDB Hardware Sizing Guide* for additional sizing information.

Nagios

In Stacklight architecture, Nagios does not perform actual checks on the hosts. Instead it provides transport and user interface for the alerting subsystem. It receives alerts from Collectors and generates notifications to the end users or cloud operators.

A typical setup would at least require a quad-core server with 8 GB of RAM and fast disks (ideally, SSDs).

Services Placement

InfluxDB and Elasticsearch subsystems of the Stacklight solution should run on dedicated servers. Additionally, clustered InfluxDB needs at least three nodes to form a quorum. Elasticsearch scales horizontally by adding instances, each running on a separate node. At least three instances of Elasticsearch are recommended.

Stacklight Aggregator service could share a physical node with other low-profile services.

Nagios server can be co-located with Stacklight Aggregator due to its low resource footprint.

The total of 5 additional physical nodes are required to install Control Plane Operations Monitoring framework based on Stacklight.

5.3.16 Services Placement Summary

The following table summarizes the placement requirements of the services described above.

Service	Number Of Instances	Number Of Dedicated Nodes	Can Share A Node	Requires a Load Balancer
keystone-all	3	•	yes	yes
nova-api	3	•	yes	yes
nova-scheduler	3	•	yes	no
nova-conductor	3	•	yes	no
nova-compute*	•	•	yes	no
neutron-server	3	•	yes	no
neutron-dhcp-agent	3	•	yes	no
neutron-l2-agent*	•	•	yes	no
neutron-l3-agent	3	•	yes	no
neutron-metadata-agent	3	•	yes	no
glance-api	3	•	yes	yes
glance-registry	3	•	yes	no
cinder-api	3	•	yes	yes
cinder-scheduler	3	•	yes	no
cinder-volume	•	•	yes	no
horizon/apache2	3	•	yes	yes
rabbitmq-server	3	3	yes	no
mysqld-server	3	3	yes	yes**
ceph-mon	3	•	yes	no
ceph-osd***	•	•	no	no
radosgw	3	•	yes	yes
lma-aggregator	1	•	yes	no
Influxdb + Grafana	3	•	yes	no
Elasticsearch + Kibana	3	•	yes	no
Nagios	1	•	yes	no
TOTAL		6		

Service	Number Of Instances	Number Of Dedicated Nodes	Can Share A Node	Requires a Load Balancer
keystone-all	3	3	yes	yes
nova-api	3	•	yes	yes
nova-scheduler	3	•	yes	no
nova-conductor	3	•	yes	no
nova-compute*	•	•	yes	no
neutron-server	3	•	yes	no
neutron-dhcp-agent	3	•	yes	no
neutron-l2-agent*	•	•	yes	no
neutron-l3-agent	3	•	yes	no
neutron-metadata-agent	3	•	yes	no
glance-api	3	•	yes	yes
glance-registry	3	•	yes	no
cinder-api	3	•	yes	yes
cinder-scheduler	3	•	yes	no
cinder-volume	•	•	yes	no
horizon/apache2	3	•	yes	yes
rabbitmq-server	3	3	yes	no
mysqld-server	3	3	yes	yes**
ceph-mon	3	•	yes	no
ceph-osd***	•	•	no	no
radosgw	3	•	yes	yes
lma-aggregator	1	•	yes	no
Influxdb + Grafana	3	•	yes	no
Elasticsearch + Kibana	3	•	yes	no
Nagios	1	•	yes	no
TOTAL	•	9	•	•

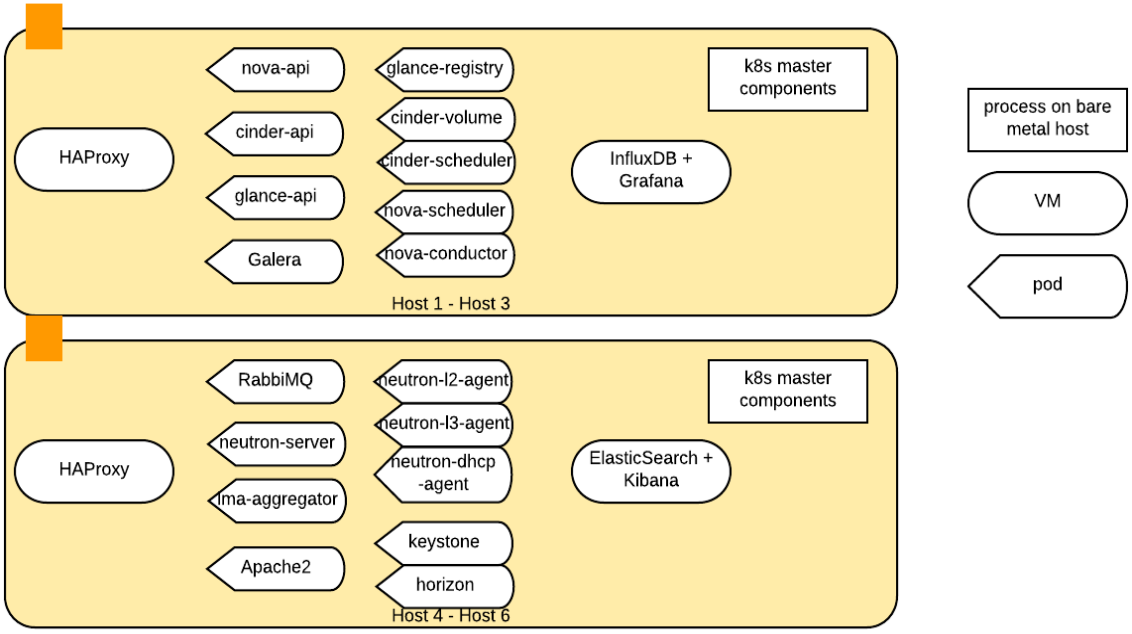
Service	Number Of Instances	Number Of Dedicated Nodes	Can Share A Node	Requires a Load Balancer
keystone-all	5	5	yes	yes
nova-api	3	•	yes	yes
nova-scheduler	5	•	yes	no
nova-conductor	3	•	yes	no
nova-compute*	•	•	yes	no
neutron-server	3	•	yes	no
neutron-dhcp-agent	3	•	yes	no
neutron-l2-agent*	•	•	yes	no
neutron-l3-agent	3	•	yes	no
neutron-metadata-agent	3	•	yes	no
glance-api	3	•	yes	yes
glance-registry	3	•	yes	no
cinder-api	3	•	yes	yes
cinder-scheduler	3	•	yes	no
cinder-volume	•	•	yes	no
horizon/apache2	3	•	yes	yes
rabbitmq-server	3	3	no	no
mysqld-server	3	3	yes	yes**
ceph-mon	3	•	yes	no
ceph-osd***	•	•	no	no
radosgw	3	•	yes	yes
lma-aggregator	1	•	yes	no
Influxdb + Grafana	3	•	yes	no
Elasticsearch + Kibana	3	•	yes	no
Nagios	1	1	yes	no
TOTAL	•	11	•	•

* this service runs on a Compute/hypervisor host

** this service might require specialized load balancer (proxysql)

*** this service runs on a Storage/OSD node

Total number of nodes for control plane is 6 which includes separate servers for monitoring infrastructure and RabbitMQ/Galera services. The following schema describes the layout of services in the test cluster.



5.3.17 Data Plane

Compute Virtualization

QEMU implementation of Linux KVM is used as a virtualization hypervisor. The KVM runs under control of libvirtd daemon, configured and managed by Nova Compute microservice (nova-compute).

Network

Depending on the picked Neutron plugin, data plane network might be realized by different technology stacks. Reference architecture employs OVS+VXLAN plugin, network data plane at compute hosts runs on OpenVSwitch virtual switching software.

Storage

Local disks and Ceph cluster RBD volumes are both used to provide block storage capability to different services in the platform.

Ephemeral

Disk files of the VM instances running in the cloud are stored in Ceph distributed storage via RBD. This enables live migration of VMs and high availability of data in certain cases.

Virtual Block Devices

Virtual block storage devices are provided by Ceph via RBD.

Images

Images used by Nova to spin up new VM instances are stored in Ceph via Glance service's back end driver.

Snapshot

Snapshots are stored in Ceph.

Database

Database files are kept on the local storage devices of nodes that run database servers. The replication and availability of the data are ensured by WSREP mechanism of Galera cluster.

Monitoring

The monitoring metrics and time-series data are written to InfluxDB database. The database keeps its files on a local storage, similar to the approach taken for state database (see above).

Logs

Logs are written to local disks of nodes, plus optionally to remote logs collector service which is the part of Telemetry component of the platform.

5.3.18 References

This section contains references to external documents used in preparation of this document.

1. Testing on scale of 1000 compute nodes
2. Performance testing of OpenStack
3. RabbitMQ Clustering
4. RabbitMQ High Availability
5. ZeroMQ Status Update

5.4 OpenStack Reference Architecture For 1000 Nodes

This document proposes a new Reference Architecture (RA) of OpenStack installation on top of Kubernetes that supports very large numbers of compute nodes, using container technologies to improve scalability and high availability of OpenStack Control Plane services. Containerization of OpenStack components will also enable provisioning, patching and upgrading large numbers of nodes in parallel, with high reliability and minimal downtime.

5.4.1 Introduction/Executive Summary

This document contains recommendations for building specific clouds depending for different use cases. All recommendations are validated and tested on the described scale in both synthetic and real-world configurations.

The proposed Reference Architecture applies the following open source tools (among others):

- OpenStack Control Plane is a scalable, modular cloud controller with support for all aspects of virtualized infrastructure.
- Ceph is a distributed storage system that provides all the most popular types of storage to a virtualized infrastructure: object storage, virtual block storage and distributed file system.
- InfluxDB is a time-series database optimized for collecting metrics from multiple sources in nearly-real time and providing access to recorded metrics.
- Docker containers are used to isolate OpenStack services from the underlying operating system and control the state of every service more precisely.

Highlights

Highlights of this document include:

- Hardware and network configuration of the lab used to develop the Reference Architecture.
- OpenStack Control Plane overview - Details how the OpenStack Control Plane is organized, including placement of the services for scaling and high availability of the control plane.
- Data plane overview - Describes the approach to the data plane and technology stack used in the Reference Architecture.
- Granular update and upgrade overview - Describes how the proposed Reference Architecture supports updating and upgrading on all levels from individual services to the whole OpenStack application.

5.4.2 Overview

Hardware and network considerations

This section summarizes hardware considerations and network layouts for the proposed solution. It defines the basic requirements to server equipment hosting the cloud based on the CCP RA. Requirements to network infrastructure in terms of L2 and L3 topologies, services like DNS and NTP and external access provided in the network.

OpenStack Control Plane

The Control Plane consists of OpenStack component services, like Nova, Glance and Keystone, and supplementary services like MySQL database server and RabbitMQ server, all enveloped in Docker containers and managed by an orchestrator (e.g. Kubernetes).

OpenStack Data Plane

OpenStack data plane is constituted by backends to various drivers of different components of OpenStack. They all fall into 3 main categories:

- Hypervisor is data plane component backing OpenStack Compute (Nova), for example, libvirt or VMWare vSphere.
- Networking is multiple data plane components under management of OpenStack Networking, for example, OpenVSwitch.
- Storage has multiple components managed by OpenStack Storage and Images services. This category includes such systems as LVM, iSCSI, Ceph and others.

Granular Life Cycle Management, Updates and Upgrades

This document describes the strategy of updating the OpenStack cloud and its components to new version. The strategy of upgrade is based on containerization of all those components. Containers effectively split the state of the system into set of states of individual container. Every container's state is managed mostly independently.

5.4.3 Hardware Overview

Server Hardware Specifications

The following server hardware was used in a lab to install and test the proposed architecture solution. For Compute nodes, two configurations are used.

Configuration One

- Server model is Dell R630
- 2x12 Core CPUs E5-2680v3
- 256GB of RAM
- 2x800GB SSD Intel S3610
- 2x10GB Intel X710 dual-port NICs

Configuration Two

- Server model is Lenovo RD550-1U
- 2x12 Core CPUs E5-2680v3
- 256GB of RAM
- 2x800GB SSD Intel S3610
- 2x10GB Intel X710 dual-port NICs

For Storage nodes, the following configuration is used.

- Server model is Lenovo RD650
- 2x12 Core CPUs E5-2670v3
- 128GB RAM
- 2x480GB SSD Intel S3610
- 10x2TB HDD

- 2x10GB Intel X710 dual-port NICs

Resource Quantities

Compute/Controller Resources

The number of Compute/Controller nodes in the environment: 516 nodes

The number of CPU Cores available to hypervisors and control plane services: 12,384 cores

The amount of RAM available to hypervisors and control plane services: 132,096 GB

Storage Resources

- The number of Storage nodes in the environment: 45 nodes
- The number of CPU Cores available to storage services: 1,080 cores
- The amount of RAM available to storage cache: 5,760 GB
- The total size of raw disk space available on storage nodes: 900 TB

Servers are installed in 35 racks connected by ToR switches to spine switches.

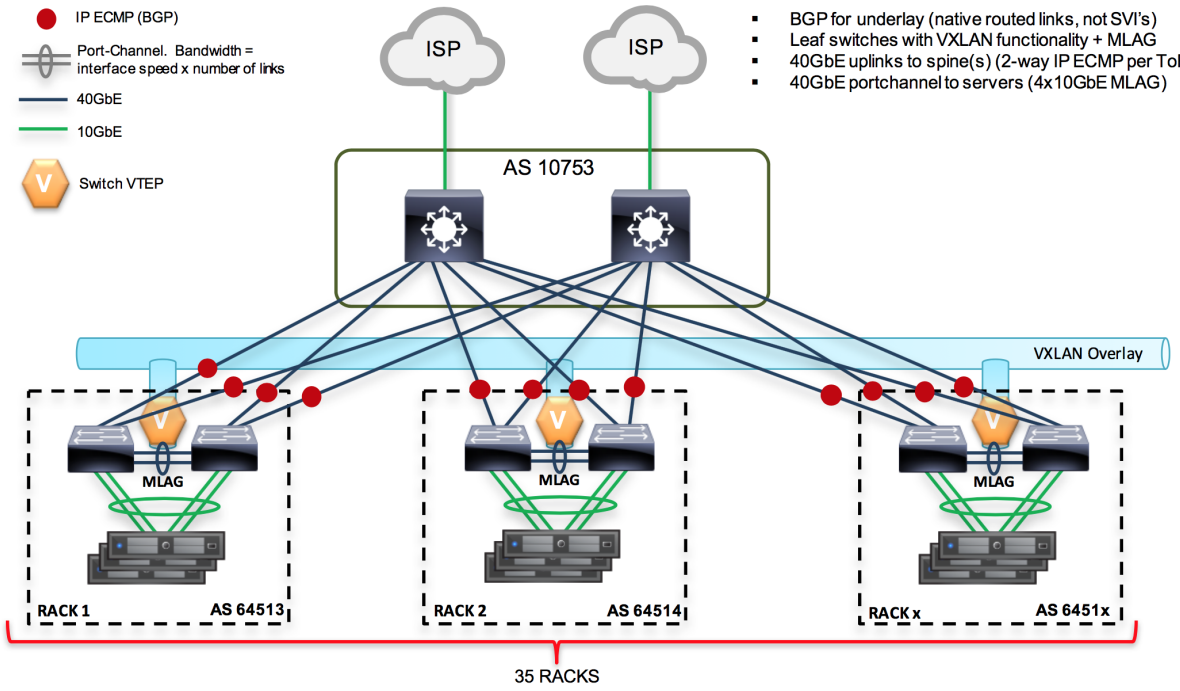
5.4.4 Network Schema

Underlay Network Topology

The environment employs leaf switches topology in the underlay network. BGP protocol used in the underlay network to ensure multipath links aggregation (IP ECMP) to leaf switches. ToR leaf switches are connected to spines with 40GbE uplinks.

The leaf switches use VXLANs to provide overlay network to servers, and MLAG aggregation to ensure availability and performance on the downstream links. Servers are connected to ToR switches with 40GbE port-channel links (4x10GbE with MLAG aggregation).

The following diagram depicts the network schema of the environment:



No specific QoS configuration was made in the underlay network. Assume that all services share the total bandwidth of network link without guarantees for individual processes or sockets.

The following models of switching hardware were used throughout testing effort in the schema described above:

- Spine switches: Arista 7508E (4x2900PS, 6xFabric-E modules, 1xSupervisorE module)
- ToR switches: Arista 7050X

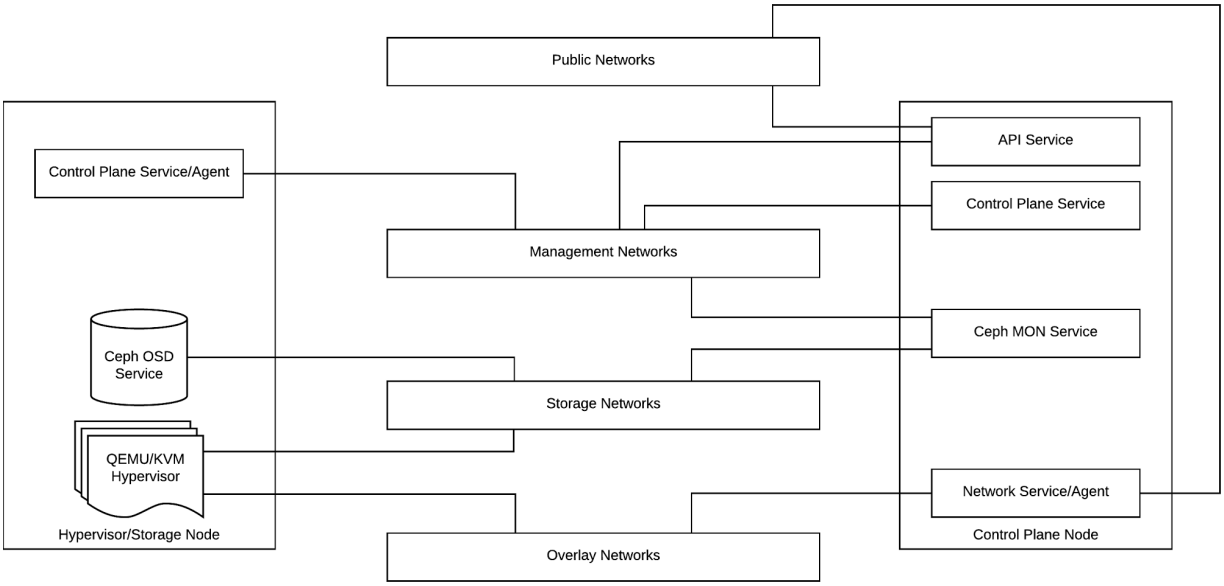
Network for OpenStack Platform

OpenStack platform uses underlay network to exchange data between its components, expose public API endpoints and transport the data of overlay or tenant networks. The following classes of networks are defined for OpenStack platform in proposed architecture.

- Management network. This is the network where sensitive data exchange happens. Sensitive data includes authentication and authorization credentials and tokens, database traffic, RPC messages and notifications. This network also provides access to administrative API endpoints exposed by components of the platform.
- **Public network. Network of this class provides access to API** endpoints exposed by various components of the platform. It also connects Floating IPs of virtual server instances to external network segments and Internet.
- Storage network. Specialized network to transport data of storage subsystem, i.e. Ceph cluster. It also connects to internal Ceph API endpoints.
- Overlay network. Networks of this class transport the payload of tenant networks, i.e. the private networks connecting virtual server instances. The current architecture employs VXLANs for L2 encapsulation.

There can be one or more networks of each class in a particular cloud, for example, Management network class can include separate segments for messaging, database traffic and Admin API endpoints, and so on.

In addition to role segmentation, networks in one class can be separated by scale. For example, Public network class might include multiple per-rack segments, all connected by an L3 router.



5.4.5 Control Plane

OpenStack Overview

OpenStack is a system that provides Infrastructure as a Service (IaaS). IaaS is essentially a set of APIs that allow for creation of elements of typical data center infrastructure, such as virtual servers, networks, virtual disks and so on. Every aspect of an infrastructure is controlled by a particular component of OpenStack:

- OpenStack Compute (Nova) controls virtual servers lifecycle, from creation through management to termination.
- OpenStack Networking (Neutron) provides connectivity between virtual servers and to the world outside.
- OpenStack Image (Glance) holds base disk images used to boot the instances of virtual servers with an operating system contained in the image.
- OpenStack Block Storage (Cinder) manages the virtual block storage and supplies it to virtual servers as block devices.
- OpenStack Identity (Keystone) provides authentication/authorization to other components and clients of all APIs.

Guidelines for OpenStack at Scale

Currently, OpenStack scalability is limited by several factors:

- Placement responsiveness desired
- MySQL scalability
- Messaging system scalability
- Scalability of the SDN
- Scalability of the SDS

Scheduler Scalability

In general, the scheduling strategy for OpenStack is a completely optimistic determination which means the state of all things is considered whenever a placement decision is made. As more factors are added to the consideration set (special hardware, more compute nodes, affinity, etc.) the time to place resources increases at a quadratic rate. There is work being done on splitting out the scheduling and placement parts of Nova and isolating them as a separate service, like Cinder or Neutron. For now, however, this is a work in progress and we should not rely on increased scheduler performance for another few releases.

OpenStack can seamlessly accommodate different server types based on CPU, memory and local disk without partitioning of the server pool. Several partitioning schemes exist that provide ability to specify pools of servers appropriate for a particular workload type. A commonly used scheme is server aggregates, which allows specific image sets to be scheduled to a set of servers based on server and image tags that the administrator has defined.

MySQL Scalability

As almost all OpenStack services use MySQL to track state, scalability of the database can become a problem. Mirantis OpenStack deploys with a MySQL + Galera cluster which allows for reads and writes to be scaled in a limited fashion. There is also the option to offload many types of reads to asynchronous slaves which was introduced in the Icehouse release. We should see support for both Galera and the asynchronous approaches increase in Kilo and subsequent releases. There are OpenStack installations in the wild with tens of thousands of nodes running under both of these scaling models. However, careful tuning and care of the databases is a very important consideration. MySQL is recommended to run on dedicated nodes. The Control Plane DB should not be used by Tenant Applications or other solutions like Zabbix; those should provision another MySQL cluster on other physical nodes.

Messaging System Scalability

A majority of OpenStack services use AMQP implementations (RabbitMQ and Qpid) for message transport and RPC. As with MySQL, there are examples of installations in the wild which have successfully scaled this messaging infrastructure to tens of thousands of nodes. However, without a deliberate federation strategy and/or careful tuning and maintenance, these systems easily become unstable around 700-1000 compute nodes depending on usage patterns. To scale OpenStack past 200 nodes, all these areas will require planning.

OpenStack compute design is modular in nature. Beginning with KVM as the only hypervisor choice does not constrain us from building a multi-hypervisor cloud later, when need arises.

Services Overview

The following paragraphs describe how the individual services and components should be placed and configured in the proposed Reference Architecture. All services are divided into two categories: stateless and stateful. Each category requires specific approach which is outlined below.

Stateless Services

Services of this type do not record their state in their environment. Those services can be killed and restarted without risk of losing data. Most of OpenStack component services are inherently stateless being either HTTP-based API servers or message queue processors.

General approach to stateless services is envelop the service into Docker container with minimal to no configuration embedded.

Stateful Services

Services that keep track of their state in some persistent storage and cannot be restarted in clean environment without losing that state are considered stateful. The main stateful component of the OpenStack platform is MySQL state database. Most of other services rely on the database to keep track of their own state.

Containers that run stateful services must have persistent storage attached to them. This storage must be made available in case if the container with stateful service should be move to another location. Such availability can be ensured on application level via replication of some sort, or on network level by redirecting connections from moving container to the same storage location.

5.4.6 OpenStack Identity (Keystone)

Keystone service provides identity to all other OpenStack services and external clients of OpenStack APIs. The main component of Identity service is an HTTP server that exposes an API of the service.

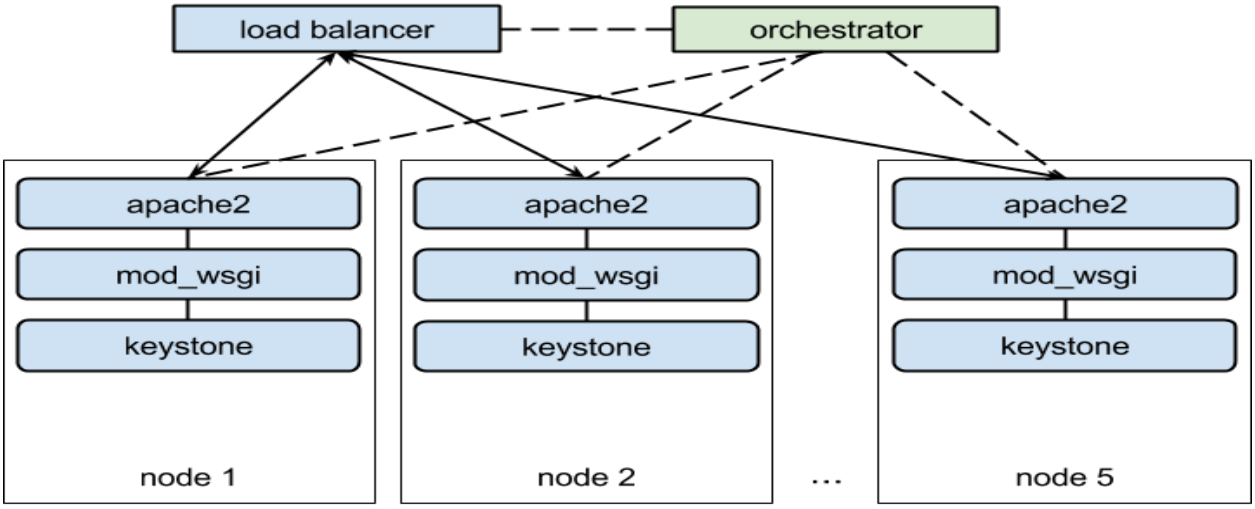
Keystone service is used by all other components of OpenStack for authorization of requests, including system requests between services and thus gets called a lot even if the platform is idle. This generates load on CPU resources of server where it runs. Thus, it is recommended to run Keystone service on dedicated node and not co-locate it with other resource intensive components.

For availability and load distribution it is recommended to run at least 5 instances of Keystone at the scale of hundreds of nodes. Load balancer with a Virtual IP address should be placed in front of the Keystone services to ensure handling of failures and even distribution of requests.

Per these recommendations, the required number of dedicated physical nodes to run Keystone service is 5. These nodes can be used to host other low footprint services. Resource intensive services should not run on those nodes.

Apache2 Web Server

Apache web server wraps Keystone and Horizon services and exposes them as a web-services. However, since Horizon can generate significant load on resources, it is not recommended to co-locate it with Keystone. See below for recommendations on scaling Horizon.



5.4.7 OpenStack Compute (Nova)

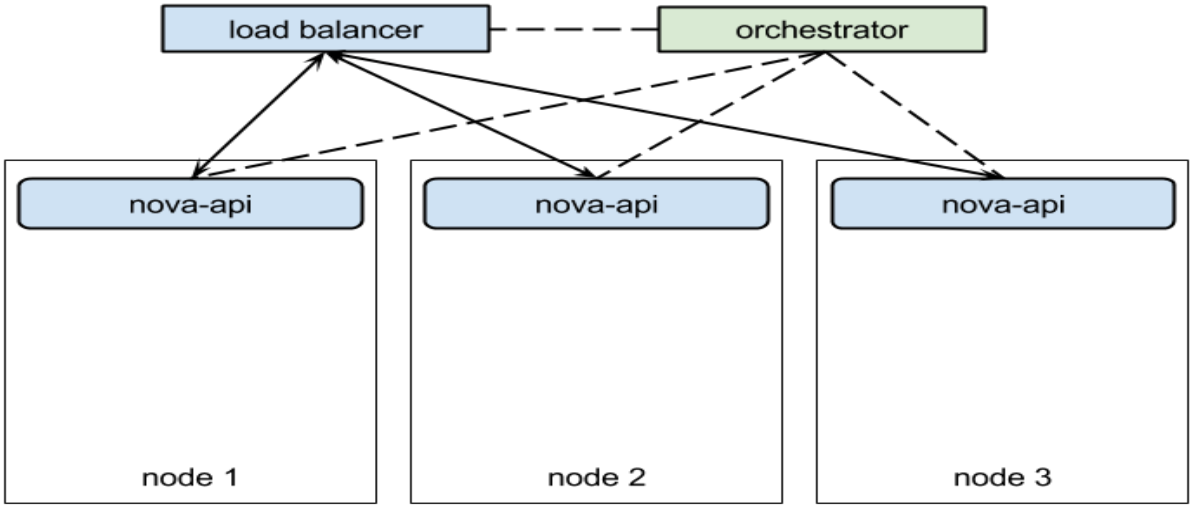
Nova has multiple service not all of which are stateless. API server, scheduler and conductor in particular are stateless, while nova-compute service is not as it manages the state of data-plane components on a compute node.

Nova API

Web server that exposes Compute API of the OpenStack platform. This is a stateless service. Nova API server consumes significant resources at scale of hundreds of nodes.

Nova API server should run on a dedicated physical servers to ensure it does not share resources with other services with comparably big footprint. It can co-locate with more lightweight services.

For availability reasons, it is recommended to run at least 3 instances of API server at any time. Load balancer with Virtual IP address shall be placed in front of all API servers.



Nova Scheduler

Scheduling service of Nova is stateless and can be co-located with services that have larger footprint. Scheduler should be scaled by adding more service instances since its architecture is inherently non-scalable due to using single thread model.

Using multiple scheduler processes might lead to unexpected oversubscription of hypervisor resources if many simultaneous placement requests are being served.

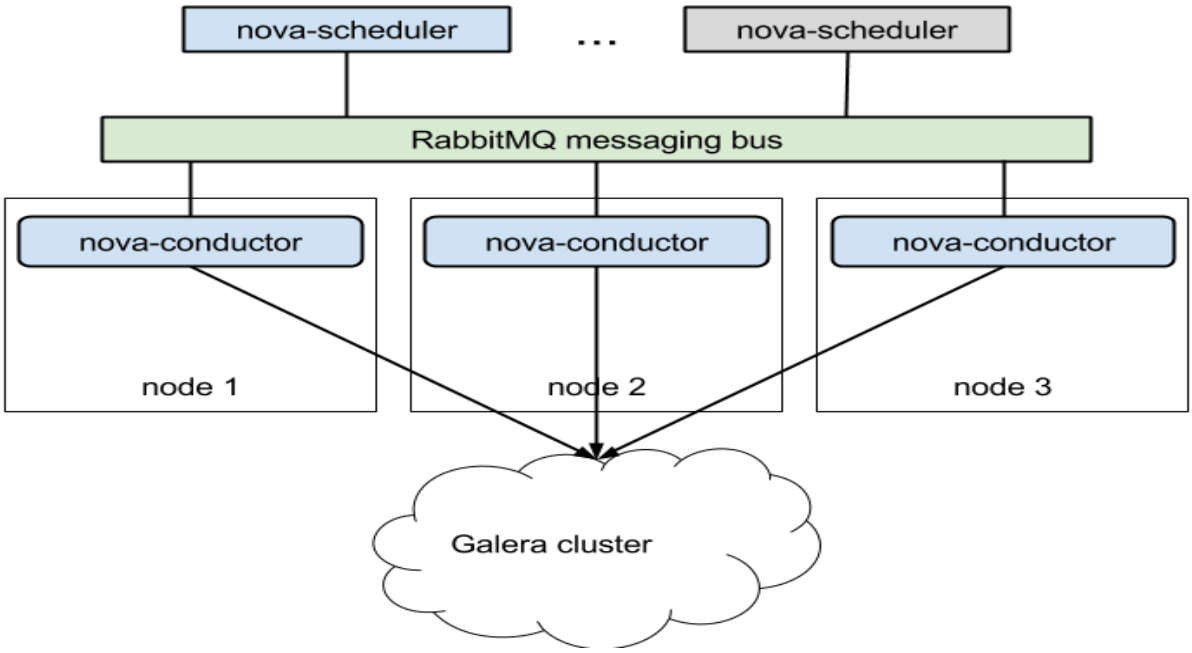
At the scale of hundreds of nodes use at least 8 instances of Scheduler. Since they can be co-located with other services, it won't take any additional physical servers from the pool. Schedulers can be placed to nodes running other Nova control plane services (i.e. API and Conductor).

Nova Conductor

Centralized conductor service is recommended for large-scale installations of Nova. It adds database interface layer for improved security and complex operations. It is a stateless service that is scaled horizontally by adding instances of it running on different physical servers.

Conductor is resources-intensive service and should not be co-located with other services. It runs on dedicated hardware nodes. One instance of Conductor service should be running per physical node.

To ensure that the Conductor service is highly available, run 3 instances of the service at any time. This will require 3 dedicated physical servers from the pool.



Nova Compute

Compute service is essentially an agent service of Nova. An instance of it runs on every hypervisor node in an OpenStack environment and controls the virtual machines and other connected resources on a node level.

Compute service in the proposed architecture is not containerized. It runs in hypervisor host and talks to different local and remote APIs to get the virtual machines up and running. One instance of Compute service runs per a hypervisor.

Services Placement

Nova control plane services use at least 6 dedicated nodes:

- 3 nodes for 3 instances of Conductor service
- 3 nodes for 3 instances of API service

8 instances of Scheduler service are distributed between the said dedicated nodes and/or nodes dedicated to other components of the platform.

Load balancer is placed in front of the API services to ensure availability and load distribution for the Compute API of OpenStack platform.

Compute services run per-node on hypervisor hosts (compute nodes) in non-redundant fashion.

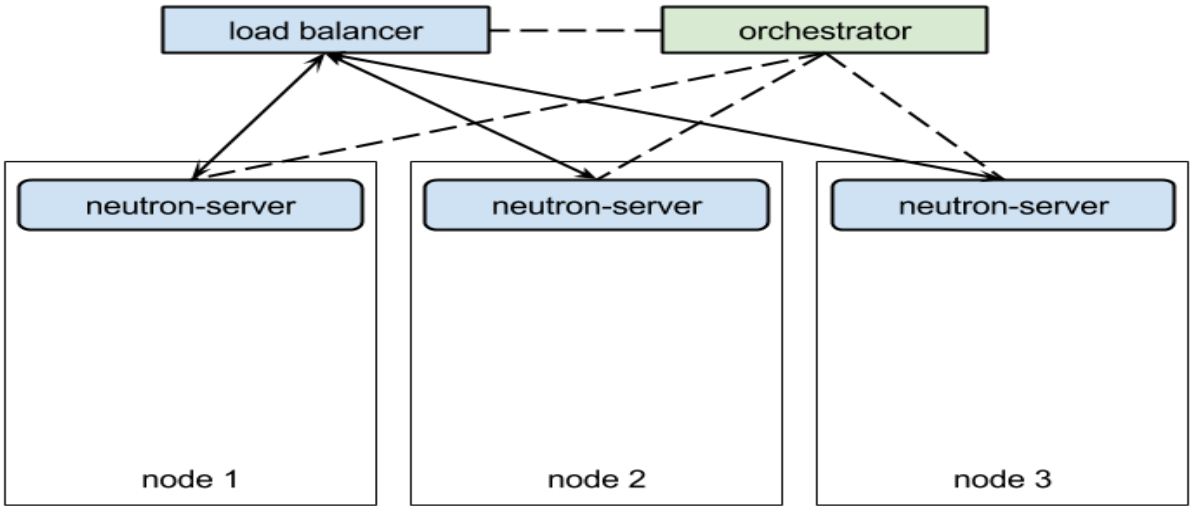
5.4.8 OpenStack Networking (Neutron)

This component includes API server that exposes HTTP-based API and a set of agents which actually manage data plane resources, such as IP addresses, firewall and virtual switches.

Neutron Server

An API server exposes Neutron API and passes all web service calls to the Neutron plugin for processing. This service generates moderate load on processors and memory of physical servers. In proposed architecture, it runs on the same nodes as other lightweight API servers. A minimal of 3 instances of the server is recommended for redundancy and load distribution.

Load balancer working in front of the API servers helps to ensure their availability and distribute the flow of requests.



Neutron DHCP agent

DHCP agent of Neutron is responsible for assigning IP addresses to VMs. The agent is horizontally scalable by adding new instances of it distributed between different nodes. DHCP agent can be co-located to any other services and can run on hypervisor hosts as well.

Neutron L3 agent

L3 agent controls routing in Public and Private networks by creating and managing namespaces, routers, floating IP addresses and network translations. The agent can be scaled by adding instances. High availability of the agent is ensured by running its instances in Corosync/Pacemaker cluster or using orchestrator-driven clustering and state tracking (e.g. Kubernetes events system with etcd).

The agent has low resources footprint and can be co-located with more resource-intensive services, for example, Neutron Server.

Neutron L2 agent

L2 agent manages data link layer configuration of the overlay network for Compute nodes. It also provides L2 functions to L3 agent. The agent is specific to Neutron plugin.

The L2 agent used with OpenVSwitch plugin generates high CPU load when creates and monitors the OVS configurations. It has to run on any host that runs nova-compute, neutron-l3-agent or neutron-dhcp-agent.

Neutron metadata agent

The metadata agent provides network metadata to virtual servers. For availability and redundancy, it is recommended to run at least 3 instances of the agent. They can share a node with other Neutron or control plane services.

Services Placement

Neutron control plane services require at least 3 nodes for redundancy of Neutron Server. Three instances are recommended for high availability. All micro services of Neutron could be co-located with the Neutron Servers or with other components' services.

Load balancer should be placed in front of the Neutron Server to ensure availability and load distribution for the Network API of the OpenStack platform.

L3 and DHCP agents are co-located with instances of Neutron Server on the same physical nodes. L2 agent works on every Compute node and on every node that runs L3 and/or DHCP agent.

5.4.9 OpenStack Images (Glance)

Images service consists of API and indexing service. Both of them are stateless and can be containerized.

Glance API

This service exposes Images API of OpenStack platform. It is used to upload and download images and snapshot.

Glance API server has low resource consumption and can be co-located with other services. It does not require dedicated physical servers.

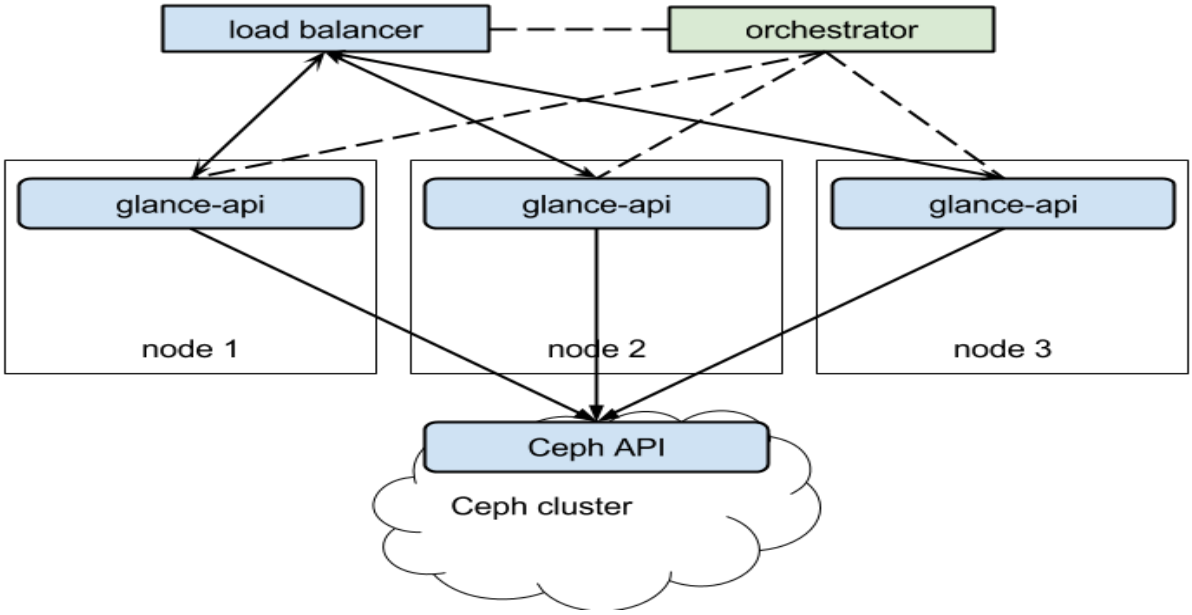
Glance API server scales by adding new instances. Load balancer is required to provide a single virtual address to access the API. The load can be evenly distributed between instances of Glance API.

Important parameter of the Image Service architecture is the storage backend. The following types of storage are proposed for Glance in this Reference Architecture:

- Local filesystem
- Ceph cluster

With local storage, the consistency of the store for all instances of Glance API must be ensured by an external component (e.g. replication daemon).

Ceph backend natively provides availability and replication of stored data. Multiple instances of Glance API service work with the same shared store in Ceph cluster.



Glance Registry

Registry serves images metadata part of the Images API. It stores and manages a catalog of images. It has low footprint and can share a physical node with other resource intensive services.

Services Placement

Micro services of Glance do not require dedicated physical servers. They can be co-located with other services.

For the purposes of high availability and redundancy, at least 3 instances of Glance API service should run at any time. Load balancer must be placed in front of those instances to provide single API endpoint and distribute the load.

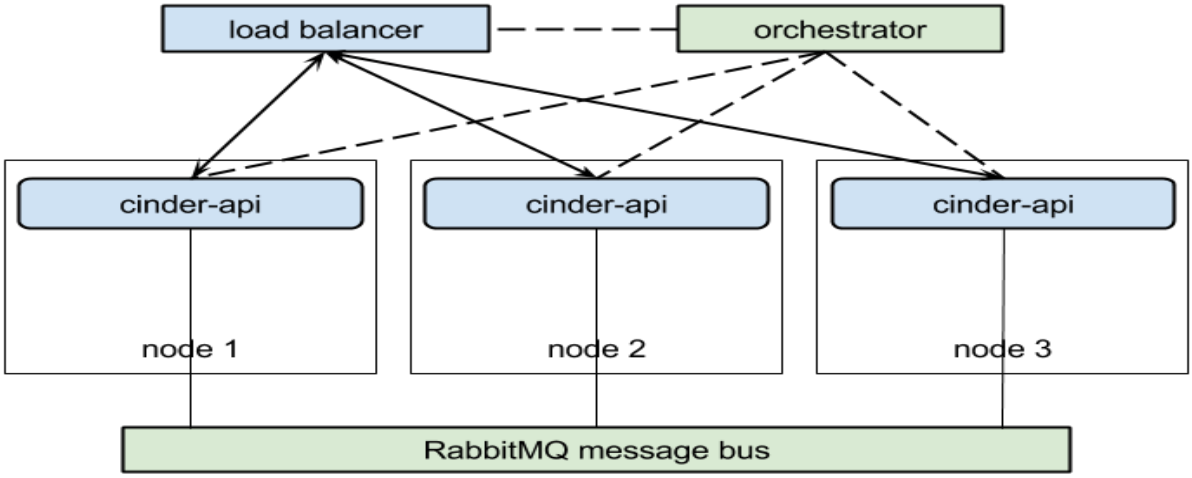
5.4.10 OpenStack Block Storage (Cinder)

Storage service manages and exposes a virtual block storage. The server that handles the management of low-level storage requires access to the disk subsystem.

Cinder API

This service exposes Volume API of the OpenStack platform. Volume API is not very often used by other components of OpenStack and doesn't consume too much resources. It could run on the same physical node with other resource non intensive services.

For availability and redundancy purposes, it is proposed to run at least 3 instances of Cinder API. Load balancer should be placed in front of these instances to provide distribution of load and failover capabilities.



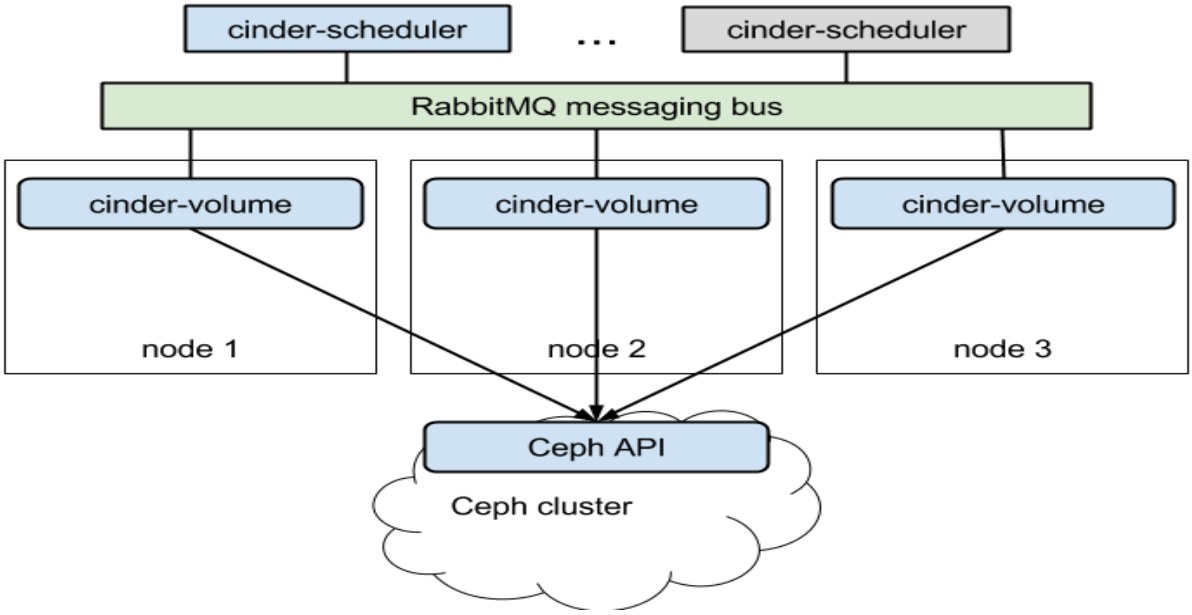
Cinder Scheduler

Scheduler selects an instance of the Volume micro service to call when a client requests creation of a virtual block volume. Scheduler is not resource intensive and can be co-located with other services. It scales horizontally by adding new instances of the service. For redundancy purposes, the instances of scheduler service should be distributed between different physical servers.

Cinder Volume

Volume service manages block storage via appropriate API which depends on the technology in use. With Ceph as a provider of virtual block storage, single instance of Cinder Volume service is required to manage the Ceph cluster via its API. If virtual block storage is provided by using LVM with local disks of Compute nodes, the Volume service must be running on every Compute node in the OpenStack environment.

In Ceph case, it is recommended to run at least 3 instances of the Volume service for the availability and redundancy of the service. Each virtual volume is managed by one instance Volume service at a time. However, if that instance is lost, another one takes over on its volumes.



Services Placement

Cinder services do not require dedicated physical nodes. They run on the same physical servers with other components of control plane of the OpenStack platform.

The instances of Cinder API service are placed behind a loca balancer to ensure the distribution of load and availability of the service.

5.4.11 OpenStack Dashboard (Horizon)

Horizon dashboard provides user interface to the cloud's provisioning APIs. This is a web application running on top of Apache2 web server. For availability purposes, multiple instances of the server are started. Load balancer is placed in front of the instances to provide load distribution and failover. Horizon does not require dedicated physical node. It can be co-located with other services of other components of the OpenStack platform.

For security reasons, it is not recommended to use the same instances Apache2 server to wrap both Horizon and Keystone API services.

5.4.12 RabbitMQ

The messaging server allows all distributed components of an OpenStack service to communicate to each other. Services use internal RPC for communication. All OpenStack services also send broadcast notifications via messaging queue bus.

Clustering

The prerequisite for High Availability of queue server is the configured and working RabbitMQ cluster. All data/state required for the operation of a RabbitMQ cluster is replicated across all nodes. An exception to this are message queues, which by default reside on one node, though they are visible and reachable from all nodes.

Cluster assembly requires installing and using a clustering plugin on all servers. Proposed solution for RabbitMQ clustering is the rabbitmq-autocluster*plugin*.

The RabbitMQ cluster also needs proper fencing mechanism to exclude split brain conditions and preserve a quorum. Proposed solution for this problem is using 'pause_minority' partition mode with the rabbit-autocluster plugin.

Replication

Replication mechanism for RabbitMQ queues is known as 'mirroring'. By default, queues within a RabbitMQ cluster are located on a single node (the node on which they were first declared). This is unlike exchanges and bindings, which can always be considered to be on all nodes. Queues can optionally be made mirrored across multiple nodes. Each mirrored queue consists of one master and one or more slaves, with the oldest slave being promoted to the new master if the old master disappears for any reason.

Messages published to the queue are replicated to all members of the cluster. Consumers are connected to the master regardless of which node they connect to, with slave nodes dropping messages that have been acknowledged at the master.

Queue mirroring therefore aims to enhance availability, but not distribution of load across nodes (all participating nodes each do all the work). It is important to note that using mirroring in RabbitMQ actually reduces the availability of queues by dropping performance by about 2 times in tests, and eventually leads to failures of RabbitMQ because of extremely high rate of context switches at node's CPUs.

There are two main types of messages in OpenStack:

- **Remote Procedure Call (RPC) messages carry commands and/or requests** between microservices within a single component of OpenStack platform (e.g. nova-conductor to nova-compute).
- **Notification messages are issued by a microservice upon specific** events and are consumed by other components (e.g. Nova notifications about creating VMs are consumed by Ceilometer).

In proposed OpenStack architecture, only notification queues are mirrored. All other queues are not, and if the instance of RabbitMQ server dies after a message sent, but before it is read, that message is gone forever. This is a trade-off for significant (at least 2 times) performance and stability boost in potential bottleneck service. Drawbacks of this mode of operation are:

- **Long-running tasks might stuck in transition states due to loss of** messages. For example, Heat stacks might never leave spawning state. Most of the time, such conditions could be fixed by the user via API.

Data Persistence

OpenStack's RPC mechanism does not impose requirements for durable queues or messages. Thus, no durability required for RabbitMQ queues, and there are no 'disk' nodes in cluster. Restarting a RabbitMQ node then will cause all data of that node to be lost. Since OpenStack does not rely on RPC as a guaranteed transport, it doesn't break the control plane. Clients shall detect failure of a server they are talking to and connect to another server automatically.

RabbitMQ service considered stateless in terms defined in this document due to the reasons mentioned above.

Networking Considerations

RabbitMQ nodes address each other using domain names, either short or fully-qualified (FQDNs). Therefore hostnames of all cluster members must be resolvable from all cluster nodes, as well as machines on which command line tools such as rabbitmqctl might be used.

RabbitMQ clustering has several modes of dealing with network partitions, primarily consistency oriented. Clustering is meant to be used across LAN. It is not recommended to run clusters that span WAN. The Shovel or Federation

plugins are better solutions for connecting brokers across a WAN. Note that Shovel and Federation are not equivalent to clustering.

Services Placement

RabbitMQ servers are to be installed on the dedicated nodes. Co-locating RabbitMQ with other Control Plane services has negative impact on its performance and stability due to high resource consumption under the load. Other services that have different resource usage patterns can prevent RabbitMQ from allocating sufficient resources and thus make messaging unstable.

Based on that, RabbitMQ will require 3 dedicated nodes out of the pool of Compute/Controller servers.

Alternatives

RabbitMQ is a server of choice for OpenStack messaging. Other alternatives include:

- **0MQ (ZeroMQ), a lightweight messaging library that integrates into** all components and provides serverless distributed message exchange.
- **Kafka, a distributed commit log type messaging system, supported by** oslo.messaging library in experimental mode.

ZeroMQ

This library provides direct exchange of messages between microservices. Its architecture may include simple brokers or proxies that just relay messages to endpoints, thus reducing the number of network connections.

ZeroMQ library support was present in OpenStack since early releases. However, the implementation assumed direct connections between services and thus a full mesh network between all nodes. This architecture doesn't scale well. More recent implementations introduce simple proxy services on every host that aggregate messages and relay them to a central proxy, which does host-based routing.

Benchmarks show that both direct and proxy-based ZeroMQ implementations are more efficient than RabbitMQ in terms of throughput and latency. However, in the direct implementation, quick exhaustion of network connections limit occurs at scale.

The major down side of the ZeroMQ-based solution is that the queues don't have any persistence. This is acceptable for RPC messaging, but Notifications may require durable queues. Thus, if RPC is using ZeroMQ, the Telemetry will require a separate messaging transport (RabbitMQ or Kafka).

Kafka

Distributed commit log based service Kafka is supported in OpenStack's oslo.messaging library as an experimental. This makes it unfeasible to include in the Reference Architecture..

5.4.13 MySQL/Galera

State database of OpenStack contains all data that describe the state of the cloud. All components of OpenStack use the database to read and store changes in their state and state of the data plane components.

Clustering

The proposed clustering solution is based on the native orchestrator-specific state management with etcd providing distributed monitoring and data exchange for the cluster. Cluster operations will be triggered by orchestrator events and handled by custom scripts.

Failover and fencing of failed instances of MySQL is provided by scripts triggered by the orchestrator upon changes in state and availability of the members of Galera cluster. State and configuration information is provided by etcd cluster.

Data Persistence

Galera implements replication mechanism to ensure that any data written to one of the MySQL servers is synchronously duplicated to other members of the cluster. When a new instance joins the cluster, one of the two replication methods is used to synchronize it: IST or SST. If the initial data set exists on the new node, incremental method (IST) is used. Otherwise, full replication will be performed (SST).

Since all nodes in the cluster have synchronous copies of the data set at any time, there is no need to use shared storage. All DB servers work with the local disk storage.

Replication

Incremental synchronous replication is used to keep MySQL databases of members in Galera cluster in sync. If a new member is added to the cluster, full replication (SST) will be performed.

Full SST replication can take indefinite time if the data set is big enough. To mitigate this risk, the proposed architecture includes a number of hot stand-by MySQL servers in addition to one Active server. The access to said servers is provided by an instance of a load balancer (see details in Networking Considerations section).

Proposed architecture allows to quickly replace failing instances of MySQL server without need to run full replication. It is still necessary to restore the pool of hot standby instances whenever the failover event occurs.

Networking Considerations

Load balancer is a key element of networking configuration of the Galera cluster. Load balancer must be coordinated with the cluster, in terms that it redirect write requests to appropriate instance of MySQL server. It also ensures failover to hot standby instances and fencing of failed active nodes.

Services Placement

MySQL servers forming the Galera cluster must run on dedicated physical servers due to their intensive use of node's resources.

5.4.14 Ceph Distributed Storage

Summary

Ceph is a distributed storage system with built-in replication capability. Architecture of Ceph is designed for resiliency and durability.

Ceph provides few different APIs for different types of supported storage:

- Distributed file system
- Object storage

- Virtual block device

OpenStack platform generally uses Object API and Block Device API of Ceph for Image and Virtual Block storage correspondingly.

Main components of Ceph are as follows.

- A Ceph Monitor maintains a master copy of the cluster map. A cluster of Ceph monitors ensures high availability should a monitor daemon fail. Storage cluster clients retrieve a copy of the cluster map from the Ceph Monitor.
- A Ceph OSD Daemon checks its own state and the state of other OSDs and reports back to monitors.
- RADOS gateway exposes the Object API of the platform. This API is generally compatible with OpenStack Object Storage API, which allows to use it as a Glance back-end without additional modifications.

Ceph Monitor

For reliability purposes, Ceph Monitors should be placed to different physical nodes. Those nodes might be the Storage nodes themselves, albeit that is not generally recommended. Proper resilience of the cluster can be ensured by using 3 or 5 instances of Ceph Monitor, each running on separate host.

Ceph OSD

Every storage device requires an instance of Ceph OSD daemon to run on the node. These daemons might be resource intensive under certain conditions. Since one node usually have multiple devices attached to it, there are usually more than one OSD process running on the node. Thus, it is recommended that no other services are placed to the OSD nodes.

RADOS Gateway

This service exposes an Object API via HTTP. It have low resource footprint and can be co-located with other services, for example, with a Ceph Monitor. Multiple radosgw daemons should be used to provide redundancy of the service. Load balancer should be placed in front of instances of radosgw for load distribution and failover.

Services Placement

Ceph scales very well by adding new OSD nodes when capacity increase is required. So the size of the Ceph cluster may vary for different clouds of the same scale. In this document, a cluster with 45 OSD nodes is described.

Instances of Monitor service require 1 dedicated node per instance, to the total of 3 nodes. RADOS gateway servers run on the same nodes as Monitors.

5.4.15 Control Plane Operations Monitoring

Summary

Monitoring the OpenStack Control Plane infrastructure is essential for operating the platform. The main goal of it is to ensure that an operator is alerted about failures and degradations in service level of the environment. Metering plays less important role in this type of monitoring.

Currently proposed solution for infrastructure monitoring is produced by Stacklight project.

Stacklight is a distributed framework for collecting, processing and analyzing metrics and logs from OpenStack components. It includes the following services:

- Stacklight Collector + Aggregator
- Elasticsearch + Kibana
- InfluxDB + Grafana
- Nagios

Stacklight Collector

Smart agent that runs on every node in the OpenStack environment. It collects logs, processes metrics and notifications, generates and sends alerts when needed. Specialized instance of Collector called Aggregator aggregates metrics on the cluster level and performs special correlation functions.

Elasticsearch + Kibana

These components are responsible for analyzing large amounts of textual data, particularly logs records and files coming from OpenStack platform nodes. Kibana provides graphical interface that allows to configure and view correlations in messages from multiple sources.

A typical setup at least requires a quad-core server with 8 GB of RAM and fast disks (ideally, SSDs). The actual disk space you need to run the subsystem on depends on several factors including the size of your OpenStack environment, the retention period, the logging level, and workload. The more of the above, the more disk space you need to run the Elaticsearch-Kibana solution. It is highly recommended to use dedicated disk(s) for your data storage.

InfluxDB + Grafana

InfluxDB is a time series database that provides high throughput and real-time queries for reduced support of standard SQL capabilities like efficiently updating records. Grafana exposes graphic interface to time series data, displays graphs of certain metrics in time and so on.

The hardware configuration (RAM, CPU, disk(s)) required by this subsystem depends on the size of your cloud environment and other factors like the retention policy. An average setup would require a quad-core server with 8 GB of RAM and access to a 500-1000 IOPS disk. For sizeable production deployments it is strongly recommended to use a disk capable of 1000+ IOPS like an SSD. See the *InfluxDB Hardware Sizing Guide* for additional sizing information.

Nagios

In Stacklight architecture, Nagios does not perform actual checks on the hosts. Instead it provides transport and user interface for the alerting subsystem. It receives alerts from Collectors and generates notifications to the end users or cloud operators.

A typical setup would at least require a quad-core server with 8 GB of RAM and fast disks (ideally, SSDs).

Services Placement

InfluxDB and Elasticsearch subsystems of the Stacklight solution should run on dedicated servers. Additionally, clustered InfluxDB needs at least three nodes to form a quorum. Elasticsearch scales horizontally by adding instances, each running on a separate node. At least three instances of Elasticsearch are recommended.

Stacklight Aggregator service could share a physical node with other low-profile services.

Nagios server can be co-located with Stacklight Aggregator due to its low resource footprint.

The total of 5 additional physical nodes are required to install Control Plane Operations Monitoring framework based on Stacklight.

5.4.16 Services Placement Summary

The following table summarizes the placement requirements of the services described above.

Service	Number Of Instances	Number Of Dedicated Nodes	Can Share A Node	Requires a Load Balancer
keystone-all	5	5	no	yes
nova-api	3	3	no	yes
nova-scheduler	8	–	yes	no
nova-conductor	3	3	no	no
nova-compute*	–	–	yes	no
neutron-server	3	–	yes	no
neutron-dhcp-agent	3	–	yes	no
neutron-l2-agent*	–	–	yes	no
neutron-l3-agent	3	–	yes	no
neutron-metadata-agent	3	–	yes	no
glance-api	3	–	yes	yes
glance-registry	3	–	yes	no
cinder-api	3	–	yes	yes
cinder-scheduler	3	–	yes	no
cinder-volume	–	–	yes	no
horizon/apache2	3	3	no	yes
rabbitmq-server	3	3	no	no
mysqld-server	3	3	no	yes**
ceph-mon	3	3	no	no
ceph-osd***	–	–	no	no
radosgw	3	–	yes	yes
lma-aggregator	1	1	yes	no
Influxdb + Grafana	3	3	no	no
Elasticsearch + Kibana	3	3	no	no
Nagios	1	–	yes	no
TOTAL	–	30	–	–

* this service runs on a Compute/hypervisor host

** this service might require specialized load balancer (proxysql)

*** this service runs on a Storage/OSD node

Total number of nodes for control plane is 30 which is not so high footprint at scale of 1000 nodes. However, it could be reduced further by using cgroups and/or container orchestrator-specific mechanisms to guarantee availability of certain amount of resources to certain resource-intensive services running on the same node. This will allow to co-locate resource-intensive services with lower risk of interference and reduce the footprint of the Platform's control plane.

5.4.17 Data Plane

Compute Virtualization

QEMU implementation of Linux KVM is used as a virtualization hypervisor. The KVM runs under control of libvirtd daemon, configured and managed by Nova Compute microservice (nova-compute).

Network

Depending on the picked Neutron plugin, data plane network might be realized by different technology stacks. Reference architecture employs OVS+VXLAN plugin, network data plane at compute hosts runs on OpenVSwitch virtual switching software.

Storage

Local disks and Ceph cluster RBD volumes are both used to provide block storage capability to different services in the platform.

Ephemeral

Disk files of the VM instances running in the cloud are stored in Ceph distributed storage via RBD. This enables live migration of VMs and high availability of data in certain cases.

Virtual Block Devices

Virtual block storage devices are provided by Ceph via RBD.

Images

Images used by Nova to spin up new VM instances are stored in Ceph via Glance service's back end driver.

Snapshot

Snapshots are stored in Ceph.

Database

Database files are kept on the local storage devices of nodes that run database servers. The replication and availability of the data are ensured by WSREP mechanism of Galera cluster.

Monitoring

The monitoring metrics and time-series data are written to InfluxDB database. The database keeps its files on a local storage, similar to the approach taken for state database (see above).

Logs

Logs are written to local disks of nodes, plus optionally to remote logs collector service which is the part of Telemetry component of the platform.

5.4.18 References

This section contains references to external documents used in preparation of this document.

1. Testing on scale of 1000 compute nodes
2. Performance testing of OpenStack
3. RabbitMQ Clustering
4. RabbitMQ High Availability
5. ZeroMQ Status Update

5.5 Kubernetes Master Tier For 1000 Nodes Scale

Table of Contents

5.5.1 Introduction

This document describes architecture, configuration and installation workflow of Kubernetes cluster for OpenStack Containerised Control Plane (CCP) on a set of hosts, either baremetal or virtual. Proposed architecture should scale up to 1000 nodes.

Scope of the document

This document does not cover preparation of host nodes and installation of a CI/CD system. This document covers only Kubernetes and related services on a preinstalled operating system with configured partitioning and networking.

Monitoring related tooling will be installed on ready to use Kubernetes as Pods, after Kubernetes installer finishes installation. This document does not cover architecture and implementation details of monitoring and profiling tools.

Lifecycle Management section describes only Kubernetes and related services. It does not cover applications that run in Kubernetes cluster.

5.5.2 Solution Prerequisites

Hardware

The proposed design was verified on a hardware lab that included 181 physical hosts of the following configuration:

- Server model: HP ProLiant DL380 Gen9
- CPU: 2 x Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz
- RAM: 264G
- Storage: 3.0T on RAID on HP Smart Array P840 Controller
- HDD: 12 x HP EH0600JDYTL
- Network: 2 x Intel Corporation Ethernet 10G 2P X710

3 out of the 181 hosts were used to install Kubernetes Master control plane services. On every other host, 5 virtual machines were started to ensure contention of resources and serve as Minion nodes in Kubernetes cluster.

Minimal requirements for the control plane services at scale of 1000 nodes are relatively modest. Tests demonstrate that three physical nodes in the configuration specified above are sufficient to run all control plane services for cluster of this size, even though an application running on top of the cluster is rather complex (i.e. OpenStack control plane + compute cluster).

Provisioning

Hosts for Kubernetes cluster must be prepared by a provsioning system of some sort. It is assumed that users might have their own provisioning system to handle prerequisites for this.

Provisioning system provides installed and configured operating system, networking, partitioning. It should operate on its own subset of cluster metadata. Some elements of that metadata will be used by installer tools for Kubernetes Master and OpenStack Control tiers.

The following prerequisites are required from Provisioning system.

Operating System

- Ubuntu 16.04 is default choice of operating system.
- It has to be installed and configured by provisioning system.

Networking

Before the deployment starts networking has to be configured and verified by underlay tooling:

- Bonding.
- Bridges (possibly).
- Multi-tiered networking.

- IP addresses assignment.
- SSH access from CI/CD nodes to cluster nodes (is required for Kubernetes installer).

Such things as DPDK and Contrail can be most likely configured in containers boot in privileged mode, no underlay involvement is required:

- Load DKMS modules
- Change runtime kernel parameters

Partitioning

Nodes should be efficiently pre-partitioned (e.g. separation of `/`, `/var/log`, `/var/lib` directories).

Additionally it's required to have LVM Volume Groups, which further will be used by:

- LVM backend for ephemeral storage for Nova.
- LVM backend for Kubernetes, it may be required to create several Volume Groups for Kubernetes, e.g. some of the services require SSD (InfluxDB), other will work fine on HDD.

Some customers also require Multipath disks to be configured.

Additional Ansible packages (optional)

Currently Kubespray project is employed for installing Kubernetes. It provides Calico and Ubuntu/Debian support.

Kubespray Ansible playbooks (or Kargo) are accepted into Kubernenes incubator by the community.

Ansible requires:

- `python2.7`
- `python-netaddr`

Ansible 2.1.0 or greater is required for Kargo deployment.

Ansible installs and manages Kubernetes related services (see Components section) which should be delivered and installed as containers. Kubernetes has to be installed in HA mode, so that failure of a single master node does not cause control plane down-time.

The long term strategy should be to reduce amount of Ansible playbooks we have to support and to do initial deployment and Lifecycle Management with Kubernetes itself and related tools.

Node Decommissioning

Many Lifecycle Management scenarios require nodes decommissioning procedure. Strategy on decommissioning may depend on the customer and tightly coupled with Underlay tooling.

In order to properly remove the node from the cluster, a sequence of actions has to be performed by overlay tooling, to gracefully remove services from cluster and migrate workload (depends on the role).

Possible scenarios of node decommissioning for underlay tooling:

- Shut the node down.
- Move node to bootstrap stage.
- As a common practise we should not erase disks of the node, customers occasionally delete their production nodes, there should be a way to recover them (if they were not recycled).

CI/CD

Runs a chain of jobs in predefined order, like deployment and verification. CI/CD has to provide a way to trigger a chain of jobs (git push trigger -> deploy -> verify), also there should be a way to share data between different jobs for example if IP allocation happens on job execution allocated IP addresses should be available for overlay installer job to consume.

Non comprehensive list of functionality:

- Jobs definitions.
- Declarative definition of jobs pipelines.
- Data sharing between jobs.
- Artifacts (images, configurations, packages etc).

User experience

1. User should be able to define a mapping of node and high level roles (master, minion) also there should be a way to define mapping more granularly (e.g. etcd master on separate nodes).
2. After the change in pushed CI/CD job for rollout is triggered, Ansible starts Kubernetes deployment from CI/CD via SSH (the access from CI/CD to Kubernetes cluster using SSH has to be provided).

Updates

When new package is published (for example libssl) it should trigger a chain of jobs:

1. Build new container image (Etcd, Calico, Hyperkube, Docker etc)
2. Rebuild all images which depend on base
3. Run image specific tests
4. Deploy current production version on staging
5. Run verification
6. Deploy update on staging
7. Run verification
8. Send for promotion to production

5.5.3 Solution Overview

Current implementation considers two high-level groups of services - Master and Minion. Master nodes should run control-plane related services. Minion nodes should run user's workload. In the future, additional Network node might be added.

There are few additional requirements which should be addressed:

- Components placement should be flexible enough to install most of the services on different nodes, for example it may be required to install etcd cluster members to dedicated nodes.
- It should be possible to have a single-node installation, when all required services to run Kubernetes cluster can be placed on a single node. Using scale up mechanism it should be possible to make the cluster HA. It would reduce amount of resources required for development and testing of simple integration scenarios.

Common Components

- Calico is an SDN controller that provides pure L3 networking to Kubernetes cluster. It includes the following most important components that run on every node in the cluster.
  - Felix is an agent component of Calico, responsible for configuring and managing routing tables, network interfaces and filters on pariticipating hosts.
  - Bird is a lightweight BGP daemon that allows for exchange of addressing information between nodes of Calico network.
- Kubernetes
  - kube-dns provides discovery capabilities for Kubernetes Services.
  - kubelet is an agent service of Kubernetes. It is responsible for creating and managing Docker containers at the nodes of Kubernetes cluster.

Plugins for Kubernetes should be delivered within Kubernetes containers. The following plugins are required:

- CNI plugin for integration with Calico SDN.
- Volume plugins (e.g. Ceph, Cinder) for persistent storage.

Another option which may be considered in the future, is to deliver plugins in separate containers, but it would complicate rollout of containers, since requires to rollout containers in specific order to mount plugins directory.

Master Components

Master Components of Kubernetes control plane run on Master nodes. The proposed architecture includes 3 Master nodes with similar set of components running on every node.

In addition to Common, the following components run on Master nodes:

- etcd
- Kubernetes
  - Kubedns
  - Kube-proxy (iptables mode)
  - Kube-apiserver
  - Kube-scheduler
  - Kube-controller-manager

Each component runs on container. Some of them are running in static pods in Kubernetes. Others are running as docker containers under management of operating system (i.e. as `systemd` service). See details in Installation section below.

Minion Components

Everything from Common plus:

- etcd-proxy is a mode of operation of etcd which doesn't provide storage, but rather redirects requests to alive nodes in etcd clutser.

Optional Components

- Contrail SDN is an alternative to Calico in cases when L2 features required.
- Flannel is another alternative implementation of CNI plugin for Kubernetes. As Calico, it creates an L3 overlay network.
- Tools for debugging (see Troubleshooting below).

Component Versions

Component	Version
Kubernetes	1.4
Etcd	3.0.12
Calico	0.21-dev
Docker	1.12.3

5.5.4 Components Overview

Kubernetes

kube-apiserver

This server exposes Kubernetes API to internal and external clients.

The proposed architecture includes 3 API server pods running on 3 different nodes for redundancy and load distribution purposes. API servers run as static pods, defined by a kubelet manifest (`/etc/kubernetes/manifests/kube-apiserver.manifest`). This manifest is created and managed by the Kubernetes installer.

kube-scheduler

Scheduler service of Kubernetes cluster monitors API server for unallocated pods and automatically assigns every such pod to a node based on filters or 'predicates' and weights or 'priority functions'.

Scheduler runs as a single-container pod. Similarly to API server, it is a static pod, defined and managed by Kubernetes installer. Its manifest lives in `/etc/kubernetes/manifests/kube-scheduler.manifest`.

The proposed architecture suggests that 3 instances of scheduler run on 3 Master nodes. These instances are joined in a cluster whith elected leader that is active, and two warm stan-dy spares. When leader is lost for some reason, a re-election occurs and one of the spares becomes active leader.

The following parameters control election of leader and are set for scheduler:

- Leader election parameter for scheduler must be "true".
- Leader elect lease duration
- Leader elect renew deadline
- Leader elect retry period

kube-controller-manager

Controller manager executes a main loops of all entities (controllers) supported by Kubernetes API. It is similar to scheduler and API server in terms of configuration: it is a static pod defined and managed by Kubernetes installer via manifest file `/etc/kubernetes/manifests/kube-controller-manager.manifest`.

In the proposed architecture, 3 instances of controller manager run in the same clustered mode as schedulers, with 1 active leader and 2 stand-by spares.

The same set of parameters controls election of leader for controller manager as well:

- Leader election parameter for controller manager must be "true"
- Leader elect lease duration
- Leader elect renew deadline
- Leader elect retry period

kube-proxy

Kubernetes proxy forwards traffic to alive Kubernetes Pods. This is an internal component that exposes Services created via Kubernetes API inside the cluster. Some Ingress/Proxy server is required to expose services to outside of the cluster via globally routed virtual IP (see above).

The pod `kube-proxy` runs on every node in the cluster. It is a static pod defined by manifest file `/etc/kubernetes/manifests/kube-proxy.manifest`. It includes single container that runs `hyperkube` application in proxy mode.

kubedns

Kubernetes DNS schedules a DNS Pod and Service on the cluster, and configures the kubelets to tell individual containers to use the DNS Service's IP to resolve DNS names.

The DNS pod (`kubedns`) includes 3 containers:

- `kubedns` is a resolver that communicates to API server and controls DNS names resolving
- `dnsmasq` is a relay and cache provider
- `healthz` is a health check service

In the proposed architecture, `kubedns` pod is controller by ReplicationController with replica factor 1, which means that only one instance of the pod is working in a cluster at any time.

Etcd Cluster

Etcd is a distributed, consistent key-value store for shared configuration and service discovery, with a focus on being:

- Simple: well-defined, user-facing API (gRPC)
- Secure: automatic TLS with optional client cert authentication
- Fast: benchmarked 10,000 writes/sec
- Reliable: properly distributed using Raft

`etcd` is written in Go and uses the Raft consensus algorithm to manage a highly-available replicated log.

Every instance of `etcd` can operate in one of the two modes:

- full mode
- proxy mode

In *full mode*, the instance participates in Raft consensus and has persistent storage.

In *proxy mode*, `etcd` acts as a reverse proxy and forwards client requests to an active etcd cluster. The etcd proxy does not participate in the consensus replication of the etcd cluster, thus it neither increases the resilience nor decreases the write performance of the etcd cluster.

In proposed architecture, `etcd` runs as a static container under control of host operating system. See details below in Installation section. The assumed version of `etcd` in this proposal is `etcdv2`.

Etcd full daemon

There are three instances of `etcd` running in full mode on Master nodes in the proposed solution. This ensures the quorum in the cluster and resiliency of service.

Etcd native proxy

Etcd in proxy mode runs on every node in Kubernetes cluster, including Masters and Minions. It automatically forwards requests to active Etcd cluster members. According to the documentation it's recommended etcd cluster architecture.

Calico

Calico is an L3 overlay network provider for Kubernetes. It propagates internal addresses of containers via BGP to all minions and ensures connectivity between containers.

Calico uses etcd as a vessel for its configuraiton information. Separate etcd cluster is recommended for Calico instead of sharing one with Kubernetes.

calico-node

In the proposed architecture, Calico is integrated with Kubernetes as Common Network Interface (CNI) plugin.

The Calico container called `calico-node` runs on every node in Kubernetes cluster, including Masters and Minions. It is controlled by operating system directly as `systemd` service.

The `calico-node` container incorporates 3 main services of Calico:

- **Felix,** the primary Calico agent. It is responsible for programming routes and ACLs, and anything else required on the host, in order to provide the desired connectivity for the endpoints on that host.
- **BIRD** is a BGP client that distributes routing information.
- ***confd* is a dynamic configuration manager for BIRD, triggered** automatically by updates in the configuration data.

5.5.5 High Availability Architecture

Proxy server

Proxy server should forward traffic to alive backends, health checking mechanism has to be in place to stop forwarding traffic to unhealthy backends.

Nginx is used to implement Proxy service. It is deployed in a static pod, one pod per cluster. It provides access to K8s API endpoint on a single by redirecting requests to instances of kube-apiserver in a round-robin fashion. It exposes the endpoint both to external clients and internal clients (i.e. Kubernetes minions).

SSL termination

SSL termination can be optionally configured on Nginx server. From there, traffic to instances of kube-apiserver will go over internal K8s network.

Proxy Resiliency Alternatives

Since the Proxy Server is a single point of failure for Kubernetes API and exposed Services, it must run in highly available configuration. The following alternatives were considered for high availability solution:

1. Keepalived Although Keepalived has problems with split brain detection there is a subproject in Kubernetes which uses Keepalived with an attempt to implement VIP management.
2. OSPF Using OSPF routing protocol for resilient access and failover between Proxy Servers requires configuration of external routers consistently with internal OSPF configurations.
3. VIP managed by cluster management tools Etcd might serve as a cluster mangement tool for a Virtual IP address where Proxy Server is listening. It will allow to converge the technology stack of the whole solution.
4. DNS-based reservation Implementing DNS based High Availability is very problematic due to caching on client side. It also requires additional tools for fencing and failover of faulty Proxy Servers.

Resilient Kubernetes Configuration

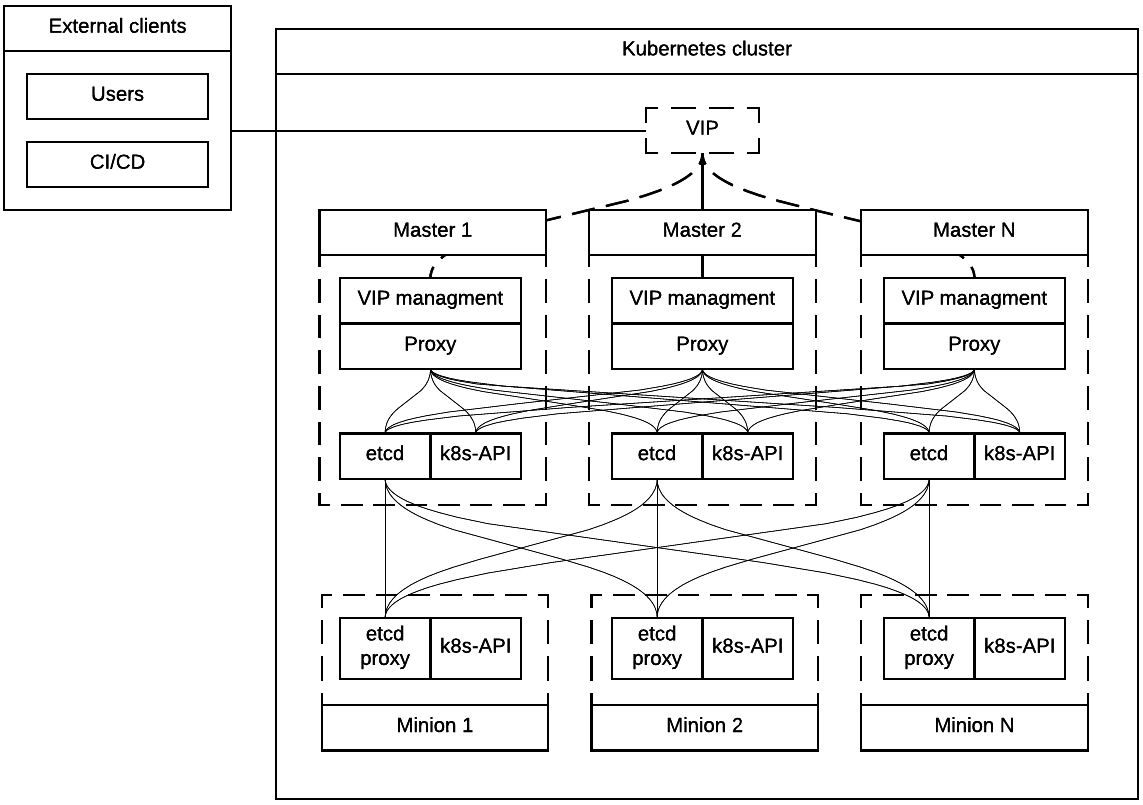
In the proposed architecture, there is a single static pod with Proxy Server running under control of Kubelet on every Minion node.

Each of the 3 Master nodes runs its own instance of `kube-apiserver` on localhost address. All services working on a Master node address the Kubernetes API locally. All services on Minion nodes connect to the API via local instance of Proxy Server.

Etcd daemons forming the cluster run on Master nodes. Every node in the cluster also runs etcd-proxy. This includes both Masters and Minions. Any service that requires access to etcd cluster talks to local instance of etcd-proxy to reach it. External access to etcd cluster is restricted.

Calico node container runs on every node in the cluster, including Masters and Minions.

The following diagram summarizes the proposed architecture.



Alternative approaches to the resiliency of Kubernetes cluster were considered, researched and summarized in *Appendix A. High Availability Alternatives*.

Next steps in development of this architecture include implementation of a Proxy server as an Ingress Controller. It will allow for closer integration with K8s in terms of pods mobility and life-cycle management operations. For example, Ingress Controller can be written to only relay incoming requests to updated nodes during rolling update. It also allows to manage virtual endpoint using native Kubernetes tools (see below).

5.5.6 Logging

Logs collection was made by Heka broker running at all nodes in the Kubernetes cluster. It used Docker logging in configuration when all logs are written to a volume. Heka reads files from the volume using Docker plugin and uploads them to ElasticSearch storage.

5.5.7 Installation

This section describes the installation of Kubernetes cluster on pre-provisioned nodes.

The following list shows containers that belong to Kubernetes Master Tier and run under control of systemd on Master and/or Minion nodes, along with a short explaination why it is necessary in every case:

- Etcd
  - Should have directory mounted from host system.
- Calico

  - Depending on network architecture it may be required to disable node-to-node mesh and configure route reflectors instead. This is especially recommended for large scale deployments (see below).
- Kubelet
  - Certificates directory should be mounted from host system in Read Only mode.

The following containers are defined as ReplicationController objects in Kubernetes API:

- kubedns

All other containers are started as static pods by Kubelet in 'kube-system' namespace of Kubernetes cluster. This includes:

- kube-apiserver
- kube-scheduler
- kube-controller-manager
- Proxy Server (nginx)
- dnsmasq

Note: An option to start all other services in Kubelet is being considered. There is a potential chicken-and-egg type issue that Kubelet requires CNI plugin to be configured prior its start, as a result when Calico pod started by Kubelet, it tries to perform a hook for a plugin and fails. Thi happens if a pod uses host networking as well. After several attempts it starts the container, but currently such cases are not handled explicitly.

Common practices

- Manifests for static Pods should be mounted (read only) from host system, it will simplify update and reconfiguration procedure.
- SSL certificates and any secrets should be mounted (read only) from host system, also they should have appropriate permissions.

Installation workflow

1. Ansible retrieves SSL certificates.
2. Ansible installs and configures docker.
   (a) Systemd config
   (b) Use external registry
3. All control-plane related Pods must be started in separate namespace `kube-system`. This will allow to restrict access to control plane pods in future.
4. Ansible generates manifests for static pods and writes them to `/etc/kubernetes/manifests` directory.
5. Ansible generates configuration files, systemd units and services for Etcd, Calico and Kubelet.
6. Ansible starts all systemd-based services listed above.
7. When Kubelet is started, it reads manifests and starts services defined as static pods (see above).
8. Run health-check.
9. This operations are repeated for every node in the cluster.

5.5.8 Scaling to 1000 Nodes

Scaling Kubernetes cluster to magnitude of 1000 nodes requires certain changes to confiugration and, in a few cases, the source code of components.

The following modifications were made to default configuration deployed by Kargo installer.

Proxy Server

Default configuration of parameter `proxy_timteout` in Nginx caused issues with long-polling "watch" requests from kube-proxy and kubelet to apiserver. Nginx by default terminates such sessions in 3 seconds. Once session is cut, Kubernetes client has to restore it, including repeat of SSL handshake, and at scale it generates high load on Kube API servers, about 2000% of CPU in given configuration.

This problem was solved by changing the default value (3s) to more appropriate value of 10m:

```
proxy_timeout: 10m
```

As a result, CPU usage of `kube-apiserver` processes dropped 10 times, to 100-200%.

The corresponding change was proposed into upstream Kargo.

kube-apiserver

The default rate limit of Kube API server proved to be too low for the scale of 1000 nodes. Long before the top load on the API server, it starts to return `429 Rate Limit Exceeded` HTTP code.

Rate limits were adjusted by passing new value to `kube-apiserver` with `--max-requests-inflight` command line option. While default value for this parameter is 400, it has to be adjusted to 2000 at the given scale to accommodate to actual rate of incoming requests.

kube-scheduler

Scheduling of so many pods with anti-affinity rules, as required by CCP architecture, puts `kube-scheduler` under high load. A few optimizations were made to its code to accommodate to the 1000 node scale.

- scheduling algorithm improved to reduce a number of expensive operations: pull request.
- cache eviction/miss bug in scheduler has to be fixed to improve handling of anti-affinity rules. It was worked around in Kubernetes, but root cause still requires effort to fix.

The active scheduler was placed to dedicated hardware node in order to cope with high load while scheduling large number of OpenStack control plane pods.

kubedns and dnsmaq

Default settings of resource limits for dnsmasq in Kargo don't fit for scale of 1000 nodes. The following settings must be adjusted to accommodate for that scale:

- `dns_replicas: 6`
- `dns_cpu_limit: 100m`
- `dns_memory_limit: 512Mi`
- `dns_cpu_requests 70m`

- `dns_memory_requests: 70Mi`

A number of instances of `kubedns` pod was increased to 6 to handle load generated by the cluster of the given size.

Following limits were tuned in `dnsmasq` configuration:

- number of parallel connections the daemon could handle was increased to 1000:

```
--dns-forward-max=1000
```

- size of cache was set to the highest possible value of 10000

Ansible

Several parameters in Ansible configuration have to be adjusted to improve its robustness in higher scale environments. This includes the following:

- `forks` for a number of parallel processes to spawn when communicating to remote hosts.
- `timeout` default SSH timeout on connection attepmts.
- `download_run_once` and `download_localhost` boolean parameters control how container images are being distributed to nodes.

Calico

In the tested architecture Calico was configured without route reflectors for BIRD BGP daemons. Therefore, Calico established a full mesh connections between all nodes in the cluster. This operation took significant time during node startup.

It is recommended to configure route reflectors for BGP daemons in all cases at scale of 1000 nodes. This will reduce the number of BGP connections across the cluster and improve startup time for nodes.

5.5.9 Lifecycle Management

Validation

Many LCM use-cases may cause destructive consequences for the cluster, we should cover such use-cases with static validation, because it's easy to make a mistake when user edits the configuration files.

Examples of such use-cases:

- Check that there are nodes with Master related services.
- Check that quorum for etcd cluster is satisfied.
- Check that scale down or node decommissioning does not cause data lose.

The validation checks should be implemented on CI/CD level, when new patch is published, a set of gates should be started, where validation logic is implemented, based on gates configuration they may or may not block the patch for promotion to staging or production.

Scale up

User assigns a role to a new node in configuration file, after changes are committed in the branch, CI/CD runs Ansible playbooks.

Master

1. Deploy additional master node.
2. Ensure that after new component is deployed, it's available via endpoints.

Minion

1. Deploy additional minion node.
2. Enable workload scheduling on new node.

Scale down

Scaledown can also be described as Node Deletion. During scaledown user should remove the node from configuration file, and add the node for decommissioning.

Master

1. Run Ansible against the cluster to make sure that the node being deleted is not present in any service's configuration.
2. Run node decommissioning.

Minion

1. Disable scheduling to the minion being deleted.
2. Move workloads away from the minion.
3. Run decommission of services managed by Ansible (see section *Installation*).
4. Run node decommissioning.

Test Plan

- Initial deploy

  Tests must verify that Kubernetes cluster has all required services and generally functional in terms of standard operations, e.g. add, remove a pod, service and other entities.

- Scaleup

  Verify that Master node and Minion node could be added to the cluster. The cluster must remain functional in terms defined above after the scaleup operation.

- Scaledown

  Verify that the cluster retains its functionality after removing Master or Minion node. This test set is subject to additional limitations to number of removed nodes since there is a absolute minimum or nodes required for Kubernetes cluster to function.

- Update

  Verify that updating single service or a set of thereof doesn't degrade functions of the cluster compared to its initial deploy state.

  - Intrusive
  - Non-intrusive

- Rollback

  Verify that restoring version of one or more components to previously working state after they were updated does not lead to degradation of functions of the cluster.

- Rollout abort

  Verify that if a Rollback operation is aborted, the cluster can be reverted to working state by resuming the operation.

5.5.10 Updating

Updating is one the most complex Lifecycle management use-cases, that is the reason it was decided to allocate dedicated section for that. We split updates use-cases into two groups. The first group "Non-intrusive", is the simplest one, update of services which do not cause workload downtime. The second "Intrusive", is more complicated since may cause updates downtime and has to involve a sequence of actions in order to move stateful workload to different node in the cluster.

Update procedure starts with publishing of new version of image in Docker repository. Then a service's metadata should be updated to new version by operator of the cloud in staging or production branch of configuration repository for Kubernetes cluster.

Non-intrusive

Non-intrusive type of update does not cause workload downtime, hence it does not require workload migration.

Master

Update of Master nodes with minimal downtime can be achieved if Kubernetes installed in HA mode, minimum 3 nodes.

Key points in updating Master related services:

- First action which has to be run prior to update is backup of Kubernetes related stateful services (in our case it is etcd).
- Update of services managed by Ansible is done by ensuring version of running docker image and updating it in systemd and related services.
- Update of services managed by Kubelet is done by ensuring of files with Pod description which contain specific version.
- Nodes has to be updated one-by-one, without restarting services on all nodes simultaneously.

Minion

Key points in updating Minion nodes, where workload is run:

- Prior to restarting Kubelet, Kubernetes has to be notified that Kubelet is under maintenance and its workload must not be rescheduled to different node.
- Update of Kubelet should be managed by Ansible.

- Update of services managed by Kubelet is done by ensuring of files with Pod description.

Intrusive

Intrusive update is an update which may cause workload downtime, separate update flow for such kind of updates has to be considered. In order to provide update with minimal downtime for the tenant we want to leverage VMs Live Migration capabilities. Migration requires to start maintenance procedure in the right order by butches of specific sizes.

Common

- Services managed by Ansible, are updated using Ansible playbooks which triggers pull of new version, and restart.
- If service is managed by Kubelet, Ansible only updates static manifest and Kubelet automatically updates services it manages

Master

Since master node does not have user workload update the key points for update are the same as for "Non-intrusive" use-cases.

Minion

User's workload is run on Minion nodes, in order to apply intrusive updates, rollout system has to move workload to a different node. On big clusters updates in butch-update will be required, to achieve faster rollout.

Key requirements for Kubernetes installer and orchestrator:

- Kubernetes installer is agnostic of which workloads run in Kubernetes cluster and in VMs on top of OpenStack which works as Kubernetes application.
- Kubernetes installer should receive rollout plan, where the order, and grouping of nodes, update pf which can be rolled out in parallel are defined. This update plan will be generated by different tool, which knows "something" about types of workload run on the cluster.
- In order to move workload to different node, installer has to trigger workload evacuation from the node.
  - Scheduling of new workload to the node should be disabled.
  - Node has to be considered as in maintenance mode, that unavailability of kubelet does not cause workload rescheduling.
  - Installer has to trigger workload evacuation in kubelet, kubelet should use hooks defined in Pods, to start workload migration.
- In rollout plan it should be possible to specify, when to fail rollout procedure.
  - If some percent of nodes failed to update.
  - There may be some critical for failure nodes, it's important to provide per node configuration, if it is important to stop rollout procedure if this node failed to be updated.

Limitations

Hyperkube

Current Kubernetes deliver mechanism relies on Hyperkube distribution. Hyperkube is a single binary file which contains all set of core Kubernetes components, e.g. API, Scheduler, Controller, etc. The problem with this approach is that bug-fix for API causes update of all core Kubernetes containers, even if API is installed on few controllers, new version has to be rolled out to all thousands of minions.

Possible solutions:

- For different roles rollout different versions of Hyperkube. This approach significantly complicates versions and fixes tracking process.
- Make split between those roles and create for them different images. The problem will remain since most of the core components are developed in a single repository and released together, hence it is still an issue, if release tag is published on the repo, rebuild of all core components will be required.

For now we go with native way of distribution until better solution is found.

Update Configuration

Update of configurations in most of the cases should not cause downtime.

- Update of Kubernetes and related services (calico, etcd, etc).
- Rotation of SSL certificates (e.g. those which are used for Kubelet authentication)

Abort Rollout

Despite the fact that this operation may be dangerous, user should be able to interrupt update procedure.

Rollback

Some of the operations are impossible to rollback, rollback may require to have different flow of actions to be executed on the cluster.

5.5.11 Troubleshooting

There should be a simple way to provide for a developer tooling for debugging and troubleshooting. These tools should not be installed on each machine by default, but there should be a simple way to get this tools installed on demand.

- Image with all tools required for debugging
- Container should be run in privileged mode with host networking.
- User can rollout this container to required nodes using Ansible.

Example of tools which may be required:

- Sysdig
- Tcpdump
- Strace/Ltrace
- Clients for etcd, calico etc

- ...

5.5.12 Open questions

- Networking node?

5.5.13 Related links

- Keepalived based VIP managament for Kuberentes
- HA endpoints for K8s in Kargo
- Large deployments in Kargo
- ECMP load balancing for external IPs

5.5.14 Contributors

- Evgeny Li
- Matthew Mosesohn
- Bogdan Dobrelya
- Jedrzej Nowak
- Vladimir Eremin
- Dmytriy Novakovskiy
- Michael Korolev
- Alexey Shtokolov
- Mike Scherbakov
- Vladimir Kuklin
- Sergii Golovatiuk
- Aleksander Didenko
- Ihor Dvoretskyi
- Oleg Gelbukh

5.5.15 Appendix A. High Availability Alternatives

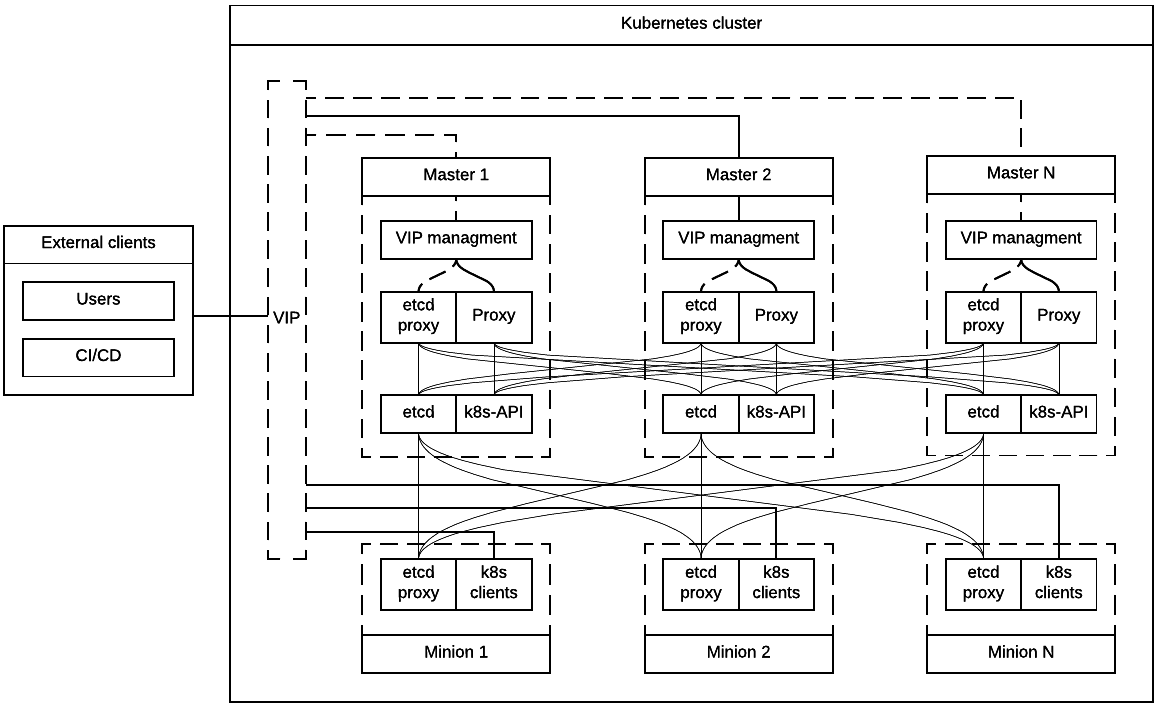
This section contains some High Availability options that were considered and researched, but deemed too complicated or too risky to implement in the first iteration of the project.

Option #1 VIP for external and internal with native etcd proxy

First approach to Highly Available Kubernetes with Kargo assumes using VIP for external and internal access to Kubernetes API, etcd proxy for internal access to etcd cluster.

- VIP for external and internal access to Kubernetes API.
- VIP for external access to etcd.

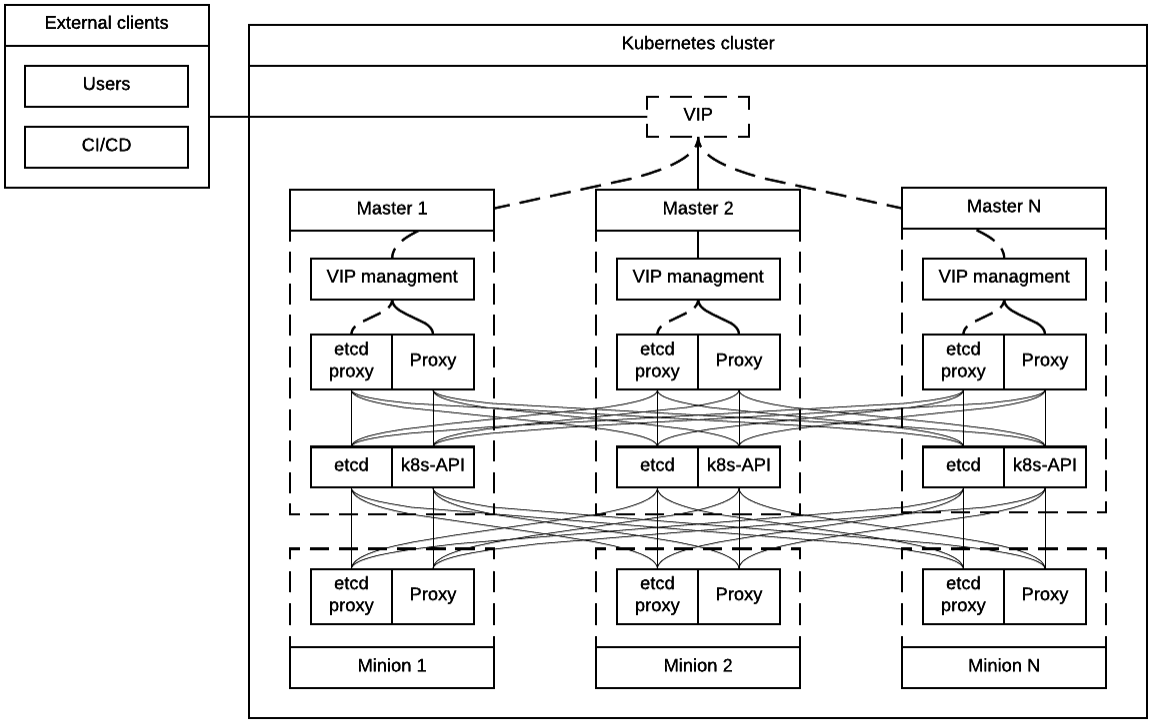
- Native etcd proxy on each node for internal access to etcd cluster.



Option #2 VIP for external and Proxy on each node for internal

The second considered option is each node that needs to access Kubernetes API also has Proxy Server installed. Each Proxy forwards traffic to alive Kubernetes API backends. External clients access Etcd and Kubernetes API using VIP.

- Internal access to APIs is done via proxies which are installed locally.
- External access is done via Virtual IP address.

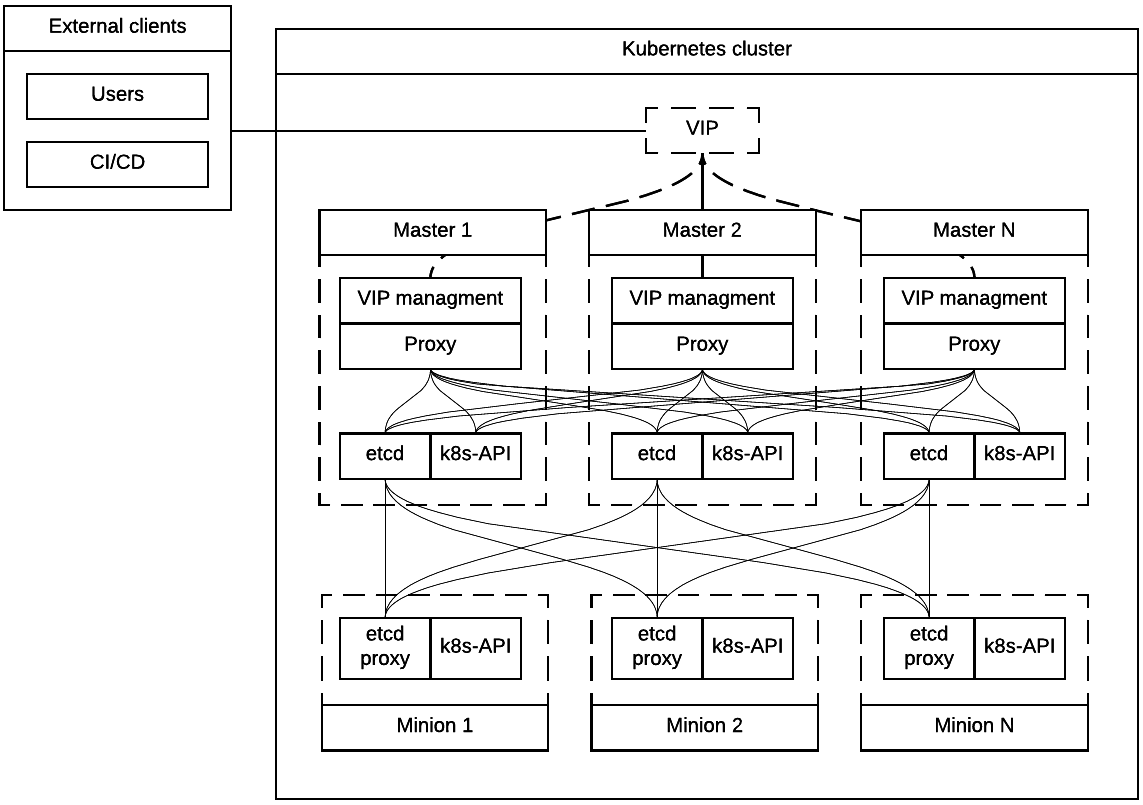


Option #3 VIP for external Kubernetes API on each node

Another similar to "VIP for external and Proxy on each node for internal" option, may be to install Kubernetes API on each node which requires access to it instead of installing Proxy which forwards the traffic to Kubernetes API on master nodes.

- VIP on top of proxies for external access.
- Etcd proxy on each node for internal services.
- Kubernetes API on each node, where access to Kubernetes is required.

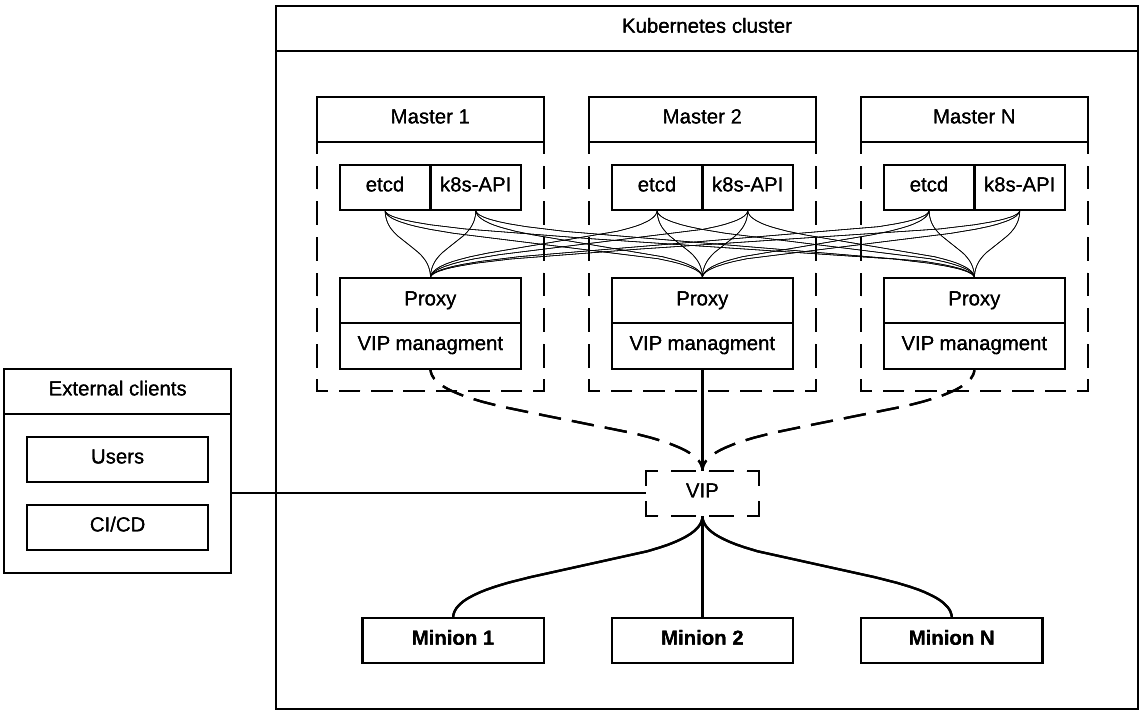
This option was selected despite potential limitations listed above.



Option #4 VIP for external and internal

In order to achieve High Availability of Kubernetes master proxy server on every master node can be used, each proxy is configured to forward traffic to all available backends in the cluster (e.g. etcd, kubernetes-api), also there has to be a mechanism to achieve High Availability between these proxies, it can be achieved by VIP managed by cluster management system (see "High Availability between proxies" section).

- Internal and External access to Etcd or Kubernetes cluster is done via Virtual IP address.
- Kubernetes API also access to Etcd using VIP.



Option #5 VIP for external native Kubernetes proxy for internal

We considered using native Kubernetes proxy for forwarding traffic between APIs. Kubernetes proxy cannot work without Kubernetes API, hence it should be installed on each node, where Kubernetes proxy is installed. If Kubernetes API is installed on each node, there is no reason to use Kubernetes proxy to forward traffic with it, internal client can access the Kubernetes API through localhost.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search