## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copyright notice</td>
<td>1</td>
</tr>
<tr>
<td>Preface</td>
<td>2</td>
</tr>
<tr>
<td>Intended audience</td>
<td>2</td>
</tr>
<tr>
<td>Documentation history</td>
<td>2</td>
</tr>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>MCP design</td>
<td>3</td>
</tr>
<tr>
<td>Deployment automation</td>
<td>5</td>
</tr>
<tr>
<td>SaltStack and Reclass model</td>
<td>5</td>
</tr>
<tr>
<td>SaltStack repository structure</td>
<td>6</td>
</tr>
<tr>
<td>Deployment templates</td>
<td>7</td>
</tr>
<tr>
<td>Repository planning</td>
<td>8</td>
</tr>
<tr>
<td>Local mirror design</td>
<td>8</td>
</tr>
<tr>
<td>Mirror image content</td>
<td>10</td>
</tr>
<tr>
<td>Infrastructure planning</td>
<td>15</td>
</tr>
<tr>
<td>Overview</td>
<td>15</td>
</tr>
<tr>
<td>DriveTrain overview</td>
<td>18</td>
</tr>
<tr>
<td>CI/CD pipeline overview</td>
<td>18</td>
</tr>
<tr>
<td>High availability in DriveTrain</td>
<td>19</td>
</tr>
<tr>
<td>Multi-cluster architecture</td>
<td>20</td>
</tr>
<tr>
<td>Staging environment</td>
<td>21</td>
</tr>
<tr>
<td>Plan Keycloak identity and authorization</td>
<td>24</td>
</tr>
<tr>
<td>Plan a Ceph Cluster</td>
<td>26</td>
</tr>
<tr>
<td>Plan a Ceph cluster</td>
<td>26</td>
</tr>
<tr>
<td>Ceph services</td>
<td>28</td>
</tr>
<tr>
<td>Plan an OpenStack environment</td>
<td>29</td>
</tr>
<tr>
<td>Virtualized control plane planning</td>
<td>30</td>
</tr>
<tr>
<td>Virtualized control plane overview</td>
<td>30</td>
</tr>
<tr>
<td>Compute nodes planning</td>
<td>32</td>
</tr>
<tr>
<td>Plan Transport Layer Security (TLS)</td>
<td>33</td>
</tr>
<tr>
<td>Network planning</td>
<td>34</td>
</tr>
<tr>
<td>Selecting a network technology</td>
<td>34</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Types of networks</td>
<td>35</td>
</tr>
<tr>
<td>Storage traffic</td>
<td>36</td>
</tr>
<tr>
<td>Neutron OVS use cases</td>
<td>38</td>
</tr>
<tr>
<td>Node configuration</td>
<td>38</td>
</tr>
<tr>
<td>VCP servers network interfaces</td>
<td>38</td>
</tr>
<tr>
<td>Neutron VXLAN tenant networks with network nodes (no DVR)</td>
<td>38</td>
</tr>
<tr>
<td>Network node configuration</td>
<td>39</td>
</tr>
<tr>
<td>Compute nodes configuration</td>
<td>40</td>
</tr>
<tr>
<td>Neutron VXLAN tenant networks with DVR for internal traffic</td>
<td>40</td>
</tr>
<tr>
<td>Network node configuration</td>
<td>41</td>
</tr>
<tr>
<td>Compute nodes configuration</td>
<td>42</td>
</tr>
<tr>
<td>Neutron VLAN tenant networks with network nodes (no DVR)</td>
<td>42</td>
</tr>
<tr>
<td>Network node configuration</td>
<td>43</td>
</tr>
<tr>
<td>Compute nodes configuration</td>
<td>44</td>
</tr>
<tr>
<td>Neutron VXLAN tenant networks with network nodes for SNAT (DVR for all)</td>
<td>45</td>
</tr>
<tr>
<td>Network node configuration</td>
<td>47</td>
</tr>
<tr>
<td>Compute nodes configuration</td>
<td>47</td>
</tr>
<tr>
<td>Neutron VLAN tenant networks with network nodes for SNAT (DVR for both)</td>
<td>48</td>
</tr>
<tr>
<td>Network node configuration</td>
<td>49</td>
</tr>
<tr>
<td>Compute nodes configuration</td>
<td>50</td>
</tr>
<tr>
<td>Plan the Domain Name System</td>
<td>51</td>
</tr>
<tr>
<td>Storage planning</td>
<td>53</td>
</tr>
<tr>
<td>Image storage planning</td>
<td>54</td>
</tr>
<tr>
<td>Block storage planning</td>
<td>54</td>
</tr>
<tr>
<td>Object storage planning</td>
<td>54</td>
</tr>
<tr>
<td>Tenant Telemetry planning</td>
<td>55</td>
</tr>
<tr>
<td>Tenant Telemetry with the legacy StackLight LMA back ends</td>
<td>55</td>
</tr>
<tr>
<td>Standalone Tenant Telemetry</td>
<td>57</td>
</tr>
<tr>
<td>Ironic planning</td>
<td>60</td>
</tr>
<tr>
<td>High availability in OpenStack</td>
<td>62</td>
</tr>
<tr>
<td>Plan a Kubernetes cluster</td>
<td>64</td>
</tr>
<tr>
<td>Kubernetes cluster overview</td>
<td>64</td>
</tr>
<tr>
<td>Kubernetes cluster components</td>
<td>65</td>
</tr>
</tbody>
</table>
Network planning 67
  Types of networks 67
  Calico networking considerations 70
  Plan the OpenContrail networking 70
  Network checker overview 71
  MetalLB support 72
Etcd cluster 73
High availability in Kubernetes 73
OpenStack cloud provider for Kubernetes 74
Virtual machines as Kubernetes pods 75
  Virtlet manager 76
  Virtlet tapmanager 77
  Virtlet vmwrapper 77
  Container Runtime Interface Proxy 78
Kubernetes federation 78
Plan OpenContrail networking 82
  Limitations 82
  Requirements for federated Kubernetes 83
OpenContrail cluster overview 83
  OpenContrail 3.2 cluster overview 84
  OpenContrail 4.0 cluster overview 85
OpenContrail components 89
  OpenContrail 3.2 components 89
  OpenContrail 4.0 components 92
OpenContrail traffic flow 95
  User Interface and API traffic 95
  SDN traffic 95
OpenContrail vRouter 96
OpenContrail HAProxy driver with LBaaSv2 97
OpenContrail and Ironic multi-tenancy integration 98
StackLight planning 101
  DevOps Portal 101
  Dependencies between services 102
<table>
<thead>
<tr>
<th>Service Overview</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Intelligence Service overview</td>
<td>103</td>
</tr>
<tr>
<td>Cloud Health service overview</td>
<td>104</td>
</tr>
<tr>
<td>Runbooks Automation overview</td>
<td>105</td>
</tr>
<tr>
<td>Capacity Management Service overview</td>
<td>106</td>
</tr>
<tr>
<td>Push Notification service overview</td>
<td>107</td>
</tr>
<tr>
<td>Security Audit service overview</td>
<td>108</td>
</tr>
<tr>
<td>Cleanup service overview</td>
<td>109</td>
</tr>
<tr>
<td>Cloud Vector service overview</td>
<td>110</td>
</tr>
<tr>
<td>Hardware Correlation service overview</td>
<td>110</td>
</tr>
<tr>
<td>Heatmaps service overview</td>
<td>111</td>
</tr>
<tr>
<td>Logging, monitoring, and alerting</td>
<td>112</td>
</tr>
<tr>
<td>StackLight LMA overview</td>
<td>112</td>
</tr>
<tr>
<td>StackLight LMA components</td>
<td>113</td>
</tr>
<tr>
<td>StackLight LMA high availability</td>
<td>116</td>
</tr>
<tr>
<td>Monitored components</td>
<td>117</td>
</tr>
</tbody>
</table>
Copyright notice

2018 Mirantis, Inc. All rights reserved.

This product is protected by U.S. and international copyright and intellectual property laws. No part of this publication may be reproduced in any written, electronic, recording, or photocopying form without written permission of Mirantis, Inc.

Mirantis, Inc. reserves the right to modify the content of this document at any time without prior notice. Functionality described in the document may not be available at the moment. The document contains the latest information at the time of publication.

Mirantis, Inc. and the Mirantis Logo are trademarks of Mirantis, Inc. and/or its affiliates in the United States and other countries. Third party trademarks, service marks, and names mentioned in this document are the properties of their respective owners.
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
This is the latest version of the Mirantis Cloud Platform documentation. It is updated continuously to reflect the recent changes in the product. To switch to any release-tagged version of the Mirantis Cloud Platform documentation, use the Current version drop-down menu on the home page.
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:

<table>
<thead>
<tr>
<th>MCP design overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
</tr>
<tr>
<td>DriveTrain</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>MCP clusters</td>
</tr>
<tr>
<td>DevOps portal</td>
</tr>
<tr>
<td>Logging, Monitoring, and Alerting, or StackLight LMA</td>
</tr>
<tr>
<td>Metal-as-a-Service (MaaS)</td>
</tr>
<tr>
<td>Feature</td>
</tr>
<tr>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>High Availability ensured by Keepalived and HAProxy</td>
</tr>
<tr>
<td>OpenContrail (optional)</td>
</tr>
<tr>
<td>Ceph cluster (optional)</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Service class</th>
<th>System class</th>
<th>Cluster class</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>A system class defines a role for a node, 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.</td>
<td>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.</td>
</tr>
</tbody>
</table>

**SaltStack repository structure**

The repository structure allows deployment engineers to store and deploy multiple MCP cluster profiles. This is especially beneficial for multi-cluster 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.

DriveTrain provides a pipeline for generating a Reclass cluster model with Cookiecutter based on parameters provided by operator through Jenkins UI.
Repository planning

The MCP software is provided as packages, containers, and configuration metadata. The packages are available through APT repositories from a combination of sources. Containers are available in the Docker image format at the Mirantis Docker registry. The configuration metadata is stored in Git repositories. The MCP Release Notes: Release artifacts section in the related MCP release documentation lists the repositories and Docker registry sources. The Salt Master node requires access to APT, Docker, and Git repositories, while all other nodes in the environment require access to the APT and Docker repositories only.

Even though it is possible to use the Mirantis and third-party repositories and registry directly with the Internet access, Mirantis recommends using local mirrors for the following reasons:

- **Repeatability**
  You can redeploy clouds exactly the same way including all dependencies.

- **Control**
  You have control over when and which packages to upgrade. By default, apt-get dist-upgrade updates the packages to the latest available version. And with a local mirror, you control when a new update is available.

- **Security**
  This is a good security practice not to download artifacts from the Internet but to control what software gets delivered into the datacenter.

To create the local mirrors and registries, the Internet access is required. Otherwise, you need to ship all artifacts manually through a medium, such as an external hard drive. By default, the local aptly repositories and local Docker registry are provided by DriveTrain. Though, MCP provides the flexibility to create local mirrors and registry anywhere within your datacenter. For more information about the local mirror design, see *Local mirror design*.

Local mirror design

The main scenarios for using local mirrors and registries include:

- **Online deployment**
  There is the Internet access from the DriveTrain node during deployment. DriveTrain is then used to set up the local mirrors and registries.

- **Offline deployment**
  There is no Internet access from any of the MCP nodes. The local mirrors and registry are not part of an MCP cluster. The existing local mirrors can be used and populated with the repositories needed by the MCP deployment. This scenario is more common at telephone and financial companies.

By default, MCP deploys local mirrors with packages in a Docker container on the DriveTrain nodes with GlusterFS volumes using the online scenario.

You can manage the local mirrors using the `aptly mirror` command.
Note

A mirror can only be updated as a whole. Individual package updates are not supported. The complete mirror of the repositories used in MCP may consume several hundreds of gigabytes of disk space.

This section explains the details of the scenarios above.

**Online deployment**

The online deployment scenario assumes that there is the Internet access from the DriveTrain nodes during deployment. By default, the local Gerrit (Git), aptly, and Docker registry are run as part of DriveTrain.

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

```
```

The high-level workflow for the online deployment using local mirrors is as follows:

1. Create a deployment model that has the default repositories included (requires the Internet access).
2. Deploy MCP DriveTrain using the MCP Deployment Guide.
3. Configure the aptly mirror to mirror the repositories described in the MCP Release Notes: Release artifacts section in the related MCP release documentation.
4. Modify the deployment model to utilize the local mirrors that are provided by DriveTrain.
5. Deploy the MCP cluster using DriveTrain.

**Offline deployment**

The offline deployment scenario assumes that there is no Internet access from any of the nodes inside the MCP cluster including DriveTrain. The requirement is that the MCP cluster instead has access to the already prepared Debian repository, Docker registry, Git server, HTTP server with QCOW images, and PyPi server. Mirantis provides a pre-configured QCOW image with already mirrored Debian packages using Aptly, Docker images using Docker Registry, Git repositories needed for MCP deployment, QCOW images, and Python packages. The actual content of this image is described in *Mirror image content*.

The high-level workflow of the offline deployment is as follows:
1. Create the deployment model using the Model Designer UI as described in the MCP Deployment Guide checking the offline option (requires the Internet access).

2. Run a VM using the mirror image.

3. Deploy the MCP DriveTrain and MCP OpenStack environment using the MCP Deployment Guide.

Note
The offline QCOW image enables you to deploy an MCP OpenStack environment with the services and features as per an OpenStack release version announced in the corresponding MCP release.

Seealso
- GitLab Repository Mirroring
- The aptly mirror

Mirror image content
The mirror image delivered for the offline MCP deployment includes:

- Services and ports
- Mirrored Debian package repositories
- Mirrored Docker images
- Mirrored Git repositories
- Mirrored QCOW images

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
<th>Protocol/Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aptly</td>
<td>Serves Debian Packages</td>
<td>HTTP/80</td>
</tr>
<tr>
<td>Registry</td>
<td>Serves Docker images</td>
<td>HTTP/5000</td>
</tr>
<tr>
<td>Git</td>
<td>Serves Git Repositories</td>
<td>HTTP/8088</td>
</tr>
<tr>
<td>HTTP</td>
<td>Serves QCOW images and other files</td>
<td>HTTP/8078</td>
</tr>
</tbody>
</table>

Mirrored Debian package repositories

<table>
<thead>
<tr>
<th>Package</th>
<th>Repository URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ceph Luminous</td>
<td><a href="http://apt.mirantis.com/xenial/">http://apt.mirantis.com/xenial/</a> MCP_RELEASE ceph-luminous</td>
</tr>
<tr>
<td>MCP extra packages</td>
<td><a href="http://apt.mirantis.com/xenial/">http://apt.mirantis.com/xenial/</a> MCP_RELEASE extra</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>MCP OS Mitaka additional packages</td>
<td><a href="http://apt.mirantis.com/xenial/">http://apt.mirantis.com/xenial/</a> MCP_RELEASE mitaka</td>
</tr>
<tr>
<td>MCP OS Newton additional packages</td>
<td><a href="http://apt.mirantis.com/xenial/">http://apt.mirantis.com/xenial/</a> MCP_RELEASE newton</td>
</tr>
<tr>
<td>MCP OS Ocata additional packages</td>
<td><a href="http://apt.mirantis.com/xenial/">http://apt.mirantis.com/xenial/</a> MCP_RELEASE ocata</td>
</tr>
<tr>
<td>Opencontrail 3.2.3</td>
<td><a href="http://apt.mirantis.com/xenial/">http://apt.mirantis.com/xenial/</a> MCP_RELEASE oc323</td>
</tr>
<tr>
<td>Salt formulas</td>
<td><a href="http://apt.mirantis.com/xenial/">http://apt.mirantis.com/xenial/</a> MCP_RELEASE salt</td>
</tr>
<tr>
<td>MCP Update packages</td>
<td><a href="http://apt.mirantis.com/xenial/">http://apt.mirantis.com/xenial/</a> MCP_RELEASE updates</td>
</tr>
<tr>
<td>cassandra</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/cassandra/">http://mirror.mirantis.com/MCP_RELEASE/cassandra/</a> 21x main</td>
</tr>
<tr>
<td>docker</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/docker/">http://mirror.mirantis.com/MCP_RELEASE/docker/</a> xenial stable</td>
</tr>
<tr>
<td>docker-1.x</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/docker-1.x/">http://mirror.mirantis.com/MCP_RELEASE/docker-1.x/</a> ubuntu-xenial main</td>
</tr>
<tr>
<td>elasticsearch-2.x</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/elasticsearch-2.x/">http://mirror.mirantis.com/MCP_RELEASE/elasticsearch-2.x/</a> stable main</td>
</tr>
<tr>
<td>elasticsearch-5.x</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/elasticsearch-5.x/">http://mirror.mirantis.com/MCP_RELEASE/elasticsearch-5.x/</a> stable main</td>
</tr>
<tr>
<td>elasticsearch-curactor</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/elasticsearch-curator/">http://mirror.mirantis.com/MCP_RELEASE/elasticsearch-curator/</a> stable main</td>
</tr>
<tr>
<td>glusterfs-3.8</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/glusterfs-3.12/">http://mirror.mirantis.com/MCP_RELEASE/glusterfs-3.12/</a> xenial main</td>
</tr>
<tr>
<td>glusterfs-3.8</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/glusterfs-3.8/">http://mirror.mirantis.com/MCP_RELEASE/glusterfs-3.8/</a> xenial main</td>
</tr>
<tr>
<td>influxdb</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/influxdb/">http://mirror.mirantis.com/MCP_RELEASE/influxdb/</a> xenial stable</td>
</tr>
<tr>
<td>kibana-4.6</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/kibana-4.6/">http://mirror.mirantis.com/MCP_RELEASE/kibana-4.6/</a> stable main</td>
</tr>
<tr>
<td>maas</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/maas/">http://mirror.mirantis.com/MCP_RELEASE/maas/</a> xenial main</td>
</tr>
<tr>
<td>saltstack</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/saltstack/">http://mirror.mirantis.com/MCP_RELEASE/saltstack/</a> xenial main</td>
</tr>
<tr>
<td>td-agent-trusty</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/td-agent-trusty/">http://mirror.mirantis.com/MCP_RELEASE/td-agent-trusty/</a> trusty contrib</td>
</tr>
<tr>
<td>td-agent-xenial</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/td-agent-xenial/">http://mirror.mirantis.com/MCP_RELEASE/td-agent-xenial/</a> xenial contrib</td>
</tr>
<tr>
<td>ubuntu-xenial</td>
<td><a href="http://mirror.mirantis.com/MCP_RELEASE/ubuntu/">http://mirror.mirantis.com/MCP_RELEASE/ubuntu/</a></td>
</tr>
</tbody>
</table>

Mirrored Docker images

<table>
<thead>
<tr>
<th>Image name</th>
</tr>
</thead>
<tbody>
<tr>
<td>${_param:docker_artifactory_mirror}/mirantis/external:MCP_RELEASE</td>
</tr>
</tbody>
</table>

©2018, Mirantis Inc.
<table>
<thead>
<tr>
<th>Mirrored Git repositories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository name</td>
</tr>
<tr>
<td>---------------------------</td>
</tr>
<tr>
<td>rundeck-cis-jobs</td>
</tr>
<tr>
<td><a href="mailto:git@github.com">git@github.com</a>:Mirantis/rundeck-cis-jobs.git</td>
</tr>
<tr>
<td>reclass-system-salt-model</td>
</tr>
<tr>
<td><a href="mailto:git@github.com">git@github.com</a>:Mirantis/reclass-system-salt-model</td>
</tr>
<tr>
<td>pipeline-library</td>
</tr>
<tr>
<td><a href="mailto:git@github.com">git@github.com</a>:Mirantis/pipeline-library</td>
</tr>
<tr>
<td>mk-pipelines</td>
</tr>
<tr>
<td><a href="mailto:git@github.com">git@github.com</a>:Mirantis/mk-pipelines</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mirrored QCOW images</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image name</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>ubuntu-14-04-x64-MCP_RELEASE.qcow2</td>
</tr>
<tr>
<td>ubuntu-16-04-x64-MCP_RELEASE.qcow2</td>
</tr>
</tbody>
</table>

1. The repositories are tagged with MCP_RELEASE
Infrastructure 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.

MCP employs modular architecture approach by using the Reclass model to describe configuration and distribution of services across the infrastructure nodes. This allows adjusting the installation scale of the MCP Management Plane and clusters by changing the Reclass model.

Overview

Infrastructure nodes are the nodes that run all required services for the MCP cluster deployment, lifecycle management, and monitoring, as well as virtual machines of Virtualized Control Plane.

The exact number of the infrastructure nodes in each MCP environment and distribution of the MCP components across the infrastructure nodes are unique for each use case and are defined in the deployment model. See recommendations in the MCP Standard Configuration guide.

Typically, 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, Monitoring, Alerting or StackLight LMA
- The StackLight 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 Salt Master.

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. Typically, in production environments, VCP services run on dedicated infrastructure nodes.

OpenContrail Controller (with MCP OpenContrail only)
- The OpenContrail controller node includes package-based services of MCP OpenContrail, such as the API server and configuration database.

OpenContrail Analytics (with MCP OpenContrail only)
- The OpenContrail analytics node includes package-based services for MCP OpenContrail metering and analytics, such as the Cassandra database for analytics and data collectors.
Note
If you run MCP OpenContrail SDN, you need to have Juniper MX or SRX hardware or virtual router to route traffic to and from compute nodes.

Network node (with Open vSwitch only)
Network node is a special kind of infrastructure node that is 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 components mapping of the infrastructure and foundation node for a mixed MCP cluster with OpenStack, Kubernetes, and Ceph.

Note
You can use either Neutron or OpenContrail SDN, but not both at the same time.
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 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 and include SaltStack, Reclass, Jenkins, MCP Registry (aptly, Docker registry), and Gerrit.

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
Executes the jobs that use SaltStack API to run Salt formulas with Reclass model 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:
An operator submits changes to a Reclass model or a SaltStack formula in Gerrit for review and approval.

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, with staging MCP cluster, you trigger deployment job in Jenkins before merging the change. This allows you to verify it before promoting to production. However, if you deploy onto production MCP cluster, you might want to approve and merge the change first.

Jenkins job invokes the required SaltStack formulas and Reclass models from Gerrit and artifacts from the MCP Registry.

SaltStack applies changes to the cloud environment.

### 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 cluster 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 cluster 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:
Multi-cluster architecture

Mirantis Cloud Platform (MCP) can manage multiple disparate clusters using the same DriveTrain and infrastructure node installation. The following clusters are supported:

- 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.

DriveTrain uses a single data model structure to describe a multi-cluster configuration of MCP. This structure resides in a single Git repository. Each cluster is defined by one directory in the repository directory tree.

A Jenkins deployment pipeline enables you to specify the URL and credentials of the Salt Master API endpoint that will be called upon the execution of the pipeline. Use the following pipeline parameters to designate the Salt Master service:

- SALT_MASTER_URL
- SALT_MASTER_CREDENTIALS

The targeted Salt Master node then distributes appropriate changes to targeted nodes.
MAAS and StackLight LMA do not support multi-cluster environments. These components are installed per cluster and used only for that cluster.

One of the most common use cases of a multi-cluster architecture is the installation of a staging cluster next to a production one. The staging environment is managed by the same instance of DriveTrain as the production cluster.

The following diagram describes a high-level multi-cluster architecture:

Staging environment
Mirantis recommends creating a staging environment for any production purposes. Thus, a typical MCP installation should consist of at least two clusters of the same kind (OpenStack or Kubernetes): for staging and production.
Mirantis recommends you install the staging environment first and reuse as much as possible of the Reclass cluster model of the staging environment to deploy production environment(s). Having a staging environment with a control plane topology that differs from the production environment is considered impractical.

Consider installing a staging environment for your production environment if:

- You will run mission-critical workloads in your production environment.
- You plan to install more than one production environment in the same or similar topology.
- You plan to develop custom components or integrate an external service with MCP.

In any case, the staging environment provides you with a testing sandbox to test any changes in configuration or versions of artifacts before you apply it to your production environment.

### Note

DriveTrain pipelines allow you to select what Salt Master node should be called during the particular build/run of the pipeline. This allows you to target staging and production environments separately. See more details in the *Multi-cluster architecture* section.

---

## Configuration update

Mirantis recommends you follow this procedure to create, test, and promote the changes using a staging environment:

1. Create a change in the Reclass model for a staging environment in the DriveTrain Gerrit repository.
2. Merge the change into the Reclass model for the staging environment.
3. Run the pipeline in DriveTrain Jenkins to apply the configuration change of the staging environment and verify the change.
4. Cherry pick the change to the Reclass model for the production environment in DriveTrain Gerrit.
5. Merge the change into the Reclass model for the production environment.
6. Run the pipeline in DriveTrain Jenkins to apply the configuration change of the production environment.

For detailed steps on how to change service configuration, see *MCP Operations Guide: Change service configuration*.

## Package update

Mirantis recommends you follow this procedure to promote updated packages using a staging cluster:

1. Sync the package(s) from the upstream repository to the local aptly mirror.
2. Run the Update System Package(s) pipeline on the staging environment and verify the status of update.
3. Run the Update System Package(s) pipeline on the production environment and verify the status of update.

For detailed steps on how to update packages, see *MCP Operations Guide: Update service packages*. 
Note

A limited number of the packages is pinned to a specific version in the Reclass cluster model. Those packages must be updated using the configuration update procedure and the Linux state must be applied to update the packages.
Plan Keycloak identity and authorization

Keycloak is an open-source identity and access management solution that provides a single entry point for MCP deployments. Keycloak supports the single sign-on functionality that enables you to authenticate with Keycloak rather than with individual services. For example, once logged in to Keycloak you do not have to log in again to access a different MCP service. You can also use different social networks and other identity providers as authentication backgrounds, connect to existing LDAP or Active Directory servers to synchronize users and groups, and so on. Keycloak Admin Console enables administrators to manage users, groups, and clients, control the authorization policies and enable or disable user federation. For more information about the Keycloak functionality, see About Keycloak.

Keycloak includes the following components:

Keycloak cluster
To ensure high availability, Mirantis recommends setting up Keycloak in a clustered mode. Keycloak runs as a JAVA application within the WildFly server. The available clustered configurations include the standalone clustered mode and the domain clustered mode.

Keycloak proxy
Keycloak has an HTTP(S) proxy that you can put in front of web applications and services where it is not possible to install a Keycloak adapter. You can set up URL filters so that certain URLs are secured either by a browser login and/or bearer token authentication. You can also define role constraints for URL patterns within your applications.

Keycloak proxy is also used to protect the services that do not have a built-in authentication mechanism, for example, Prometheus, Elasticsearch, and Kibana.

Database
In MCP, Keycloak uses the Galera MySQL cluster, deployed with OpenStack, as a database.

Deployment of a Keycloak cluster requires three virtual nodes, each running a Keycloak application. To enable concurrent connections, for example, to keep the authentication sessions open, Keycloak uses the HAProxy service that provides the load balancing capabilities.

The following diagram illustrates the Keycloak architecture in MCP:
Seealso

Keycloak documentation
Plan a Ceph Cluster

Proper planning is crucial for building a reliable, resilient, and performant Ceph cluster. The following pages will advise on planning a Ceph cluster suitable for your performance and resilience requirements.

Plan a Ceph cluster

You can use Ceph as a primary solution for all storage types, including image storage, ephemeral storage, persistent block storage, and object storage. When planning your Ceph cluster, consider the following:

Ceph version

The supported stable versions are Luminous and Jewel. For new deployments, Mirantis recommends using the Luminous release to obtain the latest performance improvements and bug fixes.

Daemon colocation

Ceph uses the Ceph Monitor (ceph.mon), object storage (ceph.osd), Ceph Manager (ceph-mgr) and RADOS Gateway (ceph.radosgw) daemons to handle the data and cluster state. Each daemon requires different computing capacity and hardware optimization.

Mirantis recommends running Ceph Monitors and RADOS Gateway daemons on dedicated virtual machines or physical nodes. Colocating the daemons with other services may negatively impact cluster operations. Three Ceph Monitors are required for a cluster of any size. If you have to install more than three, the number of Monitors must be odd.

Ceph Manager is installed on every node (virtual or physical) running Ceph Monitor, to achieve the same level of availability.

Store type

Ceph can use either the BlueStore or FileStore back end. The BlueStore back end typically provides better performance than FileStore because in BlueStore the journaling is internal and more lightweight compared to FileStore. Mirantis supports BlueStore only for Ceph versions starting from Luminous.

For more information about Ceph back ends, see Storage devices.

- **BlueStore configuration**
  
  BlueStore uses Write-Ahead Log (WAL) and Database to store the internal journal and other metadata. WAL and Database may reside on a dedicated device or on the same device as the actual data (primary device).

  - **Dedicated**
    
    Mirantis recommends using a dedicated WAL/DB device whenever possible. Typically, it results in better storage performance. One write-optimized SSD is recommended for WAL/DB metadata per five primary HDD devices.

  - **Colocated**
    
    WAL and Database metadata are stored on the primary device. Ceph will allocate space for data and metadata storage automatically. This configuration may result in slower storage performance in some environments.

- **FileStore configuration**

  Mirantis recommends using dedicated write-optimized SSD devices for Ceph journal partitions. Use one journal device per five data storage devices.
It is possible to store the journal on the data storage devices. However, Mirantis does not recommend it unless special circumstances preclude the use of dedicated SSD journal devices.

Ceph cluster networking

A Ceph cluster requires having at least the front-side network, which is used for client connections (public network in terms of Ceph documentation). Ceph Monitors and OSDs are always connected to the front-side network.

To improve the performance of the replication operations, you may additionally set up the back-side network (or cluster network in terms of Ceph documentation), which is used for communication between OSDs.

Mirantis recommends assigning dedicated interface to the cluster network. Bonding interfaces and dedicating VLANs on bonded interfaces is not supported.

For more details on Ceph cluster networking, see Ceph Network Configuration Reference.

Pool parameters

Set up each pool according to expected usage. Consider at least the following pool parameters:

- min_size sets the minimum number of replicas required to perform I/O on the pool.
- size sets the number of replicas for objects in the pool.
- type sets the pool type, which can be either replicated or erasure.

The following diagram shows the architecture of a Ceph cluster:

See also

- Ceph services
- Ceph Network Configuration Reference
Ceph services

When planning a Ceph cluster, consider the following guidelines for the Ceph Monitor and RADOS Gateway services.

Ceph Monitor service

The Ceph Monitor service is quorum-based. Three instances of Ceph Monitor are required to ensure fault tolerance and typically suffice for any number of OSD nodes. If additional Ceph Monitors are required for any reason, the total number of instances must be odd.

Run one Ceph Monitor per physical infrastructure node to minimize the risk of losing the Monitor quorum upon server hardware failure.

The recommended minimal size of a Ceph Monitor VM is:

- 4 vCPU
- 16 GB RAM
- 32 GB disk
- 10 GbE network

RADOS Gateway service

The RADOS Gateway service is required to provide Swift and S3-compatible Object Storage API on top of Ceph. RADOS Gateway is stateless and puts no constraints on the number of nodes in a cluster. Start with two instances behind a load balancer for redundancy in case of failure.

Mirantis recommends running the RADOS Gateway service on a dedicated VM with at least:

- 4 vCPU
- 8 GB RAM
- Minimum 16 GB disk space for the operating system
- 10 GbE network connection internally and externally

RADOS Gateway scales out horizontally, so you can increase the number of VMs to get more concurrent connections processed.
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 recommended configurations in MCP Standard Configuration: OpenStack environment scaling.

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 table describes the MCP virtualized control plane. Modular architecture enables you to install VCP with any subset of components listed below.

<table>
<thead>
<tr>
<th>VCP services</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Platform</strong></td>
</tr>
<tr>
<td>OpenStack</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Object Storage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Image Storage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Networking</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
| Back-end services | • Proxy (NGINX)  
|                  | • GlusterFS  
|                  | • RabbitMQ  
|                  | • MySQL/Galera  

| Logging, Monitoring, and Alerting (LMA) | • StackLight LMA  

**Note**

MCP requires that Ceph OSDs run on dedicated hardware servers. This reduces operations complexity, isolates the failure domain, and helps avoid resources contention.

**Seealso**

*MCP Standard Configuration: OpenStack environment scaling*
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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead components</td>
<td>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.</td>
</tr>
<tr>
<td>HW Components</td>
<td>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.</td>
</tr>
<tr>
<td>Oversubscription ratio</td>
<td>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.</td>
</tr>
<tr>
<td>Flavor definitions</td>
<td>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.</td>
</tr>
<tr>
<td>Flavor totals</td>
<td>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.</td>
</tr>
<tr>
<td>Resource utilization per compute node</td>
<td>The resource utilization parameter defines the percentage of memory, processing, 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.</td>
</tr>
</tbody>
</table>
Plan Transport Layer Security (TLS)

The Transport Layer Security (TLS) cryptographic protocol enables you to provide a secured encrypted communication for the client-server OpenStack applications as well as for the RabbitMQ and MySQL back ends of an MCP OpenStack environment. TLS protects the communications in your MCP cluster from trespassing and eavesdropping.

By default, only the traffic transmitted over public networks is encrypted. If you have specific security requirements, you may want to configure internal communications to connect through encrypted channels.

See also
- MCP Deployment Guide: Enable TLS support
- Introduction to TLS and SSL in the OpenStack Security Guide
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:

<table>
<thead>
<tr>
<th>OpenContrail vs Neutron OVS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OpenContrail</strong></td>
</tr>
<tr>
<td>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:</td>
</tr>
<tr>
<td>- Service chaining</td>
</tr>
<tr>
<td>- MPLS over UDP/GRE with vMX router</td>
</tr>
<tr>
<td>- Multi-site SDN</td>
</tr>
<tr>
<td>- Network analytics</td>
</tr>
</tbody>
</table>

See also

*Plan OpenContrail networking*
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.

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:

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
<th>IP Address</th>
<th>VLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>br-mesh</td>
<td>Tenant overlay traffic (VXLAN)</td>
<td>Routed, Subnet</td>
<td>Leaf switch only</td>
</tr>
<tr>
<td>br-mgm</td>
<td>Openstack and other management traffic</td>
<td>Routed, Subnet</td>
<td>Leaf switch only</td>
</tr>
<tr>
<td>br-stor</td>
<td>Storage traffic</td>
<td>Routed, Subnet</td>
<td>Leaf switch only</td>
</tr>
<tr>
<td>eth0</td>
<td>PXE Boot traffic</td>
<td>VLAN, Subnet, Default</td>
<td>Global</td>
</tr>
<tr>
<td>br-prv</td>
<td>Tenant VLAN traffic bridge</td>
<td>VLAN range</td>
<td>Global</td>
</tr>
<tr>
<td>br-floating</td>
<td>External VLAN traffic bridge</td>
<td>VLAN range</td>
<td>Global</td>
</tr>
</tbody>
</table>

VCP servers network interfaces

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

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
<th>IP Address</th>
<th>VLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>br-pxe</td>
<td>PXE Boot traffic</td>
<td>VLAN, Subnet</td>
<td>Global</td>
</tr>
<tr>
<td>br-mgm</td>
<td>Openstack and other management traffic</td>
<td>Routed, Subnet</td>
<td>Leaf switch only</td>
</tr>
<tr>
<td>br-stor</td>
<td>Storage traffic</td>
<td>Routed, Subnet</td>
<td>Leaf switch only</td>
</tr>
<tr>
<td>eth0</td>
<td>PXE Boot traffic</td>
<td>VLAN, Subnet, Default</td>
<td>Global</td>
</tr>
</tbody>
</table>

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-mgm 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 (management 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 opensvswitch.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 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.

![Diagram of 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-mgm 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 (management 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 physonet1 into the Neutron provider networks. br-floating is patched with br-prv which is mapped as physonet2 for VLAN tenant network traffic. br-mgm 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-mgm 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-mgm 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 (management 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-mgm 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-mgm 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:
Plan the Domain Name System

MCP leverages the OpenStack Domain Name System as a Service component called Designate to provide DNS with integrated Keystone authentication for OpenStack environments. Designate uses the Galera MySQL cluster as the distributed database to provide a mapping of IP addresses to domain names and hosts to access the Internet.

Designate uses RESTful API for creating, updating, and deleting DNS zones and records of OpenStack environments. Designate integrates Nova and Neutron notifications for auto-generated records as well as uses different underlying DNS servers including BIND9 and PowerDNS that are supported by MCP.

Designate includes the following components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>designate-api</td>
<td>Processes API requests by sending them to designate-central using the Remote Procedure Call (RPC) mechanism.</td>
</tr>
<tr>
<td>designate-central</td>
<td>Handles RPC requests using message queueing, coordinates persistent storage, and applies business logic to data from designate-api. Storage is provided using the SQLAlchemy plugin supporting MySQL.</td>
</tr>
<tr>
<td>designate-worker</td>
<td>Runs the zone create, zone update, and zone delete tasks as well as tasks from designate-producer.</td>
</tr>
<tr>
<td>designate-producer</td>
<td>Manages periodic Designate tasks.</td>
</tr>
<tr>
<td>designate-pool-manager</td>
<td>(Optional) Manages the states of the DNS servers that are handled by Designate.</td>
</tr>
<tr>
<td>designate-zone-manager</td>
<td>(Optional) Handles all periodic tasks of the zone shard that designate-zone-manager is responsible for.</td>
</tr>
<tr>
<td>designate-mdns</td>
<td>Pushes the DNS zone information to the customer-facing DNS servers. Can also pull information about the DNS zones hosted outside of the Designate infrastructure.</td>
</tr>
<tr>
<td>designate-sink</td>
<td>Consumes the Nova and Neutron events and notifications to produce auto-generated records that are determined by custom notification handlers.</td>
</tr>
<tr>
<td>Back end (BIND9 or PowerDNS)</td>
<td>Represents your existing DNS servers or DNS servers deployed on separate VMs of the MCP infrastructure nodes.</td>
</tr>
</tbody>
</table>

All components except the back end can run on the MCP Virtualized Control Plane (VCP) as a part of the OpenStack API.

See also

- Designate OpenStack documentation
- Designate architecture with the Pool Manager role
- Designate architecture with the Worker role
Storage planning

Depending on your workload requirements, 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 your MCP cluster with the following types of storage:

Image storage
   Storage required for storing disk images that are used in the OpenStack environment.

Ephemeral block storage
   Storage for the operating systems of virtual servers in an OpenStack environment, bound to the life cycle of the instance. As its name suggests, the storage will be deleted once the instance is terminated. Ephemeral storage does persist through a reboot of a virtual server instance.

Persistent block storage
   Block storage that exists and persists outside a virtual server instance. It is independent of virtual machine images or ephemeral disks and can be attached to virtual servers.

Object storage
   Storage for unstructured data with capabilities not provided by other types of storage, such as separation of metadata from other data and an Application Programming Interface (API).
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

A highly scalable distributed object storage that is recommended as an Image storage for environments with a large number of images and/or snapshots. If used as a back end for both image storage and ephemeral storage, Ceph can eliminate caching of images on compute nodes and enable copy-on-write of disk images, which in large clouds can save a lot of storage capacity.

GlusterFS

A distributed network file system that allows you to create a reliable and redundant data storage for image files. This is the default option for an Image store with the File back end in MCP.

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 low latency operations, such as databases operations, you can configure a combination of multiple Cinder block-storage back ends. This enables the use of a Ceph cluster for a general-purpose storage, and a separate storage back end or type for applications that require low latency or other performance characteristics.

To achieve this functionality:

- Use the Cinder multi-backend capability
- Configure a Ceph cluster as a general-purpose back end
- Configure a high-performance third-party block storage solution as an additional back end by using Cinder drivers.

Object storage planning

Ceph is the preferred option for most use cases that involve Object Storage.

Ceph provides a robust, reliable, and easy to manage object storage solution. RADOSGW service of Ceph provides Object Store API which is compatible with Swift and AWS S3 APIs.

To scale Ceph object storage performance, add instances of RADOSGW service to a back end of a load balancer, for example, the HAProxy service.
Tenant Telemetry planning

MCP 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.

If your OpenStack version is prior to Pike, deploy Tenant Telemetry that uses the legacy StackLight LMA back ends. If your OpenStack version is Pike, deploy the standalone Tenant Telemetry that uses its own back ends.

Caution!

Standalone Tenant Telemetry does not support integration with StackLight LMA.

Tenant Telemetry with the legacy StackLight LMA back ends

Caution!

Tenant Telemetry that uses the legacy StackLight LMA back ends is deprecated for OpenStack Pike.

Tenant Telemetry that uses the legacy StackLight LMA back ends 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 StackLight LMA toolchain. Therefore, Tenant Telemetry is deployed together with the StackLight 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 the 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:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central agents</td>
<td>Collect metrics from the OpenStack services and send them to the notifications.sample queue. Central agents run on the virtualized control plane nodes.</td>
</tr>
<tr>
<td>Compute agents</td>
<td>Request virtual instances metadata from Nova API every 10 minutes and send them to the notifications.sample queue. Compute agents run on the compute nodes.</td>
</tr>
</tbody>
</table>
### 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.

### Standalone Tenant Telemetry

**Caution!**

Standalone Tenant Telemetry is supported starting from the Pike OpenStack release and does not support integration with StackLight LMA.

Standalone Tenant Telemetry stores scalability metrics in the time-series database called Gnocchi and events in Panko. By default, Panko uses MySQL as a back end with the same Galera cluster as for the OpenStack API. Gnocchi uses the following back ends:

- MySQL Galera cluster as indexing storage (using the same MySQL cluster as the OpenStack API)
- Redis as incoming metrics storage set up on the same nodes as Tenant Telemetry
- Aggregation metrics storage:
  - Ceph. This option is recommended for production.
  - File back end based on GlusterFS. Use this option only for testing purposes.

**Note**

To define the amount of resources for Gnocchi, calculate the approximate amount of stored data using the [How to plan for Gnocchi's storage](#) instruction. Roughly, 1000 instances produce approximately 60 GB of telemetry data per year.

**Example:**

The cloud includes 15 compute nodes with 256 GB RAM each:

$$15 \times 256 = 3840 \text{ GB RAM raw}$$

Therefore, the cloud includes approximately 3.84 thousands of instances 1 GB each. Assuming that 1000 instances produce about 60 GB of metering data:
3.840 * 60 GB = 230 GB of telemetry data for cloud

A huge amount of short-living instances may increase this value because the data is stored with different aggregation rules. The older the data, the higher is the aggregation step.

Tenant Telemetry supports the community Aodh service that uses the Gnocchi API and provides an alarm evaluator mechanism based on metrics. Aodh allows triggering actions that are based on defined rules against sample or event data of OpenStack services that is collected by Ceilometer. After the event-driven alarm evaluation, Aodh provides instant alarm notifications to the user. The default Aodh back end is the same Galera cluster as used for the Openstack API.

To gather metrics from the compute nodes, Tenant Telemetry uses the Ceilometer Compute Agent installed on each compute node.

The following diagram displays the composition of the standalone Tenant Telemetry components across MCP nodes:

The following diagram displays the data flow across the Tenant Telemetry services:
The following table describes the components of standalone Tenant Telemetry:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
</table>

©2018, Mirantis Inc.
Ceilometer agents
- Central agents collect metrics from the OpenStack services and send them to the notifications.sample queue. Central agents run on the virtualized control plane nodes.
- Compute agents request virtual instances metadata from the Nova API 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, transform if required, and publish them to Gnocchi and Panko.

Gnocchi agent
- Metricd processes the measures, computes their aggregates, and stores them into the aggregate storage. Metricd also handles other cleanup tasks, such as deleting metrics marked for deletion.

Aodh agents
- API server (aodh-api) provides access to the alarm information in the data store.
- Alarm evaluator (aodh-evaluator) fires alarms based on the associated statistics trend crossing a threshold over a sliding time window.
- Notification listener (aodh-listener) fires alarms based on defined rules against events that are captured by the notification agents of the Telemetry Data Collection service.
- Alarm notifier (aodh-notifier) allows setting alarms based on the threshold evaluation for a collection of samples.

Ironic planning
MCP enables you to provision the OpenStack environment workloads (instances) to bare metal servers using Ironic. Ironic provisions workloads to bare metal servers through the Compute service (Nova) in almost the same way the virtual machines are provisioned.

Ironic applies to a number of use cases that include:
- An OpenStack environment contains the workloads that can not be run on virtual machines due to, for example, legacy software installed.
- An OpenStack environment contains the workloads that require high performance as well as no virtualization overhead.

Ironic consists of two main components: ironic-api and ironic-conductor. Additionally, it requires several auxiliary services for ironic-conductor including TFTP and HTTP servers. To enable the Compute service' users to provision their workloads on bare metal servers, the nova-compute service is configured to use the ironic virt-driver.

Ironic components

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ironic-api</td>
<td>Provides access to the alarm information in the data store.</td>
</tr>
<tr>
<td>ironic-evaluator</td>
<td>Fires alarms based on the associated statistics trend crossing a threshold over a sliding time window.</td>
</tr>
<tr>
<td>ironic-listener</td>
<td>Fires alarms based on defined rules against events that are captured by the notification agents of the Telemetry Data Collection service.</td>
</tr>
<tr>
<td>ironic-notifier</td>
<td>Allows setting alarms based on the threshold evaluation for a collection of samples.</td>
</tr>
<tr>
<td>ironic-conductor</td>
<td>Performs actual node provisioning. Due to security and performance considerations, it is deployed on separate brm* VMs on MCP KVM nodes along with its auxiliary services.</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ironic-api</td>
<td>Due to security considerations, two pools of ironic-api services are deployed with different access policies:</td>
</tr>
<tr>
<td></td>
<td>• The public pool processes requests from other services and users. It is deployed on ctl* nodes. REST API endpoints used by bare metal nodes are disabled for services in this pool.</td>
</tr>
<tr>
<td></td>
<td>• The deploy pool processes requests from nodes during node provisioning or cleaning. It is deployed on brm* nodes. The REST API endpoints enabled for services in this pool are those used by bare metal nodes during provisioning.</td>
</tr>
<tr>
<td>nova-compute</td>
<td>Separate pool of nova-compute services with ironic virt-driver deploys on brm* nodes.</td>
</tr>
</tbody>
</table>

The following diagram displays the Ironic network logic.

![Ironic network logic diagram]

Seealso

OpenContrail and Ironic multitenancy integration
High availability in OpenStack

The Virtual Control Plane (VCP) services in the MCP OpenStack are highly available and work in active/active and active/standby modes to enable an automatic recovery after a single node failure. An OpenStack environment contains both stateless and stateful services. Therefore, the MCP OpenStack handles them in different ways to achieve high availability (HA).

To make the OpenStack stateless services, such as nova-api, nova-conductor, glance-api, keystone-api, neutron-api, and nova-scheduler sustainable against a single node failure, MCP OpenStack runs a load balancer (two or more instances) and reserved running service instances to be switched to when the currently used service instance fails.

MCP OpenStack ensures HA for the stateless services in an active/active configuration using the HAProxy with Keepalived set of software. HAProxy provides access to the OpenStack API endpoint by redirecting the requests to active instances of an OpenStack service in a round-robin fashion. The proxy server sends API traffic to the available back ends and prevents the traffic from going to the unavailable nodes. The Keepalived daemon provides VIP management for the proxy server.

Note
Optionally, you can manually configure SSL termination on the HAProxy, so that the traffic to OpenStack services is mirrored to go for inspection in a security domain.

to make the OpenStack stateful services (the message queue and databases) highly available, the MCP:

- Deploys the stateful services in clusters with an odd number of nodes (n) to avoid the split-brain (three, five, seven, so on).

Note
The quorum (Q) in this case is determined as more than a half of the nodes (1). In case of multiple failures, so that the cluster size falls below the quorum value, the cluster itself fails:

\[
Q = \text{floor}(n/2) + 1 \quad (1)
\]

- Uses Galera to create and run the MySQL cluster of an odd number of database instances. Galera/MySQL is working in an active/active mode with synchronous replication of data, while HAProxy implements an active/standby mode of operation for the services that write to MySQL.

- Configures the RabbitMQ cluster with the ha-mode policy to run mirrored queues in an active/active mode.

- Deploys any number of proxy instances with the Horizon OpenStack service and NGINX in an active/active configuration. These services are not clustered and do not use quorum.

The following diagram describes the control flow in an HA OpenStack environment:
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, Monitoring, 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
- Runs the services related to the Kubernetes Master Tier, such as the Kubernetes control plane services.
  - The default hostname is ctl0X.

Kubernetes Node
- Runs user workloads (previously called Minion). In MCP, a Kubernetes Node is identical to the compute node. The default hostname is cmp0X.

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 with Calico:
Kubernetes cluster components

A Kubernetes cluster includes the Kubernetes components as well as supplementary services that run on all or some of the nodes.
Unless noted otherwise, all listed components run as daemon processes on a host operating system, controlled by systemd.

The components can be divided into the following types:

Common components
These components run on all nodes in a Kubernetes cluster.

- The kubelet agent service is responsible for creating and managing Docker containers on the Kubernetes cluster nodes.
- The kube-proxy service is responsible for the TCP/UDP stream forwarding or round-robin TCP/UDP forwarding across various back ends to reach cluster services (acts as a service proxy). This service is used for the Calico SDN only.

Master components
These components run on the Kubernetes Master nodes and provide the control plane functionality.

- The etcd service is a distributed key-value store that stores data across a Kubernetes cluster.
- The kubectl command-line client for the Kubernetes API enables cloud operators to execute commands against Kubernetes clusters.
- The kube-apiserver REST API server verifies and configures data for such API objects as pods, services, replication controllers, and so on.
- The kube-scheduler utility implements the scheduling functions of workloads provisioning in pods to specific Kubernetes Nodes according to the capacity requirements of workloads, Nodes allowances, and user-defined policies, such as affinity, data localization, and other custom restraints. The kube-scheduler utility may significantly affect performance, availability, and capacity.
- The kube-control-manager process embeds the core control loops shipped with Kubernetes, such as the replication controller and so on.

Networking components
These components run either on the Kubernetes nodes or KVM nodes depending on the SDN solution that you choose for your MCP cluster, Calico or OpenContrail.

- The Calico SDN solution provides pure L3 networking to a Kubernetes cluster. Calico runs as a Docker container calico-node.
- The Container Network Interface (CNI) plugin for the Calico SDN establishes a standard for the network interface configuration in Linux containers.
- The OpenContrail SDN can be installed as an alternative to Calico networking in the MCP clusters that require L2 network adjacency.

Caution!
OpenContrail 3.2 for Kubernetes is available as a technical preview only. For production MCP Kubernetes deployments, use OpenContrail 4.0.
Kubernetes Master nodes. In the OpenContrail 4.0 deployments, it runs on the Kubernetes Nodes.

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

- The etcd-proxy client redirects requests to available nodes in the etcd cluster.
- The metallb process for Calico provides external IP addresses to the workloads services, for example, NGINX, from the pool of addresses defined in the MetalLB configuration.

Network planning
Mirantis Cloud Platform supports OpenContrail and Calico networking models for Kubernetes clusters.

Calico is a distributed networking controller integrated through the Container Network Interface (CNI) plugin that provides pure L3 networking to a Kubernetes cluster. Calico runs as a Docker container calico-node on the Kubernetes nodes. This container includes all Calico services.

When using Calico, the workload network, which is analogous to the tenant network in OpenStack, is combined with the public and storage networks into one flat L3 space. You can also specify the pool.address parameter for particular hosts to define an interface for the workload network traffic. This parameter defines a host IP address that is used as a source IP address to reach other nodes in a Kubernetes cluster.

Kubernetes has one logical network for all Kubernetes workloads. Each Kubernetes Master node and Kubernetes Node has one network interface for the traffic flow between nodes.

The OpenContrail SDN can be installed as an alternative to Calico networking in the MCP clusters that require L2 overlay network adjacency. In OpenContrail 3.2, the OpenContrail cluster runs on the KVM nodes and the vRouter service runs on the Kubernetes Master nodes and on the Kubernetes Nodes. In OpenContrail 4.0, the OpenContrail cluster runs in pods on the Kubernetes Master nodes and the vRouter service runs on the Kubernetes Nodes.

Caution!
OpenContrail 3.2 for Kubernetes is available as a technical preview only. For production MCP Kubernetes deployments, use OpenContrail 4.0.

Types of networks
When planning your MCP Kubernetes cluster, consider the types of traffic that your workloads generate and design your network accordingly.

An MCP Kubernetes cluster with OpenContrail SDN contains the following types of the underlay networks:

- PXE/Management
  The non-routable network that is used for MAAS and Salt for DHCP traffic, provisioning and managing nodes. It usually requires a 1 Gbps network interface.

- Public network
  The routable network for external IP addresses of the LoadBalancer services managed by OpenContrail (or Calico).
- **Control network**
  The routable network for managing traffic between kube-api, kubelet, and OpenContrail (or Calico). It is also used to access the KVM nodes.

- **Workload network**
  The routable network for communication between containers in a cluster that is managed by OpenContrail (or Calico). It is analogous to the tenant network in OpenStack.

- **Storage network (optional)**
  The routable network used for storage traffic.

The following diagram displays an overview of the underlay networks in an MCP Kubernetes cluster with OpenContrail:

The following table provides the underlay networks configuration template for an MCP Kubernetes cluster with OpenContrail:

<table>
<thead>
<tr>
<th>Network name</th>
<th>Subnet range</th>
<th>Gateway</th>
<th>VLAN ID (optional)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PXE/Management network</td>
<td>192.168.x.0/26</td>
<td>-</td>
<td>xxx1</td>
</tr>
<tr>
<td>Control network/API</td>
<td>10.10.0.0/26</td>
<td>10.10.0.1</td>
<td>xxx2</td>
</tr>
<tr>
<td>Workload network</td>
<td>10.10.1.0/26</td>
<td>10.10.1.1</td>
<td>xxx3</td>
</tr>
<tr>
<td>Storage network</td>
<td>10.10.2.0/26</td>
<td>10.10.2.1</td>
<td>xxx4</td>
</tr>
<tr>
<td>Public network</td>
<td>10.20.0.0/26</td>
<td>10.20.0.1</td>
<td></td>
</tr>
</tbody>
</table>

The nodes must have at least two network interfaces available:

<table>
<thead>
<tr>
<th>Number of ports</th>
<th>Speed</th>
<th>Networks</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 Gbps</td>
<td>PXE/Management</td>
<td>This interface does not need to be fast. It is used for nodes PXE booting, configuring, and management.</td>
</tr>
</tbody>
</table>
10 Gbps
Control, Workload, Public, Storage
All traffic, except the configuration management transactions, flows through this interface.

An MCP Kubernetes cluster with Calico contains the same underlay networks as for OpenContrail. But in Calico, public and workload networks are combined into one flat IP address space. Network traffic can then be separated using network policies and IP pools.
Calico networking considerations

As Kubernetes does not provide native support for inter-pod networking, MCP uses Calico as an L3 networking provider for all Kubernetes deployments through the Container Network Interface (CNI) plugin. The CNI plugin establishes a standard for the network interface configuration in Linux containers.

Calico ensures propagation of a container IP address to all Kubernetes Nodes over the BGP protocol, as well as provides network connectivity between the containers. It also provides dynamic enforcement of the Kubernetes network policies. Calico uses the etcd key-value store or the Kubernetes API datastore as a configuration data storage.

Calico runs in a container called calico-node on every node in the Kubernetes cluster. 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 lightweight BGP daemon that distributes routing information between the nodes of the Calico network.

confd
Dynamic configuration manager for BIRD, triggered automatically by updates in the configuration data.

See also

- Project Calico in Kubernetes
- Calico official documentation
- Plan OpenContrail networking

Plan the OpenContrail networking

In MCP, OpenContrail integrates with Kubernetes by providing different isolation modes for virtual machines, pods, and bare metal workloads. When using OpenContrail, you can also enable the NGINX Ingress controller to provide an external access to Kubernetes services.

OpenContrail provides various features, such as network analytics, security, multitenancy, and so on. For details, see Plan OpenContrail networking.
Caution!

- OpenContrail 3.2 for Kubernetes is available as technical preview only. For production MCP deployments, use OpenContrail 4.0.
- OpenContrail with Virtlet is available as technical preview. Use such configuration for testing and evaluation purposes only.

Network checker overview

Network checker, or Netchecker, is a Kubernetes application that verifies connectivity between Kubernetes nodes. Netchecker comprises the following components:

- **Netchecker agent** is deployed on every Kubernetes node using the Daemonset mechanism which ensures automatic pod management. Agents periodically gather networking information from the Kubernetes nodes and send it to the Netchecker server.

- **Netchecker server** is deployed in a dedicated Kubernetes pod and exposed inside of the cluster through the Kubernetes service resource. All Netchecker agents connect to the Netchecker server through the service DNS name.

The communication mechanism between the user and Netchecker is the HTTP RESTful interface. You can run the following requests:

- **GET** - `/api/v1/connectivity_check` - request to test the connectivity between the Netchecker server and agents. The response contains information about possible issues.

  **Example of the request:**

  ```
  curl http://nc-server-ip:port/api/v1/connectivity_check
  ```

  This request returns the following:

<table>
<thead>
<tr>
<th>Message</th>
<th>A text that describes the status of the connection between the agent and server pods. Example: All 4 pods successfully reported back to the server.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absent</td>
<td>Indicates that the Netchecker server failed to receive reports from one or more Netchecker agents since the deployment of the Netchecker application. This field can be empty.</td>
</tr>
<tr>
<td>Outdated</td>
<td>Indicates that the Netchecker server failed to receive reports from one or more agents within the configured report interval due to a connectivity error. This field can be empty.</td>
</tr>
</tbody>
</table>

The following diagram illustrates how the Netchecker works:
MetalLB support

In MCP, MetalLB is a Kubernetes add-on that provides a network load balancer for bare metal Calico-based Kubernetes clusters using standard routing protocols.

MetalLB support is available starting Kubernetes 1.9.0 on clusters that do not have the network load balancing implemented yet.

Since MetalLB provides a standard network load balancer functionality, it is compatible with several Kubernetes networking add-ons.

In MCP, MetalLB supports only the layer-2 mode. For details, see: MetalLB in layer-2 mode. MetalLB in the Border Gateway Protocol (BGP) mode is not supported yet.

In an MCP Kubernetes cluster, MetalLB runs in Docker containers on the Kubernetes Nodes.

When using MetalLB, you can also enable the NGINX Ingress controller to provide an external access to Kubernetes services.

See also
MCP Operations Guide: Monitor connectivity between Kubernetes nodes using Netchecker
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 control plane services in MCP Kubernetes are highly-available and work in active-standby mode. MCP installs all control components on all nodes in the MCP Kubernetes cluster with one node at a time being selected as a master replica and others running in the stand-by mode.

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. The kube-controller-manager and kube-scheduler services elect their own leaders natively.

API servers work independently while external or internal Kubernetes load balancer dispatches requests between all of them. Each of the three Kubernetes Master nodes runs its own instance of kube-apiserver. All Kubernetes Master Tier services work with the Kubernetes API locally, while the services that run on the Kubernetes nodes access the Kubernetes API by directly connecting to an instance of kube-apiserver.

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.

OpenStack cloud provider for Kubernetes

The OpenStack cloud provider extends the basic functionality of Kubernetes by fulfilling the provider requirement for several resources. This is achieved through communication with several OpenStack APIs. As a rule, the Kubernetes cluster must consist of instances that are deployed on an OpenStack environment in order to activate the cloud provider.

Additionally, the OpenStack environment must have the following components installed: nova, keystone, neutron (with LBaaS), and cinder.

In the future, DNS support may become available through the Designate project.

Several of the Kubernetes components communicate with the OpenStack environment services in order to obtain information as well as create and maintain objects.

The kubelet service accesses nova to obtain the nova instance name. The kubelet node name will be set to the instance name. It also accesses cinder to mount PersistentVolumes that are requested by a pod.
The kube-apiserver service accesses nova to limit admission to the Kubernetes cluster. The service only allows the cloudprovider-enabled Kubernetes nodes to register themselves into the cluster.

The kube-controller-manager service accesses cinder and neutron to create PersistentVolumes (using cinder) and LoadBalancers (using neutron-lbaas).

Below is a diagram of the components involved and how they interact.

Virtual machines as Kubernetes pods

MCP allows running QEMU/KVM virtual machines as Kubernetes pods using Virtlet.

Virtlet is a Kubernetes Container Runtime Interface (CRI) implementation that is packaged as a Docker image and contains such components as a libvirt daemon, QEMU/KVM wrapper, and so on.

Virtlet enables you to run unmodified QEMU/KVM virtual machines that do not include an additional Docker layer as in similar solutions in Kubernetes. Virtlet supports all standard Kubernetes objects, such as ReplicaSets, Deployments, DaemonSets, and so on, as well as their operations.

Virtlet uses libvirt API to manage virtual machine and translates Kubernetes API primitives into operations over libvirt.

The following diagram describes the Virtlet components and interactions between them.
Virtlet includes the following components:

- Virtlet **manager** that implements CRI interfaces for virtualization and image handling
- A **libvirt** instance
- Virtlet **tapmanager** that is responsible for managing a VM networking
- Virtlet **vmwrapper** that is responsible for preparing environment for an emulator
- An emulator (QEMU with KVM support and with a possibility to disable KVM)
- **Container Runtime Interface (CRI) Proxy** that provides the ability to mix docker-shim and VM-based workloads on the same Kubernetes node

The image service provides VM images accessible through HTTP in a local cluster environment. It is only used as an optional helper because Virtlet manager can pull images from any HTTP server accessible from the node.

Caution!

Virtlet with OpenContrail is available as technical preview. Use such configuration for testing and evaluation purposes only.

Virtlet manager
Virtlet manager has the main binary file that is responsible for providing API that fulfills the Container Runtime Interface (CRI) specification. Virtlet manager handles the requests from kubelet and has the following functionality:

- Control the preparation of libvirt VM environment (virtual drives, network interfaces, trimming resources like RAM, CPU).
- Call CNI plugins to setup network environment for virtual machines.
- Request libvirt to call vmwrapper instead of using emulator directly.
- Query libvirt for VM statuses.
- Instruct libvirt to stop VMs.
- Call libvirt to remove a VM environment.

Seealso
Virtlet manager

Virtlet tapmanager
Virtlet tapmanager controls the setup of VM networking using CNI that is started by the virtlet command, since tapmanger uses the same virtlet binary.

The Virtlet tapmanager process has the following functionality:

- Take the setup requests from the Virtlet manager and set up networking for a VM by producing an open file descriptor that corresponds to the TAP device.
- Run DHCP server for each active VM.
- Handle requests from vmwrapper by sending the file descriptor over a Unix domain socket to vmwrapper. As a result, this file descriptor can be used in another mount namespace of a container. And you do not need a shared access to the directory containing network namespaces.
- Remove a VM network upon the Virtlet manager requests.

Virtlet vmwrapper
Virtlet vmwrapper is controlled by libvirt and runs the emulator (QVM/QEMU).

The Virtlet vmwrapper process has the following functionality:

1. Request TAP file descriptor from tapmanager.
2. Add the command-line arguments required by the emulator to use the TAP device.
3. Spawn the emulator.
Container Runtime Interface Proxy

Container Runtime Interface (CRI) Proxy provides a way to run multiple CRI implementations on the same node, for example, Virtlet and dockershim. It enables running infrastructure pods such as kube-proxy. CRI Proxy reuses the dockershim component from kubelet to have Docker as one of CRI implementations on the multi-runtime nodes. For details, see CRI Proxy design.

Kubernetes federation

Kubernetes offers federation as a way to logically group several clusters together. The Kubernetes federation allows a given host cluster to schedule resources across multiple Kubernetes clusters and expose their services with a common DNS service.

When a federated service is created, it is distributed to each cluster automatically. Pods that consume a federated service are routed to the closest available instance.

The following are the main benefits of the Kubernetes federation:

- Resilience. You can schedule a workload onto multiple locations through a single API.
- Scalability. You can place workloads on multiple clusters for load balancing purposes.

An on-premises Kubernetes federation scenario requires several extra steps to ensure that there are no single points of failure in the implementation.

The Kubernetes federation has the following key components:

- federation-apiserver that validates and configures data for the API objects. The API objects include pods, services, replication controllers, and so on. The API server handles REST operations and provides the front end to a cluster shared state through which all other components interact with each other.
- federation-controller-manager that is a daemon maintaining the core control loops shipped with federation.

The following definitions are used in the Kubernetes federation:

- kubeconfig
  A type of configuration file that any Kubernetes component uses to authenticate to a Kubernetes API server. The file is in YAML format and contains the following sections:
  - current context indicates the default context to be used for a Kubernetes component. It can be overridden with the --context option.
  - contexts consists of two components: user and cluster. Often a cluster name is the same as a context name.
  - clusters contains information about servers, such as a server field (URL), a name, and optionally a certificate authority field.
• users contains information about users, such as a name (applicable only when using the role-based access control), a certificate, and a key. The last two items can be a path or a Base64-encoded certificate.

• kubedns
  Leverages SkyDNS to serve DNS records for each service in a Kubernetes cluster. All pods must be able to resolve any domain within a Kubernetes cluster.

• CoreDNS
  Next generation of SkyDNS that can use etcd to accept updates for DNS entries. It functions as an on-premises open-source alternative to cloud DNS services (DNSaaS).

• neutron-lbaas
  Load balancer that enables the federation control plane to access the Kubernetes API server of connected clusters.

The following diagram summarizes the layout of services in a federation of MCP Kubernetes clusters:
When connecting two Kubernetes clusters deployed by the MCP Heat template, the federation control plane connects only to the public IP of the Ibaas load balancer for the child Kubernetes clusters:
See also

- Kubernetes official documentation on federation
- CoreDNS

See also

MCP Deployment Guide: Deploy ExternalDNS for Kubernetes
Plan OpenContrail networking

OpenContrail is a highly scalable distributed Software Defined Networking (SDN) solution for MCP. It provides NFV capabilities as well as powerful network analytics and visualization of results through a web UI. OpenContrail cluster is integrated with both types of the MCP cluster, OpenStack and Kubernetes, to provision, orchestrate, and deploy high-performance clusters with various networking features.

The following table describes the main features of OpenContrail in comparison with the native Calico SDN for Kubernetes and OVS for OpenStack:

<table>
<thead>
<tr>
<th>Feature name</th>
<th>OpenContrail</th>
<th>Calico</th>
<th>OVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulation</td>
<td>MPLSoGRE, MPLSoUDP, VXLAN</td>
<td>IP-in-IP</td>
<td>VLAN, VXLAN, GRE, Geneve</td>
</tr>
<tr>
<td>Security and multitenancy</td>
<td>Native overlay, label, namespaces</td>
<td>Calico policy using iptables, conntrack</td>
<td>Native overlay, label, namespaces</td>
</tr>
<tr>
<td>Multi-cloud</td>
<td>Bare metal, VMs, containers</td>
<td>KVM, containerized</td>
<td>Bare metal, VMs, containers</td>
</tr>
<tr>
<td>Network analytics</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>NFV</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dataplane acceleration</td>
<td>DPDK, SR-IOV</td>
<td>No</td>
<td>DPDK, SR-IOV</td>
</tr>
<tr>
<td>Extra features</td>
<td>Bare metal extensions, L2VPN, L3VPN</td>
<td>No</td>
<td>FWaaS, VPNaaS, LBaaS</td>
</tr>
</tbody>
</table>

Limitations

The OpenContrail deployment in MCP includes the following limitations:

1. OpenContrail does not support tenant renaming due to architecture limitations.
2. OpenContrail does not support parallel addition and removal of rules to and from security groups.
3. The Kubernetes-specific OpenContrail limitations are as follows:
   - OpenContrail 3.2 for Kubernetes is available as a technical preview only. For production environments, use OpenContrail 4.0.
   - OpenContrail with Virtlet is available as technical preview. Use such configuration for testing and evaluation purposes only.
   - The Kubernetes conformance tests from the StatefulSetBasic set may fail if the concurrency is more than 2. Inside pods, the logs may contain the network outage messages. Mirantis recommends running all tests from the StatefulSetBasic set of the conformance tests consequentially, with concurrency=1.
   - A Kubernetes container is reachable only from a node where this container is spawned. The container provides support for liveness and readiness probes. But any feature that has a requirement for all nodes to be able to communicate with all containers and vice versa is not supported. For example, proxying requests to pods from an API server. Mirantis recommends avoiding isolation between nodes and pods by configuring routing between them. It can be done...
by using a simple gateway or a hardware router. For better performance, Mirantis recommends using hardware routers supported by OpenContrail.

- Exposing a service with the NodePort type is not supported. Use the LoadBalancer type with the OpenContrail public network instead. For details, see the official Kubernetes documentation.

See also

Requirements for federated Kubernetes

The OpenContrail requirements when deployed with federated Kubernetes on the MCP cluster are as follows:

- LoadBalancer must be configured to work with the port mapping of a Kubernetes Service.
- The Kubernetes Service network must be configured for all namespaces. The federation control plane must be configured in a new namespace.
- The floating IP must be reachable inside the Kubernetes cluster or kubefed must be run outside the cluster.

See also

Limitations

OpenContrail cluster overview

OpenContrail provides the following types of networking to a Kubernetes cluster or an OpenStack environment running on MCP cluster:

- Basic networking that includes IP address management, security groups, floating IP addresses, and so on
- Advanced networking that includes DPDK network virtualization and SR-IOV

OpenContrail is based on the overlay networking, where all containers are connected to a virtual network with encapsulation (MPLSoGRE, MPLSoUDP, VXLAN). This enables you to separate underlay Kubernetes management network. And container workload requires an external gateway (a hardware EdgeRouter or a simple gateway) to route traffic outside. In the host mode, the containers run in the host network namespace and use the host IP address. It means that they run on the underlay network. Therefore, those containers cannot reach overlay containers without some external routing. This behavior is intentional for an overlay SDN. However, this brings a limitation for the host network pods to reach service endpoints and the Kubernetes DNS.

In the current network model, the OpenContrail vRouter uses different gateways for the control and data planes.
The supported versions of OpenContrail include 3.2 and 4.0 for OpenStack. For Kubernetes, OpenContrail 3.2 is available as a technical preview only. For production environments, use OpenContrail 4.0.

OpenContrail 3.2 cluster overview

An OpenContrail 3.2 cluster contains the following types of nodes:

- **Controller node**
  Includes package-based services of OpenContrail, such as the API server and configuration database.

- **Analytics node**
  Includes package-based services for OpenContrail metering and analytics, such as the Cassandra database for analytics and data collectors.

- **vRouter node**
  A forwarding plane that runs in the hypervisor of a compute node. It extends the network from the physical routers and switches into a virtual overlay network hosted on compute nodes.

- **Gateway node**
  A physical or virtual gateway router that provides access to an external network.

The following diagram describes the minimum production installation of OpenContrail 3.2 with OpenStack for MCP:
Caution!

OpenContrail 3.2 for Kubernetes is available as a technical preview only. For production MCP Kubernetes deployments, use OpenContrail 4.0.

OpenContrail 4.0 cluster overview

In OpenContrail 4.0, the OpenContrail controller and analytics modules are delivered as containers to reduce the complexity of the OpenContrail deployment and to group the related OpenContrail components.

Each container has an INI-based configuration file that is available on the host system and mounted within a specific container.

The OpenContrail containers run with the host network, without using a Docker bridge. All services within a container listen on the host network interface.
The following diagram describes the minimum production installation of OpenContrail 4.0 with OpenStack for MCP.

An OpenContrail 4.0 cluster for OpenStack contains the following types of entities:

- Controller Docker container
  - Includes the package-based OpenContrail services, such as the API server and configuration database. Runs on top of the ntw virtual machine as a Docker container initialized by docker-compose.

- Analytics Docker container

Note
For the sake of visual clarity, the diagrams in this section illustrate only the OpenContrail architecture with OpenStack or Kubernetes. The diagrams presuppose the DriveTrain and StackLight LMA nodes.
Includes the OpenContrail metering and analytics package-based services, such as analytics API, alarm generator and data collector. Runs on top of the nal virtual machine as a Docker container initialized by docker-compose.

- Analyticsdb Docker container

  Includes the OpenContrail metering and analytics package-based services. This container includes a database for Analytics container, Kafka, and ZooKeeper. Runs on top of the nal virtual machine as a Docker container initialized by docker-compose.

- vRouter node

  A forwarding plane that runs in the hypervisor of a compute node. It extends the network from the physical routers and switches into a virtual overlay network hosted on compute nodes.

- Gateway node

  A physical or virtual gateway router that provides access to an external network.

The following diagram describes the minimum production installation of OpenContrail 4.0 with Kubernetes for MCP.
An OpenContrail 4.0 cluster for Kubernetes contains the following types of entities:

- **OpenContrail pod**
  
  Includes the package-based OpenContrail controller, configuration, metering, and analytics services. Runs using Docker on the Kubernetes Master nodes and is defined through the Kubernetes API.

- **vRouter node**
A forwarding plane that runs in the hypervisor of a Kubernetes Node. It extends the network from the physical routers and switches into a virtual overlay network hosted on Kubernetes Nodes.

- **Gateway node**
  A physical or virtual gateway router, for example, Juniper MX, that provides access to an external network.

See also
- **Official Juniper documentation**

### OpenContrail components

Mirantis Cloud Platform (MCP) supports OpenContrail versions 3.2 and 4.0.

The difference between two OpenContrail versions is as follows:

- In OpenContrail 3.2, services run as supervisor or non-supervisor services. In OpenContrail 4.0, all services run as systemd services in a Docker container.
- In OpenContrail 4.0, the ifmap-server and contrail-discovery services are deprecated.

The OpenContrail services are distributed across several MCP cluster nodes:

- In the OpenStack-based deployments for both versions of OpenContrail, the control, config, analytics, and database services run on the OpenContrail controller (ntw) and analytics (nal) nodes. In the Kubernetes-based deployments, these services run on the Kubernetes Master nodes (ctl) for OpenContrail 4.0 and on the OpenContrail controller (ntw) and analytics (nal) nodes for OpenContrail 3.2.
- The network-controller services run on the Kubernetes Master nodes.
- The vrouter OpenContrail services run on the OpenStack compute nodes and Kubernetes Nodes (cmp). In OpenContrail 3.2, the vrouter service also runs on the Kubernetes Master nodes.
- The OpenContrail plugin is included to the neutron-server service that runs on the OpenStack controller nodes (ctl).

**Caution!**

OpenContrail 3.2 for Kubernetes is available as a technical preview only. For production MCP Kubernetes deployments, use OpenContrail 4.0.

This section describes the OpenContrail 3.2 and 4.0 services as well as their distribution across the MCP cluster nodes.

### OpenContrail 3.2 components
The tables in this section describe the OpenContrail 3.2 services and their distribution across the MCP cluster nodes.

**Caution!**

OpenContrail 3.2 for Kubernetes is available as a technical preview only. For production MCP Kubernetes deployments, use OpenContrail 4.0.

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrail-control</td>
<td>Communicates with the cluster gateways using BGP and with the vRouter agents using XMPP as well as redistributes appropriate networking information.</td>
</tr>
<tr>
<td>contrail-control-nodemgr</td>
<td>Collects the OpenContrail controller process data and sends this information to the OpenContrail collector.</td>
</tr>
<tr>
<td>contrail-dns</td>
<td>Using the contrail-named service, provides the DNS service to the VMs spawned on different compute nodes. Each vRouter node connects to two OpenContrail controller nodes that run the contrail-dns process.</td>
</tr>
<tr>
<td>contrail-named</td>
<td>This is the customized Berkeley Internet Name Domain (BIND) daemon of OpenContrail that manages DNS zones for the contrail-dns service.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrail-webui</td>
<td>Consists of the webserver and jobserver services. Provides the OpenContrail web UI.</td>
</tr>
<tr>
<td>ifmap-server</td>
<td>Deprecated in OpenContrail 4.0. The contrail-control, contrail-schema, contrail-svc-monitor services connect to the Interface for Metadata Access Points (IF-MAP) server using this service during configuration changes.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrail-api</td>
<td>Exposes a REST-based interface for the OpenContrail API.</td>
</tr>
<tr>
<td>contrail-config-nodemgr</td>
<td>Collects data of the OpenContrail configuration processes and sends it to the OpenContrail collector.</td>
</tr>
<tr>
<td>contrail-device-manager</td>
<td>Manages physical networking devices using netconf or ovsdb. In multi-node deployments, it works in the active/backup mode.</td>
</tr>
</tbody>
</table>
### contrail-discovery
Depreciated in OpenContrail 4.0. Acts as a registry for all OpenContrail services.

### contrail-schema
Listens to configuration changes done by a user and generates corresponding system configuration objects. In multi-node deployments, it works in the active/backup mode.

### contrail-svc-monitor
Listens to configuration changes of service-template and service-instance as well as spawns and monitors virtual machines for the firewall, analyzer services and so on. In multi-node deployments, it works in the active/backup mode.

### The supervisor analytics services, OpenContrail analytics node

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrail-alarm-gen</td>
<td>Evaluates and manages the alarms rules.</td>
</tr>
<tr>
<td>contrail-analytics-api</td>
<td>Provides a REST API to interact with the Cassandra analytics database.</td>
</tr>
<tr>
<td>contrail-analytics-nodemgr</td>
<td>Collects all OpenContrail analytics process data and sends this information to the OpenContrail collector.</td>
</tr>
<tr>
<td>contrail-collector</td>
<td>Collects and analyzes data from all OpenContrail services.</td>
</tr>
<tr>
<td>contrail-query-engine</td>
<td>Handles the queries to access data from the Cassandra database.</td>
</tr>
<tr>
<td>contrail-snmp-collector</td>
<td>Receives the authorization and configuration of the physical routers from the contrail-config-nodemgr service, polls the physical routers using the Simple Network Management Protocol (SNMP) protocol, and uploads the data to the OpenContrail collector.</td>
</tr>
<tr>
<td>contrail-topology</td>
<td>Reads the SNMP information from the physical router user-visible entities (UVEs), creates a neighbor list, and writes the neighbor information to the physical router UVEs. The OpenContrail web UI uses the neighbor list to display the physical topology.</td>
</tr>
</tbody>
</table>

### The supervisor database services, OpenContrail controller and analytics nodes

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrail-database</td>
<td>Manages the Cassandra database information.</td>
</tr>
<tr>
<td>contrail-database-nodemgr</td>
<td>Collects data of the contrail-database process and sends it to the OpenContrail collector.</td>
</tr>
<tr>
<td>kafka</td>
<td>Handles the messaging bus and generates alarms across the OpenContrail analytics nodes.</td>
</tr>
</tbody>
</table>

### The non-supervisor database services, OpenContrail controller and analytics nodes

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
</table>
cassandra | On the OpenContrail network nodes, maintains the configuration data of the OpenContrail cluster. On the OpenContrail analytics nodes, stores the contrail-collector service data.
---|---
redis | Stores the physical router UVE storage and serves as a messaging bus for event notifications.
zookeeper | Holds the active/backup status for the contrail-device-manager, contrail-svc-monitor, and the contrail-schema-transformer services. This service is also used for mapping of the OpenContrail resources names to UUIDs.

The supervisor vrouter services, OpenStack compute nodes and Kubernetes Master nodes/Nodes

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrail-vrouter-agent</td>
<td>Connects to the OpenContrail controller node and the OpenContrail DNS system using the Extensible Messaging and Presence Protocol (XMPP).</td>
</tr>
<tr>
<td>contrail-vrouter-nodemgr</td>
<td>Collects the supervisor vrouter data and sends it to the OpenContrail collector.</td>
</tr>
</tbody>
</table>

The OpenContrail network-controller services, Kubernetes Master nodes

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kube-network-manager</td>
<td>Creates a network solution for containers using the OpenContrail API to define objects such as virtual-networks, network interfaces and access control policies. This service also annotates pods with the interface UUID created by OpenContrail as well as the allocated private IP address and a MAC address. These annotations are used by kubelet.</td>
</tr>
</tbody>
</table>

The OpenContrail plugin services, OpenStack controller nodes

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>neutron-server</td>
<td>The Neutron server that includes the OpenContrail plugin.</td>
</tr>
</tbody>
</table>

OpenContrail 4.0 components

The tables in this section describe the OpenContrail 4.0 services and their distribution across the MCP cluster nodes.

In OpenStack, the control, config, database, and analytics services run as plain Docker containers managed by docker-compose. In Kubernetes, these services run as pods managed by the Kubernetes API.

The config and control services, OpenContrail controller containers

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrail-api</td>
<td>Exposes a REST-based interface for the OpenContrail API.</td>
</tr>
</tbody>
</table>
### Service Description

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrail-config-nodemgr</td>
<td>Collects data of the OpenContrail configuration processes and sends it to the OpenContrail collector.</td>
</tr>
<tr>
<td>contrail-control</td>
<td>Communicates with the cluster gateways using BGP and with the vRouter agents using XMPP as well as redistributes appropriate networking information.</td>
</tr>
<tr>
<td>contrail-control-nodemgr</td>
<td>Collects the OpenContrail controller process data and sends this information to the OpenContrail collector.</td>
</tr>
<tr>
<td>contrail-device-manager</td>
<td>Manages physical networking devices using netconf or ovsdb. In multi-node deployments, it works in the active/backup mode.</td>
</tr>
<tr>
<td>contrail-discovery</td>
<td>Deprecated. Acts as a registry for all OpenContrail services.</td>
</tr>
<tr>
<td>contrail-dns</td>
<td>Using the contrail-named service, provides the DNS service to the VMs spawned on different compute nodes. Each vRouter node connects to two OpenContrail controller containers that run the contrail-dns process.</td>
</tr>
<tr>
<td>contrail-named</td>
<td>This is the customized Berkeley Internet Name Domain (BIND) daemon of OpenContrail that manages DNS zones for the contrail-dns service.</td>
</tr>
<tr>
<td>contrail-schema</td>
<td>Listens to configuration changes done by a user and generates corresponding system configuration objects. In multi-node deployments, it works in the active/backup mode.</td>
</tr>
<tr>
<td>contrail-svc-monitor</td>
<td>Listens to configuration changes of service-template and service-instance as well as spawns and monitors virtual machines for the firewall, analyzer services and so on. In multi-node deployments, it works in the active/backup mode.</td>
</tr>
<tr>
<td>contrail-webui</td>
<td>Consists of the webserver and jobserver services. Provides the OpenContrail web UI.</td>
</tr>
<tr>
<td>ifmap-server</td>
<td>Deprecated. The contrail-control, contrail-schema, contrail-svc-monitor services connect to the Interface for Metadata Access Points (IF-MAP) server using this service during configuration changes.</td>
</tr>
</tbody>
</table>

### The analytics services, OpenContrail analytics containers

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrail-alarm-gen</td>
<td>Evaluates and manages the alarms rules.</td>
</tr>
<tr>
<td>contrail-analytics-api</td>
<td>Provides a REST API to interact with the Cassandra analytics database.</td>
</tr>
<tr>
<td>contrail-analytics-nodemgr</td>
<td>Collects all OpenContrail analytics process data and sends this information to the OpenContrail collector.</td>
</tr>
<tr>
<td>contrail-collector</td>
<td>Collects and analyzes data from all OpenContrail services.</td>
</tr>
<tr>
<td>contrail-query-engine</td>
<td>Handles the queries to access data from the Cassandra database.</td>
</tr>
<tr>
<td>contrail-snmp-collector</td>
<td>Receives the authorization and configuration of the physical routers from the contrail-config-nodemgr service, polls the physical routers using the Simple Network Management Protocol (SNMP) protocol, and uploads the data to the OpenContrail collector.</td>
</tr>
</tbody>
</table>
contrail-topology | Reads the SNMP information from the physical router user-visible entities (UVEs), creates a neighbor list, and writes the neighbor information to the physical router UVEs. The OpenContrail web UI uses the neighbor list to display the physical topology.
---|---

The database services, OpenContrail controller and analytics containers

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cassandra</td>
<td>On the OpenContrail network nodes and OpenContrail pods, maintains the configuration data of the OpenContrail cluster. On the OpenContrail analytics containers, stores the contrail-collector service data.</td>
</tr>
<tr>
<td>contrail-database</td>
<td>Manages the Cassandra database information.</td>
</tr>
<tr>
<td>contrail-database-nodemgr</td>
<td>Collects data of the contrail-database process and sends it to the OpenContrail collector.</td>
</tr>
<tr>
<td>kafka</td>
<td>Handles the messaging bus and generates alarms across the OpenContrail analytics containers.</td>
</tr>
<tr>
<td>redis</td>
<td>Stores the physical router UVE storage and serves as a messaging bus for event notifications.</td>
</tr>
<tr>
<td>zookeeper</td>
<td>Holds the active/backup status for the contrail-device-manager, contrail-svc-monitor, and the contrail-schema-transformer services. This service is also used for mapping of the OpenContrail resources names to UUIDs.</td>
</tr>
</tbody>
</table>

The vrouter services, OpenStack compute nodes and Kubernetes Master nodes/Nodes

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrail-vrouter-agent</td>
<td>Connects to the OpenContrail controller container and the OpenContrail DNS system using the Extensible Messaging and Presence Protocol (XMPP).</td>
</tr>
<tr>
<td>contrail-vrouter-nodemgr</td>
<td>Collects the supervisor vrouter data and sends it to the OpenContrail collector.</td>
</tr>
</tbody>
</table>

The OpenContrail network-controller services, OpenContrail pods on Kubernetes Master nodes

<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrail-kube-manager</td>
<td>Creates a network solution for containers using the OpenContrail API to define objects such as virtual networks, network interfaces and access control policies. This service also annotates pods with the interface UUID created by OpenContrail as well as the allocated private IP address and a MAC address. These annotations are used by kubelet.</td>
</tr>
</tbody>
</table>

The OpenContrail plugin services, OpenStack controller nodes
<table>
<thead>
<tr>
<th>Service name</th>
<th>Service description</th>
</tr>
</thead>
<tbody>
<tr>
<td>neutron-server</td>
<td>The Neutron server that includes the OpenContrail plugin.</td>
</tr>
</tbody>
</table>

OpenContrail traffic flow

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

User Interface and API traffic

The following diagram displays all types of UI and API traffic in an MCP cluster, including monitoring, OpenStack or Kubernetes 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 MCP cluster. 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 in an MCP cluster:
OpenContrail vRouter

The OpenContrail vRouter provides data forwarding to an OpenStack tenant instance or a Kubernetes container and reports statistics to the OpenContrail analytics nodes. The OpenContrail vRouter is installed on all OpenStack compute nodes and on all Kubernetes Master nodes and Nodes.

In MCP, the OpenContrail vRouter can be either kernel-based or DPDK-based. You can configure a node(s) before deployment to use the DPDK vRouter mode instead of the regular kernel mode. This option allows different nodes to use different modules of the OpenContrail vRouter. Using DPDK with OpenContrail allows processing more packets per second in comparison to the regular kernel module.

The vRouter agent acts as a local control plane. Each OpenContrail vRouter agent is connected to at least two OpenContrail control nodes in an active-active redundancy mode. The OpenContrail vRouter agent is responsible for all networking-related functions including routing instances, routes, and so on.

The OpenContrail vRouter uses different gateways for the control and data planes. For example, the Linux system gateway is located on the management network, and the OpenContrail gateway is located on the data plane network.

The following diagram shows the OpenContrail kernel vRouter setup by Cookiecutter:
On the diagram above, the following types of networks interfaces are used:

- **eth0** - for the management (PXE) network (eth1 and eth2 are the slave interfaces of Bond0)
- **Bond0.x** - for the MCP control plane network
- **Bond0.y** - for the MCP data plane network

See also: [MCP Deployment Guide: Enable OpenContrail DPDK](#)

**OpenContrail HAProxy driver with LBaaSv2**

The OpenContrail HAProxy driver with Load Balancing as a Service (LBaaS) is implemented as a special type of service instance.

The load balancer service is implemented as a network namespace with HAProxy. The service runs on two randomly chosen vRouter compute nodes to achieve high availability.

The load balancer service has two sides:

- **Right** that is the public side
- **Left** that is the private side (for the back-end and pool subnet)

In LBaaS v1, the left side subnet is determined automatically from the subnet_id of a pool. But in LBaaS v2, the pool does not associate with subnet anymore. Therefore, to overcome this architecture limitation, the pool members of the left side and listener of the right side should be associated with the same subnet.

The OpenContrail HAProxy driver provides the benefits of LBaaS v2 API along with listening of multiple ports for the same IP address by decoupling the virtual IP address from the physical port.
The Transport Layer Security (TLS) termination on the OpenContrail HAProxy load balancer requires Barbican. Barbican is a REST API that is used for secured storage as well as for provisioning and managing of secrets such as passwords, encryption keys, and X.509 certificates.

OpenContrail HAProxy driver supports the following configuration options:

<table>
<thead>
<tr>
<th>Component</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listener protocols</td>
<td>TCP</td>
</tr>
<tr>
<td></td>
<td>HTTP</td>
</tr>
<tr>
<td></td>
<td>TERMINATED_HTTPS</td>
</tr>
<tr>
<td>Load balancer algorithms</td>
<td>ROUND_ROBIN</td>
</tr>
<tr>
<td></td>
<td>LEAST_CONNECTIONS</td>
</tr>
<tr>
<td></td>
<td>SOURCE_IP</td>
</tr>
<tr>
<td>Health monitor types</td>
<td>PING</td>
</tr>
<tr>
<td></td>
<td>TCP</td>
</tr>
<tr>
<td></td>
<td>HTTP</td>
</tr>
<tr>
<td>Session persistence</td>
<td>SOURCE_IP</td>
</tr>
<tr>
<td></td>
<td>HTTP_COOKIE</td>
</tr>
<tr>
<td></td>
<td>APP_COOKIE</td>
</tr>
</tbody>
</table>

OpenContrail and Ironic multitenancy integration

Multitenancy in Ironic is supported when OpenContrail is used as SDN. Multitenancy allows using isolated tenant networks for node provisioning by connecting the OpenContrail virtual network to a bare metal server or a virtual instance through a top-of-rack (ToR) switch.

The main advantages of the OpenContrail and Ironic multitenancy integration include:

- Provisioning and cleaning of VMs can be done in separate networks
- Bare metal nodes can be attached to any tenant network
- Bare metal nodes are fully network-isolated between tenants
- Networks can be attached dynamically after provisioning

Using this feature on large deployments enhances the performance of the tenant-to-tenant networking and simplifies communication with the virtual instances that run on the OpenContrail cluster.

A basic ToR services node (TSN) of the OpenContrail cluster consists of two physical servers that host the ToR Service node and ToR agents. TSN is the multicast controller of the ToR switches. TSN receives all broadcast packets from the ToR switches. The broadcast packets of a VM are sent from the corresponding compute node to a ToR switch directly. TSN also provides DHCP and DNS services to the bare metal servers or virtual instances running behind the ToR switch ports.
One TSN can host multiple ToR agents, where one ToR agent is responsible for managing one ToR switch. The ToR agent controls the ToR switch communication with the Open vSwitch Database (OVSDB), such as receiving and translating to OVSDB the configuration information from the ToR switch as well as adding corresponding OVSDB table entries on the ToR switch.

OpenContrail supports two connection modes to the ToR switch:

- **Plain Transmission Control Protocol (PTCP) mode**
  When ToR agent initiates connection and pushes changes to ToR. Only one active ToR agent per ToR is allowed.

- **Secure Sockets Layer (SSL) mode**
  ToR switch initiates connection. To support HA, ToR agents must be installed behind HAProxy. The active ToR agent is responsible for advertising routes from the ToR using the Border Gateway Protocol (BGP) Ethernet Virtual Private Network (EVPN) to the OpenContrail controller node.

To enable multitenancy in Ironic, you must configure tenant networks. For details, see: MCP Deployment Guide: Install Ironic components.

TSN in MCP is enabled in HA mode and with SSL for the OVSDB configuration.

The following diagram displays overview of the ToR topology:
See also

- Ironic planning
- MCP Deployment Guide: Enable TSN
- Multi-tenancy in the Bare Metal service in OpenStack documentation
- Ironic Baremetal Support in OpenContrail GitHub project
- Redundancy for HAProxy in Juniper documentation

See also

The community OpenContrail Architecture Documentation
StackLight planning

StackLight is an Operations Support System (OSS) that provides a single pane of glass for cloud maintenance and day-to-day operations as well as offers critical insights into cloud health including reach operational information about all services deployed on Mirantis Cloud Platform. StackLight includes:

DevOps Portal
- Provides a number of OSS tools and acts as a main administrative dashboard for the entire MCP environment.

StackLight LMA
- Comprises the logging, monitoring, and alerting functions that provide rich operational insights for the services deployed on the platform.

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.

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.
Dependencies between services

The DevOps Portal services are interconnected and require installation of particular services to operate successfully. For example, all DevOps Portal services require the DevOps Portal web UI.

The following table describes the dependencies of the DevOps Portal services:

<table>
<thead>
<tr>
<th>DevOps Portal service</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Intelligence</td>
<td>• DevOps Portal web UI</td>
</tr>
<tr>
<td></td>
<td>• Elasticsearch cluster (version 5.x.x or higher)</td>
</tr>
<tr>
<td></td>
<td>• Runbook Automation</td>
</tr>
<tr>
<td>Cleanup</td>
<td>• DevOps Portal web UI</td>
</tr>
<tr>
<td></td>
<td>• Push Notification service</td>
</tr>
<tr>
<td></td>
<td>• Cloud Intelligence service</td>
</tr>
</tbody>
</table>
### 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.

---

2 The LMA tab is only available in the DevOps Portal with StackLight LMA deployments.

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Capacity Management</td>
<td>• DevOps Portal web UI</td>
</tr>
<tr>
<td></td>
<td>• Kibana service of StackLight LMA</td>
</tr>
<tr>
<td>Cloud Health</td>
<td>• DevOps Portal web UI</td>
</tr>
<tr>
<td></td>
<td>• Grafana service of StackLight LMA</td>
</tr>
<tr>
<td>Cloud Vector</td>
<td>• DevOps Portal web UI</td>
</tr>
<tr>
<td></td>
<td>• Cloud Intelligence service</td>
</tr>
<tr>
<td>Heatmaps</td>
<td>• DevOps Portal web UI</td>
</tr>
<tr>
<td></td>
<td>• Prometheus service of StackLight LMA</td>
</tr>
<tr>
<td>Hardware Correlation</td>
<td>• DevOps Portal web UI</td>
</tr>
<tr>
<td></td>
<td>• Cloud Intelligence service</td>
</tr>
<tr>
<td></td>
<td>• Prometheus service of StackLight LMA</td>
</tr>
<tr>
<td>LMA tab 2</td>
<td>• DevOps Portal web UI</td>
</tr>
<tr>
<td></td>
<td>• Prometheus, Grafana, and Kibana services of StackLight LMA</td>
</tr>
<tr>
<td>Push Notification</td>
<td>• DevOps Portal web UI</td>
</tr>
<tr>
<td></td>
<td>• Elasticsearch cluster (version 5.x.x or higher)</td>
</tr>
<tr>
<td></td>
<td>• PostgreSQL database</td>
</tr>
<tr>
<td>Runbooks Automation</td>
<td>• DevOps Portal web UI</td>
</tr>
<tr>
<td></td>
<td>• PostgreSQL database</td>
</tr>
<tr>
<td>Security Audit</td>
<td>• DevOps Portal web UI</td>
</tr>
<tr>
<td></td>
<td>• Push Notification service</td>
</tr>
<tr>
<td></td>
<td>• PostgreSQL database</td>
</tr>
</tbody>
</table>

©2018, Mirantis Inc.
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 include:

- 3 virtual machines
- The recommended size of a VM:
  - 8 GB RAM
  - 50 GB disk
  - 2 vCPU

Cloud Health service overview

The Cloud Health service includes a set of high level tests that provide the overall cloud status. From a user perspective, testing of such things as network, storage, compute, and image APIs enables you to detect outages. The Cloud Health service provides the following types of tests:

- Failed Customer Interactions (FCI), which simulate the OpenStack-related actions, such as provisioning a VM, attaching a volume, attaching a floating IP, and others.
Cloud Availability API, which measures the response time of a cloud when a user performs one API call to the storage, compute, network, image, or Keystone API.

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, Monitoring, 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-cluster environment, with multiple OpenStack installations, the Runbook Automation service connects to multiple nodes on one or more clusters. 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 service leverages 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.

**Push Notification service overview**

The Push Notification service enables users to send notifications, execute API calls, and open tickets on helpdesk. The service can also connect several systems through a notification queue that transforms and routes API calls from source systems into specific protocols that other systems use to communicate. With the notification system, you can send an API call and transform it into another API call, an email, an AMQP message, or another protocol.

The following diagram displays the high level architecture of the Push Notification service:
Security Audit service overview

Security Audit is a service that enables the user to define specific audit rules for cloud resources. The Security Audit service runs on a schedule that depends on reporting jobs configurations while applying these rules to the metadata of cloud resources. The service provides the output in a form of a list of findings that do not comply with these rules, and sends the found issues to the Push Notification service.

The Security Audit service is comprised of the following components:

- **Watcher**
  Checks for changes on a tenant and registers them in the Security Audit database

- **Auditor**
  Runs security rules against tenant resources and notifies the owner if any of them are broken

- **Notifier**
  Sends an email through the Push Notification service and/or generates or updates a corresponding Salesforce tickets

The following diagram displays the high level architecture of the Security Audit service:
Cleanup service overview

The Cleanup service is designed to detect the unused cloud resources to clean them up.

The Cleanup service determines whether a resource should be a candidate for a cleanup by applying a set of rules on it. The service queries CIS for the existing resources. If any of the rules determines that the resource is a cleanup candidate, the service marks the resource and schedules time to delete or stop it.

The user can configure the schedule according to which the service will scan an MCP cloud environment for the clean-up candidates.

The following diagram displays the high level architecture of the Cleanup service:
Cloud Vector service overview

The Cloud Vector service obtains the data from the Cloud Intelligence service through API and graphically represents it on the Cloud Vector dashboard of the DevOps Portal. The Cloud Vector dashboard represents a cloud environment in a form of a cloud map that enables you to easily identify the number of nodes running in your cloud environment. The entities that build the map include availability zones (AZs), hosts, VMs, and services.

The following diagram describes the high level architecture of the Cloud Vector service:

Hardware Correlation service overview

The Hardware Correlation service obtains the data from the Cloud Intelligence service and StackLight LMA Prometheus and graphically represents it on the HW Correlation dashboard of the DevOps Portal. The HW Correlation correlation dashboard provides the capability to generate an on-demand dashboard to graphically illustrate the consumption of the physical host resources by VMs running on a particular host. For example, you can view a dashboard with the CPU, memory, and disk consumption for the compute node where a specific VM is running and for the VM itself.

The following diagram describes the high level architecture of the Hardware Correlation service:
Heatmaps service overview

The Heatmaps service obtains the data from the StackLight LMA Prometheus and graphically represents it on the Heatmaps dashboard of the DevOps Portal. The Heatmaps dashboard provides the information about resource consumption by the cloud environment identifying the load of each node and number of alerts triggered for each node.

The dashboard includes heatmaps for the following data:

- Memory utilization
- CPU utilization
- Disk utilization
- Alerts triggered

The following diagram describes the high level architecture of the Heatmaps service:
Logging, monitoring, and alerting

StackLight LMA, the Logging, Monitoring, and Alerting toolchain, is the operational health and response monitoring solution of the Mirantis Cloud Platform (MCP). It is based on Prometheus, an open-source monitoring solution and a time series database.

StackLight LMA overview

StackLight LMA monitors nodes, services, cluster health, and provides reach operational insights out-of-the-box for OpenStack, Kubernetes, and OpenContrail services deployed on the platform. Stacklight LMA helps to prevent critical conditions in the MCP cluster by sending notifications to cloud operators so that they can take timely actions to eliminate the risk of service downtime. Stacklight LMA uses the following tools to gather monitoring metrics:

- Telegraf, a plugin-driven server agent that monitors the nodes on which the MCP cluster components are deployed. Telegraf gathers basic operating system metrics, including:
  - CPU
  - Memory
  - Disk
  - Disk I/O
  - System
  - Processes
  - Docker

- Prometheus, a toolkit that gathers metrics. Each Prometheus instance automatically discovers and monitors a number of endpoints such as Kubernetes, etcd, Calico, Telegraf, and others. For Kubernetes deployments, Prometheus discovers the following endpoints:
  - Node, discovers one target per cluster node.
  - Service, discovers a target for each service port.
  - Pod, discovers all pods and exposes their containers as targets.
  - Endpoint, discovers targets from listed endpoints of a service.

By default, the Prometheus database stores metrics of the past 15 days. To store the data in a long-term perspective, consider one of the following options:

- (Default) Prometheus long-term storage, which uses the federated Prometheus to store the metrics (six months)
- InfluxDB, which uses the remote storage adapter to store the metrics (30 days)

Using the Prometheus web UI, you can view simple visualizations, debug, add new features such as alerts, aggregates, and others. Grafana dashboards provide a visual representation of all graphs gathered by Telegraf and Prometheus.
StackLight LMA components

StackLight LMA consists of the following components:

Prometheus server

Collects and stores monitoring data. A Prometheus server scrapes metrics from Telegraf, exporters, and native endpoints, such as Calico, etcd, or Kubernetes, either directly or through Pushgateway. Prometheus stores all scraped samples in a local database and runs rules over this data to either record new time series from existing data or generate alerts. Prometheus stores the data as time series: streams of time-stamped values that belong to the same metric and the same set of labeled dimensions. Timestamps have a millisecond resolution, while values are always 64-bit floats. Prometheus has a dimensional data model. Any given combination of labels for the same metric name results in a separate time series. The Prometheus Query Language (PromQL) enables filtering and aggregation based on these dimensions. Grafana uses the data stored in Prometheus to provide graphs and charts.

The built-in alarms defined in Salt formulas detect the most critical conditions that may occur. However, using the Reclass model you can modify and override the built-in alarms as well as create custom alarms for a specific deployment. Both built-in and custom alarms use the same declarative YAML structure.

If more than one instance of Prometheus is deployed, they perform as independent Prometheus servers not connected to each other. However, these instances gather the same endpoints. Therefore, in case of any failure in one Prometheus server, another Prometheus server will contain the same data in the database.

Alertmanager

Handles alerts sent by client applications such as the Prometheus server. Alertmanager deduplicates, groups, and routes alerts to receiver integrations. By default, StackLight LMA is configured to send email notifications. However, you can also configure it to send notifications to Salesforce. This can be configured separately for any alert. Alertmanager also performs silencing and inhibition of alerts.

Alerta

Receives, consolidates and deduplicates the alerts sent by Alertmanager and visually represents them through a simple yet effective web UI. Using Alerta, you can easily view the most recent alerts, watched alerts, as well as group and filter alerts according to your needs. Alerta uses MongoDB as a back end.

Telegraf and exporter agents

Collect metrics from the system they are running on. Telegraf runs on every host operating system and on every VM where certain services of MCP are deployed. Telegraf collects and processes the operational data that is relevant to the scope of a node including hardware, host operating system metrics, local service checks, and measurements. Telegraf is plugin-driven and has the concept of two distinct set of plugins.
Input plugins collect metrics from the system, services, or third-party APIs
Output plugins write and expose metrics to various destinations

Pushgateway
Enables ephemeral and batch jobs to expose their metrics to Prometheus. Since these jobs may not exist long enough to be scraped, they can instead push their metrics to the Pushgateway, which then exposes these metrics to Prometheus. Pushgateway is not an aggregator or a distributed counter but rather a metrics cache. The metrics pushed are exactly the same as scraped from a permanently running program.

Grafana
Builds and visually represents metric graphs based on time series databases. Grafana supports querying of Prometheus using the PromQL language.

Long-term storage system
Uses one of the following set of components to store the data for further analysis:

- **Prometheus** long-term storage that scrapes all data from the Prometheus server. This historical data can then be used for analytics purposes. Prometheus Relay adds a proxy layer to Prometheus to merge the results from underlay Prometheus servers to prevent gaps in case some data is missing on some servers. Grafana uses the data from Prometheus long-term storage. This approach is used by default.

- **InfluxDB** long-term storage that scrapes the data using the remote storage adapter. This historical data can then be used for analytics purposes. InfluxDB Relay adds a basic high availability layer to InfluxDB by replicating the InfluxDB data to a cluster of InfluxDB servers. If you choose the Prometheus long-term storage, InfluxDB will still be used in case you deploy the Tenant Telemetry that uses the StackLight LMA back ends.

Logging system
Responsible for collecting, processing, and persisting the logs. The logging system components include:

- **Fluentd** (log collector), which parses logs, sends them to Elasticsearch, generates metrics based on analysis of incoming log entries, and exposes these metrics to Prometheus. Fluentd runs on every node in a cluster.

- **Elasticsearch**, which stores logs and notifications, and **Elasticsearch Curator**, which maintains the data (indexes) in Elasticsearch by performing such operations as creating, closing, or opening an index as well as deleting a snapshot. Additionally, Elasticsearch Curator can also manage the data retention policy in Elasticsearch clusters. Elasticsearch Curator runs on each Elasticsearch node within the log storage nodes.

The metrics derived from logs are used to alert the operator upon abnormal conditions such as a spike of HTTP 5xx errors. Elasticsearch receives and indexes the logs for viewing and searching in Kibana.

Gainsight integration service
You can integrate StackLight LMA with Gainsight. Gainsight integration service queries Prometheus for the following metrics data, combines the data into a single CSV file, and sends the file to the Salesforce Gainsight extension through API:

- The amount of vCPU, vRAM, and vStorage used and available
- The number of VMs running, compute nodes, and tenants/projects
- The availability of Cinder, Nova, Keystone, Glance, and Neutron
By default, Gainsight integration service sends the data to API once per day. Mirantis uses the collected data for further analysis and reports to improve the quality of customer support.

The following diagram illustrates data flow and connections between the StackLight LMA services. Prometheus long-term storage is illustrated as the default option.

The Prometheus, Pushgateway, Alertmanager, Alerta, Grafana, and Gainsight services run on a separate Docker Swarm Mode cluster deployed on the monitoring VMs. The following diagram illustrates the composition of StackLight LMA components across all MCP services. Prometheus long-term storage is illustrated as the default option.
StackLight LMA high availability

High availability in StackLight LMA is achieved through the deployment of three Prometheus servers, Prometheus Relay service, and InfluxDB Relay service.

To ensure high availability for Prometheus, StackLight LMA deploys three Prometheus servers at the same time. Each Prometheus server uses the same configuration file, monitors the same endpoints, and has the same alerts defined. The Alertmanager service deduplicates the fired alerts, so you will receive one alert instead of three.

For external components such as Grafana, the Prometheus Relay service handles Prometheus API calls, sends them to all discovered Prometheus servers, merges the results, and returns them to Grafana to visualize the data from all Prometheus servers. Therefore, if one Prometheus servers is down, Grafana will contain the data from the remaining Prometheus servers.

The following diagram illustrates the Prometheus HA.
High availability for Prometheus long-term storage is achieved by scraping the same data in an independent way. In case one Prometheus server fails, the other two will contain the data. To keep the time series gapless, Prometheus Relay acts as a proxy and merges the results from three underlay Prometheus servers.

High availability for the InfluxDB service is achieved using the InfluxDB Relay service that listens to HTTP writes and sends the data to each InfluxDB server through the HTTP write endpoint. InfluxDB Relay returns a success response once one of the InfluxDB servers returns it. If any InfluxDB server returns a 4xx response or if all servers return a 5xx response, it will be returned to the client. If some but not all servers return a 5xx response, it will not be returned to the client.

This approach allows sustaining failures of one InfluxDB or one InfluxDB Relay service while these services will still perform writes and serve queries. InfluxDB Relay buffers failed requests in memory to reduce the number of failures during short outages or periodic network issues.

Monitored components

StackLight LMA measures, analyzes, and reports 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.

StackLight LMA monitors the following components and their sub-components, if any:

- Linux operating system
- Salt Master node
- Salt Minion node
- Jenkins
- RabbitMQ
- Apache
- libvirt
- Memcached
- MySQL
- HAProxy
- NGINX
- NTP
- GlusterFS
- OpenStack (Nova, Cinder, Glance, Neutron, Keystone, Horizon, Heat, Octavia)
- OpenContrail (Cassandra, Contrail server, Zookeeper)
- Kubernetes (Kube-apiserver, Kube-controller-manager, Kube-proxy, Kube-scheduler, Kubelet, Docker, etc)
- Calico (Felix, BIRD, confd)
- Ceph (OSD, ceph-mon)
- StackLight LMA (Alertmanager, InfluxDB, InfluxDB Relay, Elasticsearch, Heka, Grafana, Kibana, Prometheus, Pushgateway, Telegraf, remote storage adapter)