Making OpenStack Production Ready with Kubernetes and OpenStack-Salt – Part 1

Jakub Pavlik, Lachlan Evenson - June 27, 2016 -

This document introduces and explains how to build workflow for lifecycle management and operation of an enterprise OpenStack private cloud coupled with OpenContrail SDN running in Docker containers and Kubernetes.

The following blog post is divided into five parts, the first being an explanation of a deployment journey into a continuous DevOps workflow. Secondly, steps on how to build and integrate containers with your build pipeline are listed. The third part details the orchestration of containers with a walkthrough of Kubernetes architecture including plugins and prepping OpenStack for decomposition. As the fourth step we introduce the tcp cloud Theory of a "Single Source of Truth" solution for central orchestration. In the fifth and final step we bring it all together, demonstrating a deploy and upgrade of OpenStack with OpenContrail.

We decided to divide text into two blogs for better reading. This is the first blog, which covers part 1 – 2 related mostly to journey and containers build.

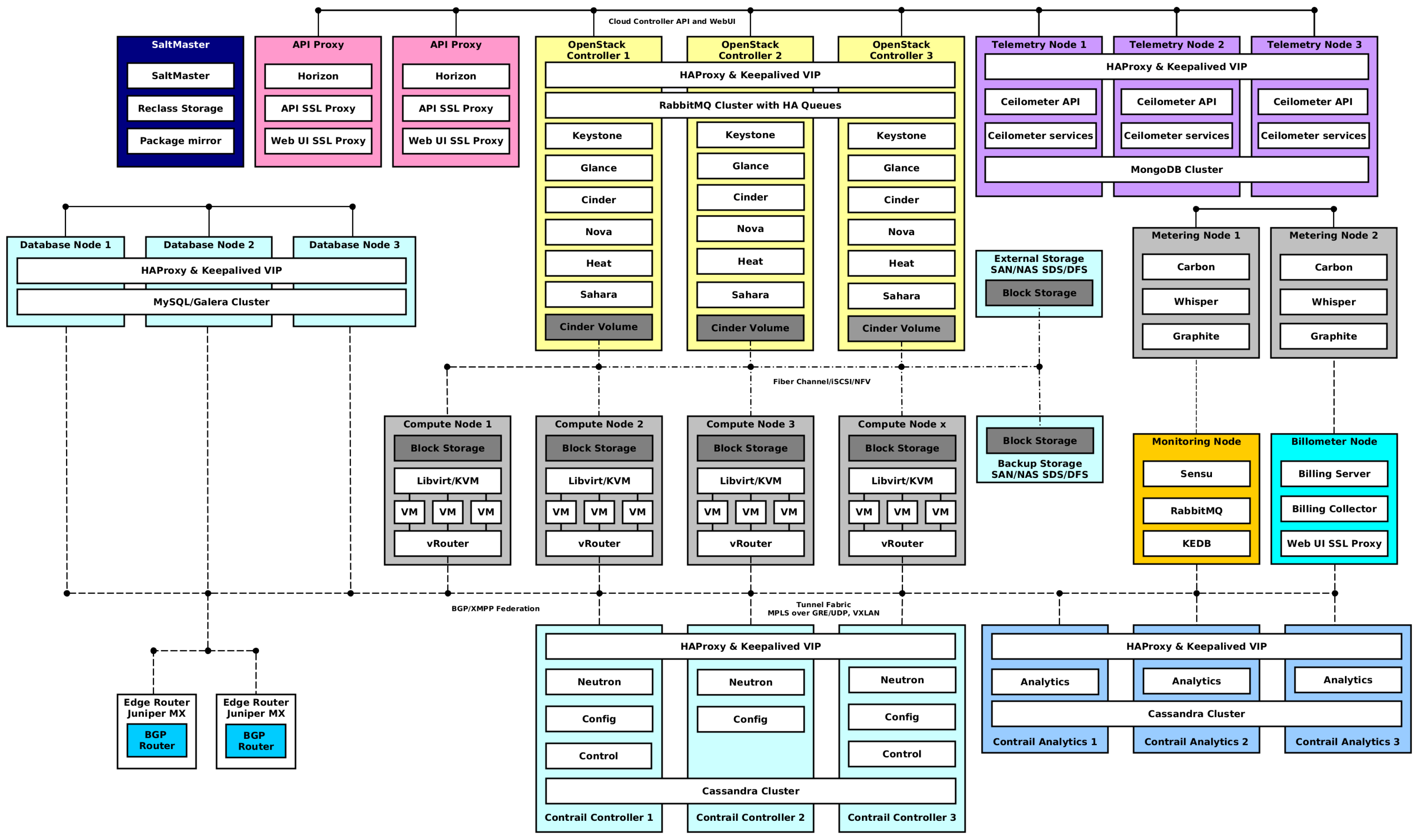
OpenStack Deployment Evolution

At first glance you might ask why would I add additional tooling on top of my existing deployment tools? It is therefore important is to explain why anyone should consider Docker and/or Kubernetes as tools for running OpenStack in production. At tcp cloud, we have deployed and operate a growing number of enterprise private cloud in production on OpenStack. Each OpenStack deployment is different (storage, network, modules) and any number of service combinations that exist given the needs of the customer. There is one thing all cloud deployments have in common however – deployment phases and initial goals. These have become evident and all customers journeys lead to an individual cloud maturity model. Lets divide these phases of evolution into following three sections.

RagTag

Characterized as untidy, disorganized, inharmonious . This is always the first step and sometimes the last for anybody starting with OpenStack. Every company or individual that is considering OpenStack as a private cloud solution has the common first goal of – deploy OpenStack!

This journey typically starts on openstack.org and lands on deployment tools like Puppet, Chef, Ansible, TripleO, Helion, Fuel, etc. It is almost impossible for anybody to identify right way to get OpenStack up and running without any experience. Even though all of them promise simple and quick setup if the whole environment, everybody will probably finish with the following logical topology of production environment. This diagram shows a typical production service oriented architecture of a single region OpenStack Deployment in High Availability. As you can see, this diagram is very complex.



Next thing which you find during this time is that current deployment tools cannot cover a complete production environment (bonding, storage setups, service separation, etc.). This is when the cloud team starts to ask themselves – do we need to really to setup an environment in one day, deploy in 5 minutes on a single machine or through a nice clickable gui? Are these really the key decision points which determine the right choice of our future production environment? Standing up a stack is easy and deployment tools are one shot! You cannot run them twice or are they repeatable with patterns! What about lifecycle like patching, upgrades, configuration changes, etc.
This brings us back to statement that nobody can choose right solution without experience of “Day 2 Operations”.

Ops

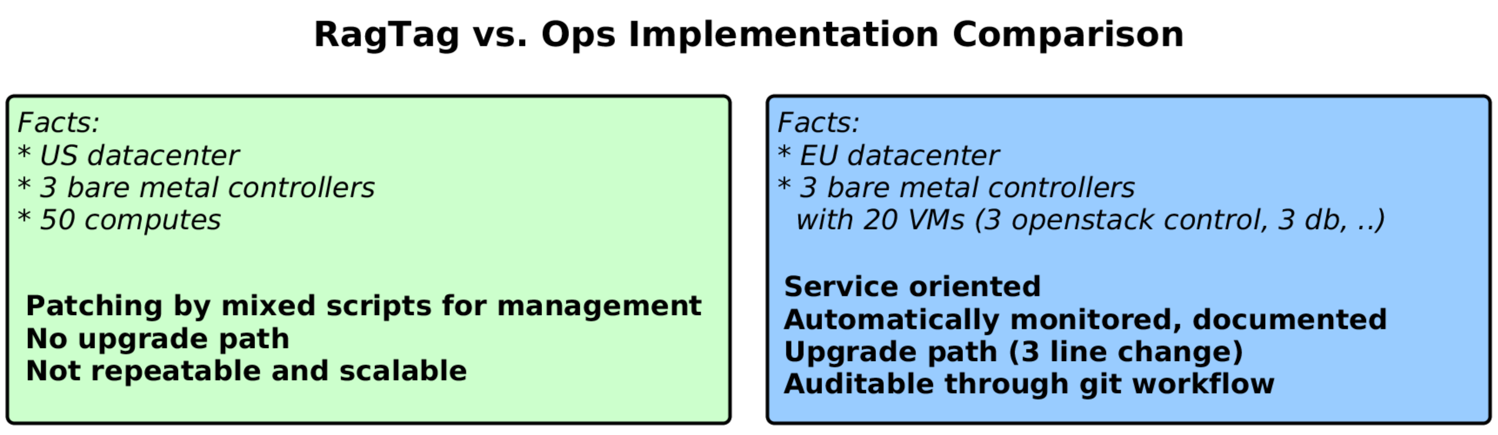
Second phase called Ops or as we mentioned “Day 2 Operations” can start with typical example:

OpenStack is up and running and then you come to work another day to an email from your security team “Please upgrade or reconfigure RabbitMQ to prevent security vulnerability”. How can you do it with confidence? Your deployment tool cannot be used again. Now comes day-to-day operation, which is more difficult than the deployment itself.

This lead us to define following criterias for day-to-day operations like patching, upgrades, business continuity, disaster recovery, automatic monitoring, documentation, backups and recovery. General expectation is that **Ops** can be managed by the **Deployment tool**. However the reality is that the **Deployment tool** does not do **Ops**. As already mentioned deployment tools are one shot. Therefore you start build and **Ops** tools like random scripts (restart service or change configuration), manual hacks, tribal knowledge (production depends on specific people who knows how to manage).

Ideal workflow needs to include terms like repeatable patterns, single source of truth (Infrastructure as a Code), best practises, rigid to agile, declarative (desired state), personalise could experience.

We did not want to develop this workflow by ourselves, therefore we found OpenStack-Salt as optimal tool. It is official project under big tent since 2016/05. It is service oriented approach and covers almost all mentioned parameters of ideal workflow. It is production ready proven architecture managed as a code.


RagTag vs. Ops Implementation Comparison

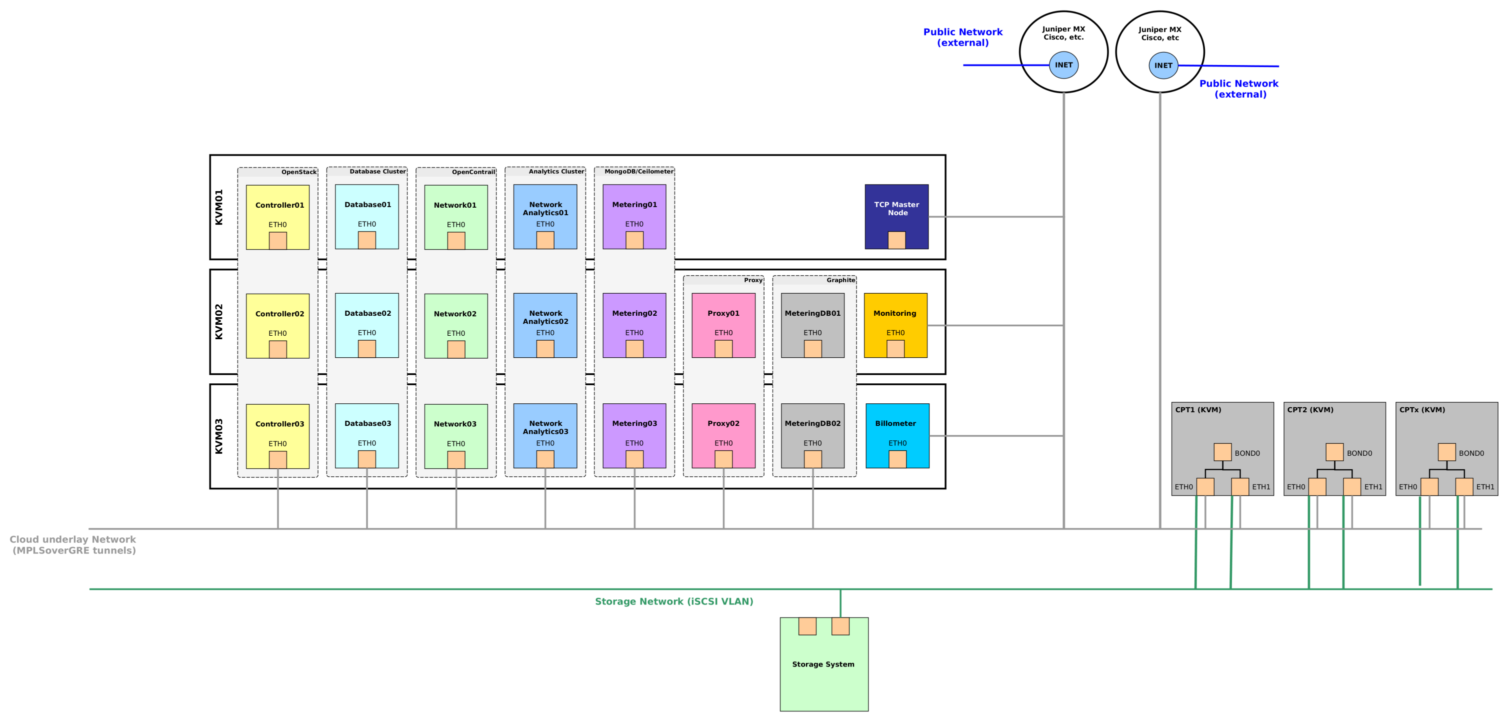
Facts:
** US datacenter*
** 3 bare metal controllers*
** 50 computes*

Patching by mixed scripts for management
No upgrade path
Not repeatable and scalable

Facts:
** EU datacenter*
** 3 bare metal controllers*
with 20 VMs (3 openstack control, 3 db, ..)

Service oriented
Automatically monitored, documented
Upgrade path (3 line change)
Auditable through git workflow

However still our production infrastructure looks like figure below. It consists about 23 Virtual Machines on at least 3 physical KVM nodes just for cloud control plane. We have to upgrade, patch and maintain 23 OS to provide flexibility and service oriented architecture.



DevOps

Based on previous ideas we asked question "what about to treat infrastructure as a microservice?". This bring us from **Ops** to **DevOps**, which really means to treat OpenStack as a set of Applications.

RagTag
Ops
DevOps

Our defined criteria is that it must be composable, modular, repeatable, immutable and split applications from infrastructure. It has to break monolithic VMs to containers and microservices.

We also decided that we do not want to reinvent wheel to create a new project, but reuse existing knowledge invested in OpenStack-Salt project.

These steps depict the evolution of OpenStack deployment in last 2-3 years. Now let's take a look how to build containers and microservices instead of monolithic deployments of the past. The following sections explains technically DevOps workflow.

How to build containers?

Configuration management era has started couple years ago, when tools like Fabric, Puppet, Chef and later SaltStack or Ansible change the approach of how to deploy application and infrastructure in companies. These tools finished the era of bash scripting and bring repeatable, idempotent and reusable patterns with serialized knowledge. Companies invest a huge effort into this approach and community brings opportunity to deploy OpenStack almost in every this configuration management tool.

Recently the era of microservices (accelerated by Docker containers) have appeared and as we described in the DevOps workflow, containers should encapsulate services to help to operate and treat infrastructure as microservice applications. Unfortunately Docker pushes configuration management tools and serialized knowledge off to the side. Even some experts predict end of configuration management with Docker. If you think about it, you realize that docker brings dockerfiles and entry points with which invoke dejavu of bash scripting again. So why have we invested so much into a single source of truth (Infrastructure as a Code), if we should started from scratch? This is the question, which we had on our mind before we started with concept for containerisation of OpenStack. So first requirement was building Docker containers by more effective way than just bash everything. We took a look at OpenStack Kolla, CoreOS and other projects around which provide approach for getting OpenStack in containers.

Kolla uses Ansible for containers build and jinja for parametrization dockerfiles. The concept is very interesting and promises a good future. However it is completely new in way of serialized knowledge and production operation of OpenStack. Kolla tries to be universal for docker containers build. There is missing orchestration or recommended workflow for running on production not only single machine with host networking. Kolla-kubernetes project has started almost month ago, but it is still too early to run it seriously in enterprise environment. There must be done a lot of work and bring more operational approach. Basically we want to reuse what we have in OpenStack-Salt as much as possible without a new project or maintaining two solutions.

We defined two criterias to leverage running OpenStack services in containers.

- Use **configuration management** for building Docker containers as well as standard OS
- **Reuse existing solution** – do not start from scratch and rewrite all knowledge into another tool just for containers build and maintain two worlds.

We created a simple repository Docker-Salt (https://github.com/tcpcloud/docker-salt), which builds containers by exactly same salt formulas used for tens of production deployments. This enabled knowledge reuse, When someone patches configuration in a standard OpenStack deployment, it automatically builds a new version of Docker container as well. It provides opportunity to use single tool for Virtual Machine OpenStack deployment as well as microservices. We can mix VM and Docker environments and operate environment from one place without a combination of 2-3 tools.

Build Pipeline

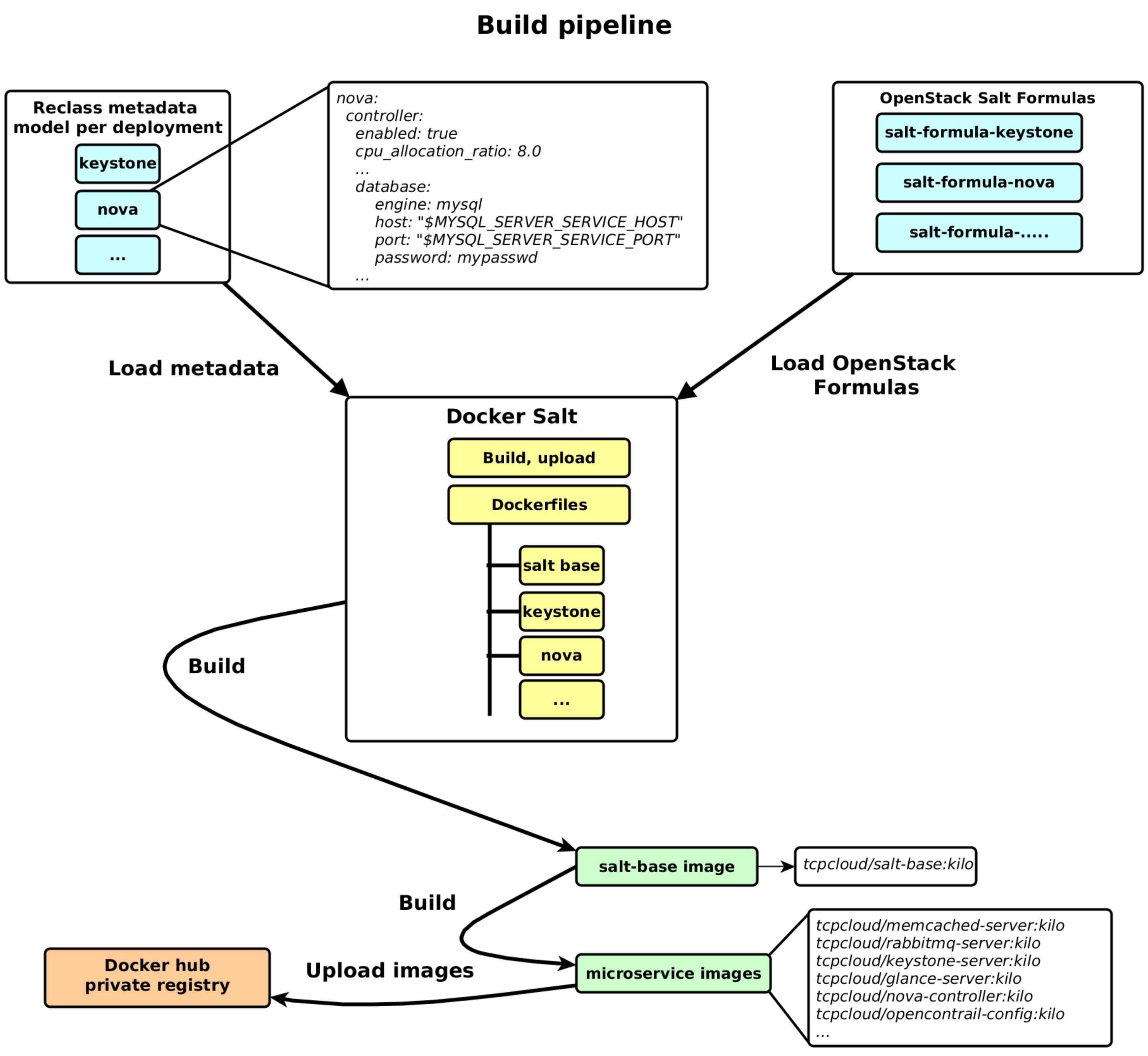
The following diagrams shows building pipeline for Docker images. This pipeline is completely automated by Jenkins CI.

Reclass metadata model is deployment specific and it is single source of truth (described later), which contains configurations like neutron plugin, cinder backends, nova cpu allocation ratio, etc. It is git repository with a simple yaml structure.

OpenStack salt formulas are currently used for tasks such as installing a package, configuring, and starting a service, setting up users or permissions, and many other common tasks.

Docker Salt provides scripts for build, test and upload Docker images. It contains dockerfiles definitions for base image and all Openstack support and core microservices.

Build process downloads the metadata repository and all salt-formulas to build salt-base image with a specific tag. Tag can be an OpenStack release version or any other internal versioning. Every configuration change in OpenStack requires a rebuild of this base image. The base image is used to build all other images These images are uploaded to private docker registry.



```
nova:
  controller:
    enabled: true
    cpu_allocation_ratio: 8.0
    ...
    database:
        engine: mysql
        host: "$MYSQL_SERVER_SERVICE_HOST"
        port: "$MYSQL_SERVER_SERVICE_PORT"
        password: mypasswd
    ...
```

```
tcpcloud/memcached-server:kilo
tcpcloud/rabbitmq-server:kilo
tcpcloud/keystone-server:kilo
tcpcloud/glance-server:kilo
tcpcloud/nova-controller:kilo
tcpcloud/opencontrail-config:kilo
...
```

Docker Salt repository contains compose files for local testing and development. OpenStack can be run locally without Kubernetes or any other orchestration tool. Docker compose will be part of functional testing during the CI process.

Changes in current formulas

Following review shows changes required for salt-formula-glance (https://review.openstack.org/#/c/320707/). Basically we had to prevent starting glance services and sync_db operations during the container build. Then we have to add entrypoint.sh, which instead of huge bash script that replaces env variables by specific values then runs salt highstate. Highstate reconfigures config files and runs sync_db.

You might note that Salt is not uninstalled from container. We wanted to know what is the difference between container with or without salt. The glance container with salt has about 20MB more than glance itself. The reason is that both is written in python and uses same libraries.

To get more information about container orchestration and live upgrade continue on second part post.

Jakub Pavlik
tcp cloud

Lachlan Evenson
Lithium Technologies