iVolve
High Level Design Document

Private Cloud Build Plus

Prepared by:	Yoshi Kadokawa <yoshi.kadokawa@canonical.com>
Prepared on:	26 March, 2021
Version:	1.0

Design Stakeholders

iVolve

Name	Email	Title
Bilal Bin Ameer	bilal.ameer@ivolve.io	Manager DevOps
Rashid Iqbal	rashid.iqbal@ivolve.io	Senior DevOps Engineer

Canonical

Name	Email	Title
Yoshi Kadokawa	yoshi.kadokawa@canonical.com	Field Software Engineer
Mark Maglana	mark.maglana@canonical.com	Field Software Engineer

Revisions

Date	Version	Notes
2020-03-26	1.0	Initial DRAFT

Table of contents

1. Introduction

This document contains the High Level Design (HLD) for the Project. This document is complemented by a Low Level Design document (LLD).

The purpose of this document is to describe the solution architecture design for the OpenStack Cloud deployment. The solution architecture design described here is based on the understanding of iVolve's key objectives & requirements, analysis of hardware bill of materials, network infrastructure, and workloads requirements for the cloud, combined with industry expertise and recommended practices.

It includes an overview of data centers, components, and technologies used to build the solution, including some of the key configuration details. However, this document does not cover operational or change processes. It lists technologies and interfaces to operations and change process but only at the overview level.

Identity management, networking and storage topics are covered at the architectural level, leaving detailed and implementation-specific specifications in the LLD document.

1.1. Intended readers

This document is intended for all readers who wish to get a logical overview of the project architecture. For technical architects and engineers, the LLD document is required to fully understand the target solution.

For project and line managers this document will serve as an efficient summary.

2. Key Objectives & Requirements

Canonical proposed solution includes the Foundation Cloud Engine (FCE) for delivering the OpenStack cloud. Following are the key requirements of the OpenStack platform solution mapped with the Canonical OpenStack features :

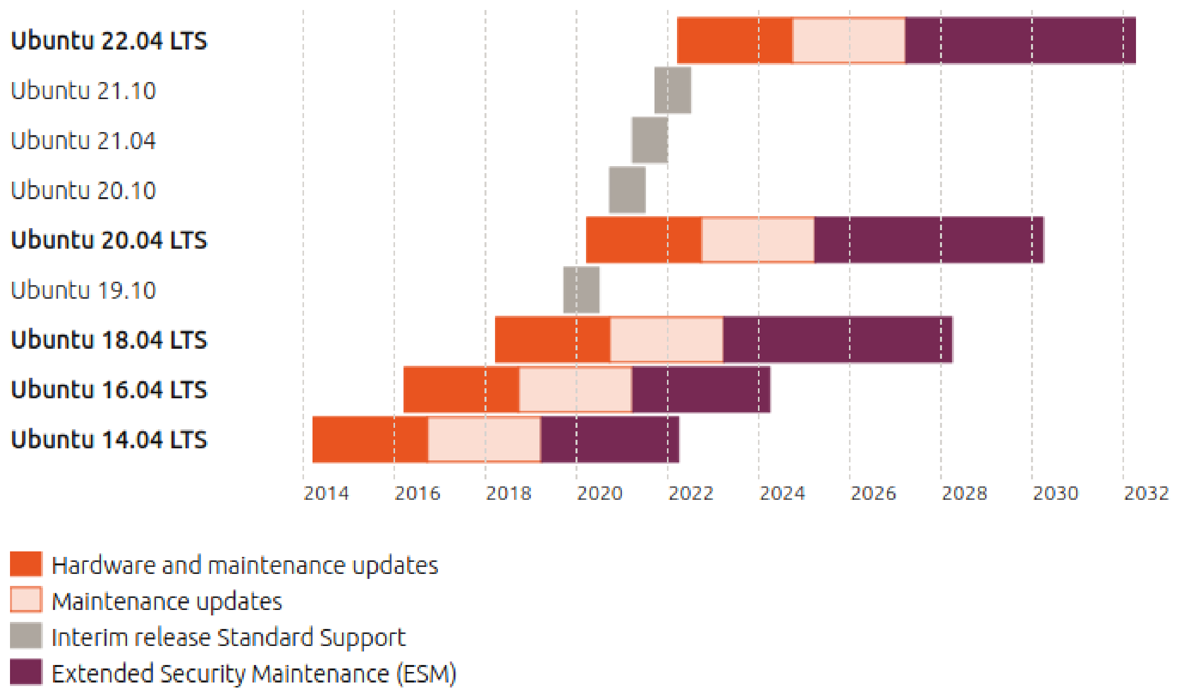
Key Requirements	Description/Features
Customer workloads/use cases	Public cloud. Only for external customers.
High Availability	Full OpenStack HA, including MAAS, Juju & Landscape
Monitoring and updates	Included, powered by Nagios, Prometheus, Grafana & Landscape
Log Aggregation	Included, powered by Elasticsearch and Graylog
Hypervisor Support	Nova KVM
Architecture	Custom Architecture (standard components only)
Encryption	OpenStack Control Plane - TLS terminated API endpoints. Ceph + Instance storage Encryption at Rest
Identity	Keystone with Active Directory backends
Storage Products	Ceph Block
Storage Performance	Ceph RBD with 3x replication
Additional Storage Options	RADOSGW Object Storage
Shared File System Service	Manila with CephFS including NFS Gateway(Ganesha)
Tuning Options	CPU overcommit - CPU overcommit 4:1 - RAM overcommit 1:1

	CPU Pinning: Yes Huge Pages: Yes
Networking Options	OVN with Geneve + Provider Networks
Network Topology	NIC bonding, unlimited underlay L2 network segregation
Load Balancing	Octavia layer-7 Application Load Balancer as a Service
Secrets Management	Barbican with Vault
IPv6 Addressing	Tenant networks only

3. Technologies

3.1. Ubuntu Server

Ubuntu Server is the operating system developed by Canonical for servers and cloud and is the base platform for the whole deployment. Ubuntu is a Linux based operating system with commercial support provided based on the following schedule:



Ubuntu Server is a subset of the overall Ubuntu operating system and LTS releases are long term supported versions. For the lifetime of a release, Canonical provides security patches and bug fixes.

The Ubuntu version selected for the project is **Ubuntu 20.04 LTS**. Key features of 20.04 release include:

- Security and bug fix support until April 2025;
- Upgrade path to Ubuntu 22.04, once released. Supported until April 2027;

Further detail on the Ubuntu release cycle is available at https://www.ubuntu.com/about/release-cycle.

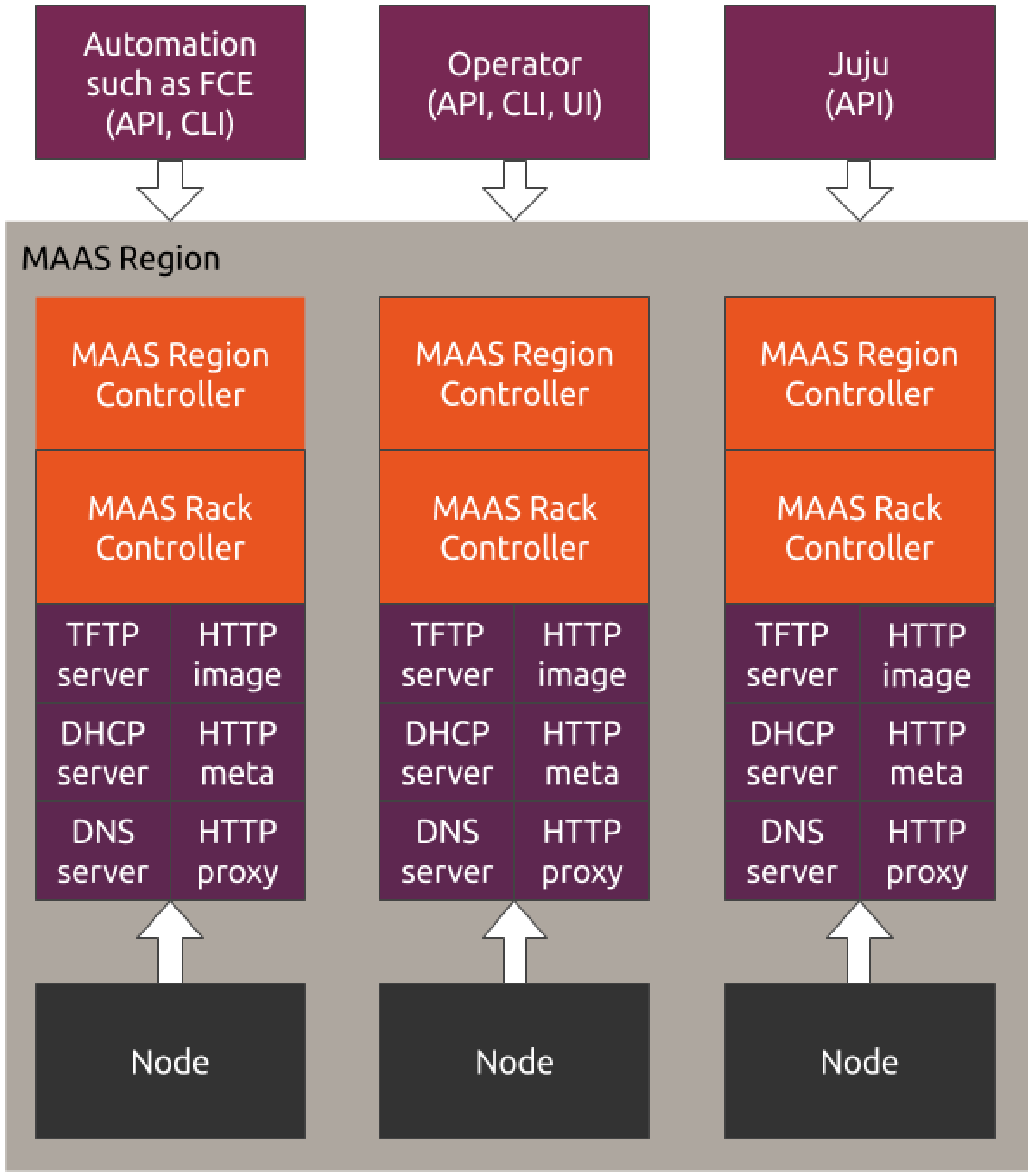
3.2. MAAS

MAAS, Metal as a Service, allows a programmatic approach to data center deployments. With built-in IPAM, inventory modules and API services, it provides a single point of contact for server management.

A data center managed by MAAS is represented as a pool of resources:

1. Compute resources: physical machines or virtual machines and their meta-information (such as tags and resource pools);
2. Storage resources: physical and logical devices attached to machines and storage pools for virtual machines;
3. Network resources: fabrics, vlans, spaces, subnets, subnet ranges and other objects.

MAAS is set up to run on all infrastructure machines to provide a highly available service to machines for PXE boot purposes and clients (API, CLI, UI).



MAAS is split in two logical units: Region Controller and Rack Controller. Region Controller provides API services and stores all information about the nodes. Rack Controllers are stateless and provide services based on information provided by Region Controllers. Rack Controllers provide TFTP, DHCP and low level services used in the PXE boot process and post-boot lifecycle. They also act as HTTP proxy servers for package downloads and MAAS metadata server queries and as DNS forwarders.

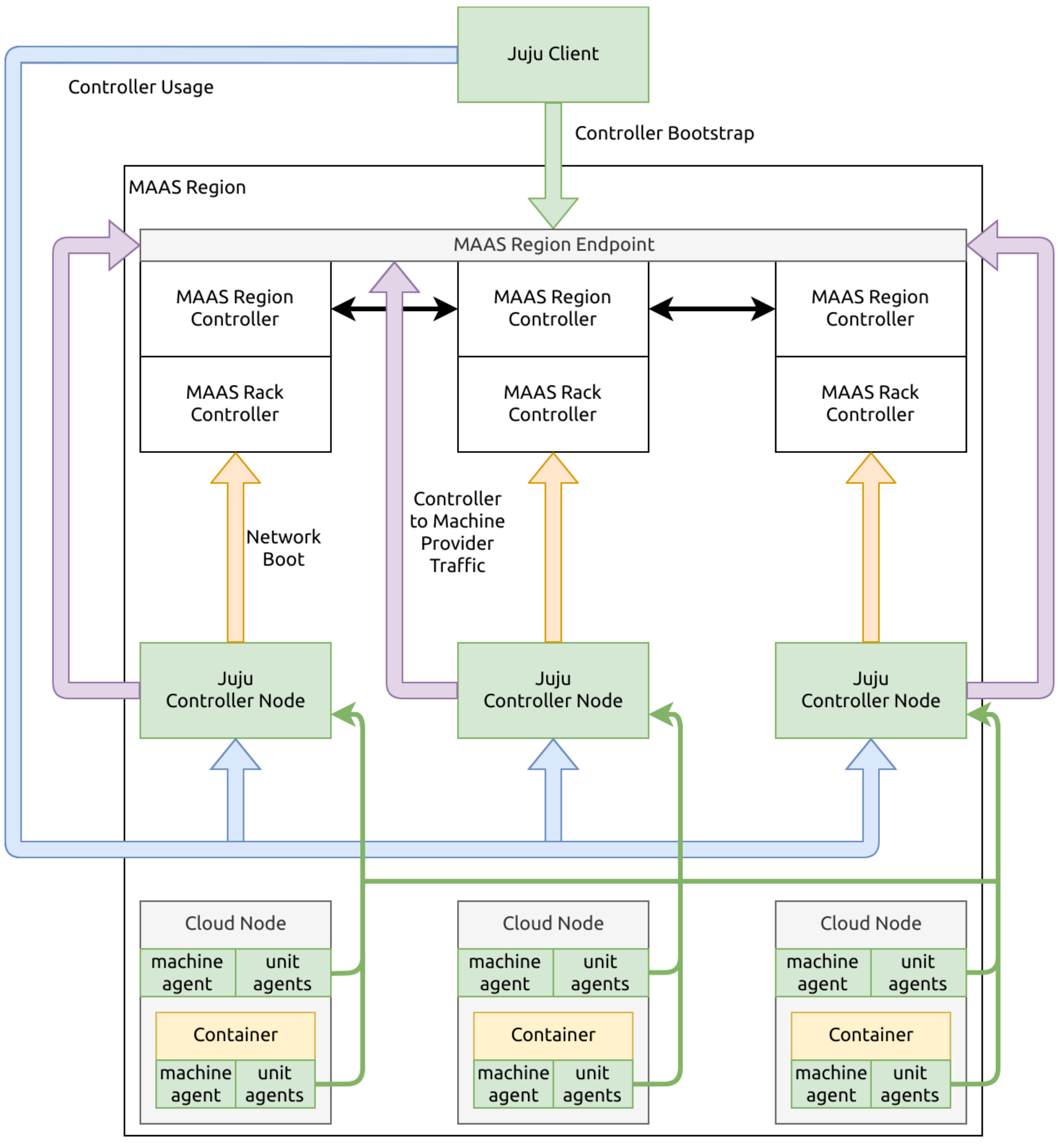
The first three AZs will have both MAAS Rack and Region services on Infra nodes. Once the cloud starts scaling out, only additional MAAS Rack controllers will be added.

The MAAS version that will be used in this project is specified in **appendix** 1. A detailed MAAS documentation can be found at https://maas.io/docs .

3.3. Juju

Juju is a service modeling software implemented as a distributed system. It uses a model description as an input and delivers an environment according to a set of requirements defined in the model. It is also used to manage the lifecycle of services in a given model.

The following diagram illustrates a typical Juju deployment:



Juju consists of a number of components:

1. Controller node (physical or virtual) which hosts API and database components of Juju;
    - Controllers include a machine provider-specific provisioning logic (MAAS acts as a machine provider in this case);
2. Juju client which is used to bootstrap and manage Juju itself and models provisioned via Juju;
3. Machine agents are used to perform post-deployment machine-related tasks that are relevant for Juju operation and report to Juju controller nodes;

4. Unit agents execute application lifecycle events. The concrete application-specific logic is encoded in Charms.

The Juju version that will be used in this project is specified in **appendix** 1. A detailed Juju documentation can be found at https://jaas.ai/.

3.4. Charms

Charms encode the operational knowledge about a particular application in a reusable way. The operational knowledge in question is about deployment, scaling, upgrades, post-deployment reconfiguration, setting up software in a highly available configuration, integrations with other applications etc.

An application provider would typically have the best operational knowledge about a given application and Charms provide a way to express this knowledge in a more sophisticated way than what deployment-focused automation and regular configuration tooling can do.

Charms rely on Juju to provide them with a platform for distributed and event-driven code execution, data exchange and a data model for common operations. However, the specifics of how to use that platform for a particular application are outside of Juju and are implemented in concrete charms.

The generic approach to operations taken in Juju and Charms makes Charms an exceptional tool not only in deploying and operating simple applications like load balancers or web servers but also complex applications such as OpenStack or Kubernetes.

Most of the software deployment in this engagement will be done using Juju and Charms. All charms that will be used in this project are open source and available at https://jaas.ai/

The major portion of the charms used in this project come from the official OpenStack project called OpenStack Charms which is developed, maintained and supported by Canonical https://docs.openstack.org/charm-guide/latest/

3.5. LXD

LXD is an evolution of LXC, a machine container technology. It provides API access to LXC which becomes a lower level construct.

Machine containers act very much like virtual machines. Containers share a kernel with the host and isolation is done via Linux kernel mechanisms such as namespaces (mount, user, network, pid, IPC, UTS and cgroup namespaces), cgroups, capabilities, AppArmor and LSM. As opposed to virtual machines, containers do not emulate a motherboard, CPU and memory, storage and network devices (albeit the latter two are typically virtual devices in the kernel). The lack of emulation reduces boot time compared to virtual machines, removes additional overhead associated with mode switching and some memory overhead ultimately increasing a maximum possible workload density.

LXD can be used for isolation between services running on the same physical machine. Regardless of the target architecture (hyper-converged, disaggregated etc.), it can be useful to run control plane and data plane components in different logical machines which is the approach taken in this project. Thus, each LXD container contains one unit of a service and its own operating system userspace components.

LXD containers are unprivileged containers as they create a separate user namespace which ensures that the root user in a container is not the root on the host. Each container is confined with an AppArmor profile on the host in addition to any AppArmor profiles for services within a LXD container.

Containers are also segregated from the networking perspective with network namespaces which isolate kernel ARP/neighbor tables, routing tables, iptables, some sysctl settings as well as interfaces assigned to a given container. Interface isolation allows a container to be connected to a subset of network segments a host system has access to. For example, a database container would not have a network interface on a public API network but an API container would, even though they would be placed on the same host.

Following are the recommended readings for LXD:

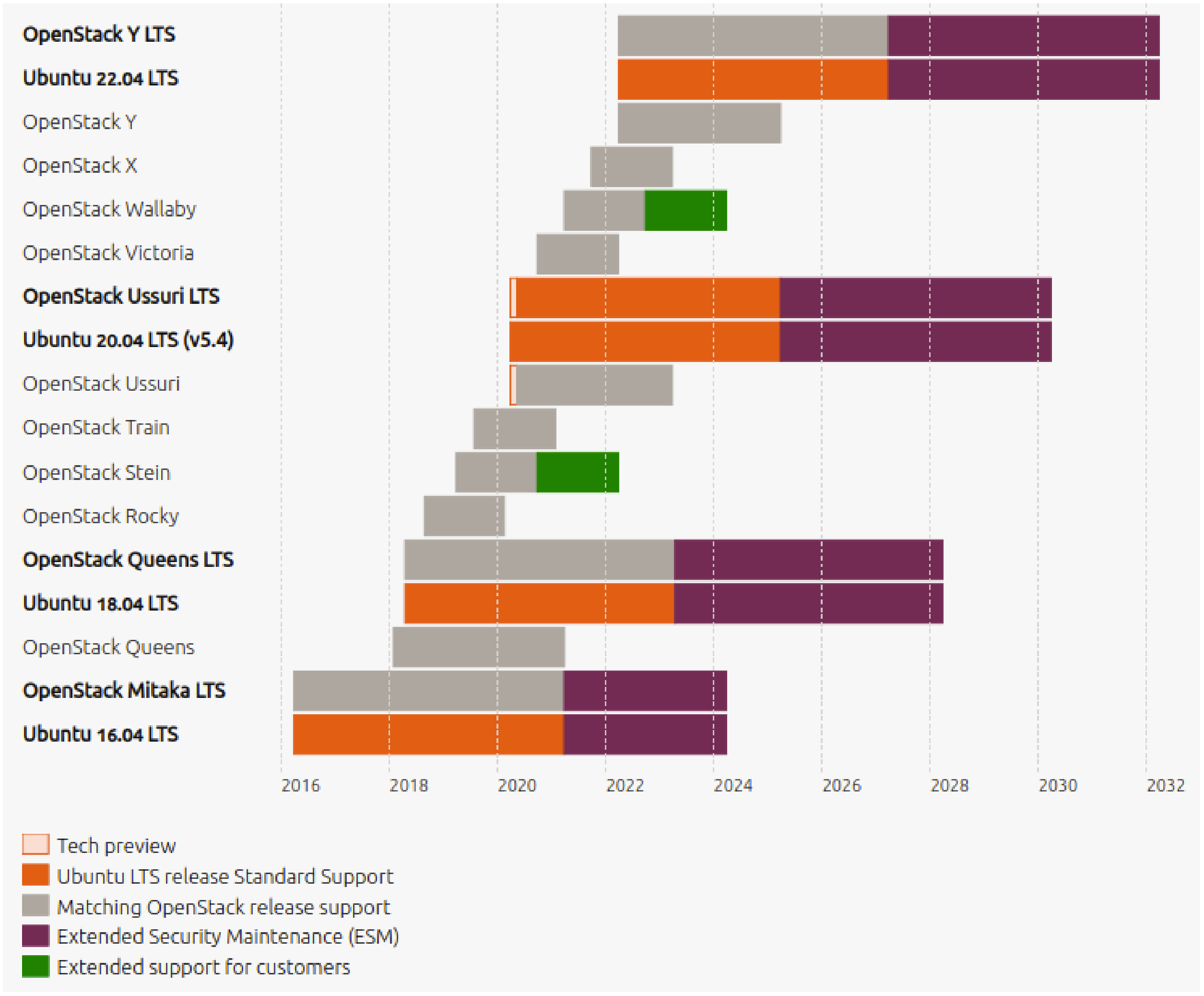
- https://linuxcontainers.org/lxd/introduction/
- https://help.ubuntu.com/lts/serverguide/lxd.html.en

3.6. Canonical OpenStack

Canonical OpenStack is a product that includes Ubuntu OpenStack distribution, processes and automation for Ubuntu OpenStack deployment, MAAS, Juju and a set of tools for monitoring and logging. MAAS and Juju deployment and configuration is done using Foundation Cloud Engine (FCE) and the OpenStack deployment is done using OpenStack Charms.

Ubuntu OpenStack is a part of the Ubuntu operating system offering and Ubuntu Cloud Archive can be used to get newer releases of OpenStack for LTS versions of Ubuntu.

Up-to-date information about Ubuntu and OpenStack version support periods is available at https://www.ubuntu.com/about/release-cycle. The following diagram is a snapshot from the release lifecycle page:



For monitoring and logging, Canonical OpenStack uses:

- Graylog for log access and persistence with FileBeat used for log collection and Elasticsearch as a backend for log storage;
- Telegraf and Prometheus for metric collection and Grafana for metric visualisation;
- Nagios for service monitoring;
- Canonical Landscape for systems management.

Canonical OpenStack provides an upgrade path to newer versions of OpenStack: once newer non-LTS or LTS versions of OpenStack are available and the respective OpenStack Charm release is made, a deployment can be upgraded to the next version via charm actions. Upgrades from OpenStack versions included into the next LTS release require an Ubuntu series upgrade. More information about upgrade mechanisms included in OpenStack Charms can be found in the documentation:

1. https://docs.openstack.org/project-deploy-guide/charm-deployment-guide/latest/app-upgrade-openstack.html

2. https://docs.openstack.org/project-deploy-guide/charm-deployment-guide/latest/app-series-upgrade.html

All OpenStack services deployed in this project are deployed in a highly available mode such that each service has at least 3 units.

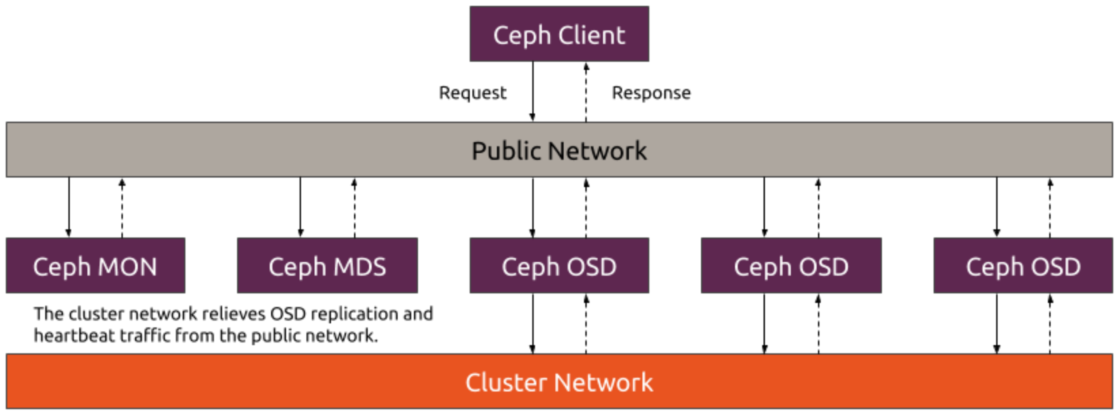
3.7. Ceph

Ceph is one of the most widely used Software Defined Storage solutions. It contains two primary components: OSDs and Monitors. OSDs provide storage for a cluster and are either collocated with compute workloads in the hyper-converged or converged architectures or are deployed on separate storage nodes in the disaggregated architecture.

Ceph is used as a storage backend for Glance, Cinder and Gnocchi services. Ceph RadosGW service also provides object storage by exposing Swift API to its clients.

Ceph access network used by Ceph clients is separated from the replication network used by OSDs.

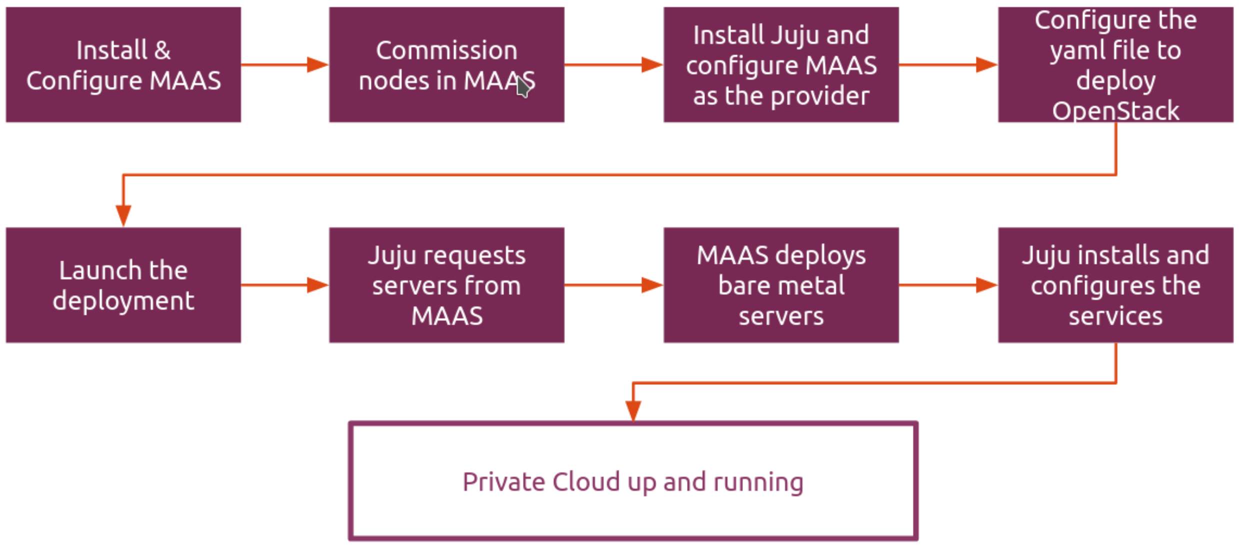
Ceph storage pools can be scaled out by adding more OSDs.



3.8. Foundation Cloud Engine

Foundation Cloud Engine (FCE) is a suite of scripts that manage customized deployments of the Private Cloud infrastructure. The Engine performs actions such as catalog available hardware, deploy and configure infrastructure, and apply customizations. Executing the individual steps in the correct order and with the prerequisite configuration is essential to a successful deployment.

Following are the high-level steps for deploying a Private Cloud Build using FCE:

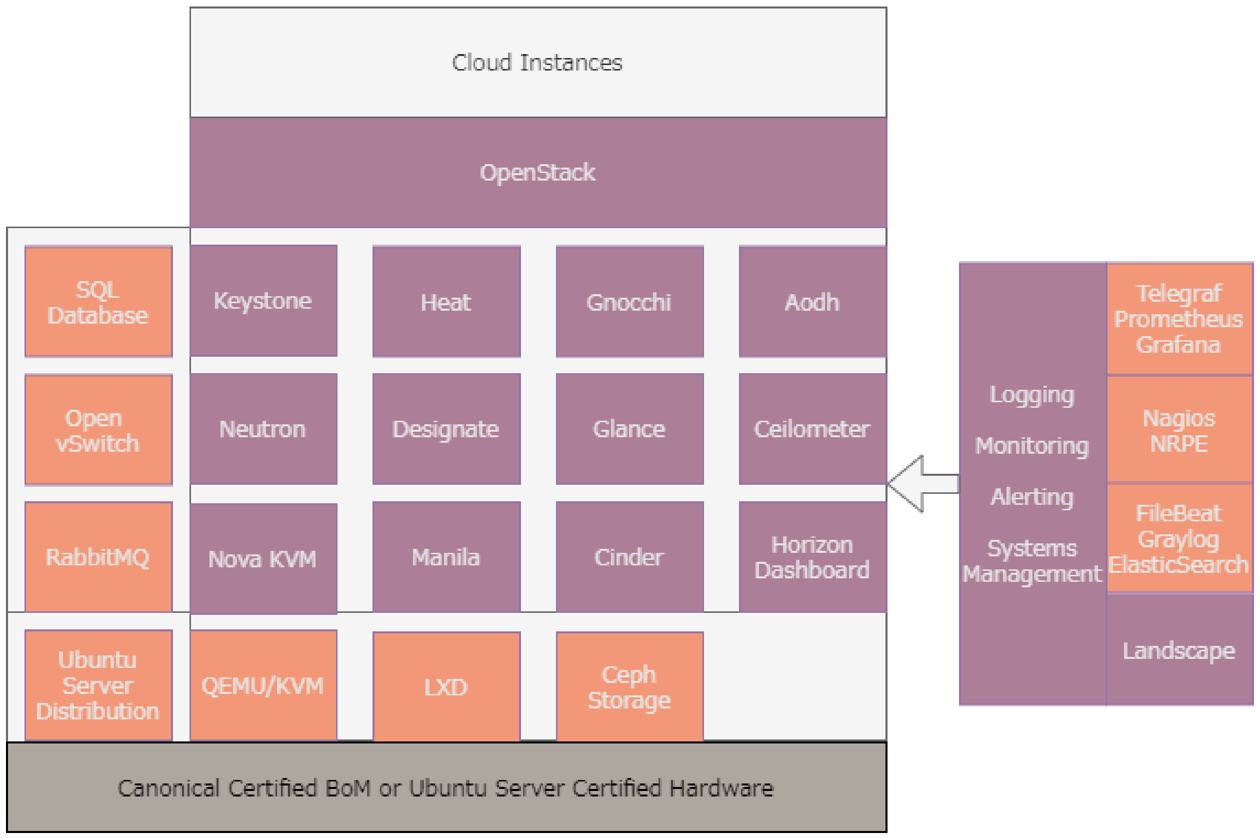

Install & Configure MAAS
Commission nodes in MAAS
Install Juju and configure MAAS as the provider
Configure the yaml file to deploy OpenStack
Launch the deployment
Juju requests servers from MAAS
MAAS deploys bare metal servers
Juju installs and configures the services
Private Cloud up and running

A more detailed deployment guide will be provided as an addendum to the low-level design document.

4. High Level Architecture

4.1. Overview

An overview of a base Canonical OpenStack configuration as deployed during the Private Cloud Build process is present at the diagram below.



The solution has a number of components included that belong to OpenStack itself (such as different API services) or exist to support the deployment and provide database, message queueing, hypervisor or packet forwarding capabilities. Moreover, the solution comes with logging, monitoring, alerting and systems management services to support operators in sustaining the cloud in production.

4.2. OpenStack Components

Canonical OpenStack contains a supported subset of OpenStack projects. The following set of OpenStack services will be deployed in this project:

Service Name	Purpose
Keystone	Identity Management, authentication and authorization.
Nova	Compute resource provisioning and scheduling.
Cinder	Block Storage provisioning.

Glance	Image storage and management.
Neutron	Network connectivity as a service.
Designate	DNS as a service.
Heat	Cloud application orchestration.
Ceilometer	Metric and event collection and forwarding.
Aodh	Alarming based on metrics and events.
Gnocchi	Time series database as a service used to store and aggregate metrics collected by Ceilometer.
Horizon	A web UI service for OpenStack.
Octavia	Load-balancing as a service.
Barbican	Storage, provisioning and management of secrets.
Manila	Shared file system service for OpenStack

In addition to core OpenStack services the following additional services are deployed:

- MySQL InnoDB Cluster - MySQL InnoDB clustered database;
- RabbitMQ - a message queuing service (AMQP);
- Memcached - a NoSQL database used as a backend for Gnocchi, Nova consoleauth and Designate;
- Hashicorp Vault- a secrets management software used for data encryption at rest and used as a secrets store for Barbican.
- Keystone LDAP - Keystone connectivity to LDAP backend provided by customer

4.3. Data Center Environment and Availability Zones

In this design initial racks used for deployment are considered to be independent entities that are placed into separate failure domains and get mapped to availability zones in OpenStack. However, this separation is logical and needs to be backed up by a physical implementation:

1. Rack placement (e.g. into different fire compartments);
2. Independent power sources and uninterruptible power supplies;
3. Independent cooling;

4. Isolation of racks into different network pods;
5. Usage of a routing and switching design with a small failure impact scope (blast radius).

A specific data center design could define a certain hierarchy of entities such as:

1. Device;
2. Rack;
3. Pod;
4. Room;
5. Data center.

The precise definition of an availability zone will depend on design decisions made during the data center design. This definition and the number of availability zones could be included for reference in the Low Level Design document based on the customer input. The term "rack" will be used going forward to discuss high availability with an assumption that racks outline availability zones.

A single rack is enough to provide all OpenStack services but 3 racks are the minimum used for high availability. Additional racks will provide only compute, storage and networking capabilities. If the number of machines in one region (control plane) scales above 200, it is recommended to create a new data center.

Nodes of each type have to be physically distributed into different availability zones. The recommended initial configuration is such that there is an infrastructure node present per rack. Cloud nodes should be evenly distributed across availability zones. If there are 3 availability zones, the scaling of the cloud is recommended to be done by adding 3 new servers each time.

4.4. Logical Architecture

4.4.1. Type of architecture

During the design of the solution, one type of architecture will be chosen for the deployment. The following table describes the different types. More details can be found in the *Canonical Cloud Reference Hardware* document.

Type	Description
Hyper-converged	All components (computing, network and storage) are software-defined components and are distributed in all the servers of the cloud. Each component is containerized in an LXD container which isolates it from the others. Ceph OSDs for storage, Openstack services and compute components on collocated on the servers.

	For general purpose workloads, this choice maximizes the utilization of the physical resources deployed, such as CPU, RAM, storage and network ports.
Converged	One type of node hosts the network and services components, while another type of node hosts the storage and compute workloads.
Disaggregated	Compute, network and storage components are all hosted on separate nodes.
Mixed	A custom solution may be designed when required.

Choice of architecture and justification:

For this deployment, the architecture *disaggregated* has been chosen.

4.4.2. Machine Classification

Machines in the deployment are classified as follows:

Machine Class	Machine Count	Purpose
Infrastructure	**3**	Machines used for cloud operations. Services deployed on them include MAAS, Juju, Landscape, ElasticSearch, Nagios, Prometheus, Grafana, Graylog, Vault.
Controller Nodes	**3**	Controller nodes host OpenStack control plane
Compute Nodes	**10**	Compute nodes host compute services. Collocated with distributed networking components.
Storage Nodes	**11**	Storage nodes host storage services

4.4.3. Service placement

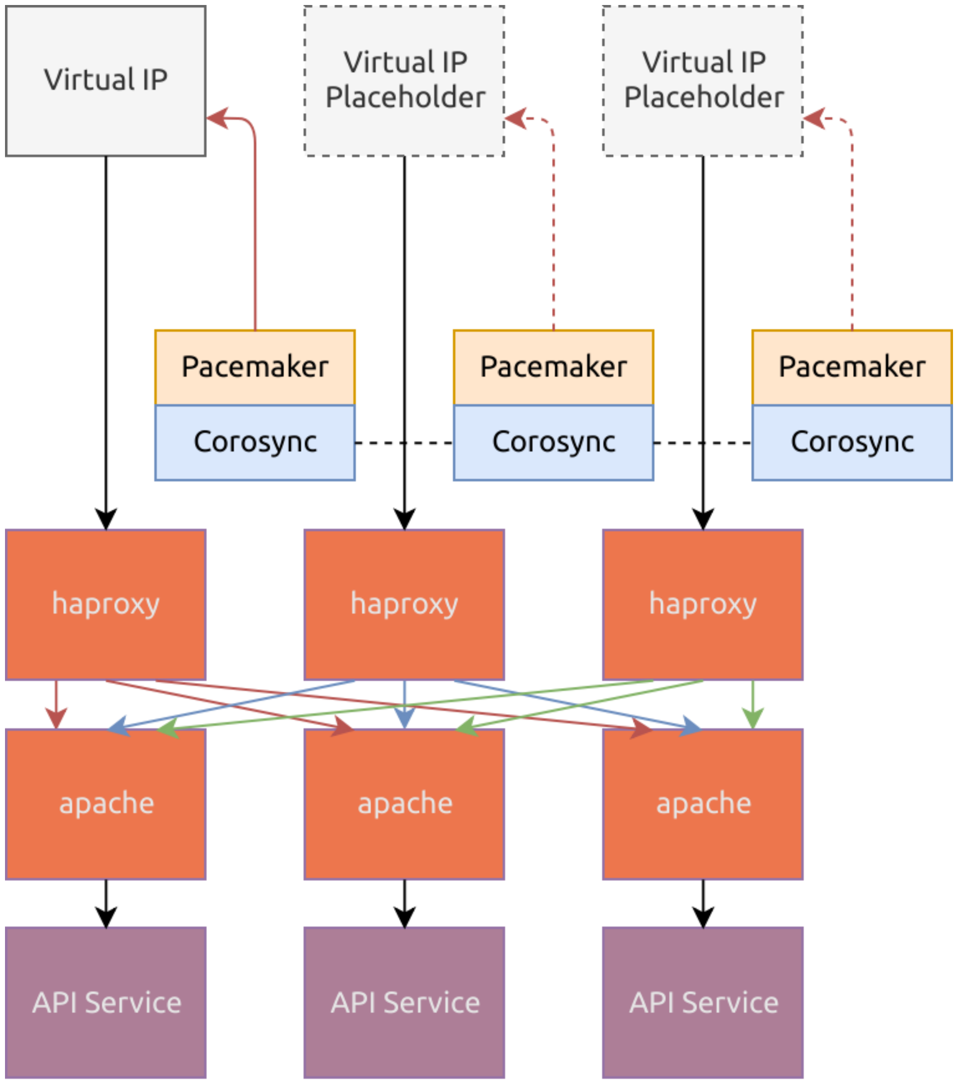
Most of the services in Canonical OpenStack are placed into LXD containers to improve isolation. Exceptions include nova-compute and Ceph OSDs. The logging and monitoring stack is placed into QEMU/KVM virtual machines deployed on infrastructure nodes.

In the disaggregated architecture, control plane services will be deployed in controller nodes, compute services in compute nodes, and storage services in storage nodes. Distributed networking components will be collocated in the compute nodes. Control plane services of the same type are spread across availability zones (for example, 3 Keystone services will be placed into containers in different availability zones).

The exact placement of services to nodes will be given in the low-level design guide.

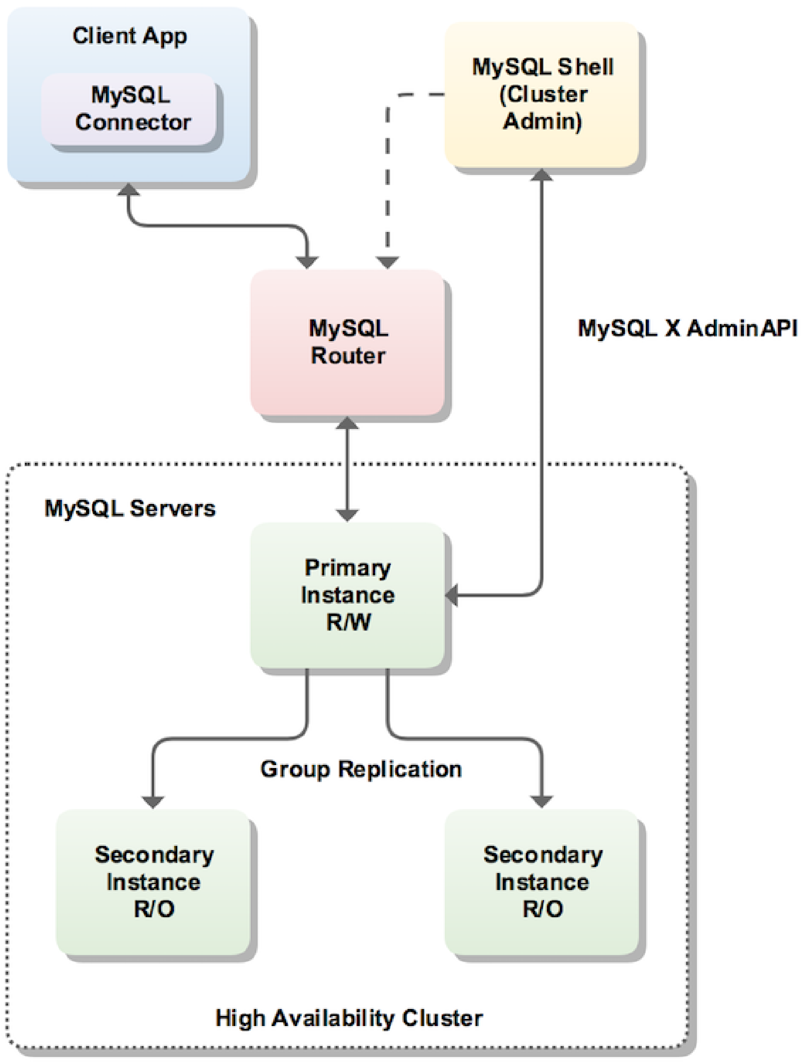
4.5. High Availability

There are several types of high availability scenarios that are used in Canonical OpenStack. For setups that rely on a single broadcast domain[1] for all API and support service units, a virtual IP per service is set up. This allows a single endpoint to access the service via one of the deployed haproxy load balancers.. The load is shared across backends that are alive. Upon a failure of haproxy or the whole node (which also means that corosync would be down) the virtual IP is migrated to a different node. A gratuitous ARP is then sent out from it to the broadcast domain, which updates the ARP tables of other nodes and gateways. The figure below describes this configuration.

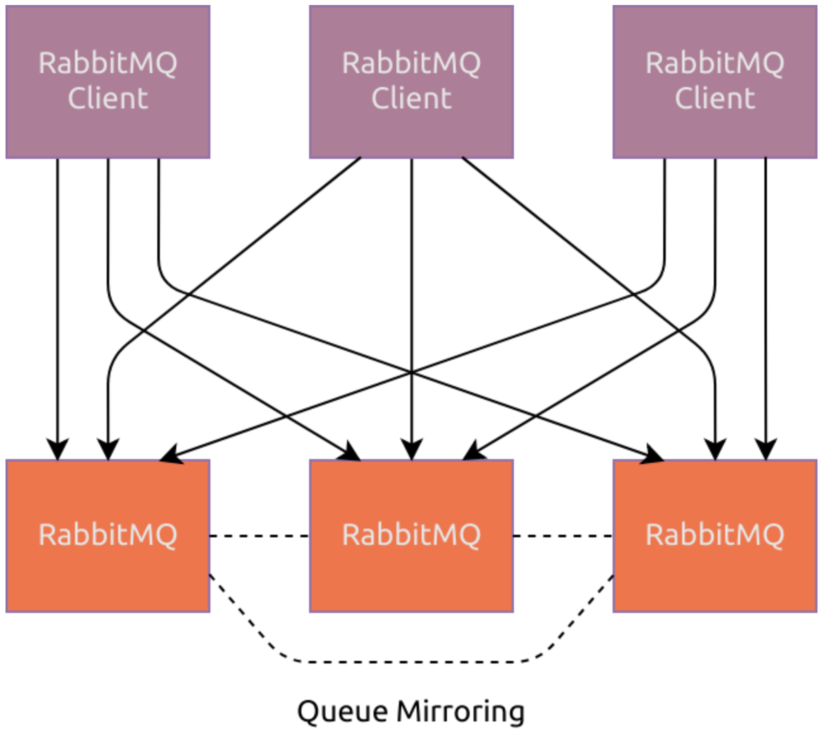


[1] For setups with multi-subnet logical network configurations, DNS-HA is used instead of virtual IPs.

MySQL InnoDB Cluster provides a complete high availability solution for MySQL. MySQL Shell includes AdminAPI which enables you to easily configure and administer a group of at least three MySQL server instances to function as an InnoDB Cluster. Each MySQL server instance runs MySQL Group Replication, which provides the mechanism to replicate data within InnoDB Clusters, with built-in failover. AdminAPI removes the need to work directly with Group Replication in InnoDB Clusters, but for more information see Chapter 17, Group Replication which explains the details. MySQL Router can automatically configure itself based on the cluster you deploy, connecting client applications transparently to the server instances. In the event of an unexpected failure of a server instance the cluster reconfigures automatically. In the default single-primary mode, an InnoDB Cluster has a single read-write server instance - the primary. Multiple secondary server instances are replicas of the primary. If the primary fails, a secondary is automatically promoted to the role of primary. MySQL Router detects this and forwards client applications to the new primary. Advanced users can also configure a cluster to have multiple-primaries. Source.



RabbitMQ clients are aware of multiple RabbitMQ endpoints and, therefore, do not rely on a clustering solution to provide them with a single endpoint. While mirrored RabbitMQ queues are used, all OpenStack services maintain a list of all active brokers and will connect to the first available broker. If that broker fails, they will switch to a new broker.



4.5.1. Availability Zones and Quorum Logic

Services managed by Pacemaker in combination with Corosync are placed into separate availability zones. The clustering logic is configured such that a cluster partition with a corosync quorum gets resources such as virtual IPs placed on its nodes. For a 3-unit cluster, a partition with 2 out of 3 units will have a virtual IP resource. However, if a partition contains only 1 unit (i.e. 2 units in the cluster fail), the virtual IP resource will not be running.

4.6. Security

4.6.1. OpenStack Security

Canonical follows security best practices from the official OpenStack Security guide located at http://docs.openstack.org/sec/

4.6.2. Control Plane TLS Termination

iVolve has opted to use a wildcard certificate signed by a publicly trusted root CA. This certificate will be used by all OpenStack endpoints to ensure secure communications with clients. The exact name of the subdomain will be covered in the LLD.

4.6.3. Logging

Audit processes apply to Canonical OpenStack at two levels:

1. Ubuntu Server;
2. OpenStack components.

OpenStack components provide standard logging available from the OpenStack distribution. Graylog integrated with ElasticSearch and FileBeat provides a centralized view of logs gathered by different components across the deployment.

Ubuntu Server includes a number of features to support audit processes such as centralized logging for services via systemd-journald, logging of kernel events via dmesg, logging of AppArmor profile violation attempts, logging of various system events via auditd, sending logs to a centralized log collection system via rsyslog.

More information is available in the Ubuntu Server documentation:

- http://manpages.ubuntu.com/manpages/bionic/man8/systemd-journald.service.8.html
- https://wiki.ubuntu.com/AppArmor;
- https://wiki.ubuntu.com/DebuggingApparmor;

4.6.4. Patching

Landscape allows an operator to define upgrade profiles in order to keep systems updated according to a defined schedule. It also allows alerts to be configured such that an operator is notified about regular package or security package update availability.

LivePatch can be optionally deployed onto cloud nodes in order to perform live security patching of general availability Ubuntu kernels.

In addition to patching the cloud nodes, it is also a good security practice to periodically update cloud images stored in Glance as well to include the latest patches for the OS and any applications installed within those images.

4.6.6. ⚠ Encryption at Rest

There are several classes of data that are protected by the “Encryption at Rest” feature:

1. Ephemeral instance storage - data placed in Nova-managed virtual disks associated with instances that reside directly on a hypervisor server;
2. Data stored in Ceph;
    a. Volume storage - data placed into Cinder-managed block devices provided by Ceph;
    b. Object storage - data placed into Ceph RADOS Gateway via Swift API;
    c. Metric - metrics stored via Gnocchi into Ceph objects.

The implementation relies on the Vault service for storing and retrieving secrets used to decrypt data placed onto physical block devices.

4.7. Identity Management

In this project the following integration with external systems are used for user authentication:

- Keystone integration with Active Directory via LDAP;

4.7.1. LDAP

Keystone supports user authentication via an external user directory using LDAP. In this case users and groups are mapped into the Keystone database and authentication is performed by using an LDAP client with credentials provided by a user via CLI or Web UI. If the remote directory accepts user credentials, user login succeeds and Keystone generates a token for the user.

LDAP identity backend is read-only in Keystone so any users will need to be pre-created outside of Keystone. Moreover, Keystone identity backend does not retrieve project or role information via LDAP (only users and groups). Projects and roles are managed in Keystone via the SQL backend.

Keystone will still use the SQL backend for initially created users, such as a cloud admin user, in the admin domain. The LDAP backend will only be used for a chosen Keystone domain.

Connectivity to the LDAP service can be done via LDAPS or StartTLS to provide a secure user authentication workflow. A CA certificate that can be used to verify LDAPS endpoint certificates needs to be provided to Keystone for creating secure connections successfully.

Details such as user and group subtrees and other details will be mentioned in the low-level design guide.

4.8. Networking

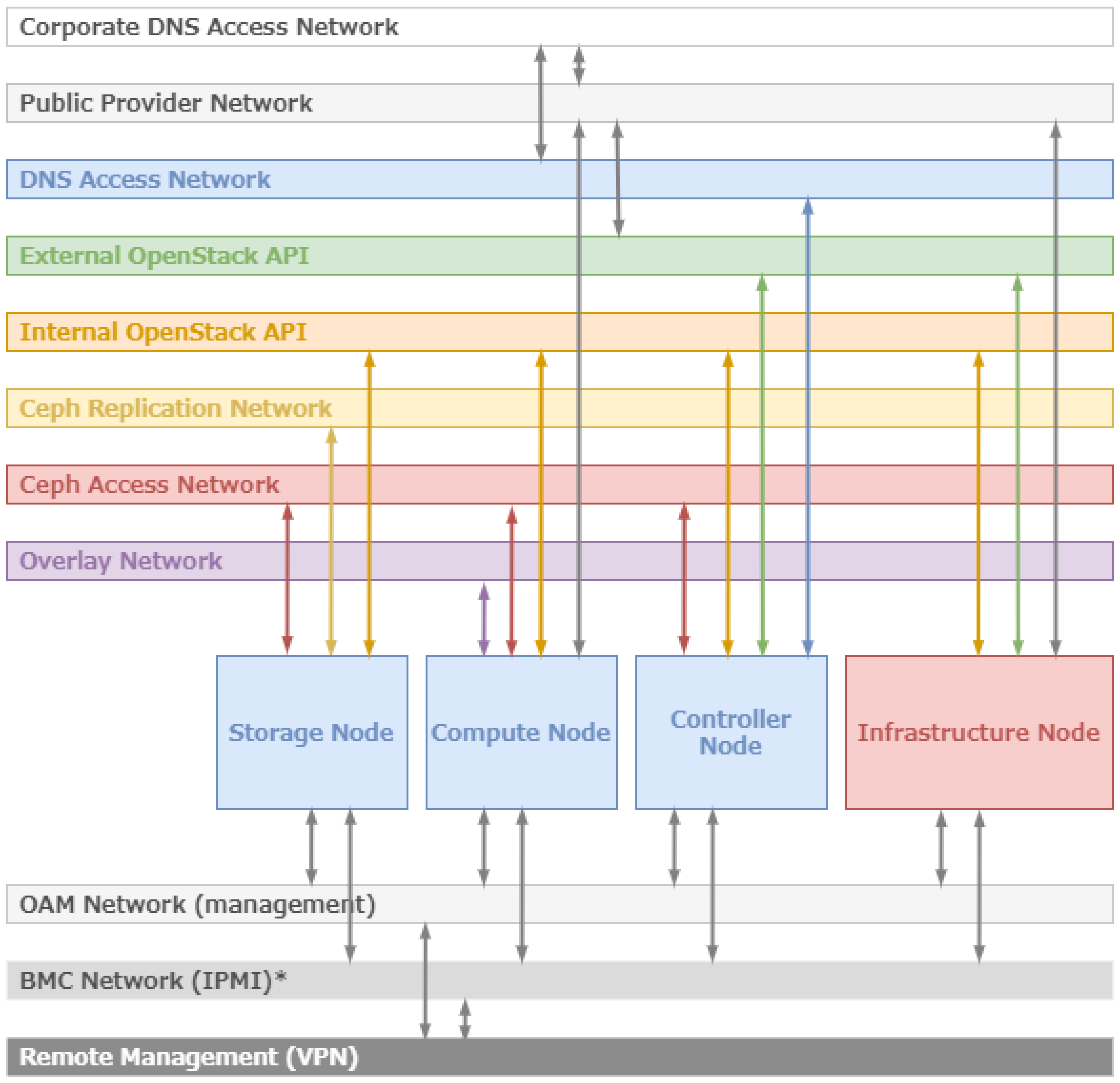
4.8.1. Logical Network Design

The logical networks are represented below. More precise details such as MTU values in use and subnets allocated to network segments will be present in the low-level design document.

Name	VLAN Tagging	L2 Segments
Provisioning	N (native vlan)	1
OAM network (can be combined to provisioning network)	*Same as provisioning network*	1
IPMI network	Y	1 or multiple
Remote Management	Y	1 or multiple
Internal OpenStack communication	Y	1
Ceph Replication Network	Y	1
Ceph Access Network	Y	1
Underlay network for Overlay network traffic	Y	1
DNS access network for Designate Bind services	Y	1

Provider network for Instance Floating IPs	Y	1 or multiple
Externally consumable OpenStack API	Y	1

The following diagram shows nodes to network connectivity and routing between networks that needs to be set up on the physical network infrastructure (represented as arrows between networks).



The network connectivity setup is kept identical for all nodes to facilitate post-deployment testing from infrastructure nodes.

In general, each network is associated with a routing domain. For single-segment networks that are self-contained (such as storage access) routing is solved trivially by using ARP broadcasts.

Default gateways of nodes are configured to use the OAM network gateway so traffic to any non-direct destinations will be sent through that gateway. Policy-based routing is used for generating responses to the requests sent by clients from arbitrary subnets to public API endpoints. Replies are sent back out the same interface the request came in on, not via default gateway.

Individual services will have access only to a subset of networks depending on their requirements. For example, only Ceph clients (QEMU, RadosGW, Cinder, Glance, Gnocchi) and Ceph services themselves will have access to the Ceph Access network. This is enforced for services placed into LXD containers by attaching them only to bridges they need access to.

Provider network access is separated into its own datapath as Open vSwitch bridges and Linux network namespaces are used by Neutron, therefore, services located on converged nodes cannot directly access provider networks. Therefore, routing arrangements need to be in place to make sure that end hosts (including VMs) on provider network subnets can access OpenStack Public API network and Corporate DNS Access network.

There are a several multi-hop scenarios present in this design which need routing on the network infrastructure side:

1. Provider networks to Corporate DNS Access network;
2. Provider networks to OpenStack Public API;
3. Remote management (VPN) network to OAM network and IPMI network;
4. OAM network to IPMI network. This is only necessary for Infrastructure nodes;

4.8.2. Underlay

In this deployment there will be two switching fabrics:

1. Workload switching fabric;
2. Management switching fabric.

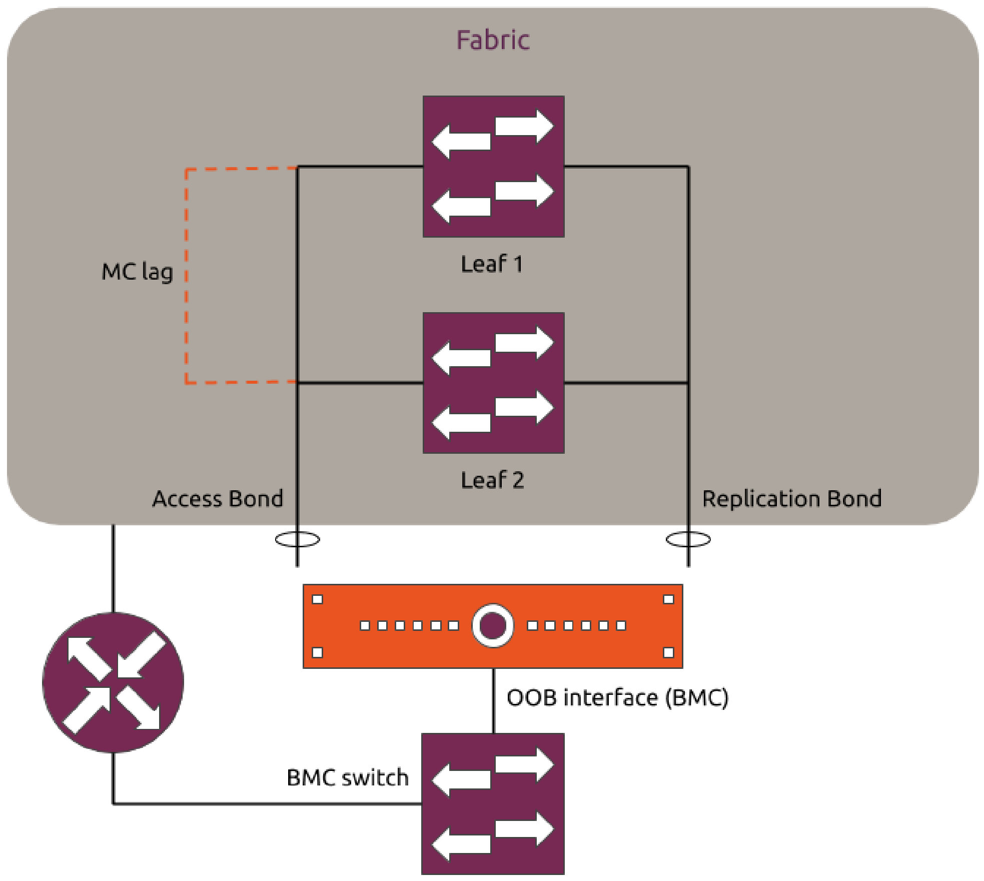
Trunking will be set up between switches for nodes to access the relevant VLANs, therefore, both fabrics will be L2-oriented.

The specific fabric implementation details (multi-tier, Leaf-Spine, EVPN) may be included into the low level design guide if provided by the customer.

4.8.2.1. Access and Replication Bonds

Two bonds will be used for workload traffic both of which will have identical characteristics:

- LACP;
- Fast LACP failover;
- Jumbo frames enabled at switch ports for all VLANs (host-side MTU configuration will vary depending on the purpose of the network).



4.8.2.2. Management Bond

Management bond is an LACP bond created out of on-motherboard network ports that allows a node to PXE boot. This bond is also used for management traffic (ssh, logging, monitoring, alerting, package downloads and updates). The characteristics of this bond are as follows:

- LACP;
- LACP fallback/bypass/suspend-individual[2] mode;

[2]Feature naming differs across network device and NOS vendors.

    - A special mode needs to be enabled on the switch side (sometimes called "LACP fallback" or "LACP bypass") in order to bypass the lack of an LACP peer on the host side during the initial boot process.
- Fast LACP failover;
- MTU 1500.

4.8.3. Overlay to Provider Network Connectivity

In this project, tenant networks are implemented using VXLAN. Connectivity to physical provider networks is provided via "data ports" which are attached to Open vSwitch bridges to forward traffic from overlay networks to underlay networks and back. In the distributed virtual routing (DVR) case or for direct VM attachment to provider networks, data ports must be available on every compute node to provide direct access to provider networks from every machine. With traditional setups where gateway components are placed on special nodes, only those nodes need to have physical connectivity to the provider network VLANs.

There are two scenarios that are generally used for data-port configuration depending on the number of interfaces available for consumption at a given node:

1. A dedicated physical interface or a bond is allocated to Open vSwitch which manages VLAN tagging based on segmentation ID information configured for a Neutron network;
    - This allows for dynamic creation of provider networks of type "vlan" with different VLAN tags via Neutron;
2. Operating-system VLAN interfaces (pre-created in MAAS at the deployment time) are given to Open vSwitch which also gets multiple per-VLAN bridges. Each bridge is associated with a physnet and Neutron provider networks are created with the "flat" type;
    - This approach is more static as the underlying node configuration is required to add more provider networks, however, it allows using the same network interface or bond for both the underlay traffic (such as public API or VXLAN traffic) and provider network traffic.

The first approach is used for this project.

4.8.4. Virtual Routing

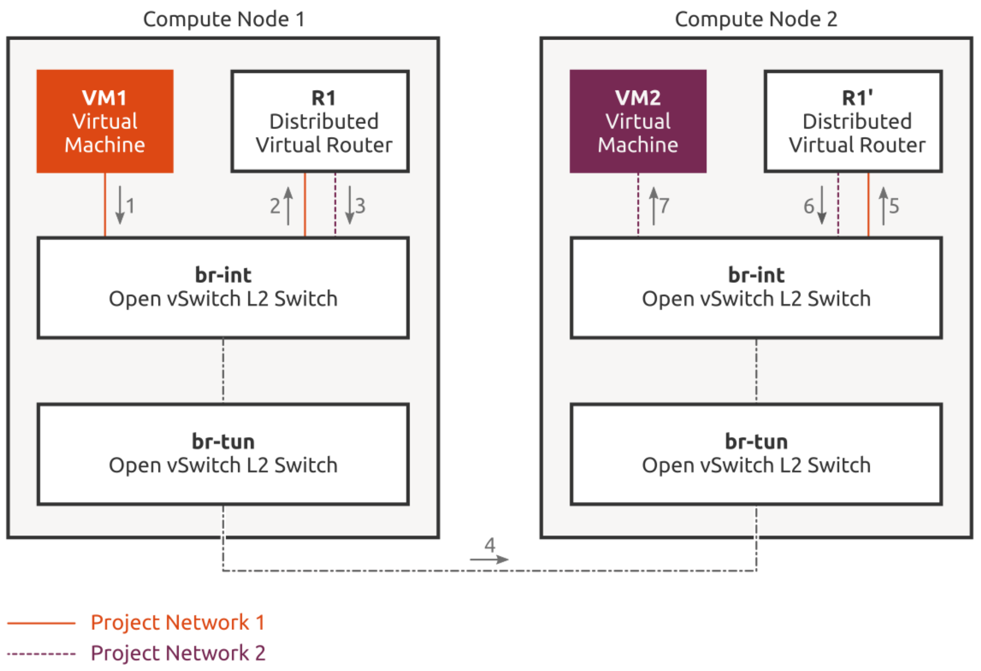
In this deployment, Canonical OpenStack is integrated with Open Virtual Network(OVN) by using OVN ML2 plugin. OVN complements the existing capabilities of OVS by adding native support for virtual network abstractions, such as virtual L2 and L3 overlays and security groups.

4.8.4.1. Distributed Virtual Routing

Canonical OpenStack is configured to support Distributed Virtual Routing (DVR) with OVN.

Virtual routers can be placed directly into the Compute Nodes instead of centralized Network Nodes. In this case, East-West traffic between Virtual Machines attached to different project networks flows between Compute Nodes only, bypassing Network Nodes. This architecture effectively reduces a failure domain to a single Compute Node, rather than a centralized Network Node.

The following diagram illustrates East-West communication between two VMs attached to different project networks:


Compute Node 1
Compute Node 2
VM1 Virtual Machine
R1 Distributed Virtual Router
VM2 Virtual Machine
R1' Distributed Virtual Router
1
2
3
4
5
6
7
br-int Open vSwitch L2 Switch
br-tun Open vSwitch L2 Switch
Project Network 1
Project Network 2

Each Compute Node hosting Virtual Machines with Floating IP addresses assigned is configured with a Floating IP namespace (`fip-<external-network-id>`). This allows for North-South traffic between Virtual Machines and external endpoints (such as the Internet for example) to bypass Network Nodes as well. It is required however, that Compute Nodes have Open vSwitch data ports attached to all Provider Networks that are intended to be used.

4.8.4.5. Provider Networks

Canonical OpenStack supports the following types of Provider Networks:

- Provider Networks with RFC 1918 private IPv4 subnets,
- Provider Networks with public IPv4 subnets.

In order to conserve public IPv4 addresses, centralized routers with HA can be used on Provider Networks with public IPv4 subnets. In this case, a single Floating IP address is consumed by the router per Provider Network.

Neutron RBAC can be used to expose a single project network to multiple projects. The exposed project network is connected with a centralized HA router to the public Provider Network. This allows for connecting VMs created in different projects to a single, exposed project network and thus connect them to a public Provider Network via a single router. This router consumes a single IP address on the Provider Network.

For the Provider Networks with RFC1918 private IPv4 subnets, Distributed Virtual Routers can be used.

Provider Network Visibility

Neutron RBAC can also be used to hide external networks for projects that do not need them. Initially, provider networks can be created with the type “internal” and later shared with relevant projects as external using Neutron RBAC. That way only the projects for which the necessary RBAC rules were created will see provider networks as external and will be able to attach their router-external ports to them.

4.9. DNS

Each MAAS region will maintain its local zone and will provide DNS services to cloud nodes by forwarding requests to upstream DNS servers.

Each region controller will have a Bind DNS server collocated with it to provide a DNS resolution service. Each node deployed in a cloud region will receive at least 3 DNS resolver addresses that will be pointing to MAAS region controllers for a given region.

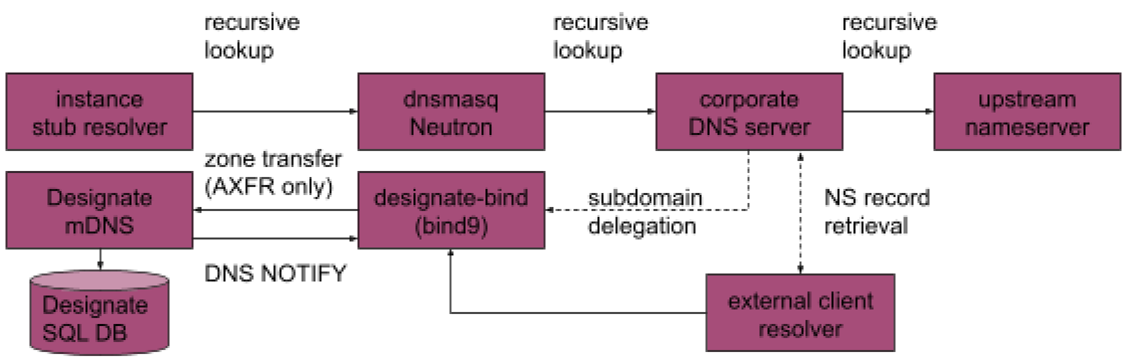
All DNS records for endpoints pertaining to Openstack Services will be hosted in Corporate DNS:

- api.region-one.openstack.example. (Corporate DNS);
    - keystone.api.region-one.openstack.example;
    - cinder.api.region-one.openstack.example;
    - glance.api.region-one.openstack.example;
    - nova.api.region-one.openstack.example;
    - neutron.api.region-one.openstack.example;
    - designate.api.region-one.openstack.example;
    - radosgw.api.region-one.openstack.example;
    - aodh.api.region-one.openstack.example.

Tenant zones will be delegated by MAAS DNS to Designate Bind:

- <tenant-zone>.compute.region-one.openstack.example. (Designate Bind);
    - ws-vm1.<tenant-zone>.compute.region-one.openstack.example;
    - ...
    - vm-name.<tenant-zone>.compute.region-one.openstack.example;

DNS records based on instance names will be provided by the Designate service in conjunction with Bind servers deployed with the cloud that will provide resolutions for those records. The high-level overview of the individual components and interactions is provided on the diagram below:



4.10. Storage

4.10.1. Services

Canonical OpenStack provides the following storage services:

1. Ephemeral storage for instances created via Nova
2. Block storage via Cinder and Ceph RBD backend
3. Object storage via Ceph RadosGW and Swift API
4. Image storage via Glance with Ceph used as an image storage backend
5. Shared filesystem via Manila and CephFS

Instances can be booted from Cinder volumes instead of ephemeral storage. While this improves the resilience of instance primary storage devices, it comes with a performance impact as this storage will be accessed over the network. The table below provides suggestions on how to use the storage capabilities provided by the platform.

Type of Instance	Storage Usage Suggestion
Workloads that use a lot of storage on the primary	Boot from volume.

device without low-latency requirements.	
Workloads that need persistent and reliable storage with a base image that can be easily restored in case of an unrecoverable compute node failure. Alternatively, database instances deployed in a highly-available configuration with native replication that need to periodically back up their data to a persistent volume using database-native mechanisms.	Ephemeral storage for the primary device with an additional volume for storing important data (such as backups).
Stateless workloads.	Ephemeral storage.

4.10.2. Ephemeral Storage Resiliency

In order to improve resiliency of local storage used for ephemeral instance disks, RAID-1 configuration will be used for the underlying physical disks.

The exact layout will be provided in the low-level design guide.

4.10.3. Ceph Storage Service Resiliency

Ceph pools are configured such that 3 replicas of the same data are created across the cluster. Moreover, CRUSH maps are generated with availability zone information exposed by MAAS and Juju taken into account. This allows for availability zone failures which will not result in data loss as replicated data is available on OSDs in different availability zones.

Ceph monitors are configured to have at least 3 units and Ceph clients such as QEMU, Rados Gateway, Cinder, Glance or Gnocchi are aware of multiple monitor endpoints.

Swift API is provided by Ceph Rados Gateway which is set up in the same way as OpenStack API services in terms of high availability.

5. Appendix 1: Software Versions

Component	Version or Series
Ubuntu Server	20.04 LTS
MAAS	2.9 or latest stable version
Juju	2.8 or latest stable version
OpenStack	Ussuri