MCP Reference Architecture

version 1.0

Contents

Copyright notice

Preface

This documentation provides information on how to use Mirantis products to deploy cloud environments. The information is for reference purposes and is subject to change.

Intended audience

This documentation is intended for deployment engineers, system administrators and developers; it assumes that the reader is already familiar with network and cloud concepts.

Documentation history

The following table lists the released revisions of this documentation:

Revision date	Description
March 30, 2017	1.0 GA

Introduction

Mirantis Cloud Platform (MCP) is a deployment and lifecycle management (LCM) tool that enables deployment engineers to deploy MCP clusters and then update software and configuration through the continuous integration and continuous delivery pipeline.

MCP design

MCP includes the following key design elements:

MCP design overview

Component	Description
DriveTrain	A general term for the MCP Lifecycle Management (LCM) system that includes the CI/CD pipeline, including Gerrit, Jenkins, and MCP Registry, SaltStack, and the Reclass model. The components perform the following functions: • SaltStack is a flexible and scalable deployment and configuration management tool that is used for lifecycle management of MCP clusters. • Reclass is an External Node classifier that, coupled with SaltStack, provides an inventory of nodes for easy configuration management. • Gerrit is a Git repository and code review management system in which all MCP codebase and configurations are stored and through which all changes to MCP clusters are delivered. • Jenkins is a build automation tool that, coupled with Gerrit, enables continuous integration and continuous delivery of updates and upgrades to the MCP clusters. • MCP Registry is a repository with binary artifacts required for MCP cluster deployment and functioning.
MCP clusters	Using DriveTrain, you can deploy and manage multiple MCP clusters of different types. An MCP cluster can be an OpenStack environment or a Kubernetes cluster. Different types of MCP clusters can co-exist in one enclosed environment and be governed by the same set of monitoring and lifecylce management components.
DevOps portal	DevOps portal provides a single point of entry for cloud operators. Using DevOps portal, cloud operators can access functionality provided by DriveTrain and StackLight.
Logging, monitoring, and alerting (LMA), or StackLight	Responsible for collection, analysis, and visualization of application critical data, as well as alerting and error notifications through a configured communication system, such as email.

Metal-as-a-Service (MaaS)	Automation software that allows you to manage physical servers as easy as virtual machines.
High Availability ensured by Keepalived and HAProxy	Keepalived is routing software that provides virtual IP addresses. HAProxy is software that provides load balancing for network connections.
Mirantis OpenContrail (optional)	MCP enables you to deploy Mirantis OpenContrail as an network virtualization solution. Mirantis OpenContrail uses official OpenContrail packages with additional customizations by Mirantis.
Ceph cluster (optional)	Distributed object storage for an OpenStack environment.

Deployment automation

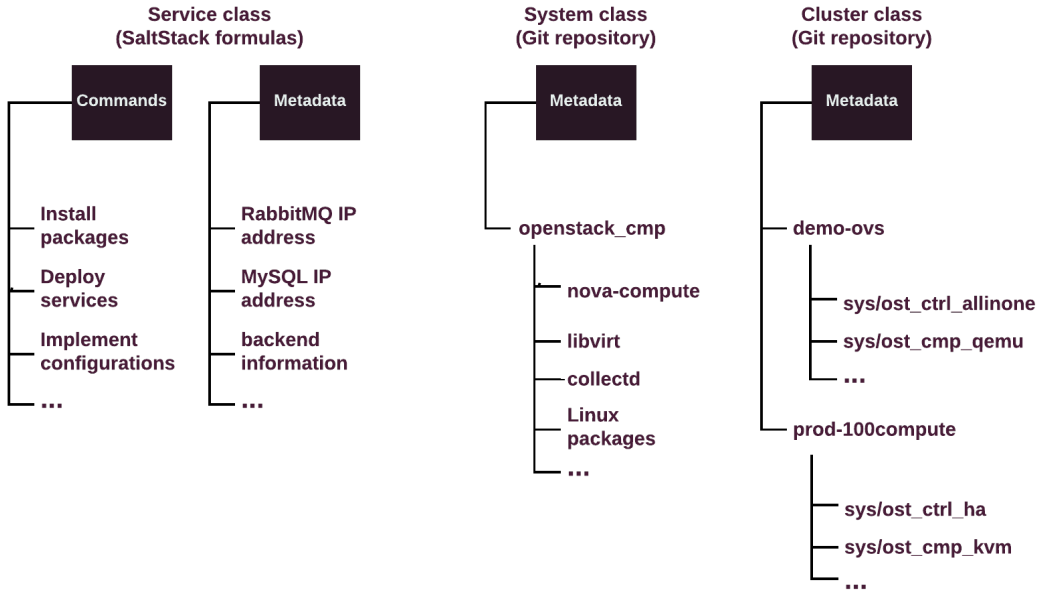
MCP uses SaltStack as an automation tool coupled with the Reclass and Cookiecutter tools to deploy MCP clusters. For example, all virtualized control plane (VCP) node configurations, as well as Kubernetes Master Tier configurations are defined by SaltStack formulas, service and cluster level classes (in Reclass terminology), and metadata.

SaltStack and Reclass model

SaltStack is an automation tool that executes formulas. Each SaltStack formula defines one component of the MCP cluster, such as MySQL, RabbitMQ, OpenStack services, and so on. This approach enables you to combine the components on a cluster level as needed so that services do not interfere with each other and can be reused in multiple scenarios.

Reclass is an external node classifier (ENC) which enables cloud operators to manage an inventory of nodes by combining different classes into MCP cluster configurations. Reclass operates classes which you can view as tags or categories.

The following diagram displays the Mirantis Reclass model hierarchy of classes:



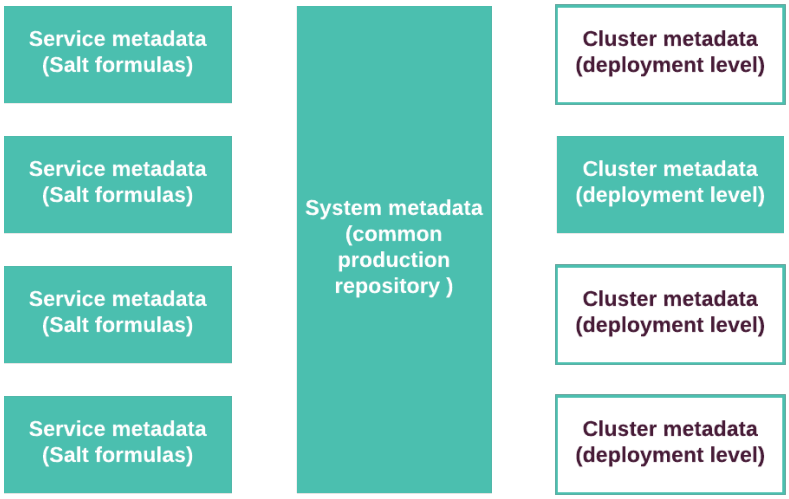
MCP reclass classes

Service class	System class	Cluster class
A service class defines one component of the MCP cluster, such as RabbitMQ, OpenStack services, MySQL, and so on. Service classes are defined in SaltStack formulas provided as .deb packages that you deploy on the Salt Master node during the bootstrap. SaltStack formulas consist of the execution part that defines the commands that need to be deployed and metadata that defines sensitive parameters, such as IP addresses, domain names, and so on. A service class inherits metadata from the system and cluster classes.	A system class defines nodes, such as compute and controller nodes, and the components required to be deployed on those nodes. System classes are the sets of the service classes combined in a way that produces a ready-to-use system on an integration level. For example, in the service ‘haproxy’ there is only one port configured by default (haproxy_admin_port: 9600), but the system ‘horizon’ class adds to the service ‘haproxy’ several new ports, extending the service model and integrating the system components with each other. System classes are provided in a Git repository. You must clone the repository on your SaltStack Master node to use the system classes. System classes inherit metadata from cluster classes.	A cluster class defines an MCP cluster profile, such as a demo or production cluster. A cluster class combines system classes into new solutions corresponding to the needs of the deployment. A set of predefined environment profiles is provided in a Git repository. You must clone the repository on your SaltStack Master node to use the cluster classes. Alternatively, you can generate cluster classes from the templates using Cookiecutter. This approach significantly speeds up metadata pre-population.

SaltStack repository structure

The repository structure allows deployment engineers to store and deploy multiple MCP cluster profiles. This is especially beneficial for multi-site and multi-cloud deployments that require different types of clouds to co-exist while using similar metadata at service and system levels. Reusing the system and service level metadata prevents data degradation over time, as well as provides consistency and ability to customize the system and service level classes as needed.

The following diagram displays the system and service classes repository structure:



Deployment templates

Although you can deploy an MCP cluster using an environment profile defined in the cluster level class, the Cookiecutter templates significantly simplify the process of deployment.

Cookiecutter is a tool that Mirantis deployment engineers use to create project templates and later use these templates to deploy different types of MCP clusters based on customer-specific data. An example of a Cookiecutter template can be a demo MCP cluster or a production MCP cluster.

Repository planning

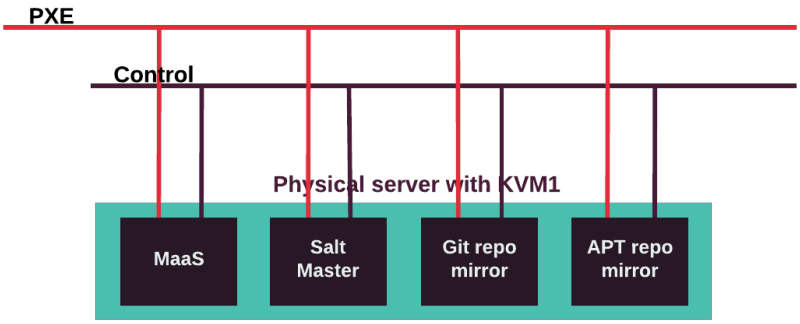
The MCP lifecycle management tool is provided as source code and configuration metadata stored in APT and Git repositories respectively. The Salt Master node requires access to both APT and Git repositories, while all other nodes in the environment require access to the APT repositories.

If you can provide an Internet connection to your OpenStack environment, you can use the repositories directly. However, if, for security or other reasons, your environment does not have an Internet connection, you need to configure local repository mirrors for both APT and Git repositories. Another option is to provide an Internet connection to the Salt Master node to access Git repository only while creating an APT repository mirror.

Local mirror design

You can place the APT and Git repository mirrors as virtual machines on the same KVM node where you run MaaS and Salt Master nodes.

The following diagram is an example of the virtual machines layout:



You will need to set up a local repository mirror and configure the APT repositories described in List of repositories. If your environment is completely isolated from the Internet, then you will also need to configure Git repositories with the required configuration files and provide Salt Master with the access to that repository.

You can use GitLab to configure Git mirror and Aptly for APT mirrors.

Seealso

- GitLab Repository Mirroring
- Aptly Mirror

List of repositories

The following table lists the APT repositories required for a MCP installation:

List of MCP APT repositories

Repositories	Description
http://archive.ubuntu.com/ubuntu/	(Required) Includes Ubuntu packages
http://apt.tcpcloud.eu/nightly	(Required) Includes TCP cloud nightly builds.
http://mirror.fuel-infra.org/mos-repos/ubuntu/<version>	(Required) MOS packages for the required version of Mirantis OpenStack. For example: http://mirror.fuel-infra.org/mos-repos/ubuntu/8.0-mu-2/
http://repo.saltstack.com/apt/ubuntu/14.04/amd64/2016.3	Includes SaltStack packages.

In addition to APT repositories that contain common configurations, you will need to configure Git repositories with the customer-specific metadata.

Infrastructure node planning

In large data centers, the services required for managing user workloads reside on separate servers from where the actual workloads run. The services that manage the workloads coupled with the hardware on which they run are typically called the control plane, while the servers that host user workloads are called the data plane.

In MCP, the control plane is hosted on the infrastructure nodes. Infrastructure nodes run all the components required for deployment, lifecycle management, and monitoring of your MCP cluster. A special type of infrastructure node called the foundation node, in addition to other services, hosts the bare-metal provisioning service called Metal-as-a-Service and the SaltStack Master node that provides infrastructure automation.

Overview

Infrastructure nodes are the nodes that run all required services for the MCP cluster deployment, lifecycle management, and monitoring.

The following main components run on the infrastructure nodes:

DriveTrain

DriveTrain includes containerized lifecycle management services, such as SaltStack, Reclass, Jenkins, Gerrit, and the MCP Registry, as well as Operational Support System tooling, including Rundeck automation, cloud health, cloud intelligence, and capacity management services. SaltStack and Reclass run on the foundation node only.

Logging, Metering, Alerting (LMA) or StackLight

The LMA toolchain includes containerized services required for cloud data collection and the visualization tools, such as Kibana and Grafana.

Provisioning tools (the foundation node only)

The foundation node includes the tools required for initial provisioning of your cloud environment, such as MaaS and Cookiecutter.

Virtualized Control Plane (VCP) (OpenStack environments only)

Virtualized Control Plane includes packaged-based OpenStack services such as Compute service, Networking service, and so on. VCP also includes other related services that enable cloud environment operation.

OpenContrail Controller (with Mirantis OpenContrail only)

The OpenContrail controller node includes package-based services of Mirantis OpenContrail, such as the API server and configuration database.

OpenContrail Analytics (with Mirantis OpenContrail only)

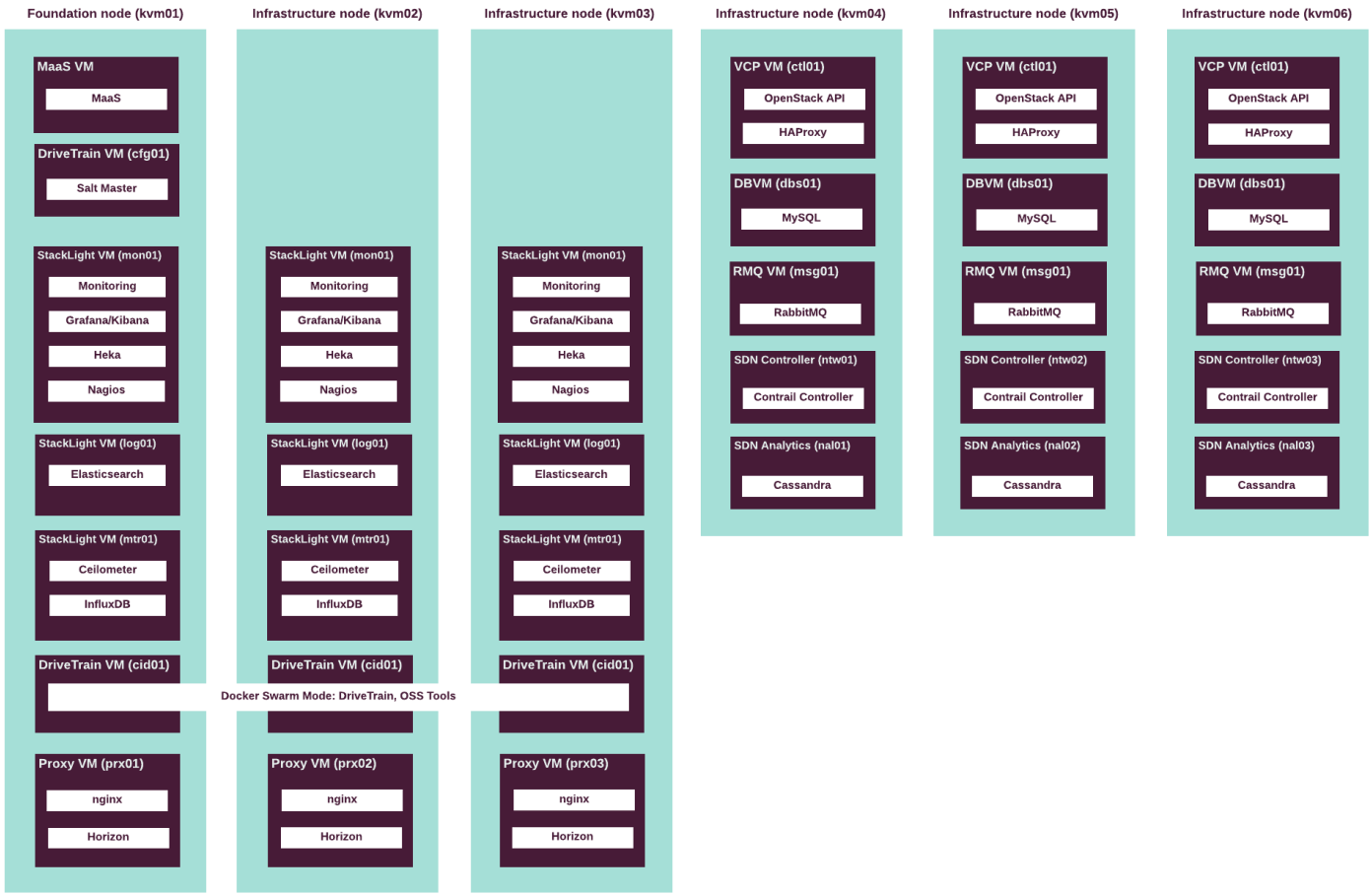
The OpenContrail analytics node includes package-based services for Mirantis OpenContrail metering and analytics, such as the Cassandra database for analytics and data collectors.

> Note
>
> If you use Open vSwitch Neutron plugin for tenant networking, all networking components run on the infrastructure nodes as virtual machines.

Network node (with Open vSwitch only)

Network nodes are required if you use Neutron OVS as your network solution. Network nodes run the Neutron L3 Routers through which the compute nodes traffic flows.

The following diagram displays the infrastructure and foundation node services mapping with Mirantis OpenContrail.



The following diagram displays the infrastructure and foundation node services mapping for tenant networking with Neutron Open vSwitch.

Multi-site and multi-cluster architecture

Mirantis Cloud Platform (MCP) can manage multiple disparate clusters using the same DriveTrain and infrastructure node installation. The cluster of environments might consist of the following types:

- OpenStack environments
- Kubernetes clusters

MCP provides the means to manage these sets of clusters using one DriveTrain installation over the L3 network. The cloud operator can execute such operations as applying the global configuration changes to a set of clusters or to an individual cluster, update cluster components, such as OpenStack services, and so on.

SaltStack uses a single data model structure to describe a multi-cluster and multi-site configuration of MCP. This structure resides in a single Git repository. Each cluster is defined by one directory in the repository directory tree. Each cluster has the site parameter that defines which site this cluster belongs to and allows to target Salt states to specific sites or subsets of sites.

SaltStack distributes appropriate changes to targeted sites, clusters, and nodes.

DriveTrain overview

DriveTrain is a set of tools and services that enable you to deploy, monitor, and manage your cloud environment. The DriveTrain components run on the infrastructure nodes alongside with the Virtualized Control Plane (VCP) and other components.

DriveTrain implements the Infrastructure-as-Code (IaC) technology that treats provisioning and lifecycle management of a compute infrastructure through the use of source code and through applying changes to the source code through a CI/CD pipeline.

DriveTrain includes the following components:

- Lifecycle management tools that enable cloud administrators to apply configuration changes to a cloud environment. The lifecycle management tools are a part of the CI/CD pipeline.

  The lifecycle management tools include:

  - SaltStack
  - Reclass
  - Jenkins
  - MCP registry
  - Gerrit
- Operational Support System (OSS) tooling that provides support for maintenance and day-to-day operations in your cloud.

  OSS tooling includes:

  - Cloud Intelligence service (CIS)
  - Cloud Health service
  - Runbooks

CI/CD pipeline overview

The MCP Continuous Integration and Continuous Deployment (CI/CD) pipeline enables the delivery of configuration changes and updates to your MCP cluster.

The MCP CI/CD pipeline overview includes the following components:

Gerrit

Stores the source code, SaltStack formulas, and Reclass models, as well as provides code review capabilities.

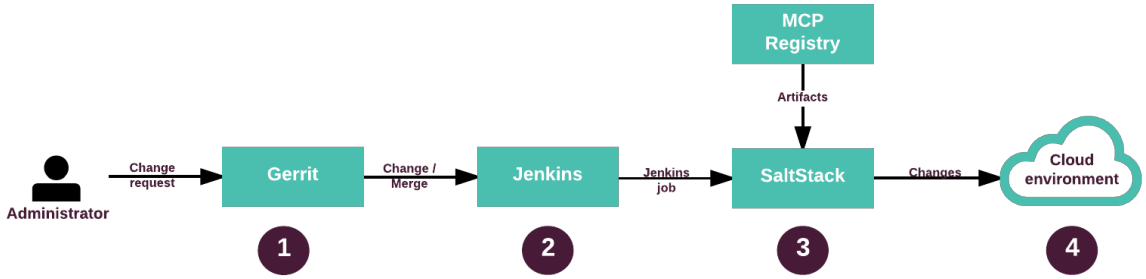
Jenkins

Detects the changes submitted to the MCP cluster configuration through Gerrit and executes the jobs that run the related SaltStack formulas and Reclass models to apply the changes.

MCP Registry

Stores the software artifacts, such as Docker images and Debian packages, required for MCP clusters.

The following diagram describes the workflow of the CI/CD pipeline:



CI/CD pipeline workflow

#	Description
1	An operator submits changes to a Reclass model or a SaltStack formula in Gerrit for review.
2	Depending on your configuration and whether you have a staging environment or deploy changes directly to a production MCP cluster, the workflow might slightly differ. Typically, if you have a production MCP cluster, Jenkins triggers a deployment job only after you merge your changes. However, you have a staging environment, your changes may be applied at the moment of submission.
3	Jenkins triggers a job that invokes the required SaltStack formulas and Reclass models from Gerrit and artifacts from the MCP Registry.
4	SaltStack applies changes to the cloud environment.

DevOps portal

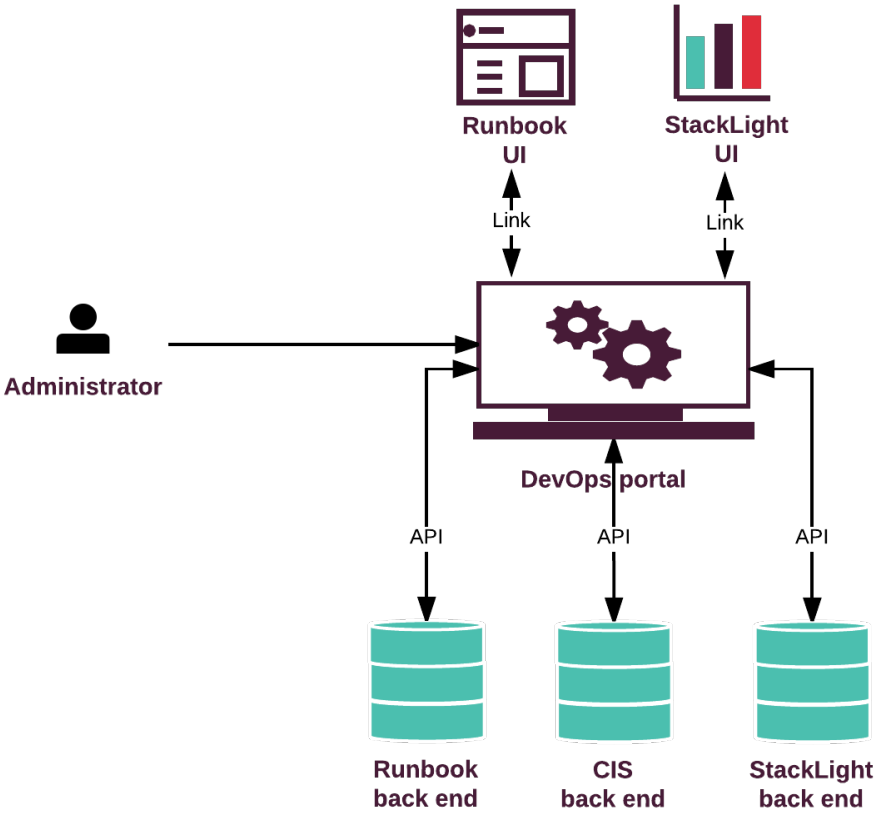
The DevOps portal is the main entry point for users who manage the Mirantis Cloud Platform (MCP) The main goal of the DevOps portal is to provide information about the cloud management status. In that sense, the DevOps portal acts as a main administrative dashboard for the entire MCP environment.

> Note
>
> The DevOps portal is provided as a technical preview.

The DevOps portal performs the following main tasks:

- Displays information about the cloud in the form of graphs, statuses, success dashboards, and so on.
- Provides links to other sets of tools, including Grafana, Kibana, and Rundeck.

The following diagram shows a high level architecture of the DevOps portal components.

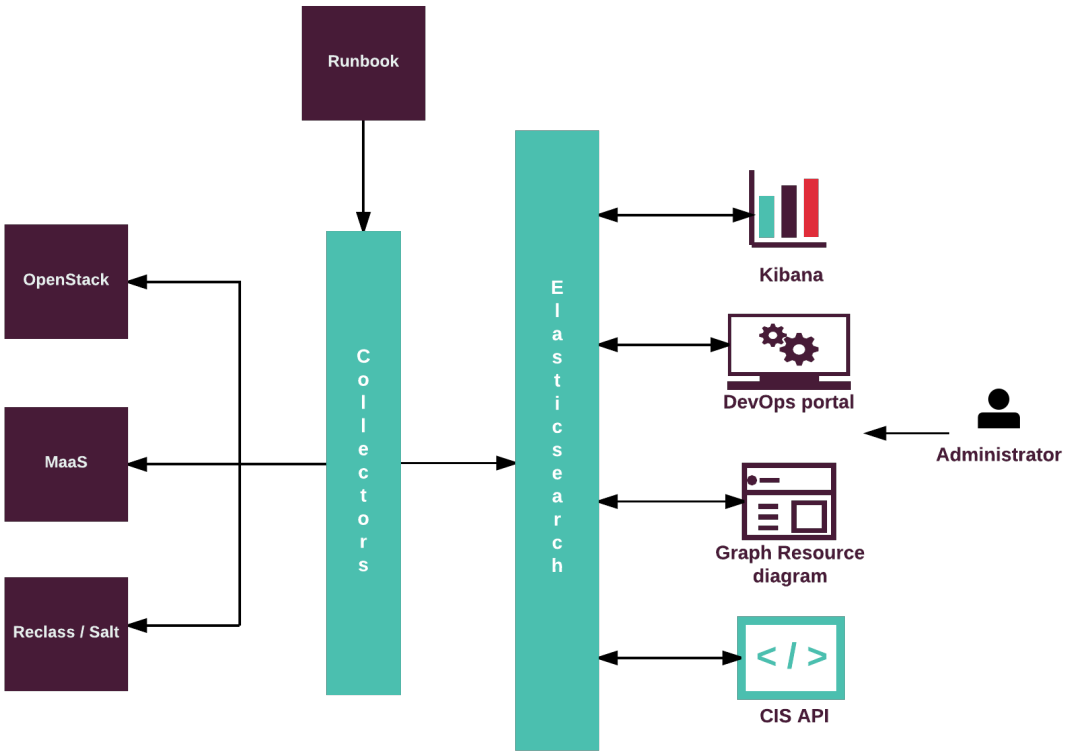


Cloud Intelligence Service overview

The Cloud Intelligence Service (CIS) consists of a set of collectors that gather information on the service level, including OpenStack, MaaS, and so on. CIS stores that information, tracks changes, as well categorizes that information for easier consumption by users or other services. CIS can query services APIs, as well as connect to specific databases in order to accelerate data collection. Although CIS has powerful search capabilities, it cannot modify any of the resources.

The DevOps portal provides a user interface, as well as an API for CIS through which users can create search queries that CIS submits to the search engine and displays the list of resources, outputs, and graphs.

The following diagram displays the CIS high level architecture.



Runbook runs the collectors that gather information about the cloud resources every five minutes and stores them in Elasticsearch. The cloud operator queries Cloud Intelligence Service information through the UI or API.

Minimum hardware requirements for CIS:

- Virtual machines - 3
- RAM - 8 GB
- Disk 50 GB
- vCPU - 2

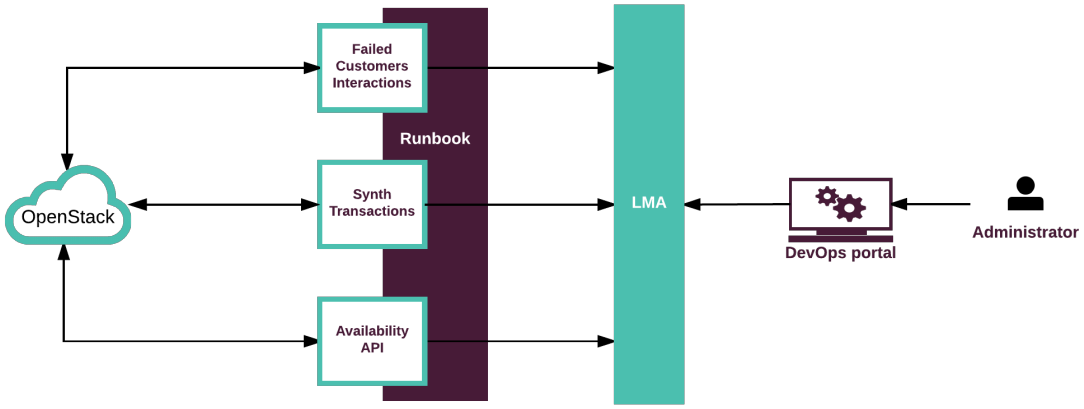
In a multi-cloud environment, that includes multiple OpenStack isolations, the DevOps portal runs one copy of the collectors per an OpenStack isolation. Such approach enables CIS to collect data from multiple resources while keeping them separate, so the data from one OpenStack environment is not mixed with the data from the other OpenStack environment.

Cloud Health Service overview

Cloud health includes a set of collectors that provide a high level overview of the cloud status through the notifications about network, storage, or compute nodes outages, the control plane status changes, and so on. Cloud health functions similarly to a monitoring system in which collectors perform specific tests to verify Failed Customer Interactions (FCI), as well as test

cloud API availability and response time. The results are displayed in the DevOps portal. When the FCI graph goes to zero, it typically signifies a problem in the system.

The following diagram describes the high level architecture of the Cloud Health Service:

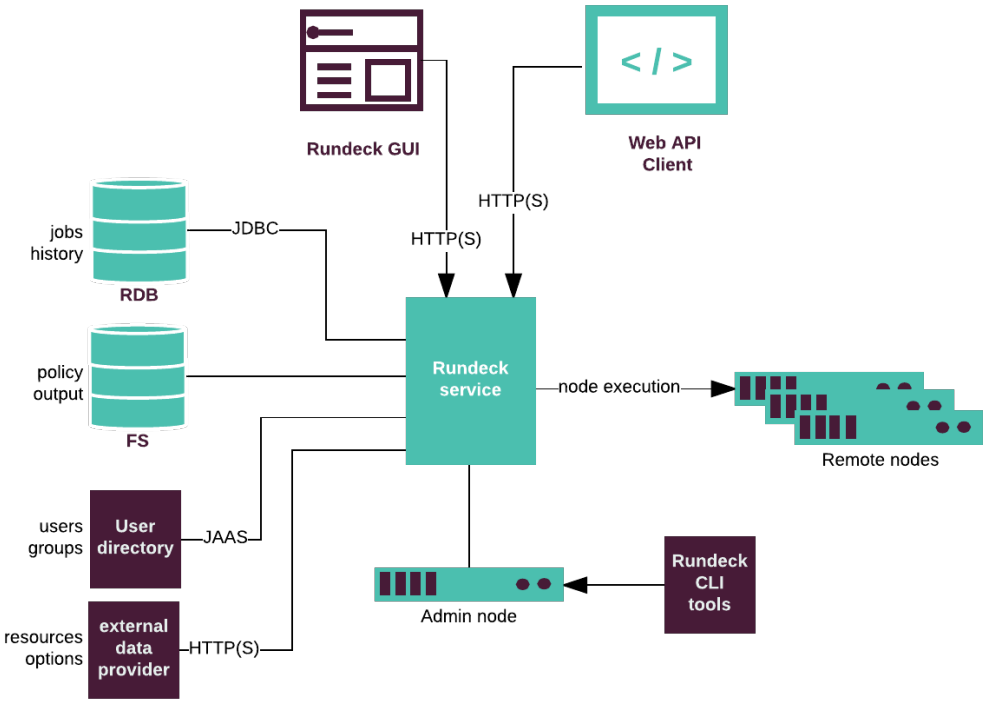


Runbook executes FCI, synth transactions and API tests and stores the corresponding data in the Logging, Metering, and Alerting (LMA) toolchain. The DevOps portal queries LMA and creates graphs and heatmaps for the cloud status. The cloud operator monitors the heatmaps and graphs through the DevOps portal.

Runbooks Automation overview

Runbooks Automation is an automation service that enables users to create a workflow of jobs that are executed at a specific time or interval The OSS components use the Runbooks automation service in order to automate such tasks as backup creation, metrics querying, report generation, cloud maintenance, and so on. Runbooks automation is provided through a tool called Rundeck which enables the users easily add scripts as Rundeck jobs and chain them into workflows. While Jenkins and the SaltStack LCM engine are mainly used for deployment and lifecycle management, Rundeck enables users to perform day-to-day operations and maintenance tasks.

The following diagram illustrates the architecture of the Runbooks Automation service:



Minimum hardware requirements for the Runbooks Automation service include:

- Virtual Machines - 3
- RAM - 8GB
- Disk - 50 GB
- CPU - 2

In a multi-cloud environment, with multiple OpenStack isolations, the Runbook Automation service connects to multiple nodes on one or more clouds. Nodes are filtered and specified on each job.

Capacity Management Service overview

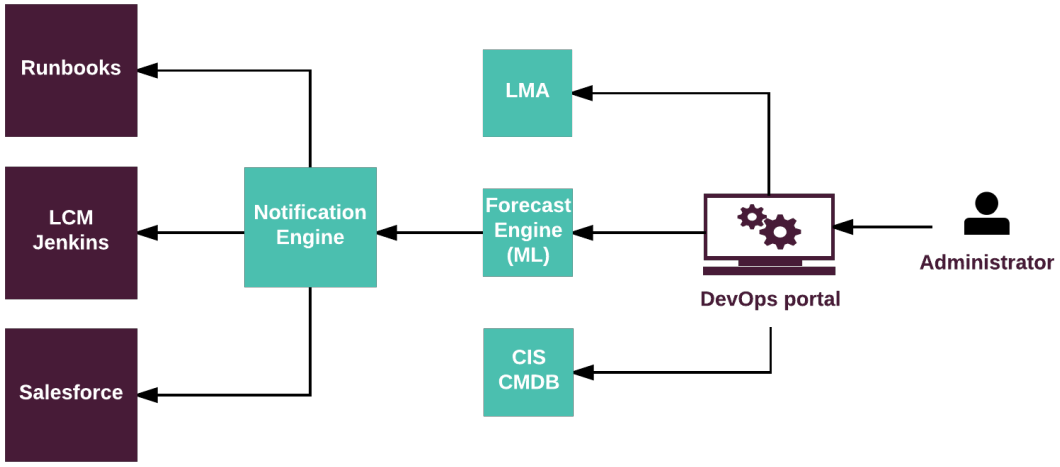
The Capacity Management Service comprises multiple dashboards that provide an overview of the resource utilization in the cloud. The leverages service the data collected by the LMA and CIS systems.

The Capacity Management Service dashboards include the following and other metrics:

- CPU utilization by hypervisor grouped by availability zones
- Total amount of RAM (GB) in use
- Total amount of storage (GB) in use
- Memory utilization by hypervisor grouped by availability zones

- Total number of hypervisors

The following diagram displays the high level Capacity Management Service architecture:



The Forecast Engine sends notifications with the required actions, such as add more nodes or clean up the storage, and creates a ticket in the corresponding task management system, such as Salesforce. The Forecast Engine pulls data from LMA and CIS and predicts the future resource utilization. The cloud operator accesses graphs and heatmaps through the DevOps portal that displays the capacity of the cloud.

High availability in DriveTrain

DriveTrain is one of the critical components of the MCP solution. Therefore, its continuous availability is essential for the MCP solution to function properly. Although, you can deploy DriveTrain in the single node Docker Swarm mode for testing purposes, most production environments require a highly-available DriveTrain installation.

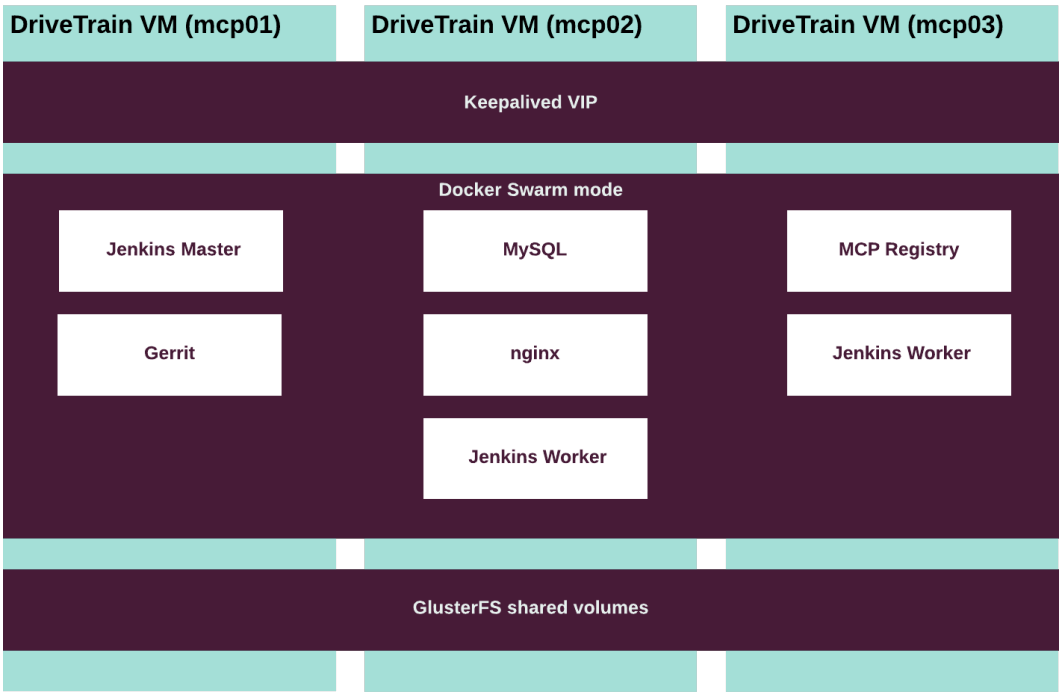
All DriveTrain components run as containers in Docker Swarm mode which ensures services are provided continuously without interruptions and are susceptible to failures.

The following components ensure high availability of DriveTrain:

- Docker Swarm mode is a special Docker mode that provides Docker cluster management. Docker Swarm ensures:
  - High availability of the DriveTrain services. In case of failure on any infrastructure node, Docker Swarm reschedules all services to other available nodes. GlusterFS ensures the integrity of persistent data.
  - Internal network connectivity between the Docker Swarm services through the Docker native networking.
- Keepalived is a routing utility for Linux that provides a single point of entry for all DriveTrain services through a virtual IP address (VIP). If the node on which the VIP is active fails, Keepalived fails over the VIP to other available nodes.

- nginx is web-server software that exposes the DriveTrain service's APIs that run in a private network to a public network space.
- GlusterFS is a distributed file system that ensures the integrity of the MCP Registry and Gerrit data by storing the data in a shared storage on separate volumes. This ensures that persistent data is preserved during the failover.

The following diagram describes high availability in DriveTrain:



Plan an OpenStack environment

MCP enables you to deploy one or multiple OpenStack environments to address the needs of your data center.

Coupled together with the deployment automation, native logging, monitoring, and alerting component, as well as with support for OpenContrail and Open vSwitch networking, an MCP OpenStack environment represents a reliable, scalable, and flexible cloud solution that supports numerous types of workloads and applications.

Virtualized control plane planning

The MCP virtualized control plane (VCP) provides all services and components required to manage your cloud. When planning a physical and virtual servers that will run VCP services, consider the size of the OpenStack environment, redundancy requirements, and the hardware you plan to use.

Virtualized control plane overview

Virtualized control plane (VCP) consists of the services required to manage workloads and respond to API calls. VCP is the heart and brain of your OpenStack deployment that controls all logic responsible for OpenStack environment management. To ensure high availability and fault tolerance, VCP must run on at least three physical nodes. However, depending on your hardware you may decide to break down the services on a larger number of nodes. The number of virtual instances that must run each service may vary as well. See Virtual control plane requirements.

Initially, you may add a minimum number of virtual instances required for each service to the VCP and later increase or decrease this number as needed.

The following tables describes the MCP virtualized control plane.

VCP services

Platform	Service
OpenStack	• Identity service (Keystone) • Image service (Glance) • Compute service (Nova) • Networking service (Neutron OVS or OpenContrail plugin) • Dashboard (Horizon) • (optional) Block Storage service (Cinder)
Bare-metal provisioning	MaaS
Configuration management service	
Storage	Ceph • Ceph monitors • RadosGW
Networking	OpenContrail • Control Config DB • Config • Analytics • DB

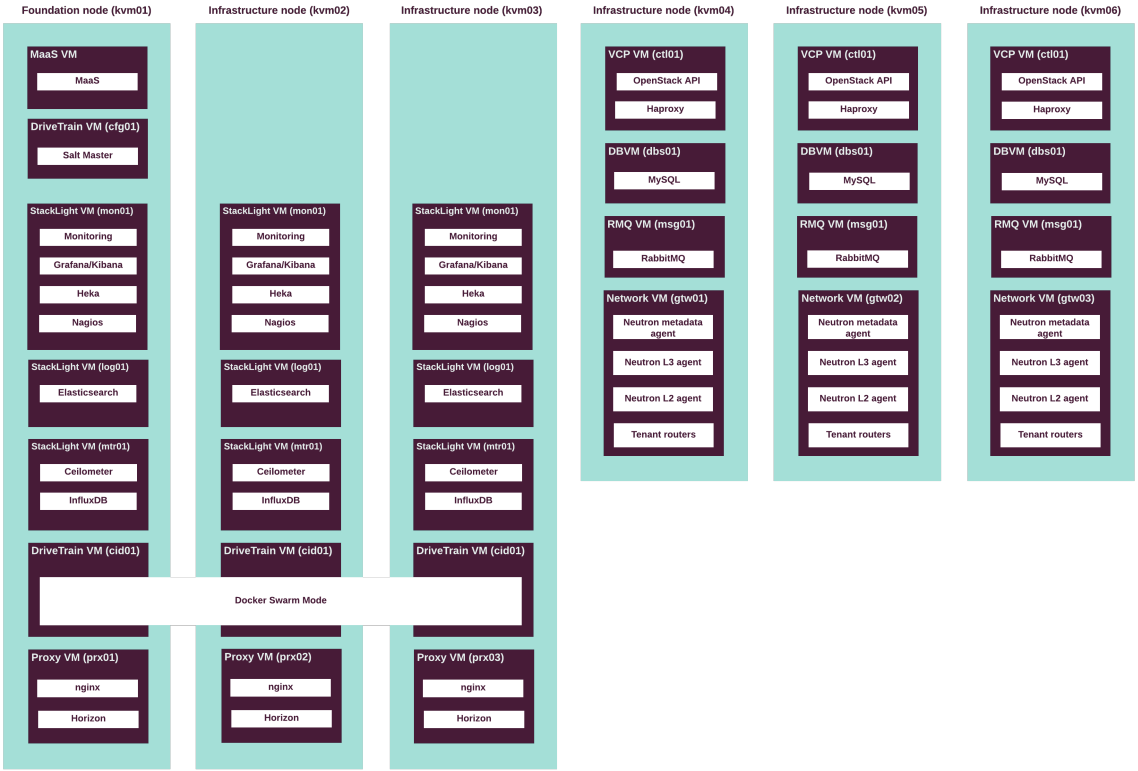
Back-end services	• Proxy (NGINX) • GlusterFS • RabbitMQ • MySQL/Galera
(optional) Logging, Metering, and Alerting (LMA)	• StackLight

Minimum virtualized control plane design

The SaltStack cluster class describes the minimum virtualized control plane configuration that is the deployment of the VCP services on three physical servers. This configuration can be altered according to the needs of your cloud environment and the hardware you use.

For scale out purposes, Mirantis recommends dedicating three additional servers for MySQL / Galera and three servers for RabbitMQ. The LMA toolchain may also be run on three additional servers.

The following diagram describes the minimum VCP design.



Seealso

- Example of virtualized control plane design
- Virtual control plane requirements

Virtual control plane requirements

Depending on the environment size, a certain number of virtual instances is required for each VCP service. The following table lists the minimum virtual control plane requirements for an OpenStack environment that runs up to 500 virtual instances.

Virtual control plane requirements

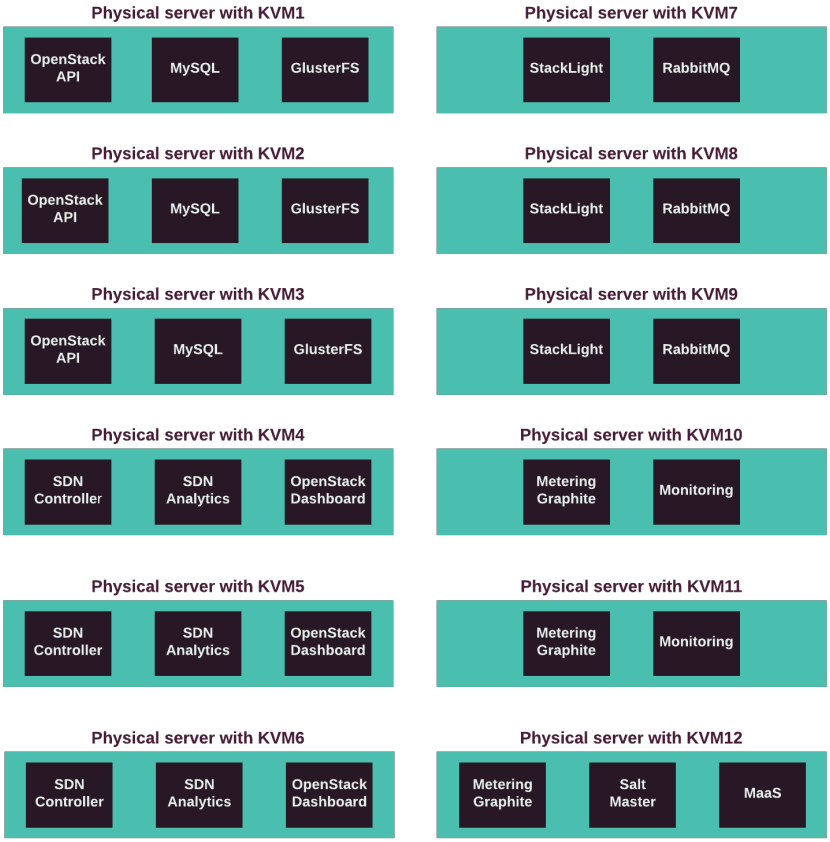
Component	Number of nodes	vCPU	Virtual Memory (GB)	Virtual Disk (GB)	vNICs	Scale
RabbitMQ	3	4	8	50	2	Vertical. Horizontal scaling may decrease performance.
MySQL	3	4	8	80	2	Horizontal and vertical. Horizontal scaling must be limited to about 5 members.
OpenContrail Controllers	3	4	16	50	2	Horizontal for configuration database Cassandra. The database does not include analytics.
OpenContrail Analytics	3	4	12	300	2	Horizontal and vertical, analytics database (Cassandra).
Logging, Metering, Alerting	3	8	16	300	2	Vertical
OpenStack Dashboard	2	4	8	50	3	Horizontal and vertical.
OpenStack controllers	3	8	16	100	4	Horizontal: Compute service, Image service, Block Storage service, Identity service, Orchestration service, Hadoop service, Vertical: HAProxy
GlusterFS	3	2	4	300	2	Vertical
Benchmark	1	2	4	80	2	Vertical

Example of virtualized control plane design

This section provides an example of VCP design that includes 12 physical nodes. Multiple virtual instances of each component spread across the physical servers ensuring redundancy and fault tolerance. Each physical node runs a KVM hypervisor and the virtual instances run on top of KVM.

> Note
>
> This example is for your reference only. It is not a recommended layout, but a design used for a particular environment and particular hardware. Your use case may or may not completely differ from this example.

The following diagram describes how VCP services are distributed among physical servers in the provided example.



Compute nodes planning

Determining the appropriate hardware for the compute nodes greatly depends on the workloads, number of virtual machines, and types of applications that you plan to run on the nodes. Typically, you need a two-socket server with the CPU, memory, and disk space that meet your project requirements.

That said, it is essential to understand your cloud capacity utilization tendencies and patterns to plan for expansion accordingly. On one hand, planning expansion too aggressively may result in underuse and financial losses for the company, while underestimating expansion trends threatens oversubscription and eventual performance degradation.

Mirantis provides a spreadsheet with the compute node calculation. You need to fill the following parameters in the spreadsheet:

Compute nodes planning

Parameter	Description
Overhead components	Describe components that put additional overhead on system resources, such as DVR/vRouter and Hypervisor. The parameters specified in the spreadsheet represent standard workloads. The DVR / vRouter parameters represent a Compute node with 2 x 10 Gbps NICs. If you use a larger capacity network interfaces, such as 40 Gbps, this number may increase. For most deployments, the hypervisor overhead parameters equal represented numbers.
HW Components	Compute profile represents the hardware specification that you require for the specified number of virtual machines and the selected flavor. The adjusted version of the compute profile represents the hardware specification after correction to overhead components.
Oversubscription ratio	Defines the amount of virtual resources to allocate for a single physical resource entity. Oversubscription ratio highly depends on the workloads that you plan to run in your cloud. For example, Mirantis recommends to allocate 8 vCPU per 1 hyper-thread CPU, as well as 1:1 ratio for both memory and disk for standard workloads, such as web application development environments. If you plan to run higher CPU utilization workloads, you may need to decrease CPU ratio down to 1:1.
Flavor definitions	Defines a virtual machine flavor that you plan to use in your deployment. The flavor depends on the workloads that you plan to run. In the spreadsheet, the OpenStack medium virtual machine is provided as an example.
Flavor totals	Defines the final hardware requirements based on specified parameters. Depending on the number and the virtual machine flavor, you get the number of compute nodes (numHosts) with the hardware characteristics.

Resource utilization per compute node	The resource utilization parameter defines the percentage of memory, proccessing, and storage resource utilization on each compute node. Mirantis recommends that vCPU, vMEM, and vDISK are utilized at least at 50 %, so that your compute nodes are properly balanced. If your calculation results in less than 50 % utilization, adjust the numbers to use the resources more efficiently.

Seealso

- Download Compute nodes calculation

Network planning

Depending on the size of the environment and the components that you use, you may want to have a single or multiple network interfaces, as well as run different types of traffic on a single or multiple VLANs.

Selecting a network technology

Mirantis Cloud Platform supports the following network technologies:

- OpenContrail
- Neutron Open vSwitch

The following table compares the two technologies and defines use cases for both:

OpenContrail vs Neutron OVS

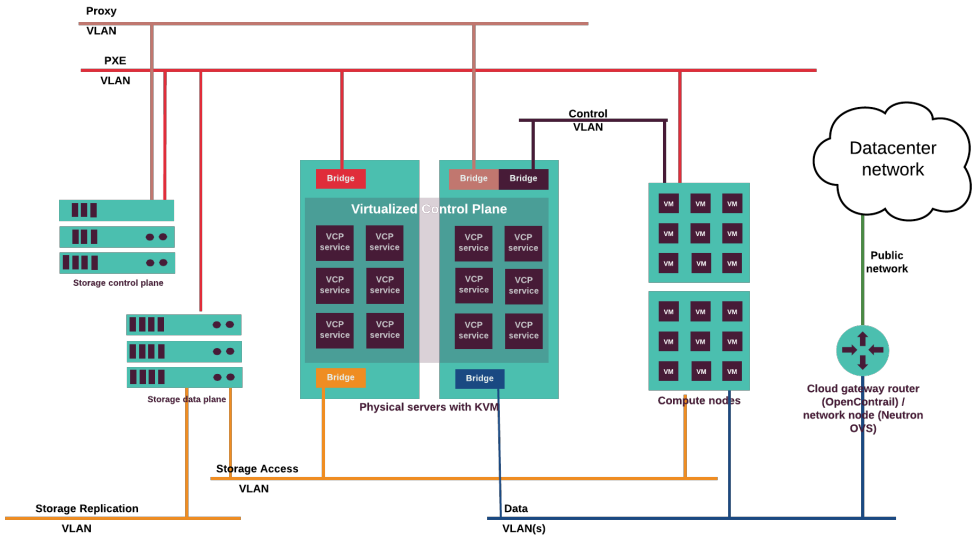
OpenContrail	Neutron OVS

Mirantis recommends OpenContrail for both staging and production environments. It provides both basic networking, such as IP address management, security groups, floating IP addresses, and advanced networking functions, including DPDK network virtualization and SR-IOV.
The following functionality is supported by OpenContrail only:
• Service chaining
• MPLS over UDP/GRE with vMX router
• Multi-site SDN
• Network analytics
	If you cannot use OpenContrail due to hardware or other limitations, your other option is to use Neutron OVS. Neutron OVS supports the following use cases:
	• L3 fabric:
	  • Neutron VXLAN tenant networks with network nodes without DVR
	  • Neutron VXLAN tenant networks with network nodes with DVR east-west and a network node for north-south traffic.
	• The following use cases require L2 networks for external provider networks to be available across all compute nodes. Therefore, these use cases are not for multi-rack strategy or L3 fabric networking:
	  • Neutron VLAN tenant networks with network nodes without DVR
	  • Neutron VXLAN tenant networks with network nodes for SNAT (DVR for both)
	  • Neutron VLAN tenant networks with network nodes for SNAT (DVR for both)

Types of networks

When planning your OpenStack environment, consider what types of traffic your workloads generate and design your network accordingly. If you anticipate that certain types of traffic, such as storage replication, will likely consume a significant amount of network bandwidth, you may want to move that traffic to a dedicated network interface to avoid performance degradation.

The following diagram provides an overview of the underlay networks in an OpenStack environment:



In an OpenStack environment, you typically work with the following types of networks:

- Underlay networks for OpenStack that are required to build network infrastructure and related components. Underlay networks include:
  - PXE / Management

    This network is used by SaltStack and MaaS to serve deployment and provisioning traffic, as well as to communicate with nodes after deployment. After deploying an OpenStack environment, this network runs low traffic. Therefore, a dedicated 1 Gbit network interface is sufficient. The size of the network also depends on the number of hosts managed by MaaS and SaltStack. Although not required, routing significantly simplifies the OpenStack environment provisioning by providing a default gateway to reach APT and Git repositories.
  - Public

    Virtual machines access the Internet through Public network. Public network provides connectivity to the globally routed address space for VMs. In addition, Public network provides a neighboring address range for floating IPs that are assigned to individual VM instances by the project administrator.
  - Proxy

    This network is used for network traffic created by Horizon and for OpenStack API access. The proxy network requires routing. Typically, two proxy nodes with

Keepalived VIPs are present in this network, therefore, the /29 network is sufficient. In some use cases, you can use Proxy network as Public network.

- Control

  This network is used for internal communication between the components of the OpenStack environment. All nodes are connected to this network including the VCP virtual machines and KVM nodes. OpenStack components communicate through the management network. This network requires routing.

- Data

  This network is used by OpenContrail to build a network overlay. All tenant networks, including floating IP, fixed with RT, and private networks, are carried over this overlay (MPLSoGRE/UDP over L3VPN/EVPN). Compute and Juniper MX routers connect to this network. There are two approaches to organizing data network: flat VLAN and L3 separation. Flat VLAN presumes that you have one L2 domain that includes all compute nodes and vMXs. In this case, this network does not require routing. The L3 separation presumes that groups of compute nodes and vMX routers are located in different L3 networks and, therefore, require routing.

- Storage access (optional)

  This network is used to access third-party storage devices and Ceph servers. The network does not need to be accessible from outside the cloud environment. However, Mirantis recommends that you reserve a dedicated and redundant 10 Gbit network connection to ensure low latency and fast access to the block storage volumes. You can configure this network with routing for L3 connectivity or without routing. If you set this network without routing, you must ensure additional L2 connectivity to nodes that use Ceph.

- Storage replication (optional)

  This network is used for copying data between OSD nodes in a Ceph cluster. Does not require access from outside the OpenStack environment. However, Mirantis recommends reserving a dedicated and redundant 10 Gbit network connection to accommodation high replication traffic. Use routing only if rack-level L2 boundary is required or if you want to configure smaller broadcast domains (subnets).

- Virtual networks inside OpenStack

  Virtual network inside OpenStack include virtual public and internal networks. Virtual public network connects to the underlay public network. Virtual internal networks exist within the underlay data network. Typically, you need multiple virtual networks of both types to address the requirements of your workloads.

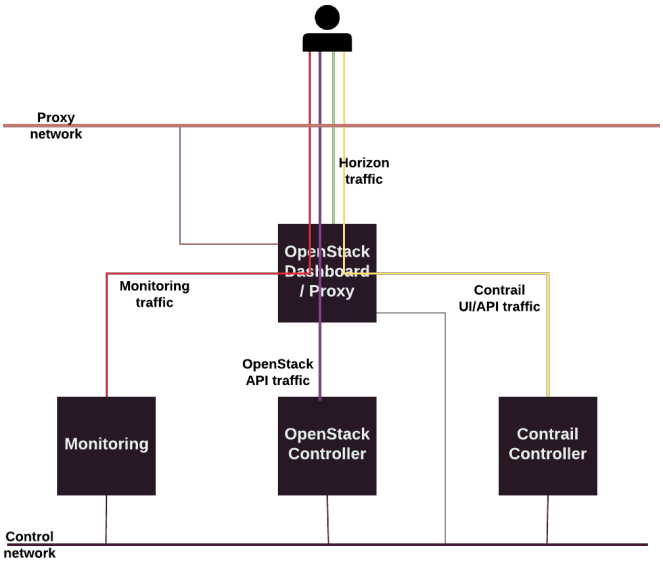
Disregarding the size of your cloud, you must have isolated virtual or physical networks for PXE, Proxy, and all other traffic. At minimum, allocate one 1 Gbit physical network interface for PXE network, and two bonded 10 Gbit physical network interfaces for all other networks. Allocate VLANs on the bonded physical interface to isolate Proxy, Data, and Control logical networks. All other networks are optional and depend on your environment configuration.

OpenContrail traffic flow

This section provides diagrams that describe types of traffic and the directions of traffic flow in an OpenStack environment.

User Interface and API traffic

The following diagram displays all types of UI and API traffic in an OpenStack environment, including Horizon, monitoring, OpenStack API, and OpenContrail UI/API traffic. The Openstack Dashboard node hosts Horizon and acts as proxy for all other types of traffic. SSL termination occurs on this node as well.

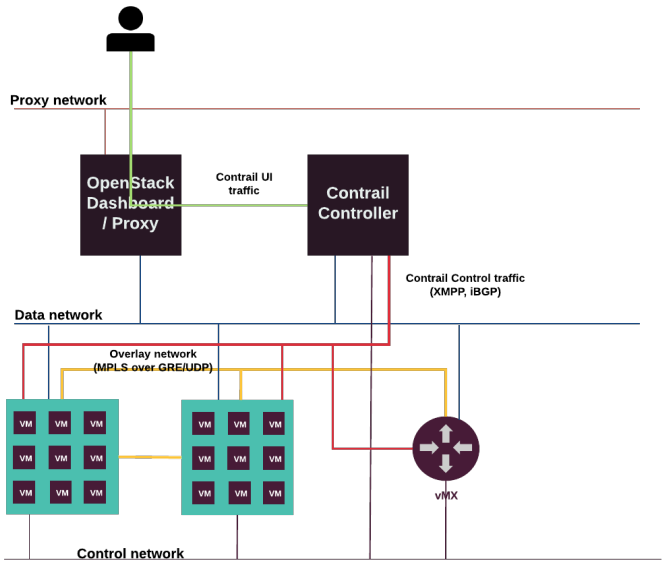


SDN traffic

SDN or OpenContrail traffic goes through the overlay Data network and processes east-west and north-south traffic for applications that run in an OpenStack environment. This network segment typically contains tenant networks as separate MPLS over GRE and MPLS over UDP tunnels. The traffic load depends on workload.

The control traffic between OpenContrail controllers, edge routers, and vRouters use iBGP and XMPP protocols. Both protocols produce low traffic which does not affect the MPLS over GRE and MPLS over UDP traffic. However, this traffic is critical and must be reliably delivered. Mirantis recommends configuring higher QoS for this type of traffic.

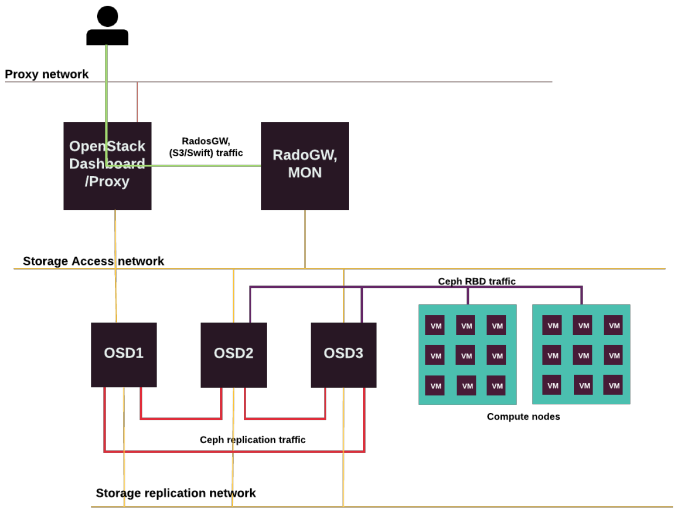
The following diagram displays both MPLS over GRE / MPLS over UDP and iBGP and XMPP traffic examples:



Storage traffic

Storage traffic flows through dedicated storage networks that Mirantis recommends to configure if you use Ceph.

The following diagram displays the storage traffic flow for Ceph RBD, replication, and RadosGW.



Neutron OVS use cases

Neutron OVS applies to a number of use cases. This section provides traffic flow diagrams, as well as compute and network nodes configurations for all use cases.

Neutron OVS requires you to set up a specific network node, which is sometimes called gateway that handles the routing across the internal networks, as well as the outbound routing.

Node configuration

For all Neutron OVS use cases, configure four VLANs and four IP addresses in separate networks on all compute and network nodes. You will also need two VLAN ranges for tenant traffic and external VLAN traffic.

The following table lists node network requirements:

Node network configuration

Port	Description	IP Address	VLAN
br-mesh	Tenant overlay traffic (VXLAN)	Routed, Subnet	Leaf switch only
br-mgmt	Openstack and other management traffic	Routed, Subnet	Leaf switch only
br-stor	Storage traffic	Routed, Subnet	Leaf switch only
eth0	PXE Boot traffic	VLAN, Subnet, Default	Global
br-prv	Tenant VLAN traffic bridge	VLAN range	Global
br-floating	External VLAN traffic bridge	VLAN range	Global

VCP servers network interfaces

Each physical server that hosts KVM on which Virtualized Control Plane (VCP) services run must have the following network configuration:

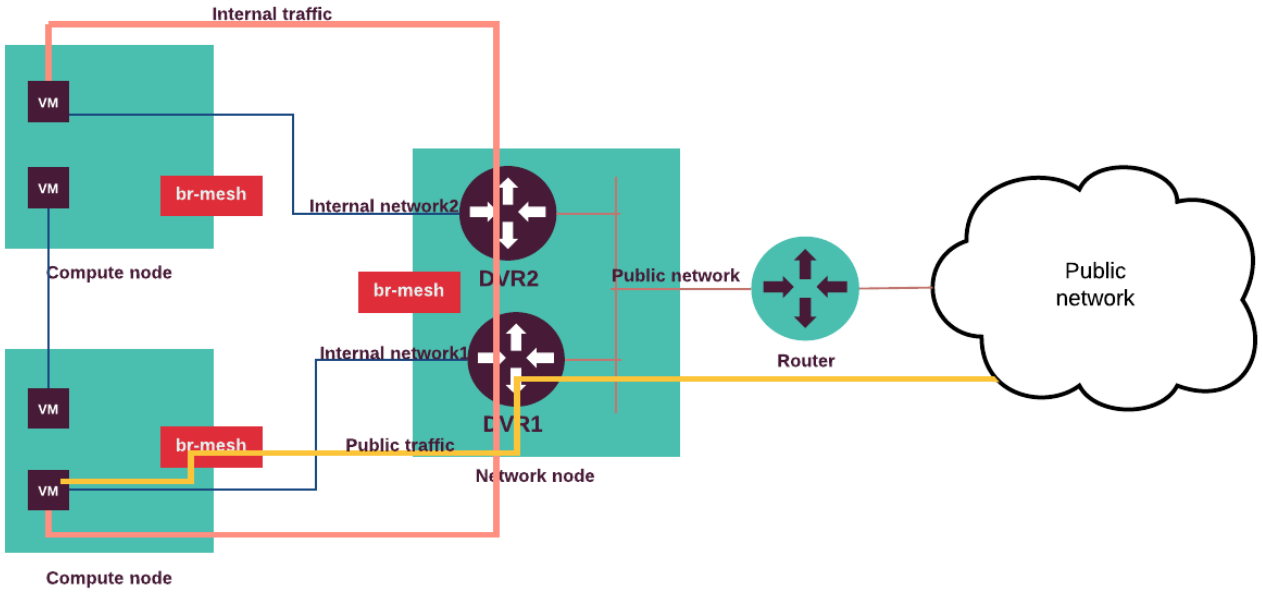
VCP servers network configuration

Port	Description	IP Address	VLAN
br-pxe	PXE Boot traffic	VLAN, Subnet	Global
br-mgmt	Openstack and other management traffic	Routed, Subnet	Leaf switch only
br-stor	Storage traffic	Routed, Subnet	Leaf switch only
eth0	PXE Boot traffic	VLAN, Subnet, Default	Global

Neutron VXLAN tenant networks with network nodes (no DVR)

If you configure your network with Neutron OVS VXLAN tenant networks with network nodes and without a Distributed Virtual Router (DVR) on the compute nodes, all routing happens on the network nodes. This is a very simple configuration that typically works for test clouds and production environments with little tenant traffic. The disadvantage of this approach is that internal traffic from one virtual internal network to the other virtual internal network has to go all the way to the network node rather than being transmitted directly through the data network as when you use a DVR. This results in networks nodes becoming a performance bottleneck for the tenant traffic.

The following diagram displays internal and external traffic flow.

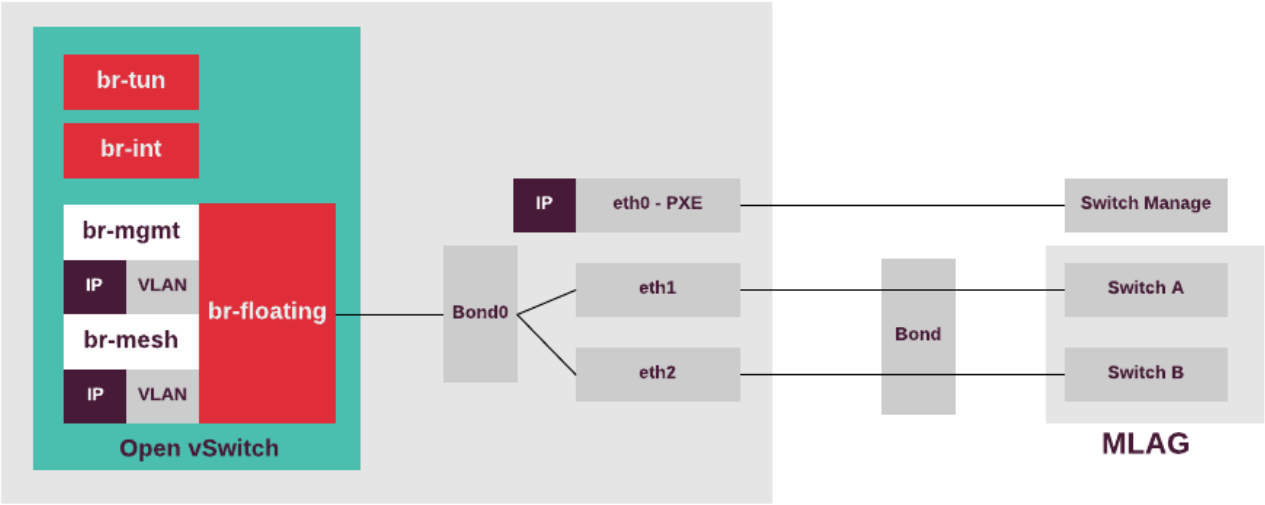


The internal traffic from one tenant virtual machine located on virtual Internal network 1 goes to another virtual machine located in the Internal network 2 through the DVR1 and DVR2 on the network node to the target VM. The external traffic from a virtual machine goes through the Internal network 1 and the tenant VXLAN (br-mesh) to the DVR 1 on the network node and the Public network to the outside network.

Network node configuration

In this use case, the network node terminates VXLAN mesh tunnels and sends traffic to external provider VLAN networks. Therefore, all tagged interfaces must be configured directly in Neutron OVS as internal ports without Linux bridges. Bond0 is added into br-floating, which is mapped as physnet1 into the Neutron provider networks. br-mgmt and br-mesh are Neutron OVS internal ports with tags and IP addresses. As there is no need to handle storage traffic on the network nodes, all the sub-interfaces can be created in Neutron OVS. This also allows for the creation of VLAN providers through the Neutron API.

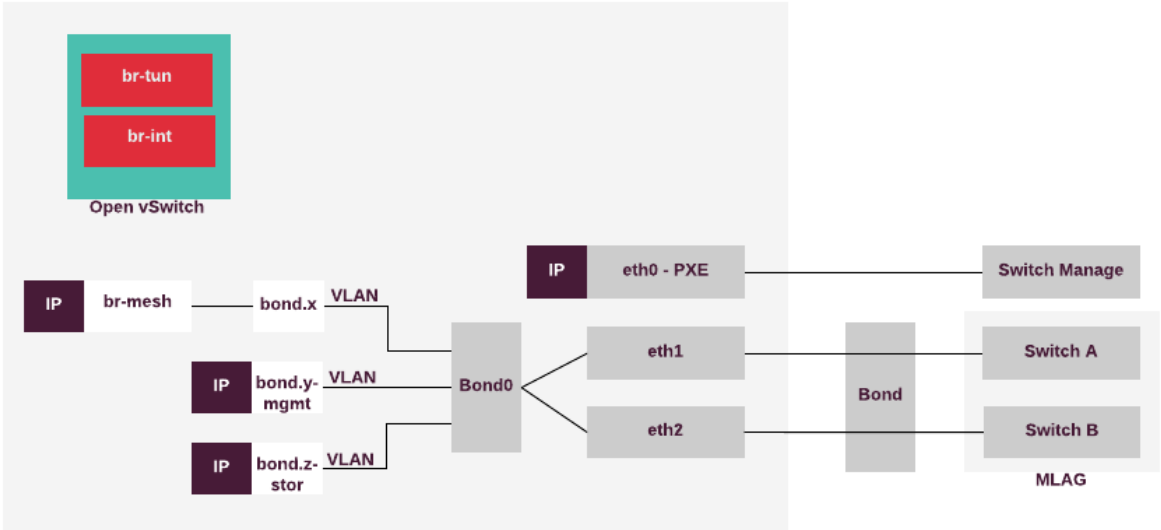
The following diagram displays network node configuration for the use case with Neutron VXLAN tenant networks with network nodes and DVRs configured on the network node only.



Compute nodes configuration

In this use case, compute nodes do not have access to the external network. Therefore, you configure a Linux bridge br-mesh, which is unidirectionally connected with br-tun and used for VXLAN tunneling. All Open vSwitch bridges are automatically created by the Neutron OVS agent. For a highly-available production environment, network interface bonding is required. The separation of types of traffic is done by bonded tagged sub-interfaces, such as bond.y for the virtualized control plane traffic (mgmt IP), bond.x for data plane bridge (br_mesh) which provides VTEP for OVS and bond.z for storage etc. IP address of br-mesh is used as local IP in the openvswitch.ini configuration file for tunneling.

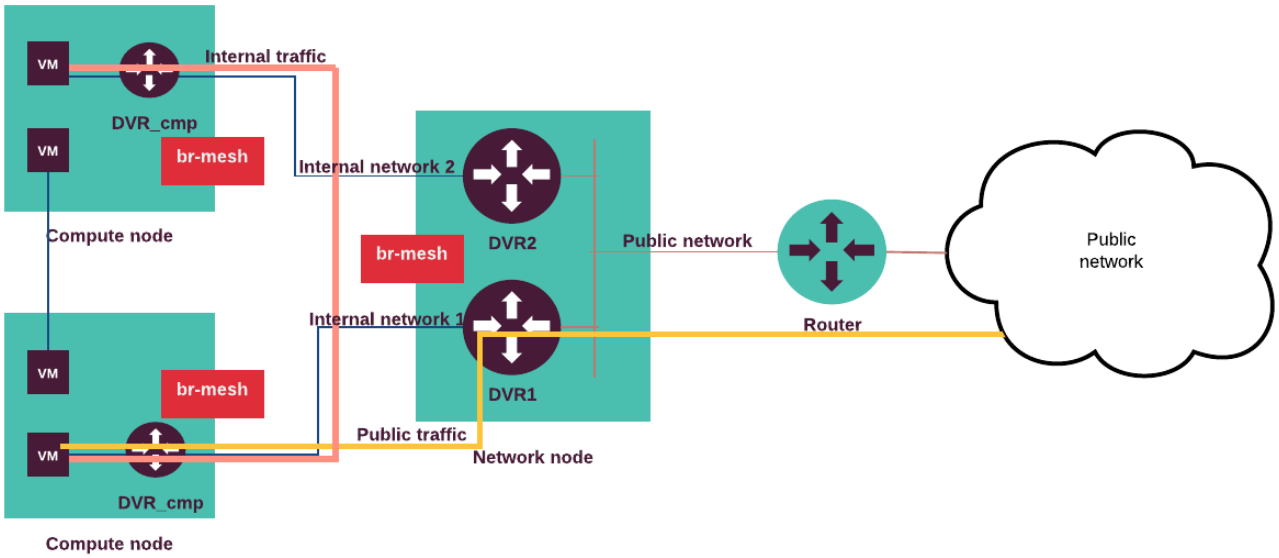
The following diagram displays compute nodes configuration for the use case with Neutron VXLAN tenant networks with network nodes and DVRs configured on the network node only.



Neutron VXLAN tenant networks with DVR for internal traffic

If you configure your network with Neutron OVS VXLAN tenant networks with network nodes and a Distributed Virtual Router (DVR) for internal traffic, DVR routers are only used for traffic that is routed within the cloud infrastructure and that remains encapsulated. All external traffic is routed through the network nodes. Each tenant requires at least two 2 DVRs - one for internal (East-West) traffic and the other for external (North-South) traffic. This use case is beneficial for the environments with high traffic flow between the virtual machines.

The following diagram displays internal and external traffic flow.

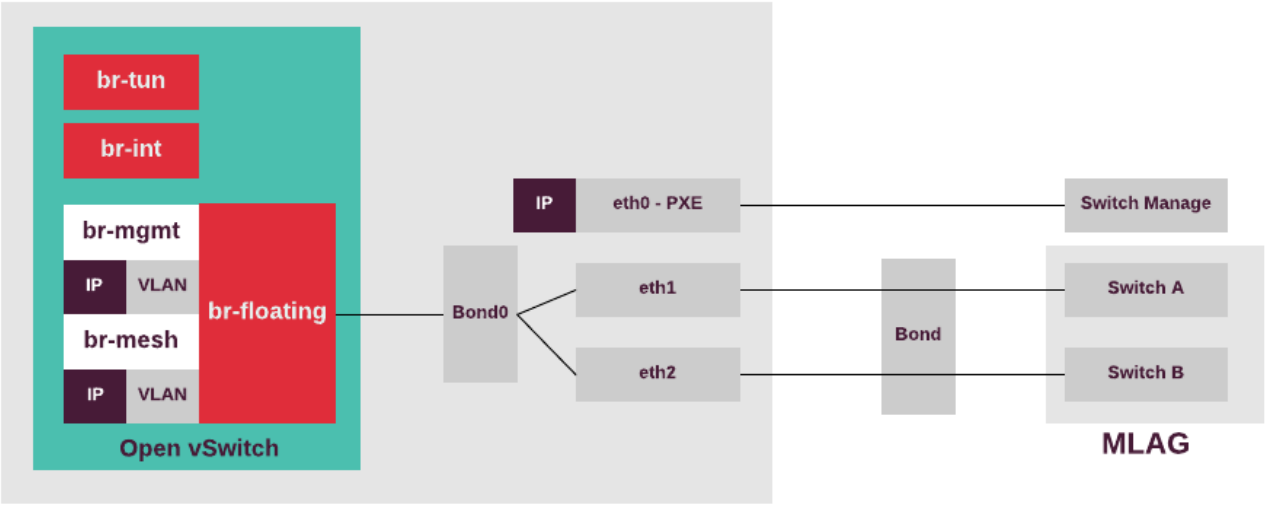


The internal traffic from one tenant virtual machine located on the virtual Internal network 1 goes to another virtual machine located in the Internal network 2 through the DVRs on the compute nodes without leaving the data plane. The external traffic from a virtual machine goes through the Internal network 1 to the DVR on the compute node, then to the DVR on the network node, and finally through the Public network to the outside network.

Network node configuration

In this use case, the network node terminates VXLAN mesh tunnels and sends traffic to external provider VLAN networks. Therefore, all tagged interfaces must be configured directly in Neutron OVS as internal ports without Linux bridges. Bond0 is added into br-floating, which is mapped as physnet1 into the Neutron provider networks. br-mgmt and br-mesh are Neutron OVS internal ports with tags and IP addresses. As there is no need to handle storage traffic on the network nodes, all the sub-interfaces can be created in Neutron OVS. This also allows for the creation of VLAN providers through the Neutron API.

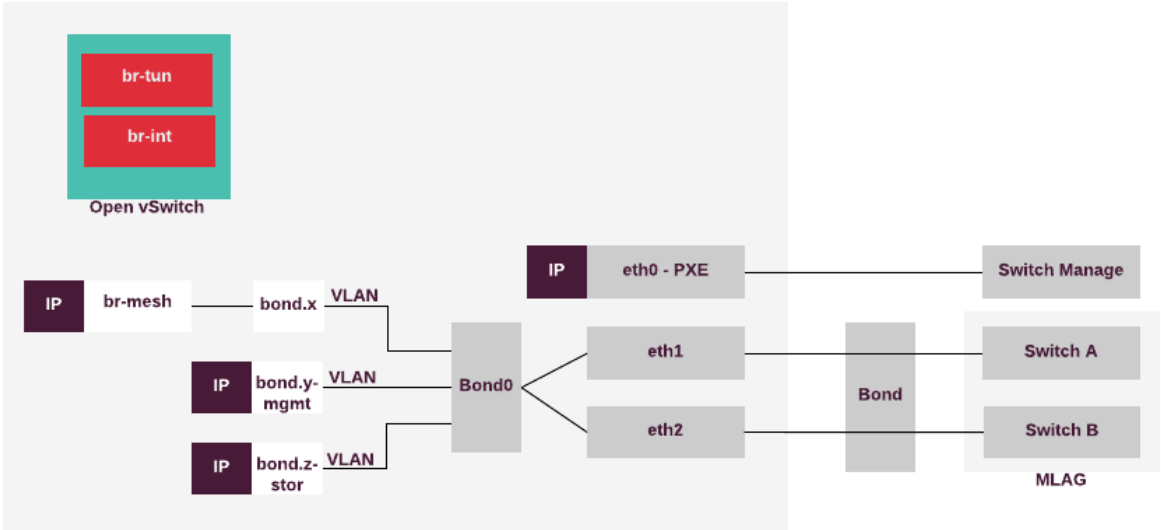
The following diagram displays network node configuration for the use case with Neutron VXLAN tenant networks with network nodes and DVRs configured on the network node only.



Compute nodes configuration

In this use case, compute nodes do not have access to the external network. Therefore, you configure a Linux bridge br-mesh, which is unidirectionally connected with br-tun and used for VXLAN tunneling. All Open vSwitch bridges are automatically created by the Neutron OVS agent. For a highly-available production environment, network interface bonding is required. The separation of types of traffic is done by bonded tagged sub-interfaces, such as bond.y for the virtualized control plane traffic (mgmt IP), bond.x for data plane bridge (br_mesh) which provides VTEP for OVS and bond.z for storage etc. IP address of br-mesh is used as local IP in the openvswitch.ini configuration file for tunneling.

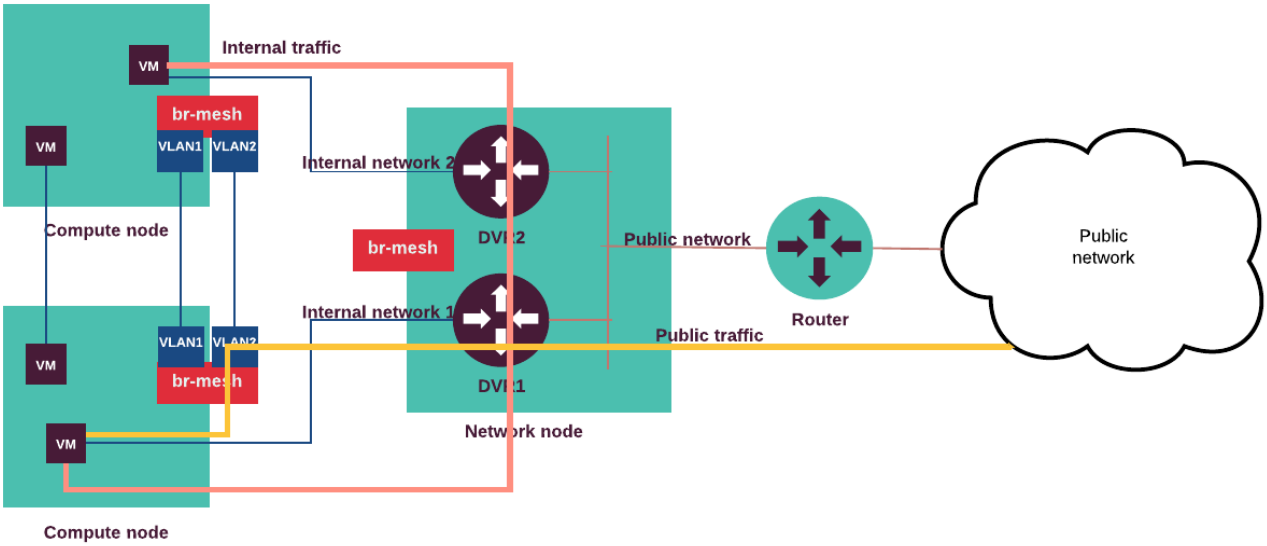
The following diagram displays compute nodes configuration for the use case with Neutron VXLAN tenant networks with network nodes and DVRs configured on the network node only.



Neutron VLAN tenant networks with network nodes (no DVR)

If you configure your network with Neutron OVS VXLAN tenant networks with network nodes and without a Distributed Virtual Router (DVR) on the compute nodes, all routing happens on the network nodes.

The following diagram displays internal and external traffic flow.

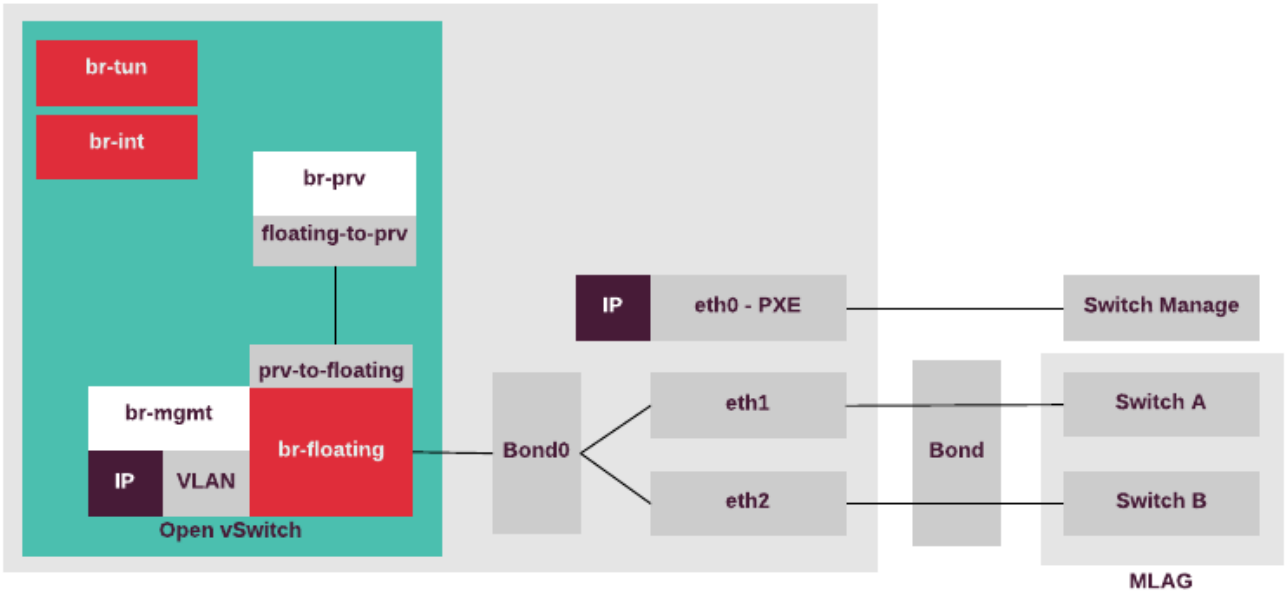


The internal traffic from one tenant virtual machine located on virtual Internal network 1 goes to another virtual machine located in the Internal network 2 through the DVRs on the network node. The external traffic from a virtual machine goes through the Internal network 1 and the tenant VLAN (br-mesh) to the DVR on the network node and through the Public network to the outside network.

Network node configuration

In this use case, the network node terminates private VLANs and sends traffic to the external provider of VLAN networks. Therefore, all tagged interfaces must be configured directly in Neutron OVS as internal ports without Linux bridges. Bond0 is added into br-floating, which is mapped as physnet1 into the Neutron provider networks. br-floating is patched with br-prv which is mapped as physnet2 for VLAN tenant network traffic. br-mgmt is an OVS internal port with a tag and an IP address. br-prv is the Neutron OVS bridge which is connected to br-floating through the patch interface. As storage traffic handling on the network nodes is not required, all the sub-interfaces can be created in Neutron OVS which enables creation of VLAN providers through the Neutron API.

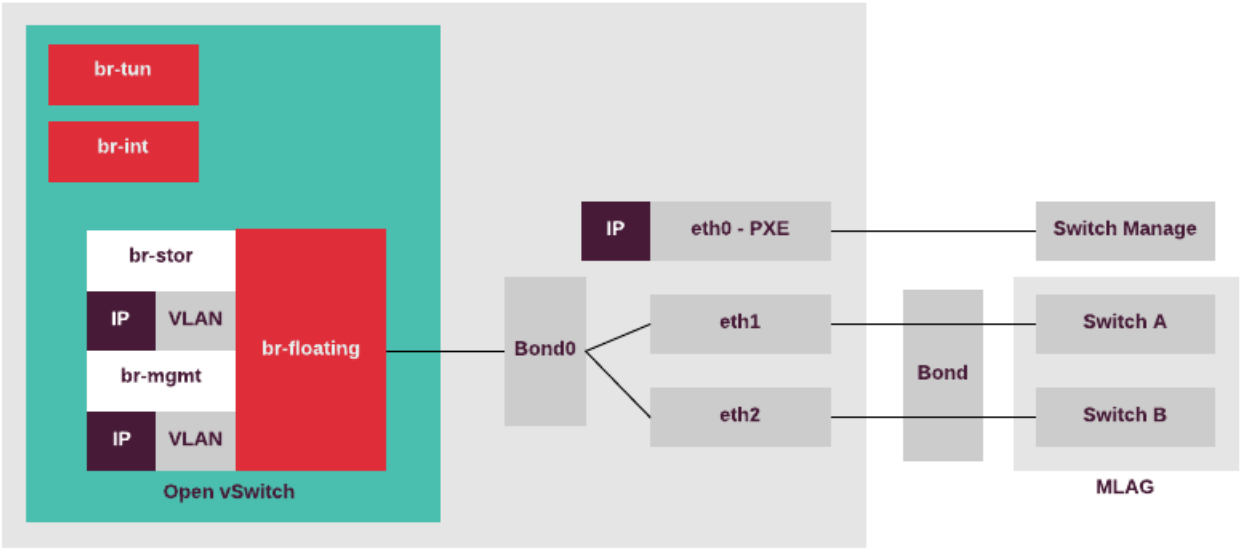
The following diagram displays network node configuration for the use case with Neutron VLAN tenant networks with network nodes and DVRs configured on the network node only.



Compute nodes configuration

In this use case, the network node terminates private VLANs and sends traffic to the external provider of VLAN networks. Therefore, all tagged interfaces must be configured directly in Neutron OVS as internal ports without Linux bridges. Bond0 is added into br-floating, which is mapped as physnet1 into the Neutron provider networks. br-floating is patched with br-prv which is mapped as physnet2 for VLAN tenant network traffic. br-mgmt is an OVS internal port with a tag and an IP address. br-prv is the Neutron OVS bridge which is connected to br-floating through the patch interface. As storage traffic handling on the network nodes is not required, all the sub-interfaces can be created in Neutron OVS which enables creation of VLAN providers through the Neutron API.

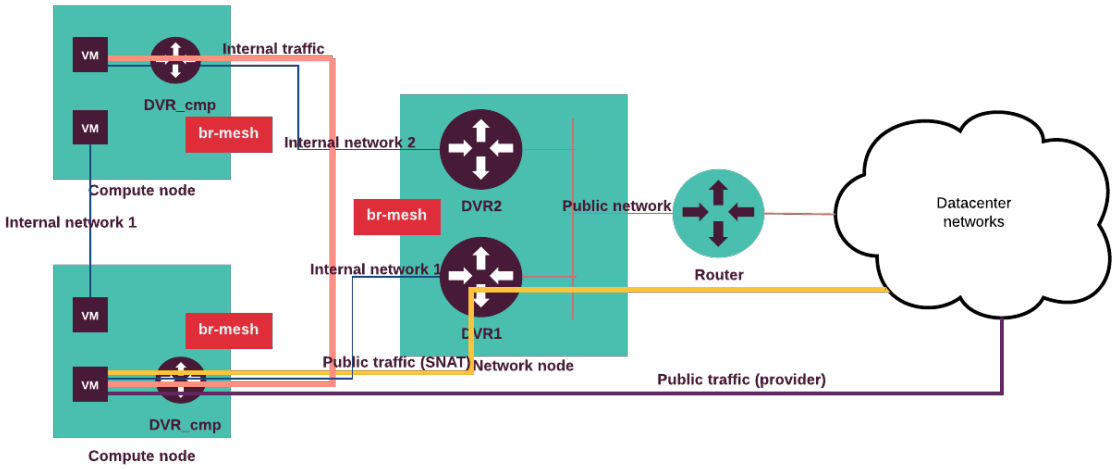
The following diagram displays compute nodes configuration for the use case with Neutron VLAN tenant networks with network nodes and DVRs configured on the network node only.



Neutron VXLAN tenant networks with network nodes for SNAT (DVR for all)

If you configure your network with Neutron OVS VXLAN tenant networks with network nodes for SNAT and Distributed Virtual Routers (DVR) on the compute nodes, network nodes perform SNAT and routing between tenant and public networks. The compute nodes running DVRs perform routing between tenant networks, as well as routing to public networks in cases when public networks (provider, externally routed) are exposed or Floating IP addresses are used.

The following diagram displays internal and external traffic flow.



The internal traffic from one tenant virtual machine located on the virtual Internal network 1 goes to another virtual machine located in the Internal network 2 through the DVRs on the compute nodes. The external traffic (SNAT) from a virtual machine goes through the Internal network 1 and the DVR on the compute node to the DVR on the network node and through the Public network to the outside network. The external routable traffic from a virtual machine on the compute nodes goes through the Internal network 1 and the DVR on the compute node through the Control or Public network to the outside network.

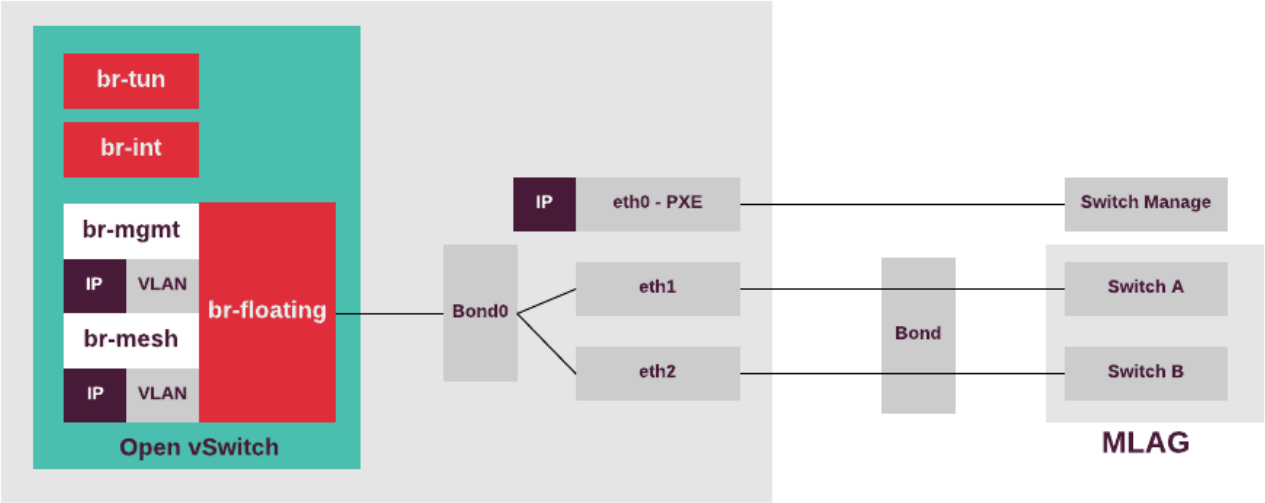
Traffic flow examples:

- A virtual machine without a floating IP address sends traffic to a destination outside the Public network (N-S). The Internal network 1 is connected to a public network through the Neutron router. The virtual machine (VM) is connected to the Internal network 1.
    1. The VM sends traffic through the DVR to the network node.
    2. The network node performs SNAT, de-encapsulates and forwards traffic to the public network's external gateway router.
    3. Return path same.
- A virtual machine with a floating IP address sends traffic to a destination outside the Public network (N-S). The compute node with a DVR hosting the VM is connected to a public network. An Internal network 1 is connected to the external network through the Neutron router. The VM is connected to the Internal network 1.
    1. The VM sends traffic through the compute node DVR to a public network (egress).
    2. The compute node DVR performs SNAT, de-encapsulates and forwards traffic to the public network's external gateway router.
    3. Return path (ingress) same (DNAT).
- A virtual machine on an internal (private, tenant) network sends traffic to a destination IP address on a public (provider, externally routed) network (E-W). The compute node with DVR hosting the VM is connected to the provider network. The Internal network 1 is connected to the provider network through the Neutron router. The VM is connected to the Internal network 1.
    1. The VM sends traffic through the compute node DVR to a destination IP on a public network.
    2. The compute node DVR de-encapsulates and forwards traffic to a public network (no NAT).
    3. Return path same.
- A virtual machine (VM1) sends traffic to another VM (VM2) located on separate host(E-W). The Internal network 1 is connected to the Internal network 2 through the Neutron router. The (VM1) is connected to the Internal network 1 and the VM2 is connected to the Internal network 2.
    1. The VM1 sends traffic to the VM2 through the compute node DVR.
    2. The DVR on the compute node hosting VM1 forwards encapsulated traffic to the DVR on the compute node hosting VM2.
    3. Return path same.

Network node configuration

In this use case, the network node terminates VXLAN mesh tunnels and sends traffic to external provider VLAN networks. Therefore, all tagged interfaces must be configured directly in Neutron OVS as internal ports without Linux bridges. Bond0 is added into br-floating, which is mapped as physnet1 into the Neutron provider networks. br-mgmt and br-mesh are Neutron OVS internal ports with tags and IP addresses. As there is no need to handle storage traffic on the network nodes, all the sub-interfaces can be created in Neutron OVS. This also allows for the creation of VLAN providers through the Neutron API.

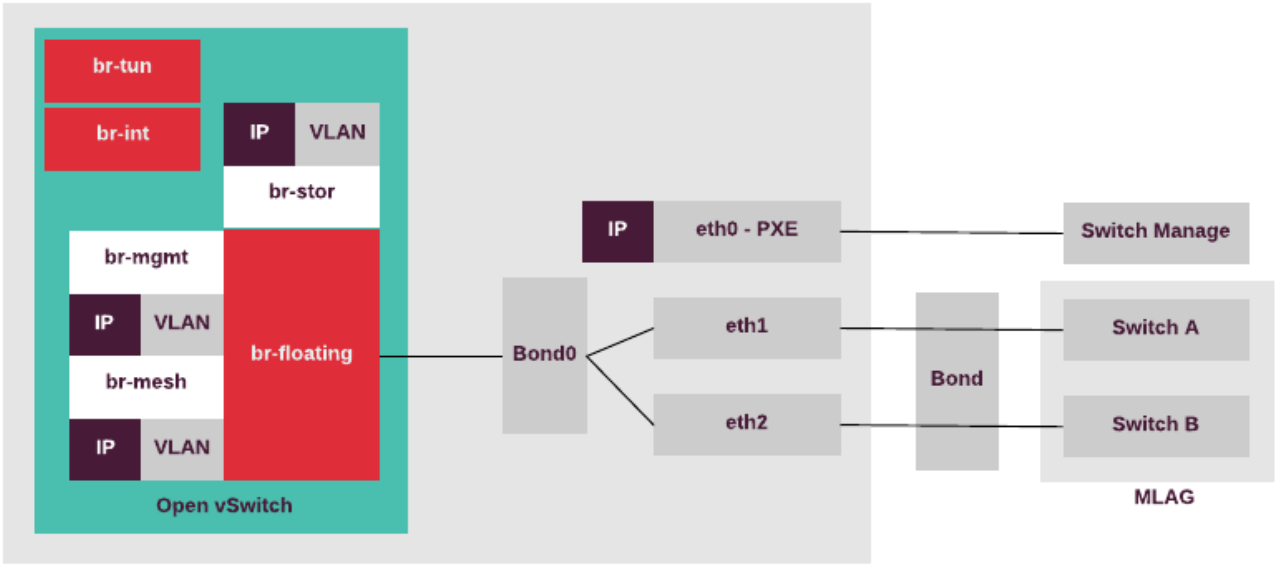
The following diagram displays network node configuration for the use case with Neutron VXLAN tenant networks with network nodes and DVRs configured on the network node only.



Compute nodes configuration

In this use case, compute nodes can access external network, therefore, there is the OVS bridge called br-floating. All Open vSwitch bridges are automatically created by the Neutron OVS agent. For a highly-available production environment, network interface bonding is required. The separation of types of traffic is done by the bonded tagged sub-interfaces, such as bond.x for the virtualized control plane traffic (mgmt IP), bond.y for data plane bridge (br_mesh) which provides VTEP for OVS and bond.z for storage etc. IP address of br-mesh is used as local IP in the openvswitch.ini configuration file for tunneling.

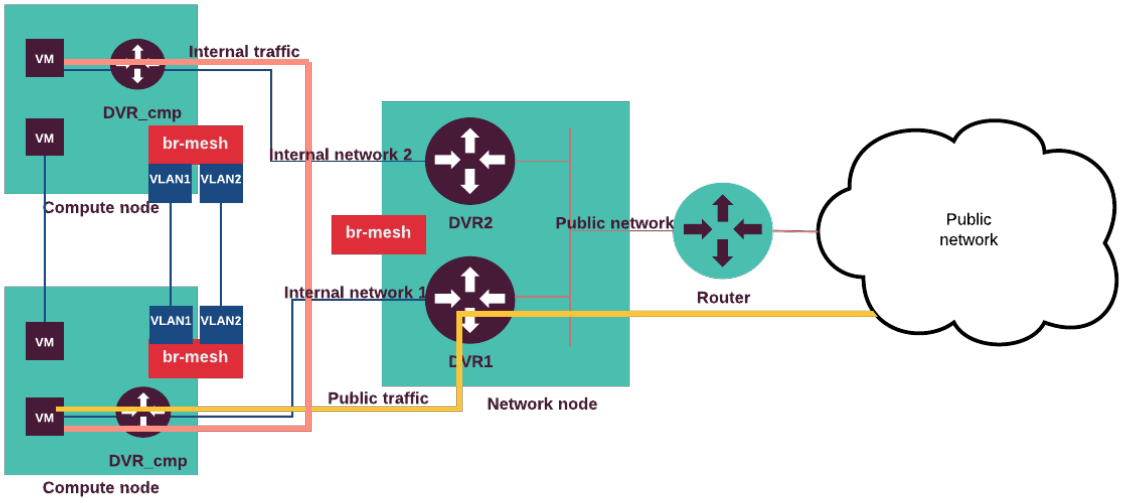
The following diagram displays the compute nodes configuration for the use case with Neutron VXLAN tenant networks with network nodes for SNAT and DVRs for all.



Neutron VLAN tenant networks with network nodes for SNAT (DVR for both)

If you configure your network with Neutron OVS VLAN tenant networks with network nodes for SNAT and Distributed Virtual Routers (DVR) on the compute nodes, SNAT traffic is managed on the network nodes while all other routing happens on the compute nodes.

The following diagram displays internal and external traffic flow.



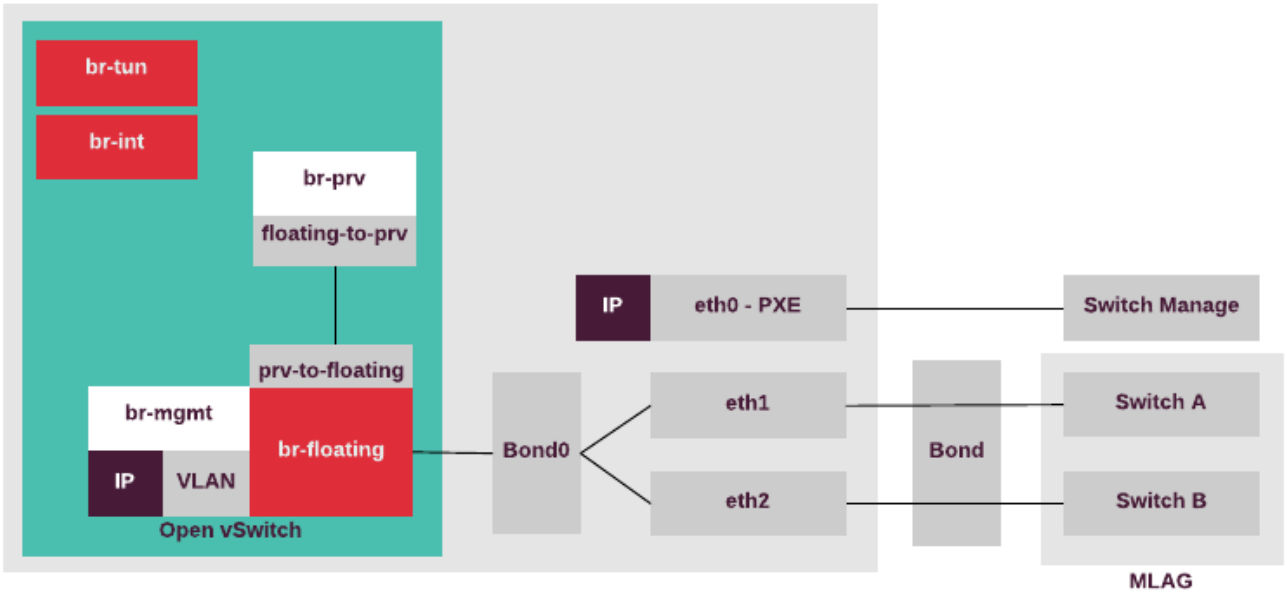
The internal traffic from one tenant virtual machine located on the virtual Internal network 1 goes to another virtual machine located in the Internal network 2 through the DVRs on the

compute nodes. The external traffic from a virtual machine goes through the Internal network 1 and the DVR on the compute node to the DVR on the network node and through the public network to the outside network.

Network node configuration

In this use case, the network node terminates private VLANs and sends traffic to the external provider of VLAN networks. Therefore, all tagged interfaces must be configured directly in Neutron OVS as internal ports without Linux bridges. Bond0 is added into br-floating, which is mapped as physnet1 into the Neutron provider networks. br-floating is patched with br-prv which is mapped as physnet2 for VLAN tenant network traffic. br-mgmt is an OVS internal port with a tag and an IP address. br-prv is the Neutron OVS bridge which is connected to br-floating through the patch interface. As storage traffic handling on the network nodes is not required, all the sub-interfaces can be created in Neutron OVS which enables creation of VLAN providers through the Neutron API.

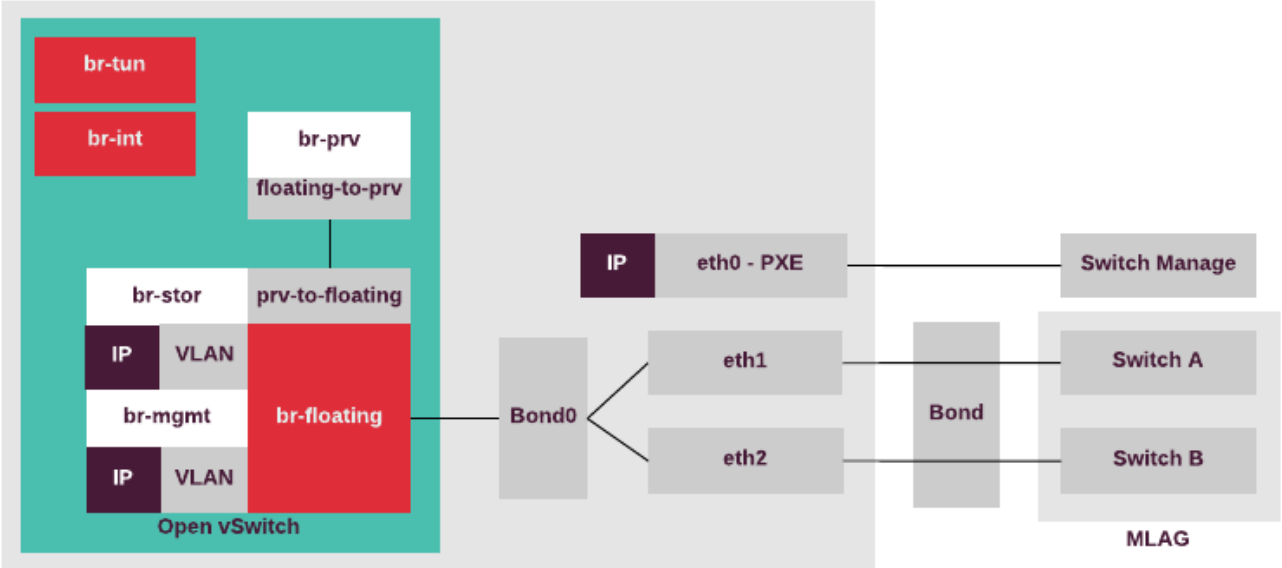
The following diagram displays network node configuration for the use case with Neutron VLAN tenant networks with network nodes and DVRs configured on the network node only.



Compute nodes configuration

In this use case, the network node terminates private VLANs and sends traffic to the external provider of VLAN networks. Therefore, all tagged interfaces must be configured directly in Neutron OVS as internal ports without Linux bridges. Bond0 is added into br-floating, which is mapped as physnet1 into the Neutron provider networks. br-floating is patched with br-prv which is mapped as physnet2 for VLAN tenant network traffic. br-mgmt is an OVS internal port with a tag and an IP address. br-prv is the Neutron OVS bridge which is connected to br-floating through the patch interface. As storage traffic handling on the network nodes is not required, all the sub-interfaces can be created in Neutron OVS which enables creation of VLAN providers through the Neutron API.

The following diagram displays compute nodes configuration for the use case with Neutron VLAN tenant networks with Network Nodes for SNAT and DVR for both:



Storage planning

Depending on your workload requirements, you need to consider different types of storage. This section provides information on how to plan different types of storage for your OpenStack environment.

You typically need to plan for the following types of storage:

Image storage

The storage required for storing virtual machine images that are used to span virtual machines in the OpenStack environment.

Ephemeral block storage

Temporary storage for the operating system in a guest virtual machine. Ephemeral storage is allocated for an instance in an OpenStack environment. As its name suggests, the storage will be deleted once the instance is terminated. This means that the VM user will lose the associated disks with the VM termination. Ephemeral storage persists through a reboot of a VM.

Persistent block storage

In contrast to ephemeral storage, persistent block storage exists outside an instance. Persistent block storage exists independently of virtual instances and can be attached to an arbitrary instance. Persistent block storage can only be attached to one instance at a time.

Object storage

Object storage enables you to store data as objects as opposed to block storage that manages data as blocks combining data, metadata, and a unique identifier in one piece. Object storage is highly scalable, does not have any limit on the amount of metadata, and can be used for various types of files.

Mirantis recommends to use Ceph cluster deployed by a separate software tool called Decapod for all types of storage. However, a variety of other storage options required for an OpenStack environment are supported as well.

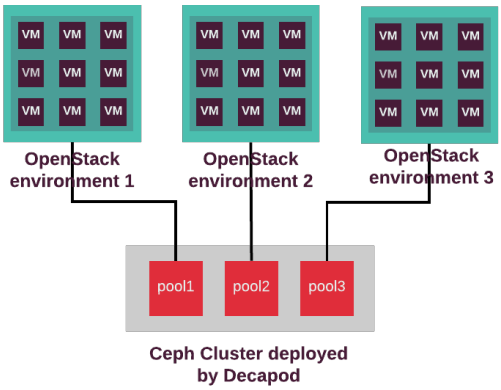
Ceph cluster deployed by Decapod

Ceph is the preferred option for all storage types, including image storage, ephemeral and persistent block storage, and object storage. Mirantis recommends deploying a standalone Ceph cluster using the Ceph lifecycle management tool called Decapod.

Decapod is a software tool that drastically simplifies the deployment and lifecycle management of Ceph. Using Decapod, you can deploy clusters with best known practices, add new nodes to a cluster, remove them, and purge a cluster, if required. You can use the Decapod web UI to manage your clusters. Also, Decapod provides a simple API to manage cluster configurations.

In addition, when you use Decapod, you can deploy one Ceph cluster with multiple storage pools isolated from each other that can serve multiple OpenStack environments.

The following diagram displays the use case with one Ceph cluster serving multiple OpenStack environments:



Decapod uses Ansible with the Ceph-Ansible community project to deliver the best user experience. For tasks, you can use plugins that encapsulate the appropriate settings. You can customize any configuration before execution.

Decapod provides the following functionality:

- Deploying Ceph on remote nodes
- Adding and removing Ceph roles, for example, deploying an OSD or removing a monitor
- Upgrading, updating, and purging clusters
- Managing partitions on disk devices for Ceph

However, Decapod does not cover:

- Providing a server for PXE
- Managing DHCP
- Managing networks by all means
- Managing host OS packages
- Deploying an OS
- Managing partitions on disks that are not related to Ceph

Data models

This section describes the Decapod data models, entities, and workflows.

User model

A user is an entity that contains common information about the Decapod user. It has a login, email, password, full name, and a role. The user model is used for authentication and authorization purposes.

When creating a user model in the system, Decapod sends a new password to a user's email. The user can reset the password and set a new one.

A user created without a role can do a bare minimum with the system because even entities listing requires permissions. Authorization is performed by assigning a role to a user. A user may have only one role in Decapod.

Seealso

- Role model

Role model

A role has two properties: name and permissions. Consider the role as a named set of permissions. Decapod has two types of permissions:

- API permissions enable using different API endpoints and, therefore, a set of actions available for usage. For example, to view the list of users, you must have the view_user permission. To modify the information about a user, you also require the edit_user permission.

  Note

  Some API endpoints require several permissions. For example, user editing requires both view_user and edit_user permissions.

- Playbook permissions define a list of playbooks that a user can execute. For example, a user with any role can execute service playbooks to safely update a host package or add new OSDs. But a user requires special permissions to execute destructive playbooks, such as purging a cluster or removing OSD hosts.

Server model

The server model defines a server used for Ceph purposes. Servers are detected during the server discovery process. Each server has a name (FQDN by default), IP, FQDN, state, cluster ID, and facts. A user is only allowed to modify the server name, other attributes are updated automatically on the server discovery. The facts property is a set of facts collected by Ansible and returned as is. By default, Ansible collects only its own facts, ignoring Ohai and Facter.

Note

Do not create a new server manually using the API. Servers must be discovered by the discovery protocol.

Server discovery is an automatic process of discovering new servers in Decapod. During this process, Decapod works passively.

> **Important**
>
> Decapod does not perform a node operating system deployment. The server discovery is performed using cloud-init, so the only requirement for the node OS is to support cloud-init.

The cloud-init package is required to create a user for Ansible, set the deployment SSH public key for the user's authorized keys, and update the /etc/rc.local script. Then, the /etc/rc.local script registers the host in Decapod.

Seealso

- Ohai
- Facter
- The cloud-init documentation

Cluster model

A cluster model defines a separate Ceph deployment. You can create as many cluster models as required. Each cluster model has a default name that you can edit only explicitly. You can delete only that cluster model that does not contain servers.

An explicit cluster model is required because it defines a name of FSID for Ceph. By default, the name of the model is used as a name of the Ceph cluster and its ID as FSID.

The cluster model configuration is a simple mapping of roles to the list of servers. You cannot manage this configuration explicitly. Instead, you can use playbooks. For example, when executing the playbook for adding a new OSD host, this host will be added to the list of servers for the role osds. If you remove Rados Gateways from the clusters using an appropriate playbook, these servers will be deleted from the list.

Several models are required to deploy a cluster. Basically, cluster deployment contains the following steps:

1. Creating an empty cluster model. This model is a holder for the cluster configuration. Also, it defines the Ceph FSID and name.

2. Creating a playbook configuration model for the deploy_cluster playbook. This will allow you to deploy the cluster.

   > Note
   >
   > Cluster deployment is an idempotent operation and you may execute it multiple times.

3. Executing that playbook configuration by creating a new execution. If required, examine the execution steps or logs.

> Seealso
>
> - Playbook configuration
> - Playbook execution

Decapod playbooks

To deliver Ceph management functionality, Decapod uses plugins called playbooks. A plugin is a Python package that contains Ansible playbooks, a configuration file, and the Python code itself. Each plugin requires configuration.

Playbook configuration

In most cases, Ansible playbooks are generic and have the capability to inject values, such as the hosts where a playbook has to be executed and also some arbitrary parameters, for example, Ceph FSID. These parameters are injected into the Ansible playbooks using the --extra-vars option or by setting them in inventory. A playbook configuration defines the name of the playbook and its parameters. For simplicity, parameters are split into two sections:

- The global_vars section contains global variables for a playbook. Each parameter in this section is defined for all hosts. However, the inventory section redefines any parameters. Parameters from the global_vars section will be passed as --extra-vars parameters. For details, see the Ansible documentation.
- The inventory section is used as the Ansible inventory. Mostly, this will be a real inventory. You can modify the section to exclude sensitive information, for example. But in most cases, the inventory parameters are used as is.

Decapod generates the best possible configuration for a given set of Server model models that can be modified afterward.

Note

Decapod uses the server IP as a host. This IP is the IP of the machine visible to Decapod and does not belong to any network other than the one used by Decapod to SSH on the machine.

Creating a playbook configuration supports optional hints that allow generating a more precise configuration. For example, if you set the dmcrypt hint for a cluster deployment, Decapod will generate the configuration with dmcrypted OSDs. For available hints, see the documentation for a particular plugin.

Seealso

- Playbook execution

Playbook execution

You can run each playbook configuration multiple times. The playbook execution model defines a single execution of a playbook configuration. As a result, you will get the execution status, such as completed, failed, and others, and the execution log. The execution log can be shown as:

- Execution steps, which are the parsed steps of the execution.
- The raw log, which is a pure Ansible log of the whole execution as is, taken from stdout and stderr.

Each execution step has timestamps (started, finished), ID of the server that issued the event, a role and task name of the event, the task status, and detailed information on the error, if any.

Seealso

- Playbook configuration

Supported Ceph packages

Mirantis provides its own Ceph packages with a set of patches that are not included in the community yet but are crucial for customers and internal needs. The supported LTS release of Ceph is Jewel. And the only supported distribution is 16.04 Xenial Xerus.

Mirantis keeps the patches as minimal and non-intrusive as possible and tracks the community releases as close as reasonable. To publish an urgent fix, intermediate releases can be issued. The packages are available from the following APT repository:

```
deb http://mirror.fuel-infra.org/decapod/ceph/jewel-xenial jewel-xenial main
```

The following table lists packages provided for upgrades only:

Ceph release	Ubuntu release	APT repository
Jewel	14.04	deb http://mirror.fuel-infra.org/decapod/ceph/jewel-trusty jewel-trusty main
Hammer (0.94.x)	14.04	deb http://mirror.fuel-infra.org/decapod/ceph/hammer-trusty hammer-trusty main
Hammer (0.94.x)	12.04	deb http://mirror.fuel-infra.org/decapod/ceph/hammer-precise hammer-precise main
Firefly	14.04	deb http://mirror.fuel-infra.org/decapod/ceph/firefly-trusty firefly-trusty main
Firefly	12.04	deb http://mirror.fuel-infra.org/decapod/ceph/firefly-precise firefly-precise main

Important

Packages for old LTS releases and Jewel for Ubuntu 14.04 are intended for upgrade purposes only and are not maintained other than fixing bugs hindering the upgrade to Jewel and Ubuntu 16.04.

Note

Create and use your own repository by following the instructions in the corresponding Create APT repository mirror playbook.

This playbook creates only the repository. You must then set up the web server like NGINX or Caddy.

Seealso

- Ceph
- Ansible
- Ceph-Ansible community project
- Decapod API reference

Image storage planning

The OpenStack Image service (Glance) provides a REST API for storing and managing virtual machine images and snapshots. Glance requires you to configure a back end for storing images.

MCP supports the following options as Glance back end:

Ceph cluster

The default and preferred option for most use cases.

GlusterFS

A highly-scalable distributed network file system that allows you to create a reliable and redundant data storage. Typically, you select GlusterFS as your image storage if you have a specific business requirement that Ceph cannot address.

OpenStack Object Storage (Swift)

You can use the OpenStack Object Storage if you deploy it as your object storage. However, Ceph cluster is still the preferred choice.

Block storage planning

The OpenStack component that provides an API to create block storage for your cloud is called OpenStack Block Storage service, or Cinder. Cinder requires you to configure one or multiple supported back ends.

Mirantis Cloud Platform (MCP) supports the following Cinder back ends:

Cinder drivers

If you already use a network storage solution, such as NAS or SAN, you can use it as a storage back end for Cinder using a corresponding Cinder driver, if available.

Ceph cluster

Ceph supports object, block, and file storage. Therefore, you can use it as OpenStack Block Storage service back end to deliver a reliable, highly-available block storage solution without single points of failure.

In environments that require high performance operations, such as databases and high performance computing, you can configure a combination of two block storage back ends: a high performance storage back end and a standard operations storage back end.

To achieve this functionality, one option is to configure a segment of the Ceph cluster with solid state drives (SSD) and the rest of the Ceph cluster with regular hard disk drives (HDD). The other option includes a high performance third party block storage running as a second block storage back end through a Cinder driver.

Object storage planning

Options for the MCP object storage include:

Ceph

Ceph is the preferred option for most use cases except for the ones listed below. Ceph provides a robust, reliable, and easy to manage object storage solution.

OpenStack Object Storage (Swift)

The OpenStack Object Storage typically used for providing non-server-specific data, such as web assets or media directly to the client. Typically, Ceph is sufficient for most use cases. However, for the following use cases, you may want to use Swift:

- Multi-tenancy

  OpenStack allows you to run virtual machines of different tenants on the physical hardware. This practice, although bringing certain economic benefits, may potentially introduce the risk of a security breach. Multi-tenancy was recently introduced in Ceph and, therefore, may lack production-ready stability. Swift is an alternative solution that at the moment provides better reliability.

- Multi-region

  If parts of your OpenStack environment are geographically separated, Swift performs data distribution and synchronization in terms of minutes while other solutions require much more time. For this use case, you can configure two object storage back ends: one for global object storage based on Swift and one for local object storage based on Ceph.

- Amazon S3

  Generally, Swift provides better compatibility with Amazon S3 cloud storage solution comparing to Ceph. However, depending on a use case, Ceph might be sufficient and if so must be used instead of Swift.

Logging, metering, and alerting planning

StackLight, also known as the Mirantis Logging, Metering, Alerting (LMA) toolchain, is the operational health and response monitoring solution of the Mirantis Cloud Platform (MCP).

StackLight monitors all devices, resources, and services running in standalone or cluster mode on top of the infrastructure clusters of MCP. This includes OpenStack, OpenContrail, Kubernetes, and Ceph.

StackLight provides rich operational insights about the monitored entities with over 700 collected metrics and 150 different alarms.

Using StackLight, cloud operators are quickly notified of critical conditions that may occur on MCP to prevent service downtimes. As a core part of MCP, StackLight is released together with MCP and is automatically installed and activated through deploying a supported Reclass model. The supported Reclass models for MCP are generated by the Cookiecutter project. Currently, Cookiecutter contains the following Reclass model templates:

- Kubernetes_mk for a standalone Kubernetes control plane
- Openstack_mk_contrail for an OpenStack control plane with Contrail SDN
- Openstack_mk_ovs for an OpenStack control plane with Open vSwitch networking

StackLight contains the following components:

Component name	Description
Log Collector	Collects, processes and persists the logs
Local Metric Collector	Creates metrics, evaluates alarms, and persists metrics at a local point of collection
Remote Metric Collector	Creates metrics, evaluates alarms, and persists metrics at a remote point of collection
Aggregator	Aggregates and correlates anomalies detected by the collectors to derive cluster level health indicators that are sent to Sensu or other alerting destinations like Nagios
collectd	Performs service checks and various resource usage measurements
InfluxDB	Stores the LMA metrics in a time-series database
Grafana	Visualizes the time-series in graphs
Elasticsearch	Stores the logs and the OpenStack notifications
Kibana	Visualizes the logs and the OpenStack notifications

Sensu	Performs the StackLight self-monitoring checks and self-healing functions, triggers the alerting notifications
Redis	Used as data persistency for Sensu
RabbitMQ	Used as a message transporting tool for Sensu
Uchiwa	Used as a monitoring dashboard for Sensu
Nagios (optional)	Visualizes the alerts and triggers alert notifications (alternative to Sensu)
Horizon	Contains two plugins to visualize health status of the OpenStack environment and resource usage of tenants (tenant monitoring)

This section describes the architecture and workflow of the essential LMA toolchain components as well as back ends and services integrated with StackLight operational insights pipeline.

StackLight operational insights pipeline components

The StackLight operational insights pipeline constitutes the Log Collector, the Local and Remote Metric Collectors, and the Aggregator.

The intent of the StackLight operational insights pipeline is to measure, analyze, and report in a timely manner everything that may fail in any of the devices, resources, and services running in the standalone or cluster mode on top of the infrastructure clusters of Mirantis Cloud Platform.

The StackLight operational insights pipeline includes monitoring of the following components:

Component	Sub-component
Linux operating system	n/a
OpenStack	• Nova • Cinder • Glance • Neutron • Keystone • Horizon • Heat • Swift
RabbitMQ	n/a
Apache	n/a
libvirt	n/a

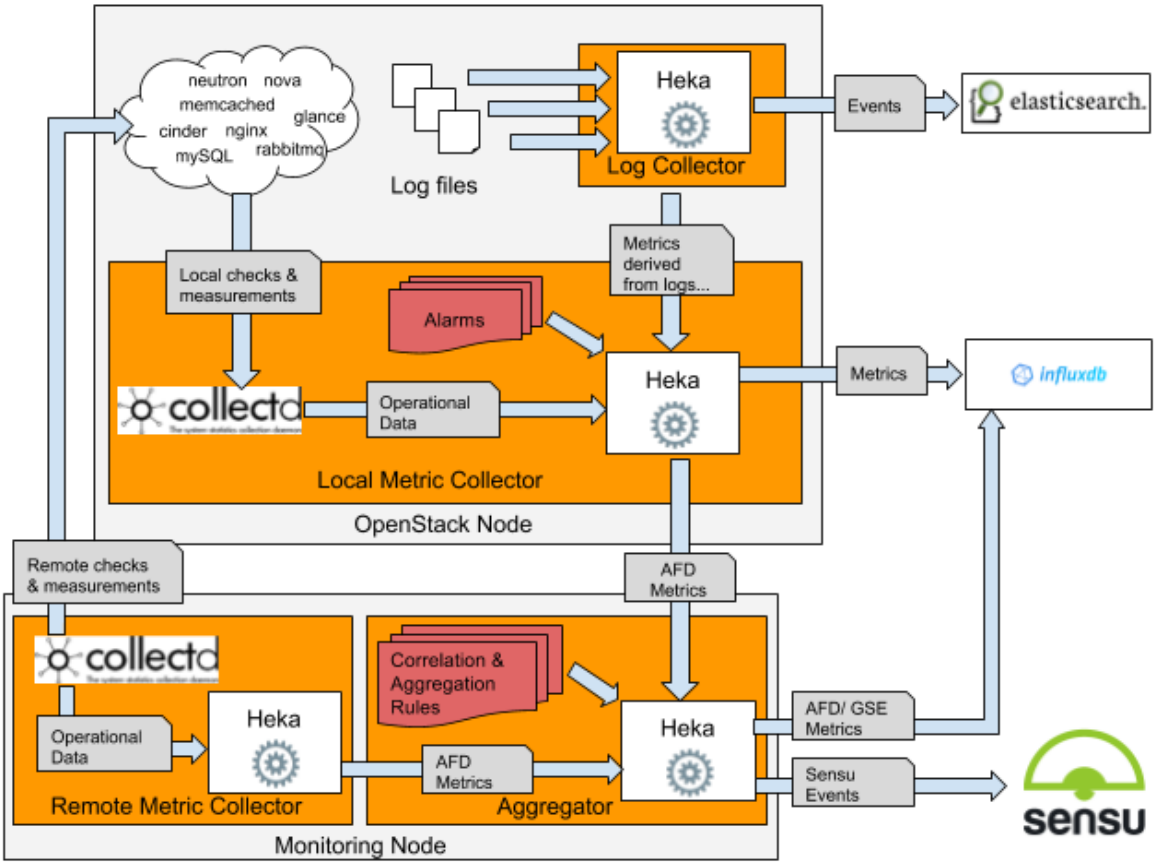
Keepalived	n/a
Memcached	n/a
MySQL	n/a
NGINX	n/a
NTP	n/a
OpenContrail	• Cassandra • Contrail server • Zookeeper (not available yet) • Kafka (not available yet)
Kubernetes	• Kube-apiserver • Kube-controller-manager • Kube-proxy • Kube-scheduler • Kubelet • Docker • etcd
Calico	• Felix • BIRD • confd
Ceph	• OSD • ceph-mon
StackLight	• InfluxDB • Elasticsearch • Grafana • Kibana

The StackLight operational insights pipeline incorporates a pragmatic approach to what StackLight monitors and how. The core value of the pipeline is to collect all necessary data and make a deep analysis of those data to define the actual health status of the services running in the MCP infrastructure. StackLight performs this analysis both at a discrete and cluster level which is a distinguishing capability of StackLight considering its built-in understanding of the services behavior, role, and relationship.

The StackLight operational insights pipeline is not a general-purpose monitoring system and does not aim to become one. The goal is to instead make that operational insights pipeline

readily available in a universally pluggable manner to third-party monitoring systems such as Nagios and/or Sensu, which are widely used in the industry.

The diagram below shows the data workflow inside the StackLight operational insights pipeline.



Log Collector

The Log Collector is installed on each of the nodes managed by Mirantis Cloud Platform. It is responsible for streaming the log files of the processes running on the nodes. It is based on the Heka stream processing technology.

The logs are parsed and sanitized using an internal message structure representation so that they can be further analyzed and processed for anomaly detection and indexing in Elasticsearch by field.

The important feature of the Log Collector is to create metrics about errors found in the logs. The specific Lua plugins accumulate these messages in memory to create statistics such as rate of errors by application. These metrics are then sent to the Local Metric Collector every ticker interval (every 10 seconds by default) to be evaluated in alarms and correlated with other metrics.

The metrics derived from logs are used to alert the operator upon abnormal conditions such as a spike of HTTP 5xx errors. Finally, the logs are sent to Elasticsearch where they are indexed for viewing and searching in Kibana.

The Log Collector has the following components:

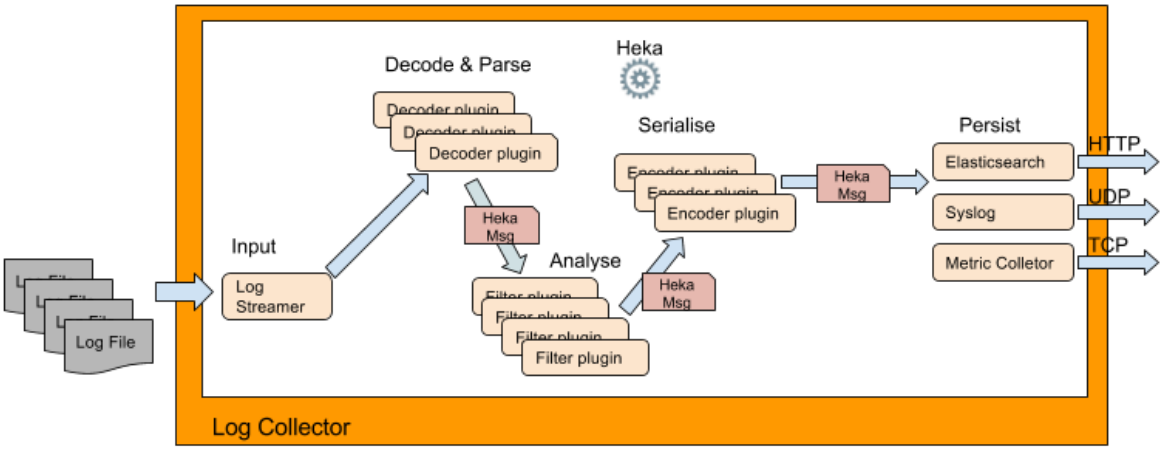
1. Heka, which provides a LogStreamer plugin that allows scanning, sorting, and reading logstreams in a sequential user-defined order.
2. A collection of Lua plugins running in the Heka Lua Sandbox to run the logs processing logic.

> Note
>
> A logstream is a single linear data stream that is spread across one or more sequential log files. For example, an Apache or NGINX server typically generates two log streams for each domain: an access log and an error log. See Heka documentation for additional details.

Out of the box, the Log Collector handles the following types of outputs:

- An HTTP output to Elasticsearch where the logs and the notifications are sent
- A TCP or UDP output to a syslog server where the logs are sent
- A TCP output to the Local Metric Collector where the metrics derived from logs are sent

The diagram below shows the data workflow inside the Log Collector.



Metric Collector

StackLight for Mirantis Cloud Platform has two types of Metric Collectors:

- The Local Metric Collector
- The Remote Metric Collector

The term “Collector” is rather a misnomer since the Metric Collector does much more than simply collecting data. The Metric Collector is a smart monitoring agent that performs advanced monitoring functions at the point of data collection.

For both Local and Remote Metric Collectors, the processing logic of the data, especially the alarms handling, is distributed across all the nodes where the Metric Collector is running, as opposed to being executed on a central server. Such logic ensures an excellent scalability and resilience of StackLight.

The Metric Collector is based on the same Heka technology as the Log Collector. By design, the Metric Collector is liberal with the data it accepts and conservative with the data it sends. This makes Heka a natural fit for the task because of its fast and flexible stream processing engine and standard plugins that cover a large array of the StackLight plugability and interoperability requirements with native monitoring systems.

> Note
>
> The future StackLight version may use HindSight instead of Heka for the Metric Collector. Hindsight is a C-based data processing infrastructure based on the same lua sandbox project to replace Heka.

The Local Metric Collector should be installed on each of the nodes monitored by StackLight. The Remote Metric Collector should be installed as a part of the monitoring cluster of the virtualized control plane of MCP in an active/passive HA cluster managed by Keepalived and HAProxy. Keepalived manages failover of the IP address for the HAProxy monitoring VIP.

Only one Remote Metric Collector in active state should be bound to the monitoring VIP at a time. However, there should be more than one Remote Metric Collector installed in active/passive failover mode to ensure that no single point of failure of the Remote Metric Collector functions.

As opposed to the Local Meric Collector, the Remote Metric Collector monitors the services remotely from the node where they are running. It also collects and processes the OpenStack notifications using the Heka RabbitMQ input plugin. The Remote Metric Collector is typically used to check the availability of the services through their assigned VIP. On the Remote Metric Collector, the CADF-compatible notifications are encoded differently depending on whether they are stored in Elasticsearch and/or sent to a syslog-compatible service, for example, a SIEM system such as QRadar.

Several types of measurements are also performed, for example, to get statistics about global resources utilization or health status information, such as those reported by the built-in heartbeat monitoring of OpenStack.

The Metric Collector has the following components:

1. The collectd daemon that performs the service health checks and resource usage measurements. The collectd daemon has a collection of the purpose-built plugins to perform various types of checks and measurements. In general, the checks and measurements are implemented in separate Python plugins so that the measurements can be turned off in configuration independently of the checks.

2. Heka and a collection of the Lua plugins running in the Heka Lua Sandbox are used to normalize and process the operational data obtained from collectd and other sources, such as the Log Collector or RabbitMQ for the OpenStack notifications. The normalization process consists of decoding and sanitizing the raw data obtained from collectd into a standard message structure. This structure contains the following items that are consumed by other plugins of the stream processing pipeline to process the alarms:

    - metric name
    - metric value
    - metric timestamp
    - metric metadata

    Finally, there are the encoder and output plugins responsible for serializing and sending the metric messages to an external destination such as InfluxDB, Nagios, or Sensu.

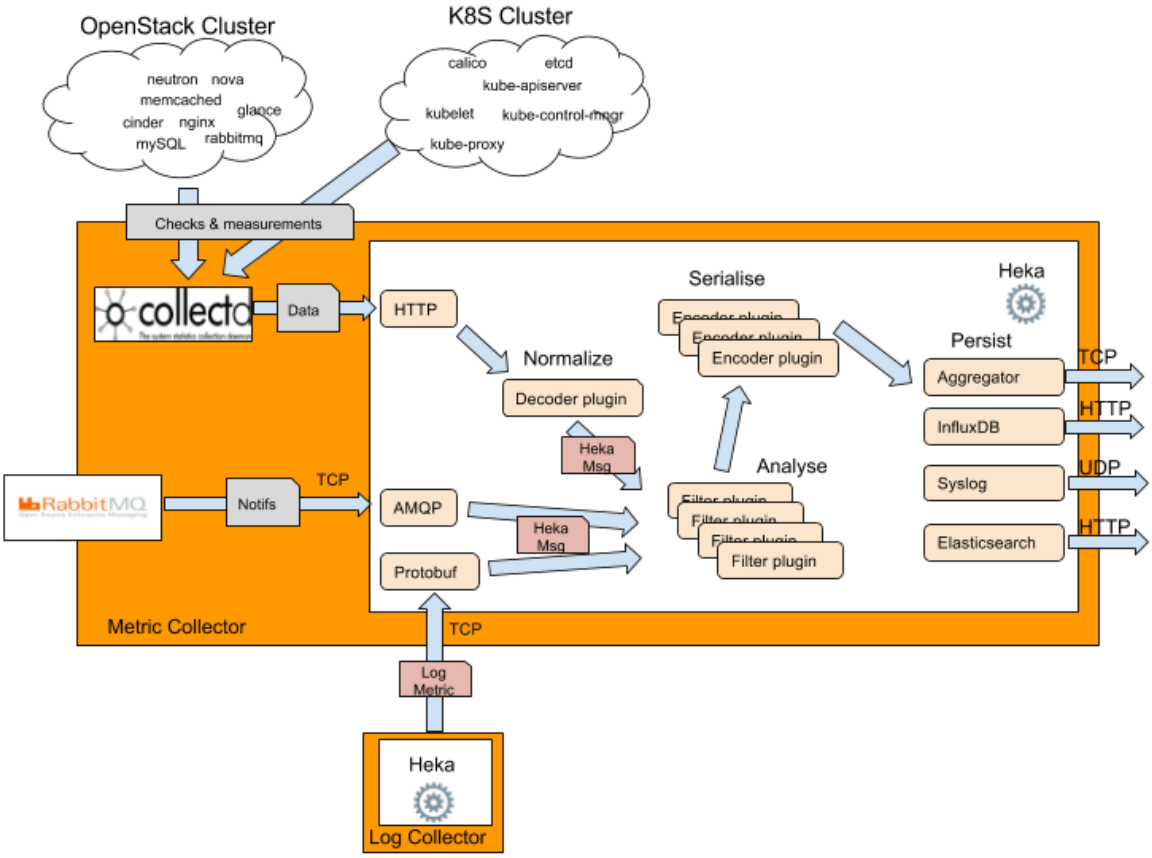
The Heka messages traversing the pipeline are persisted to disk that prevents data loss when the hekad process crashes. Also, Heka is configured with a buffer of 1 GB to prevent transient network congestion issues. When the buffer is full, Heka drops all new incoming messages. This policy allows reducing the load during ingestion of the collectd data into Heka to prevent collectd blocking. The collectd daemon blocks when it crosses its internal buffer high watermark. Relaxing the back pressure on collectd is critical since in general collectd does not recover from a block state without a restart.

> **Important**
>
> The configuration of StackLight in MCP is data-model driven. In practice, it means that the parameterization of what plugin is installed on what node in Heka and collectd is defined in the Reclass model as opposed to be defined in the Heka and collectd formulas themselves.

Out of the box, the Metric Collector handles the following types of outputs:

- A TCP output to the Aggregator where the AFD metrics are sent
- An HTTP output to InfluxDB where all metrics are sent
- An HTTP output to Elasticsearch where all notifications are sent
- A TCP or UDP output to a syslog server where the CADF notifications are sent (only the Remote Metric Collector)

The diagram below shows the data workflow inside the Metric Collector.



Aggregator

The Aggregator must be installed on the virtualized control plane of the MCP monitoring cluster in an active/passive HA cluster managed by Keepalived and HAProxy. Only one Aggregator should be bound to the monitoring VIP at a time. However, there should be more than one Aggregator installed in active/passive failover mode to ensure no single point of failure of the Aggregator functions. Keepalived is used to configure the IP address of the monitoring VIP.

The Aggregator is based on the same Heka technology as the Collectors, but it does not perform any collectd checks or measurements. Instead, it receives a stream of metrics from all the Collectors called the Anomaly and Fault Detection (AFD) metrics.

The AFD metrics contain the health status of the entities being monitored by the Collectors. The role of the Aggregator is to apply a set of declaratively defined correlation policies. The result of this processing produces another kind of metrics called the Global Status Evaluation (GSE) metrics.

The GSE metrics contain the health status of the clusters. These metrics are logical constructs that are declaratively defined in aggregation rules.

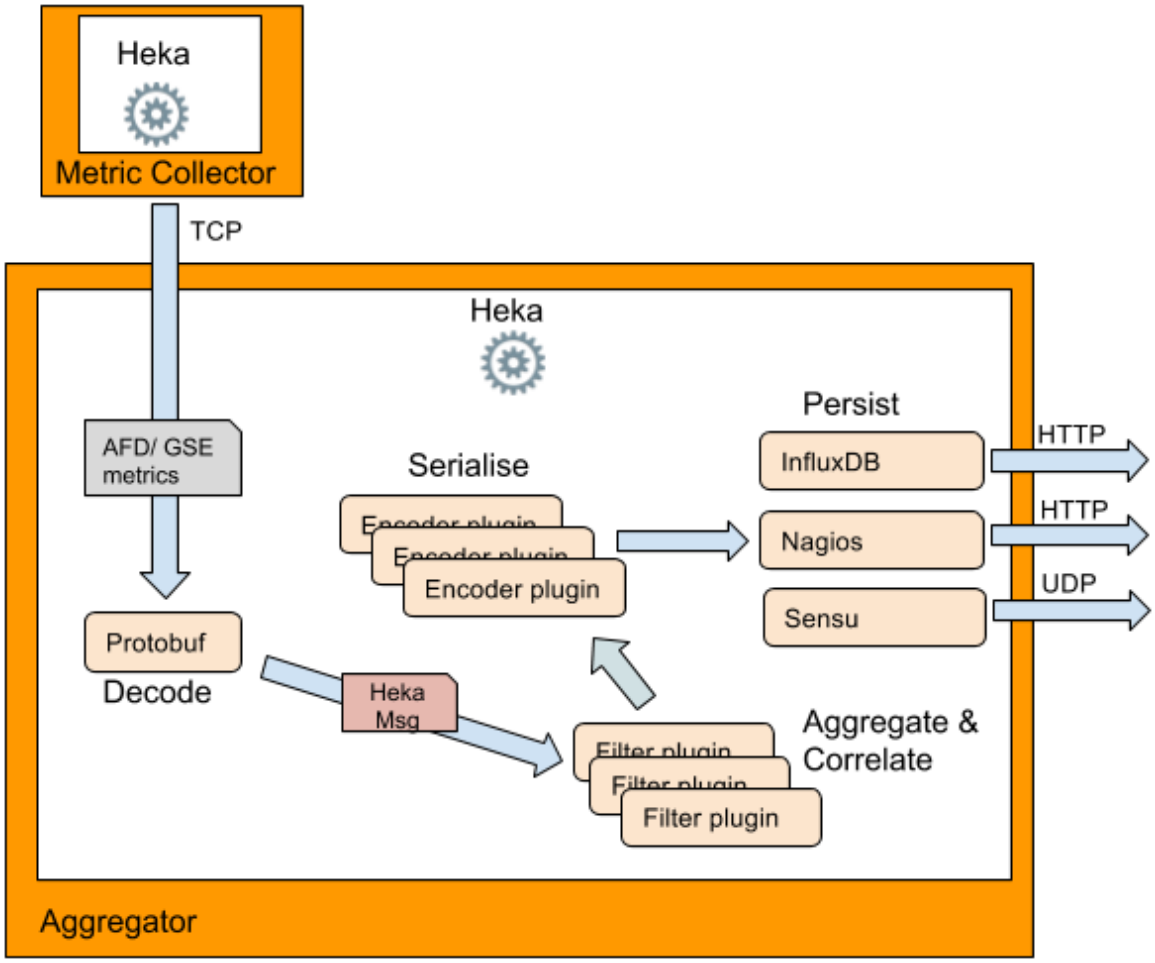
Both the AFD and GSE metrics are also called health status metrics. Ultimately, the AFD and GSE metrics are transformed into health status events that are sent to Sensu for alerting.

> **Caution!**
>
> In StackLight, Sensu is used by default instead of Nagios. Sending of passive checks to Nagios is still supported, but it is not the preferred alerting mechanism in StackLight for MCP.

Out-of-the-box, the Aggregator handles the following types of outputs:

- An HTTP output to InfluxDB where all metrics are sent
- An HTTP output to Nagios where the passive checks are sent (optional)
- A UDP output to the local Sensu client where all the AFD and GSE metrics are sent

The diagram below shows the data workflow inside the Aggregator.



Seealso

StackLight alerts in the MCP Operations Guide

Components integrated with StackLight operational insights pipeline

The LMA toolchain includes several back-end servers integrated with the StackLight operational insights pipeline to perform the visualization of an environment monitoring and health status.

InfluxDB and Grafana clusters

InfluxDB is a powerful time-series database to store and search metrics time-series. The InfluxDB cluster must be installed as part of the virtualized control plane of MCP in an active/passive HA mode managed by Keepalived and HAProxy. Only one InfluxDB server should be active at a time.

Note

The current implementation of StackLight for MCP uses InfluxDB community version 1.2. If you want to use a fully supported InfluxDB cluster for HA and scale-out, install the InfluxEnterprise version separately.

The Grafana server is installed in an active/active HA mode on the monitoring cluster of the virtualized control plane. With Grafana, HAProxy is used for load balancing with sticky sessions.

InfluxDB and Grafana are mainly used to diagnose and visualize trends (what has changed since when) in time-series over a certain retention period that is configurable in the Salt model. The Grafana dashboards are not defined in the support metadata of the Grafana formula but in the support metadata formula of the monitored entities.

In the StackLight architecture, InfluxDB is not in the critical path of alerting, because the alarms are evaluated by the Collectors and the alerts displayed in Sensu and Nagios convey the same alerting information that the Grafana annotations.

Elasticsearch cluster and Kibana server

The Elasticsearch cluster must be installed as part of the virtualized control plane for HA and scale-out. The Elasticsearch clustering is supported natively. It must be installed on at least three nodes to avoid split-brain issues. All Elasticsearch servers store data and can be elected master. By default, five shards per index type per day and two replicas, but this setting can be changed in the Salt model. The Elasticsearch cluster is reconfigured automatically when new nodes are added (or removed) to the cluster. Access to the Elasticsearch and Kibana servers is done through the monitoring VIP for failover and load balancing. The Kibana server is installed in an active/active HA mode managed by HAProxy and Keepalived.

Sensu monitoring cluster

Sensu is the overarching monitoring system that the StackLight operational insights pipeline is integrated with.

Sensu replaces Nagios as a widely used and more efficient monitoring system that includes and API to query events, as opposed to Nagios. This API allows displaying the health status of the OpenStack services in Horizon. Sensu also contains clustering and multi-domain monitoring support that allow aggregating the monitoring of multiple domains (datacenters) using Uchiwa.

Uchiwa is a simple dashboard for the Sensu monitoring framework running on top of multiple instances of the Sensu API server and presenting a unified monitoring view of a multi-domain deployment.

In the StackLight architecture for MCP, Sensu plays a dual role:

- It monitors the components of the StackLight operational insight pipeline itself (collectd, Local Metric Collector, Remote Metric Collector, and Aggregator). The monitoring of these components covers not only the processes statuses but also functional checks, such as verifying that the stream processing pipeline of Heka is up.
- It handles the StackLight health status events for alerting and escalation. Those events convey health status information about all monitored devices, resources, and services running in standalone or in cluster mode on top of the infrastructure clusters of MCP. This Sensu integration comes as a complete replacement of the passive checks handled by Nagios in the former StackLight architecture.

The Sensu monitoring cluster is installed in a distributed fashion on the monitoring cluster for high availability and scale-out. As for other services of StackLight, HAProxy is used for load-balancing and failover using a VIP managed by Keepalived.

Sensu components

The Sensu framework has the following components:

Component	Description
Secure transport	The Sensu services use RabbitMQ as a messaging queue to communicate with one another.
Data store	The Sensu server and Sensu API instances use Redis as a data store registry for the checks, events, and clients. By storing those data in Redis, the Sensu services themselves can remain stateless. Therefore, there can be several instances of the Sensu server and Sensu API instance running in parallel.
Sensu server	The Sensu server is a scalable event processor that processes event data and takes action. Typically, the Sensu server can process the events received from StackLight using filters, mutators, and handlers. The Sensu server is installed with a default handler which is defined in the alarms of the Reclass model.
Sensu API	Sensu provides access to monitoring data and core functionality through a RESTful HTTP JSON-based API. This API is used by Uchiwa and Horizon.

Sensu client	The Sensu client is the monitoring agent that is responsible for monitoring the StackLight operational insights pipeline services. It also directed by the Sensu server to perform regular Keepalived checks to verify the availability of the nodes as a whole.
Uchiwa	The monitoring dashboard for Sensu.

A special Sensu client instance supports the integration of the StackLight operational insights pipeline with Sensu. The Aggregator sends health status events to the local Sensu client through UDP. Technically, the Aggregator is viewed by Sensu as a proxy client that converts the AFD and GSE metrics traversing the Heka pipeline into the Sensu-compatible events.

Sensu events handling

The health status events sent by the Aggregator to Sensu may contain the name of a Sensu handler associated with the event. This name is configurable in the service metadata of the Heka formula that can be overridden in the Salt model.

A Sensu event may or may not have a handler name depending on the alarm definition in the support metadata of the monitored entity formula:

- If the alarm is enabled with alerting, then the event will be sent with the noop handler name. The noop handler on the Sensu server does nothing (cat /dev/null). The noop handler must be enabled for the Sensu server in the Salt model.

  Example of an alarm with alerting enabled:

```
nova_compute:
    policy: highest_severity
    alerting: enabled
    match:
      service: nova-compute
    members:
    - nova_compute
    dimension:
      service: nova-data
      nagios_host: 01-service-cluster
```

- If the alarm is enabled with notification (see example below), then the event will be sent with the name of the notification handler name that must be defined in the model for the Aggregator. This handler should also be defined in the Salt model for Sensu. Otherwise, this will be logged by the Sensu server as an error every time an event is received. Several handlers can be defined with the Sensu Salt formula including Graphite, Hipchat, Pagerduty, SalesForce.com, Slack, Statsd, and others. A default handler is used for events that have no handler defined. The Sensu handler used by the Aggregator must be defined in the Salt model.

  Example of an alarm with alerting with notification enabled:

```
nova_control:
    policy: highest_severity
    alerting: enabled_with_notification
```

```
match:
  service: nova-control
members:
- nova_logs
- nova_api_endpoint
- nova_api_check
{%- for nova_service in ('cert', 'consoleauth', 'conductor', 'scheduler') %}
- nova_{{ nova_service }}
{%- endfor %}
dimension:
  cluster_name: nova-control
  nagios_host: 00-top-clusters
```

Seealso

The StackLight alarm structure section in the MCP Operations Guide

Horizon

In addition to Sensu, the StackLight operational insights pipeline has an integration with Horizon using two plugins:

- The horizon-telemetry-dashboard plugin provides access to the Telemetry data stored in InfluxDB for the control plane and data plane nodes. A user with the admin role has access to the Telemetry Dashboard of Horizon. A user with a non-admin role has access to the Telemetry data overview of instances on a per tenant basis.
- The horizon-monitoring-dashboard plugin provides access to the StackLight health status events stored in Sensu. Using the Monitoring dashboard of Horizon, a user with the admin role can visualize those events the same way as in Uchiwa.

LMA reference deployment

A deployment of the StackLight back-end servers in a cluster is primarily intended to ensure high availability before scale-out putting aside Elasticsearch that can handle both scale-out through shards and replicas and HA clustering natively.

Resilience to hardware failures is ensured as long as there are at least three nodes configured for hosting the StackLight monitoring cluster. This is required to avoid the split-brain effect in the master election process.

It is possible to add and remove nodes in the monitoring cluster after the initial deployment. All the heavy-lifting configuration management is handled transparently by the formulas. The constraints and limitations are described in the back-end server formulas README file.

As for standalone deployments, the back-end servers can be installed on the same node or on different nodes. Your deployment blueprint may vary, but to reduce cost, it is possible to collocate all the StackLight back-end servers on the same node. Or you can use virtual

machines of the virtualized control plane to segregate the back-end servers by function or workload profile.

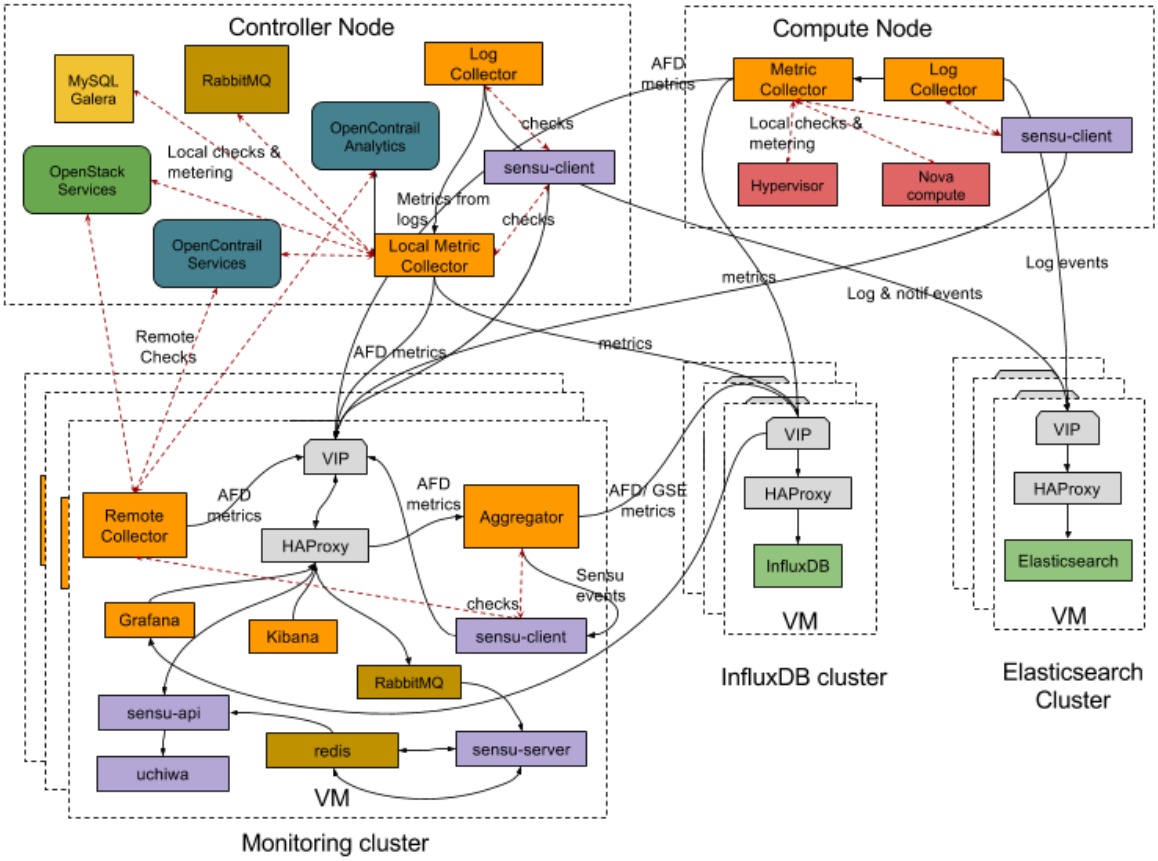
The limitations and constraints of the monitoring cluster are as follows:

1. The InfluxDB server that is installed by default on the Mirantis Cloud Platform is the latest available community version. However, the community version does not support HA clustering. Therefore, only one active instance of the InfluxDB server runs in the monitoring cluster at a time.

   InfluxDB is not used for alerting. Therefore, the crash of the active InfluxDB server will only affect the scope and time window of the time-series that can be visualized in Grafana after a failover of the active InfluxDB server to a new node of the monitoring cluster. To mitigate that issue, we recommend performing regular backups of the InfluxDB database that can be restored on the failover node after a crash of the primary node. If you still want to run InfluxDB in a fully supported cluster mode for HA and scale-out, you should install the InfluxEnterprise commercial edition of InfluxDB separately.

2. The StackLight integration with an LDAP server for authentication and authorization is supported natively for Grafana. For additional details, see the Grafana formula. However, it is not supported for Kibana and Sensu. If you want to add support for LDAP authentication and authorization in Kibana and/or Sensu, you should install the commercial version separately.

The following diagram shows a reference deployment of Stacklight on Mirantis Cloud Platform based on nodes classification defined in the Cookiecutter project for StackLight with OpenStack:

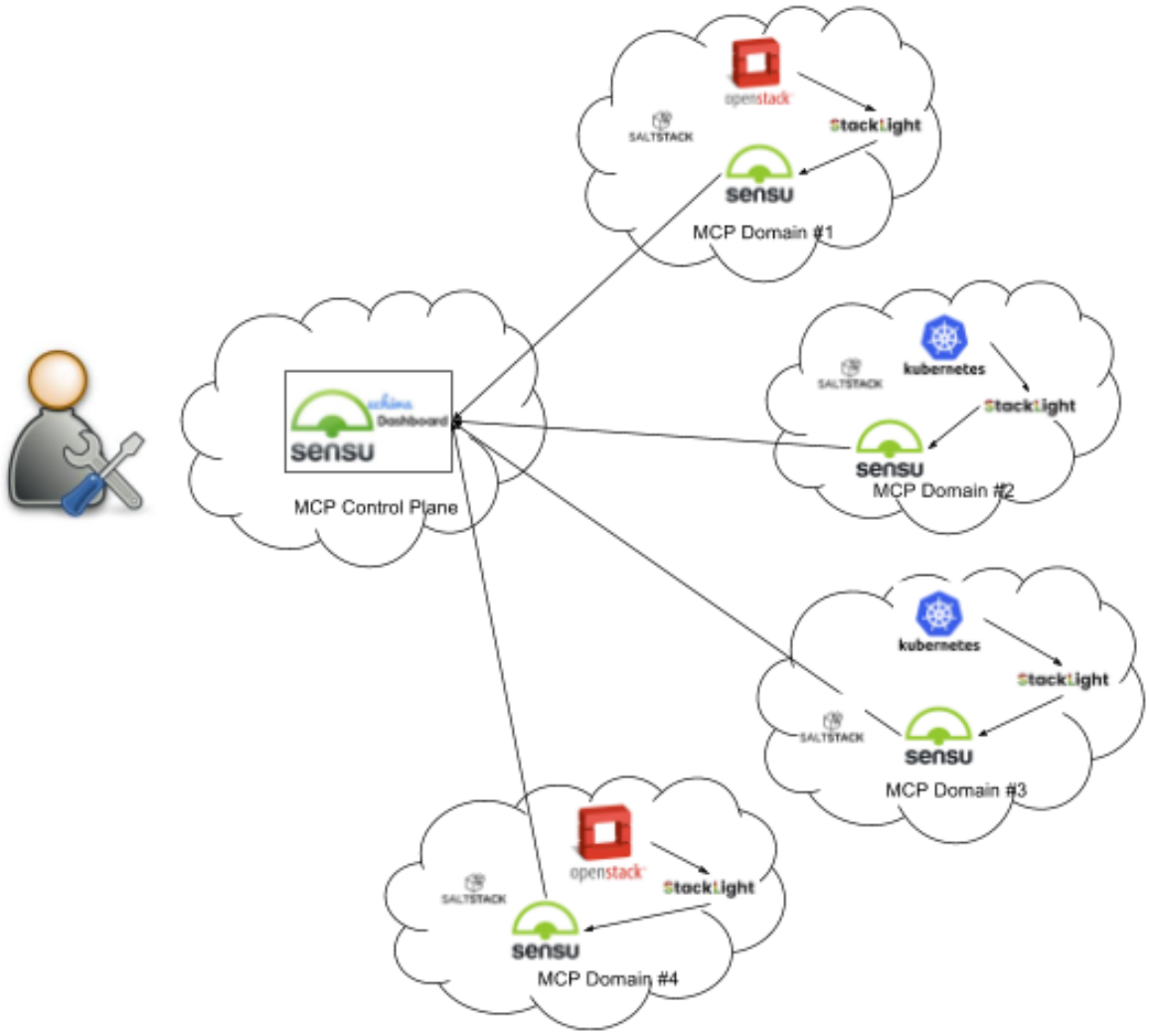


Multi-domain monitoring support

The multi-domain monitoring allows controlling multiple Mirantis Cloud Platform (MCP) clusters using a single DriveTrain in a centralized fashion.

In MCP, a domain typically refers to a Reclass model that is deployed in an MCP cluster. As such, a multi-domain MCP cluster includes one or several OpenStack environments and one or several Kubernetes clusters that can be operated in a somewhat centralized fashion from DriveTrain as shown in the diagram below.

> Note
>
> The name of an MCP cluster is defined by the cluster_domain parameter in the Reclass model.



Since the StackLight operational insights pipeline must be installed and configured as part of the same MCP cluster as the entities it monitors, the StackLight operational insights pipeline components must be deployed separately from DriveTrain. For example, the Metric Collector and the Aggregator use the service and support metadata of the Nova formula to configure the collectd checks and the Heka plugins used to precisely monitor the OpenStack Compute service. Therefore, the StackLight operational insights pipeline must be installed in the same Virtualized Control Plane as the OpenStack environment and Kubernetes cluster.

It should be technically possible to run the Sensu cluster (sensu-api, sensu-server, rabbitmq and redis) in a central location as long as there is a Layer-3 IP connectivity between the Sensu clients and RabbitMQ. But we do not recommend this scenario for the Sensu deployment to avoid the name clashing problems for the registered clients (that is the name of the physical hosts and the name of the virtual hosts representing the clusters, such as 00-top-clusters) across the Virtualized Control Plane.

The InfluxDB and the Elasticsearch clusters can be installed in a central location as long as there is a Layer-3 IP connectivity between the StackLight operational insights pipeline components

and InfluxDB and/or Elasticsearch. Furthermore, all the metrics stored in Influxdb are tagged with the cluster_domain name, known as the environment_label tag, as well as the OpenStack region and availability zone tags when applicable. Therefore, you can execute the InfluxDB queries filtered by any of those tags if you want to segregate results. The same tagging and queries filtering applies to the logs and the notifications stored in Elasticsearch.

Uchiwa can be installed in a central location for multi-domain monitoring since you can federate several Sensu clusters in Uchiwa as a logical representation of several MCP clusters.

Tenant Telemetry for OpenStack

MCP LMA toolchain provides Tenant Telemetry for OpenStack environments based on the OpenStack Telemetry Data Collection service, or Ceilometer. Tenant Telemetry assists in resource utilization planning and expansion, addresses scalability issues by collecting various OpenStack resource metrics, as well as provides the metrics to such auto-scaling systems as OpenStack Orchestration service, or Heat, that is used to launch stacks of resources, such as virtual machines.

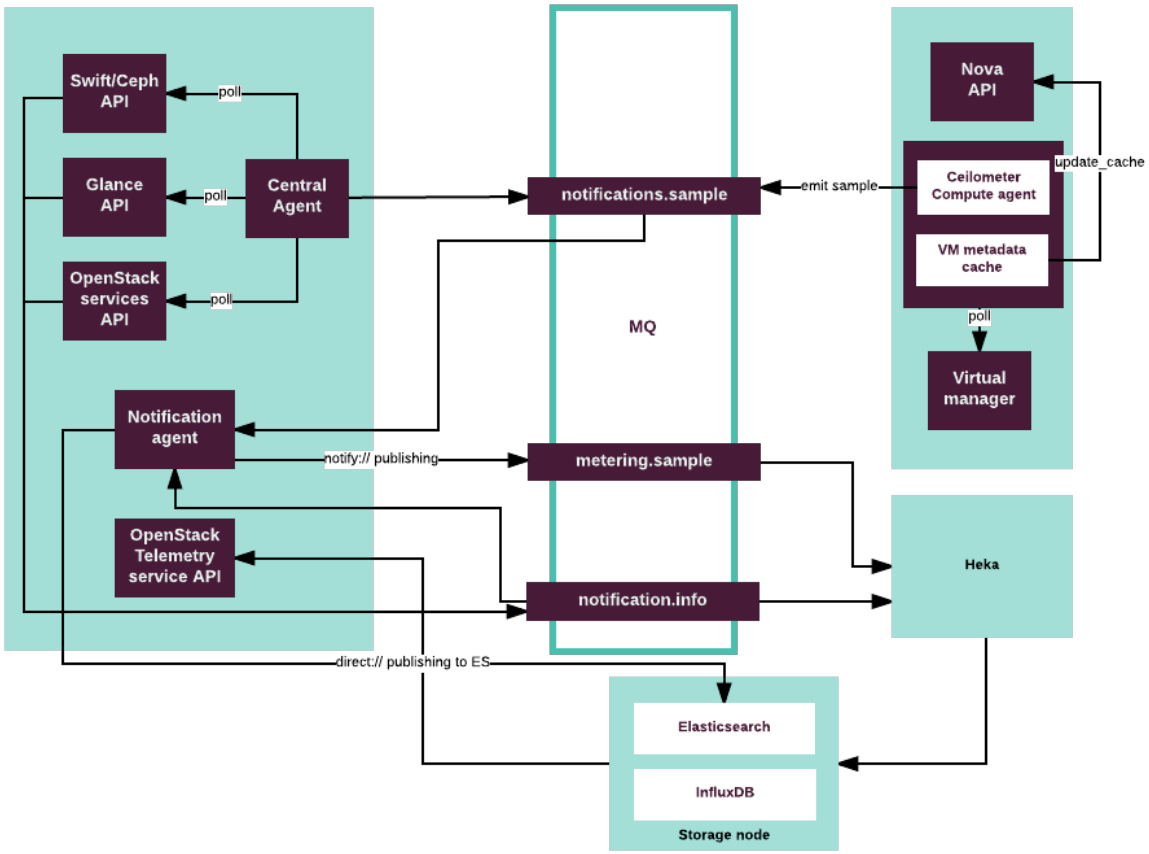
Tenant Telemetry stores scalability metrics in the time-series database called InfluxDB and information about the OpenStack resources in Elasticsearch. In the default Reclass models, Tenant Telemetry shares these resources with other components of the LMA toolchain. Therefore, Tenant Telemetry is deployed together with the LMA toolchain.

By default, Tenant Telemetry supports only sample and statistics API. However, you can enable full Ceilometer API support. Tenant Telemetry implements a complete Ceilometer functionality except complex queries with InfluxDB and Elasticsearch as back ends for samples and events.

Tenant Telemetry supports the community Aodh service that uses the Ceilometer API and provides an alarm evaluator mechanism based on metrics. Aodh allows triggering actions that are based on defined rules against sample or event OpenStack services data that is collected by Ceilometer. After the event-driven alarm evaluation, Aodh provides instant alarm notifications to the user.

Instead of the native Ceilometer Collector service, Heka is used to consume messages for the Ceilometer queues and send them to InfluxDB and Elasticsearch. The Heka Salt formula contains the ceilometer_collector pillar.

The following diagram displays Tenant Telemetry architecture:



Tenant Telemetry uses the Ceilometer agents to collect data and Heka to transfer data to InfluxDB and Elasticsearch. The Ceilometer API is used to retrieve data from back ends and provide it to the end user.

The following table describes the components of Tenant Telemetry:

Tenant Telemetry components

Component	Description
Central agents	Collect metrics from the OpenStack services and sends them to the notifications.sample queue. Central agents run on the virtualized control plane nodes.
Compute agents	Request virtutal instances metadata from Nova API every 10 minutes and send them to the notifications.sample queue. Compute agents run on the compute nodes.

Notification agents	Collect messages from the OpenStack services notification.sample and notifications.info queues and send them to the metering.sample queue. All OpenStack notifications are converted into Ceilometer Events and published to Elasticsearch. Events are published to Elasticsearch using the direct publishing mechanism provided by Ceilometer. Heka does not participate in events processing.
Heka	Processes the data collected from RabbitMQ and OpenStack notifications using a set of Lua plugins and transforms the data into metrics that are sent to InfluxDB and Elasticsearch.

Plan a Kubernetes cluster

Kubernetes is an orchestrator for containerized applications. MCP enables you to deploy a Kubernetes cluster as a standalone deployment or side by side with an OpenStack environment(s). MCP provides lifecycle management of the Kubernetes cluster through the continuous integration and continuous delivery pipeline, as well as monitoring through the MCP Logging, Metering, and Alerting solution.

Kubernetes cluster overview

Kubernetes provides orchestration, scheduling, configuration management, scaling, and updates to the containerized customer workloads. Kubernetes components are typically installed on bare metal nodes.

At a high level, a Kubernetes cluster includes the following types of nodes:

Kubernetes Master node

Runs the services related to the Kubernetes Master Tier, such as the Kubernetes control plane services.

Kubernetes Minion node

Runs user workloads. In MCP, a Kubernetes Minion node is identical to the compute node.

The MCP Kubernetes design is flexible and allows you to install the Kubernetes Master Tier services on an arbitrary number of nodes. For example, some installations may require you to dedicate a node for the etcd cluster members. The minimum recommended number of nodes in the Kubernetes Master Tier for production environments is three. However, for testing purposes, you can deploy all the components on one single node.

The following diagram describes the minimum production Kubernetes installation:

Kubernetes cluster components

A Kubernetes cluster includes Kubernetes components as well as supplementary services and components that run on all or some of the nodes.

The components can be divided into the following types:

Common components

These components run on all nodes in the Kubernetes cluster. The common components include:

- Calico is an SDN solution that provides pure L3 networking to the Kubernetes cluster. Calico includes the following main components that run on every node in the Kubernetes cluster:
  - Calico Felix is a network agent responsible for managing routing tables, network interfaces, and filters on the participating hosts.
  - BIRD is a lightweight BGP daemon that allows for exchange of addressing information between the nodes of the Calico network.
- kube-dns provides discovery capabilities to Kubernetes services.
- kubelet is an agent service of Kubernetes that is responsible for creating and managing Docker containers on the Kubernetes cluster nodes.
- The Container Network Interface (CNI) plugin for Calico SDN establishes a standard for network interface configuration in Linux containers.

Master components

These components run on the Kubernetes Master nodes and provide the control plane functionality. Most of the components run as Docker containers, with a few exceptions that run in static pods in Kubernetes.

- etcd is a distributed key-value store that stores data across the Kubernetes cluster.
- etcd-proxy is a process that redirects requests to available nodes in the etcd cluster.
- kubectl is a command line client for the Kubernetes API that enables cloud operators to execute commands against Kubernetes clusters.
- kubedns services DNS requests, as well as monitors the Kubernetes Master node for changes in Services and Endpoints. The DNS pod includes the following containers kubedns, dnsmasq, and healthz.
- kube-proxy is responsible for the TCP/UDP stream forwarding or round-robin TCP/UDP forwarding across various back ends.
- kube-apiserver is a REST API server that verifies and configures data for such API objects as pods, services, replication controllers, and so on.
- kube-scheduler is a utility that implements functions of workloads provisioning scheduling in pods to specific Kubernetes minions according to workloads capacity requirements, minions allowances and user-defined policies, such as affinity, data localization and other custom restraints. kube-scheduler may significantly affect performance.
- kube-control-manager is a process that embeds the core control loops shipped with Kubernetes, such as the replication controller and so on.

Minion components

These components run on all Kubernetes Minion nodes and include all Common components, as well as etcd-proxy.

Optional components

You may need to install some of these components if your environment has specific requirements:

- The OpenContrail SDN can be installed as an alternative to Calico networking in the MCP clusters that require L2 network connectivity.

> Note
>
> The OpenContrail SDN for Kubernetes is available as technical preview only.

Calico networking considerations

As Kubernetes does not provide native support for networking, MCP uses Calico as an L3 overlay networking provider for all Kubernetes deployments through the Common Network Interface (CNI) plugin. Calico ensures propagation of internal container IP addresses to all Kubernetes Minion nodes over the BGP protocol, as well as provides network connectivity between the containers. Calico uses the etcd key-value store as storage for the configuration data. Mirantis recommends setting up a standalone etcd cluster, separate from the one that Kubernetes uses.

Calico runs in a container called calico-node on every node in the Kubernetes cluster, including both Kubernetes Master nodes and Kubernetes Minion nodes. The calico-node container is controlled by the operating system directly as a systemd service.

The calico-node container incorporates the following main Calico services:

Felix

The primary Calico agent which is responsible for programming routes and ACLs, as well as for all components and services required to provide network connectivity on the host.

BIRD

A BGP client that distributes routing information. confd is a dynamic configuration manager for BIRD, triggered automatically by updates in the configuration data.

Etcd cluster

In the MCP Kubernetes cluster deployment, etcd is used for both Kubernetes components and Calico networking. Etcd is a distributed key-value store that allows you to store data from cluster environments. Etcd is based on the Raft consensus algorithm that ensures fault-tolerance and high performance of the store.

Every instance of etcd can operate in one of the following modes:

Etcd full daemon

In the full daemon mode, an etcd instance participates in Raft consensus and has persistent storage. Three instances of etcd run in full mode on the Kubernetes Master nodes. This ensures quorum in the cluster and resiliency of service.

Etcd native proxy

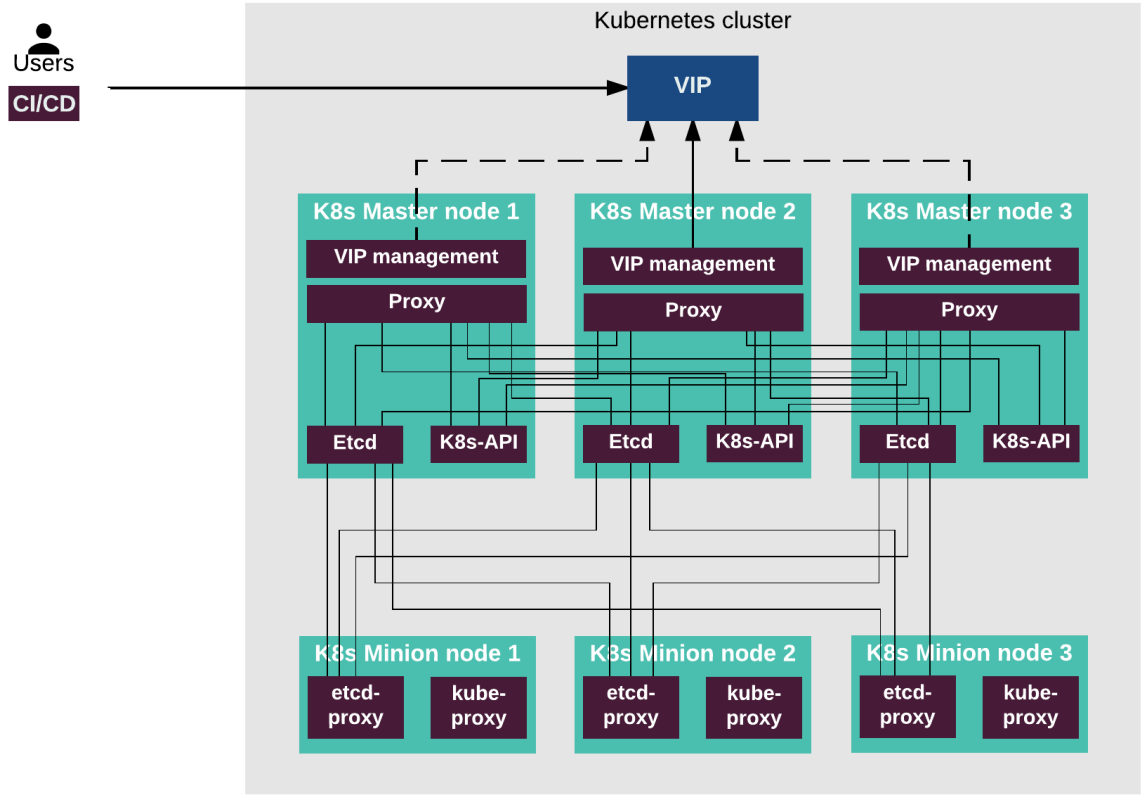
In the native proxy mode, etcd forwards client requests to an available node in the etcd cluster, therefore, acting as a reverse proxy. In this mode, etcd does not take part in the Raft consensus mechanism.

Both etcd and etcd-proxy run as systemd services.

High availability in Kubernetes

The Kubernetes Master Tier is highly-available and works in active-standby mode. MCP installs all control components on all nodes in the Kubernetes Master Tier with one node at a time being selected as a master replica and others running in the stand-by mode. API servers work independently while external or internal Kubernetes load balancer dispatches requests between all of them.

The following diagram describes the API flow in a highly available Kubernetes cluster:



High availability of the proxy server is ensured by the software called HAProxy. HAProxy provides access to the Kubernetes API endpoint by redirecting the requests to instances of kube-apiserver in a round-robin fashion. The proxy server sends API traffic to available back ends and HAProxy prevents the traffic from going to the unavailable nodes. The Keepalived daemon provides VIP management for the proxy server. Optionally, SSL termination can be configured on the HAProxy, so that the traffic to kube-apiserver instances goes over the internal Kubernetes network.

Each of the three Kubernetes Master nodes runs its own instance of kube-apiserver on the localhost address. All Kubernetes Master Tier services work with the Kubernetes API locally, while the services that run on the Kubernetes Minion nodes access the Kubernetes API through the HAProxy server.

Every Kubernetes Master node runs an instance of kube-scheduler and kube-controller-manager. Only one service of each kind is active at a time, while others remain in the warm standby mode. This behavior is controlled by the etcd-based clustering with leaders election.

Kubernetes Master Tier high availability

All Kubernetes Master Tier services run as static pods defined by the kubelet manifests in the /etc/kubernetes/manifests/ directory. One Docker image runs all static pods.

The DNS service is defined in Kubernetes API after installation of the Kubernetes cluster. The kubedns pod is controlled by the replication controller with replica factor of 1, which means that only one instance of the pod is active in a cluster at any time.

The etcd daemons that form the cluster run on the Kubernetes Master nodes. Every node in the cluster also runs the etcd-proxy process. Any service that requires access to the etcd cluster communicates with the local instance of etcd-proxy to reach it. External access to the etcd cluster is restricted.

The Calico node container runs on every node in the cluster, including Kubernetes Master and Minion nodes. The calico-node container runs as a plain Docker container under the control of systemd.

Calico announces address spaces of pods between hosts through BGP protocol. It supports the following nodes of of operation of BGP protocol, configurable in Salt installer:

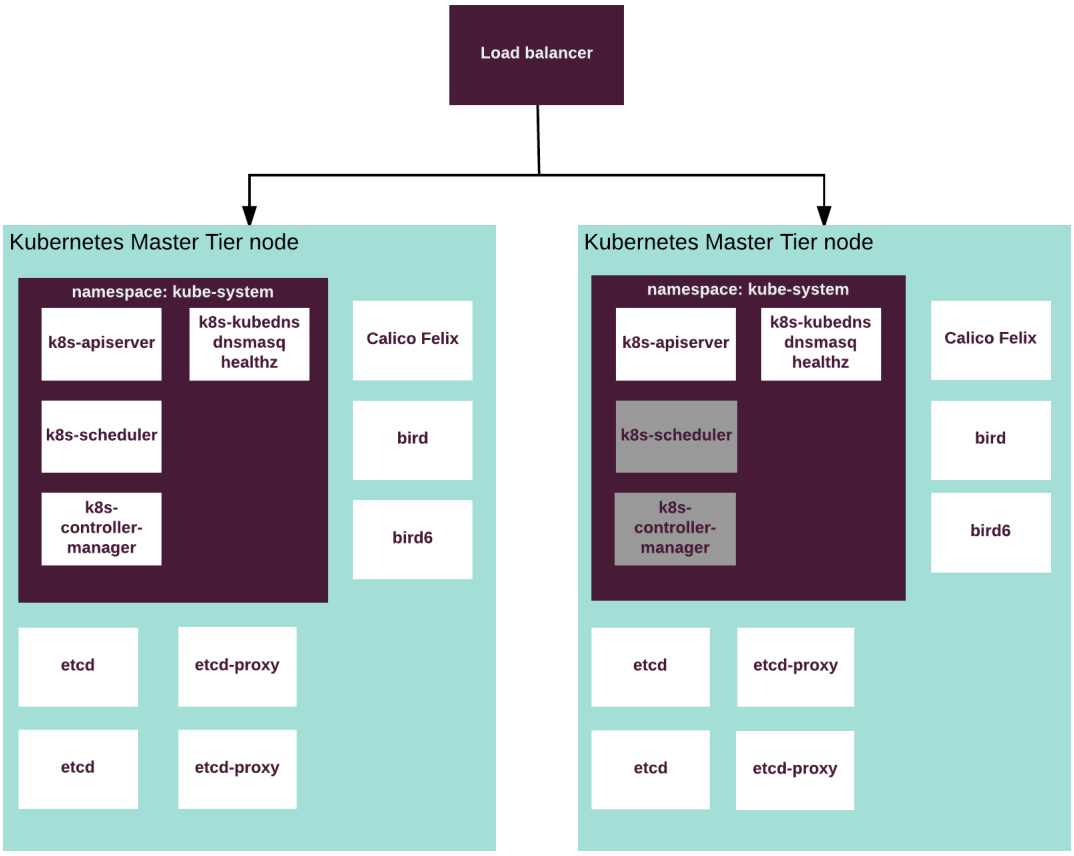
Full mesh BGP

This topology only requires that nodes support BGP using calico-node. The underlay network can have any topology with no specific requirements to ToR hardware. However, in the case of full mesh, number of BGP connections grows exponentially at scale which may result in various performance and capacity issues and makes troubleshooting increasingly difficult. Correspondingly, this configuration only recommended for small environments and testing environments.

ToR BGP peering

This option requires that ToR switches have L3 and BGP capabilities, which increases the cost. On the other hand, it drastically reduces the complexity of the BGP topology, including the number of connections. This configuration recommended for all staging and production environments.

The following diagram describes high availability in Kubernetes Master Tier.


Load balancer
Kubernetes Master Tier node
namespace: kube-system
k8s-apiserver
k8s-kubedns
dnsmasq
healthz
k8s-scheduler
k8s-
controller-
manager
Calico Felix
bird
bird6
etcd
etcd-proxy
etcd
etcd-proxy
Kubernetes Master Tier node
namespace: kube-system
k8s-apiserver
k8s-kubedns
dnsmasq
healthz
k8s-scheduler
k8s-
controller-
manager
Calico Felix
bird
bird6
etcd
etcd-proxy
etcd
etcd-proxy