# Kube-on-OS

by David Visscher

> The following is an extract from the wiki of the kube-on-openstack repository. Some diagrams and section layouts will not appear very nicely. This document is better when viewed live at: https://gitlab.com/ecida/kube-on-openstack

This wiki documents the kube-on-os deployment environment.

### Index


#### Home

#### Getting Started

#### Architecture

- Deployment Architecture
- Network Architecture
- CI

#### Terraform

- Deployment Parameters
- Initial Provisioning
- Full Resource Graph

#### Salt

- State Reference
- The Microk8s Module
- The Pillar
- The Mine

#### Notable Limitations

#### Future Opportunities


# Getting Started

This section will guide you through setting up Kube-on-Openstack from scratch.

# Prerequisites

- An Openstack project with sufficient quota to run the workload needed. Notably, make sure that enough floating IP's are available.
- An Ubuntu 20.04 Focal image should be available in the Openstack cluster. These are usually provided by the Openstack cluster's maintainer. Such images can also be acquired from https://cloud-images.ubuntu.com/focal/current/
- One of either:
  - *If running in Gitlab CI*:
    * Set up Terraform state backend for Gitlab CI. Other than that, the provided `.gitlab-ci.yaml` should work for you.
    * Access to the Openstack API from the Gitlab runner you're using. When using public runners, that usually means the API needs to be accessible from the internet.
  - *If running locally*:
    * A local clone of the repository
    * A local installation of terraform
    * Access to an Openstack cluster from your workstation.
    * If you want to keep state locally, make sure to comment out the lines indicating the `http` backend in `terraform/terraform.tf`.

# Setting Environment Variables

Configuration of Kubernetes-on-Openstack happens primary by setting the variables for Terraform to use. Here's an example of how to set these examples:

## Connecting to Openstack

For connecting to Openstack we use the default environment variables that are provided via an openrc file. Such a file can be downloaded via the "API Access" page on the dashboard. The file can simply be sourced using the same method as when using the Openstack CLI.

Example:

```
$ source openrc.sh
```

As an alternative, an Application Credential can be created. This is usually the more secure method, especially when the credentials have to live outside your own workstation.

Terraform will read the environment variables set by the openstack rc file, and use those to connect to the API.

## Variables that must be set

While most variables have (sane) defaults set, we do need to specify some of them to fit our specific case. We can do this in multiple ways, in this example we'll use environment variables, as those will work in Gitlab and locally.

We will override the defaults that are defined in `terraform/terraform.tf` using environment variables. Doing this every time you open your shell can be very tedious, so writing these to a file you can later source is recommended.

Make sure to check that you've enough available quota in the openstack project for the default instances. If Not, override `kube_node_flavor` to select a smaller instance size or lower `num_kube_nodes`.

For all possible variables that can be set, see Deployment Parameters.

### The external network

Our cluster needs to be connected to the internet somehow. Openstack has provider networks for this. We can connect routers to them and allocate floating ips.

```
export TF_VAR_external_network_name=[...]  # FIll in your provider network's name here
```

### The base image

The default value here is set to the ubuntu focal image's ID in merlin cloud. If we're using a different image or a different cluster the ID will also differ, so we need to set the corresponding variable.

```
export TF_VAR_base_image=[SOME ID]  # FIll in the id of your image here
```

### Users

We want to be able to log into our cluster once its deployed. To do so, we set the variable for it. This needs to be set as a terraform map.

Here's an example for defining two users:

```
export TF_VAR_users='[{ username = "waldorf", ssh_public_key = "[...]"},{ username = "statle
```

### A root password

Just for debugging it's very useful to be able to be able to log in as root via the console. So, especially for this first deploy, we'll set a root password.

For this we use the following two variables:

```
export TF_VAR_set_root_password="true"
export TF_VAR_root_password="VerySecurePassword"
```

For the rest we'll just accept the defaults. These are documented at Deployment Parameters, and can be found in the code in the file `terraform/terraform.tf`.

# Running the deployment

Now that we've set up our environment, we can start deploying. If you're using gitlab CI, the pipeline will start running and you can manually trigger the deploy step if a pipeline succeeds.

## Deploying from the local machine

To do so, follow these steps from the root of the local git clone:

```
cd terraform
terraform plan  # This will show what terraform will do.
terraform apply  # If you're happy with the plan.
```

Kube-on-OS should now start deploying, and make your cluster ready for you. The standard network layout it deploys can be found at Network Architecture

The rest of the process is fully automated.

# Further Reading: Useful pages to understand what's happening

- To learn about how the automated deployment process works, see Deployment Architecture and CI.
- To learn about what salt configures, see the State Reference
- To learn about how salt and microk8s interact, see The Microk8s Module
- To learn about how salt is parameterized to adapt to different deployment environments, see The Pillar.

# Project Architecture

This section describes the architecture for the kube-on-os deployment.

For information on how the deployment as a whole is designed, see Deployment Architecture

For information about the network, and how it is designed, see Network Architecture

For information on continuous integration and corresponding pipelines, see CI

# Deployment Architecture

## Deployment steps:

In broad terms, the steps of the deployment are as follows:

```
flowchart TD
    subgraph Terraform
        create_net[Create Network Layout]
        deploy_bastion[Deploy Bastion Machine]
        deploy_salt[Deploy Salt Master]
        deploy_others[Deploy other instances]

        create_net -- when network configuration is done --> deploy_bastion
        deploy_bastion -- when bastion is ready to forward connections --> deploy_salt
        deploy_salt -- after salt master is configured --> deploy_others
    end

    subgraph Salt
        highstate_salt[Ensure salt master is in desired state]
        highstate_others[Ensure other machines are in desired state one-at-a-time]

        highstate_salt --> highstate_others
    end

    subgraph Microk8s
        init[Initialise local node]
        enable_addons[Ensure addons are enabled or disabled as desired]
        subgraph connect_ha
            generate_token[K8s Master: Generate token for HA connection]
            mine_publish[K8s master: Publish token to salt mine]
            join_ha[Others: join cluster using token]

            generate_token --> mine_publish
            mine_publish --> join_ha
        end
    end

    deploy_others -- apply desired state --> Salt

    highstate_others -- as part of highstate --> init
    highstate_others -- as part of highstate --> enable_addons
    highstate_others -- as part of highstate --> connect_ha
```

### Terraform

Terraform creates and manages the resources in Openstack, and does the necce-

sary basic provisioning such that salt can take over. These resources are things like networks, subnets, ports, routers, machines etc.

The provisioning in this stage (via cloud-init and ssh) is considered as a one-off, and thus not persistent.

More on how terraform is employed can be found here

### Salt

Salt then does the full configuration of the machines based on their roles and the parameters passed via the pillar. This configuration is persistent and the final result should be fully configured machines, as specified in the state tree.

Read more about how salt works here

### Microk8s

Microk8s is used to deploy and manage the Kubernetes cluster running on the machines that salt has configured. It manages the kubernetes services and which features should be enabled for the cluster.

This happens via the microk8s module which was written for salt. The module allows salt to talk to microk8s, and as such pass along what it wants the Kubernetes cluster to look like.

## Network Architecture

This section describes how the network was designed for kube-on-openstack. In the diagram below you can see the layout of the network:

```
flowchart TD
  internet{{Internet}}
  rtr((Router))
  pubnet[[Public-facing network \n 10.0.0.0/16 ]]
  prvnet[[Private network \n 10.1.0.0/16 ]]
  salt(Salt Master \n Salt States:\n core, salt-master \n IP: 10.1.0.5)
  bastion(Bastion Machine \n Salt States:\n core, bastion\n IP: 10.0.0.10/10.1.0.1)
  k8smaster(Kubernetes Master \n Salt States:\n core, microk8s, microk8s.master\n IP: 10.0.
  k8sworker0(Kubernetes Worker 0 \n Salt States:\n core, microk8s\n IP 10.1.0.100)
  k8sworker1(Kubernetes Worker 1 \n Salt States:\n core, microk8s\n IP 10.1.0.101)
  k8sworker2(Kubernetes Worker 2 \n Salt States:\n core, microk8s\n IP 10.1.0.102)
  k8sworkerN(Kubernetes Worker n \n Salt States:\n core, microk8s\n IP 10.1.0.100 +n )

  internet --- rtr
  rtr      --- pubnet
  pubnet   --- bastion   --- prvnet
  pubnet   --- k8smaster --- prvnet
```

```
prvnet   --- salt
prvnet   --- k8sworker0
prvnet   --- k8sworker1
prvnet   --- k8sworker2
prvnet   --- k8sworkerN
```

### Components

| Component | Description |
| --- | --- |
| Router | This is the main router that provides access the to the network. This is also where floating IPs get translated to internally used IPs. |
| Public-Facing Network | Any machine with a port in this network can directly access the internet via the router. Inbound traffic is only possible with a floating IP assigned to a port in this network. |
| Bastion | This machines serves as a bridge between the inner and outer network layers. Administrators can connect to this machine from outside, and can then hop through to the private network. The machines also performs NAT for the private network machines that need to retrieve information, like updates, from the internet. |
| Kubernetes Master | This is the primary master node for the kubernetes cluster. It is connected to the public network so it can perform ingress for the cluster. |
| Kubernetes Worker | Worker node for the kubernetes cluster. Can also be part of the control plane for HA purposes. |
| Salt Master | Central node for configuration management using salt |

## Terraform

This chapter covers how terraform is used to deploy the environment.

It contains the following sections: - Deployment Parameters - Initial Provisioning - Full Resource Graph

Initial provisioning is performed by terraform via cloud-init.

The operations performed by cloud-init are documented here (in case you want to do it yourself, or are just curious). The cloud-init files are stored in `terraform/cloud-init` and are templated by terraform.

It is worth noting that all the settings we do here are assumed not to be persistent. Most of these settings here are repeated or overwritten by the core salt state. **If you want to do something that lasts, do it via salt**.

# For the salt master

(`terraform/cloud-init/salt.tftpl`)

In order to get the machine ready for our salt master, the following things are set: - Set the timezone. - Set its hostname, both the short one, and the FQDN - Change the root password (if requested) - Set up the disks, as the salt master has two disks by default: * One ephemeral root disk. This requires no further configuration as it is handled by Openstack. This disk gets deleted each time the salt master is recreated by Terraform. * A disk to contain salt's stateful components. This ensures continuity between rollouts. If we hadn't had this disk, we'd need to refresh the trust for all keys every time we replace the salt master.

```
By default, this disk is contains two partitions: one for `/srv/salt` covering 80% of the
```

- Write files needed to get up-and-running:
  - `/etc/resolv.conf` to get DNS working
  - `/tmp/install_script.sh` to contain the script for installing salt and its dependencies.
  - `/tmp/wait-for-it.sh`, a useful script that lets us wait for a certain connection to exits. Can be found here.3
  - `/etc/salt/minion_id`, we set the minion_id to be equal to the fqdn. This makes life easier, as it avoids any naming confusion in larger environments.
  - `/etc/hosts`, so the machine knows its own name
- Run a full system update (so we know we're current)
- Print a helpful recognizable message to the logs so we know when cloud-init has finished.

# For salt minions

(`terraform/cloud-init/salt-minion.tftpl`)

In order to get the machine ready to be a salt minion, the following things are set: - Set the timezone. - Set its hostname, both the short one, and the FQDN - Change the root password (if requested) - Write files needed to get up-and-running: * `/etc/resolv.conf` to get DNS working * `/tmp/install_script.sh` to contain the script for installing salt and its dependencies. * `/etc/salt/minion_id`, we

set the minion_id to be equal to the fqdn. This makes life easier, as it avoids any naming confusion in larger environments. * `/etc/hosts`, so the machine knows its own name and that of the salt master. - Run a full system update (so we know we're current) - Print a helpful recognizable message to the logs so we know when cloud-init has finished.

## For the bastion

(`terraform/cloud-init/bastion.tftpl`)

In order to get the bastion ready, the following things are set: - Set the timezone. - Set its hostname, both the short one, and the FQDN - Change the root password (if requested) - Write files needed to get up-and-running: * `/etc/resolv.conf` to get DNS working * `/tmp/install_script.sh` to contain the script for installing salt and its dependencies. * `/etc/salt/minion_id`, we set the minion_id to be equal to the fqdn. This makes life easier, as it avoids any naming confusion in larger environments. * `/etc/sysctl.conf`, to allow ipv4 forwarding * `/etc/hosts`, so the machine knows its own name and that of the salt master. - Run a full system update (so we know we're current) - Configure the network routes using netplan, so the machine knows where to find certain hosts - Run commands to configure IPv4 forwarding and masquerading. - Print a helpful recognizable message to the logs so we know when cloud-init has finished.

This page lists and explains all the relevant deployment parameters for terraform. These can all be found in the `terraform.tf` file.

The variables listed have all have sane defaults set to work with Merlin. This is to minimize the required setup for that environment. However, if you're running in a different Openstack cluster, then you may need to override these variables.

See the following pages on methods for overriding these default variables: - https://www.terraform.io/language/values/variables#variables-on-the-command-line - https://www.terraform.io/language/values/variables#variable-definitions-tfvars-files - https://www.terraform.io/language/values/variables#environment-variables

For running in gitlab CI, using environment variables is recommended. These can be set in `Settings -> CI/CD -> Variables`.

#### num_kube_nodes

*default:* `3`

This variable determines the number of nodes should be spawned in the kubernetes cluster. This number includes the master. So a value of 3 will create 1 master + 2 workers.

### kube_node_flavor

*default:* `m1.large`

This variable determines which openstack flavor to use for the kubernetes nodes. Make sure that the combination of flavor and the number of kube nodes does not exceed your project quota. A list of available flavors can be retrieved using the `openstack flavor list` command, with an openrc file activated.

### external_network_name

*default:* `vlan1066`

This variable determines which external network the cluster is to be connected to. The main router for the cluster is attached to this network, and this network is expected to provide the required floating ip's. Please make sure at least 3 are available.

### base_image

*default:* `ad156007-77bb-4e32-ac97-7f8340cab73d` *(this corresponds to the Ubuntu focal cloud image as it exists in Merlin)*

This variable determines which image to use when spawning instances in openstack. Kube-on-OS is designed to work on top of Ubuntu 20.04 focal.

When running in a different openstack cluster than Merlin: Upload an ubuntu 20.04 cloud image, if there isn't one available, and place its ID in this variable.

### deploy_domain_name

*default:* `kube-on-os`

This determines which domain to append to the machine names. As such, the salt master will be named `salt.kube-on-os`

Example: setting this to `example.local` will cause any machines to have an fqdn ending with that domain.

### deploy_user

*default:* `ubuntu`

The default user to use when provisioning. Must be changed when using a different OS with a different default username.

### set_root_password

*default:* `false`

If set to true, will set the root password to whatever's in the root_password variable. If set to false, no root password will be configured by terraform.

Only use this for debugging, access normally happens by logging in with your own user. SSH root login is always disabled, this is for console use only.

#### root_password

*default: none*

Will set all machines' root passwords to be set to this value, when `set_root_password` is set to `true`.

#### kubernetes_formula_url

Deprecated since migration to microk8s.

#### kubernetes_formula_version

Deprecated since migration to microk8s.

#### users

*default: none*

Contains a map of users that should be configured to have SSH access. Terraform passes this information to salt via the pillar. So these users are created at a later of the rollout process, via the core state.

Example value:

```
[
    { username = "waldorf", ssh_public_key = "[...]"},
    { username = "statler", ssh_public_key = "[...]"}
]
```

Below you can find the full resource dependency graph of all terraform resources. (it's probably easiest to view when downloaded and opened in an image viewer)

## State Tree

This section lists all salt states in the tree and their purpose.

## Core

The core state is applied to all machines in order to create a common base to work from for other states. It ensures the OS is configured correctly, providing a solid foundation for other states.

It contains several submodules, all linked in via `salt/salt/core/init.sls`.

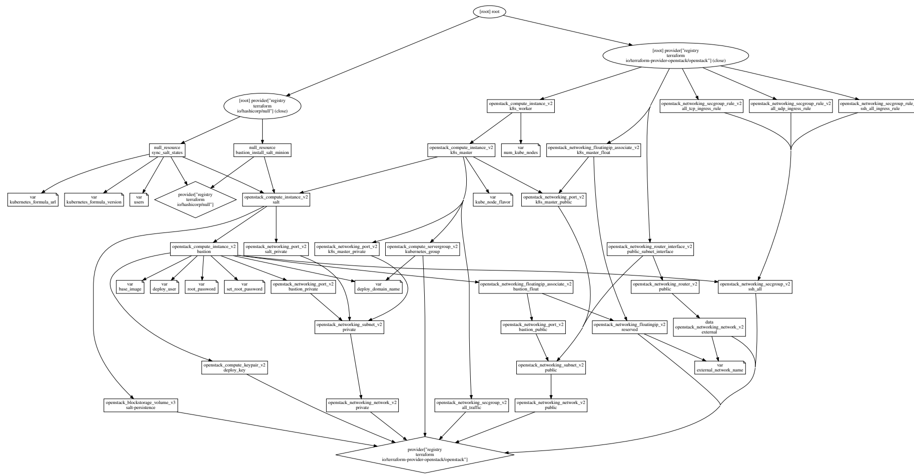


Figure 1: tfgraph.svg

## Core.Sysctl

The sysctl state manages system configuration flags, usually set in `/etc/sysctl.conf`. It sets the following things: - It increases the maximum number of open files - Disables IPv6 networking (as openstack doesn't support it anyway)

## Core.Locale

This sets the system locale to `C.UTF-8`.

## Core.MOTD

This state sets the message of the day for when users log in. It is split into the issue and the full motd, one for before login, and one after. This way, no information is leaked, but a message can still be shown that unauthorised access is prohibited.

This state also disables Ubuntu's default motd generator, as we're setting our own MOTD.

Templates for the `/etc/motd`, and `/etc/issue` files live in the `salt/salt/core/files` directory.

The state assumes that other states will later configure other services like `sshd` to actually use these files. It only manages the default files for the MOTD.

### Core.SSHD

This state ensures that the `/etc/ssh/sshd_config` file is in the state according to the template in the `salt/salt/core/files` directory. It also ensures the `sshd` service is running, and is reloaded whenever a configfile changes.

### Core.Resolv

The resolv state ensures the correct nameservers are set in the `/etc/resolv.conf` to make sure DNS is predictable. By default, we use the following nameservers: `1.1.1.1`, `1.0.0.1`.

### Core.Hosts

A python-based state that manages the hostsfile of every minion. It ensures that each minion's hostname is known to every other minion. The salt mine is used for this.

### Core.Packages

This state ensures that the system is up-to-date and certain core packages are installed.

### Core.Minion

This state ensures the salt-minion is running

### Core.Users

This state creates users on the minion based on the `users` pillar value.

The value for this is usually inherited from the terraform variable of the same name. Terraform sets this pillar value as part of the salt master config.

## File Transfer

Allows arbitrary files to be transferred to minions via the salt file server, configured via the pillar. Currently unused.

## Bastion

This state configures the required services for the bastion machine. It contains a few submodules:

### Bastion.Sysctl

Sets `net.ipv4.ip_forward` and `net.ipv4.conf.all.forwarding` sysconfig flags to allow ip forwarding.

### Bastion.IPTables

Ensures iptables is installed, along with iptables-persistent. It also sets the rules and loads them into iptables.

Changing the order of the network interfaces in openstack/terraform will break this state.

### Bastion.Fail2ban

Ensures fail2ban is installed and configured on the machine.

As this machine is addressable from the internet, we want to block repeated auth attempts.

## Kubernetes-config

*Deprecated by the microk8s state*

## Microk8s

Uses the custom microk8s module to set the desired state for microk8s on the machine.

The base state always does the following: - Make sure the microk8s snap is installed - Make sure microk8s is running - Add an extra IP to the CSR template to allow the using a floating IP. - Make sure the certificates are regenerated when the certificates are regenerated. - Enable the following addons: - dns - hostpath_storage - helm3 - ha_cluster - prometheus - Disable the following addons (in case they were activated previously): - registry

This state also has two submodules: - `joined`, for connecting nodes to a cluster - `master`, for master-specific configuration

## Salt_Master

This state configures to salt master to ensure the service is running and the correct file_roots are set.

# The Microk8s Module

This section documents the design of the custom microk8s module for salt.

All paths denoted here are relative to the repository root. Bear in mind that on the salt master, `salt/salt` is mapped to `/srv/salt`.

# Architecture

The microk8s module follows the standard architecture for salt modules.

```
flowchart TD
  state[The state at\n salt/salt/microk8s]
  state_module[State Module at \n salt/salt/_states/salt]
  execution_module[Execution Module at \n salt/salt/_modules/salt]

  state -- uses to declare desired configuration for our specific situation --> state_module
  state_module -- uses to gather information and perform operations  --> execution_module
```

It consists of three parts: - **An execution module**, which contains the atomic operations that can be performed and gather information. Can be found at `salt/salt/_modules/microk8s`. - **A state module**, which uses the execution module to attain a certain desired end-state. Can be found at `salt/salt/_states/microk8s`. - **A state in the state tree** at `salt/salt` which uses the state module to perform the specific configuration we want. We use the microk8s state here just like any other standard state module salt provides. The documentation for this lives with the other states in the tree.

# The Execution Module

This python module provides atomic operations for information-gathering and manipulation of microk8s. It can be found at `salt/salt/_modules/microk8s`, or at `/srv/salt/microk8s` on the salt-master.

It provides the following (sub)modules: - **status**, which provides commands for gathering information on the running cluster. - **addons**, which allows for gathering information on the status of addons. It also provides functions for enabling/disabling addons. - **constants**, which provides some module-wide constants. - **crosscall**, which provides entrypoints for calling other modules outside this one. - **ha**, for joining/leaving clusters and issuing tokens used to join.

For information on specific functions in these (sub)module look in the source files. Each function is as small as possible and has extensive doc-strings.

# The State Module

This python module uses the execution module to attain a certain state on the minion. It lives at `salt/salt/_states/microk8s`, or `/srv/salt/_states/microk8s` on the salt-master.

It exposes the following states for the user: - **microk8s.running**, which ensures that microk8s is running on the minion. - **microk8s.addon_enabled**, which ensures that a certain module is enable on the minion. - **microk8s.addon_disabled**, the mirror image of the previous. - **microk8s.cluster_joined**, which ensures the minion is joined to a certain cluster.

# Distribution

Salt automatically synchronizes the modules to all the minions whenever required. It can be done manually by running the following command as root on the salt master `salt * saltutil.sync_all`. In our case, terraform does this manually, just so the synchronization is logged.Kube-on-Openstack uses the salt mine to exchange information between master and minion that (potentially) periodically changes.

Mine functions are enabled via the pillar.

It is used for the following things in our case: - **Tracking the IP address of all minions.** This is so we can use it in the `core/hosts` state. An easy way to let minions know each other's names without configuring a DNS server. - **Publising short-lasting microk8s tokens.** Minions need these tokens to be able to join a cluster. The pillar is the way salt parameterizes states. It stores and exchanges data with minions securely.

All the yaml files in the pillar get flattened to a single python dict, which is then accessible to the minions. The `top.sls` file in the pillar directory determines which minions get to see which data.

# Viewing the pillar

Each minion has its own view on the pillar, which can be retrieved with the `pillar.items` function.

Example (on any minion as root):

```
root@bastion:~# salt-call pillar.items
local:
    ----------
    microk8s:
        ----------
        allowed_csr_ip:
```

```
            194.171.203.19
        master:
            k8s_master.test.kube-on-os.ecida.org
    mine_functions:
        ----------
        network.ip_addrs:
            ----------
            cidr:
                10.1.0.0/16
    network:
        ----------
        external_gateway:
            10.0.0.1
        external_interface:
            ens3
[...and so on...]
        ----------
```

## How the pillar is configured

Our pillar setup is a bit more complicated that the standard way, but not enormously so. It is set up in such a way, that allows for a little more flexibility while keeping the file tree simple.

### How the pillar is structured

The top file looks for the following files (in order from general to specific): 1. `_from_terraform.sls` 2. `environments/defaults.sls` 3. `environments/{domainname}.sls` 4. `minions/defaults/{hostname}.sls` 5. `minions/{domainname}/{hostname}.sls`

The pillar will go through these files in this order to construct the final dictionary. Later files override earlier files.

Every minion's id in our setup corresponds to its fully qualified domain name. This gets split into two parts at the first `.`: the `hostname` and the `domainname`.

For example `box.example.com` would get split into `box` and `example.com`.

The way this ordering of files is designed, we could define pillar values as globally or locally as we wanted: - Setting a value in `environments/defaults.sls` would set it for every minion - Setting a value in `environments/ecida.org.sls` would set it for all minions in that domain, overriding the default. - Setting a value in `environments/defaults/box.sls` would set it for all minions with the name `box` in any domain, overriding the default and the domain. - Setting a value in `environments/ecida.org/box.sls` would set it just for a minion called `box` in the `ecida.org` domain. This overrides all others.

If a file in this list doesn't exists, its absence is ignored. (salt doesn't do this by default)

This allows for maximum flexibility for configuration, while still being able to keep everything in the same repo and branch.

# Future Improvements

This section documents the ways Kube-on-Openstack could be improved in the future.

# Master-of-Masters

Right now, multiple environments are managed entirely separately. To reduce future operational load when managing multiple environments, a master-of-masters could be added using salt-syndic.

This would allow for greater redundancy and less inconsistency between clusters

# External loadbalancing with Octavia

Using Openstack's Octavia module, the successor of neutron_lbaasv2, we could add external loadbalancing to the kubernetes clusters. This would solve any connection bottlenecks.

This module is however somewhat complex to configure initially.