Mirantis OpenStack Monitoring Guide

version 9.2

Contents

Copyright notice

Preface

This documentation provides information on how to use Fuel to deploy OpenStack environments. The information is for reference purposes and is subject to change.

Intended Audience

This documentation is intended for OpenStack administrators and developers; it assumes that you have experience with network and cloud concepts.

Documentation History

The following table lists the released revisions of this documentation:

Revision Date	Description
February 6, 2017	9.2 GA

Introduction

This document does not attempt to tout a particular solution or monitoring system for OpenStack. Instead, it strives to provide best practices and provide specific guidelines about how to monitor OpenStack effectively irrespectively of the technology being used. This includes specific examples about how to collect and process key metrics to increase your operational visibility, check various health indicators to detect critical failure conditions, index and search the logs for root cause analysis and troubleshooting. Also, it must be highlighted from the start that this document provides guidelines for monitoring the OpenStack infrastructure and host services. It is not a guide for the monitoring the virtual machines nor the applications running on top of them.

The expected outcome is two-fold:

- Gain insights into what is critically important to watch in OpenStack so that operators can be alerted in near real-time to anticipate and react to undesirable situations.
- Provide a comprehensive set of guidelines to implement your own monitoring system. In that sense, this document can also be viewed as a specification you can use to implement your own solution using technologies like Zabbix or the LMA Toolchain that are provided as Fuel plugins for Mirantis OpenStack 6.1 onward.

In addition, we think that an effective monitoring solution for OpenStack should have the following main characteristics.

- Provide near real-time insights and alerting.
- Support discovery and configuration management automation so that the error prone manual setup can be completely avoided.
- The monitoring system supports its own self-monitoring and high availability.

Assumptions

We assume that the reader is already familiar with the concepts, architecture principles, and day-to-day administration tasks of Mirantis OpenStack. It further assumes that you have deployed Mirantis OpenStack following the recommendations in the Mirantis OpenStack Planning Guide, as well as deployed your environment using Neutron with VLAN or GRE or networking segmentation.

Intended audience

The primary audience of this document are the architects and technical staff involved in the design and deployment of an OpenStack cloud. The other audiences are the members of the operations staff that are in charge of managing and maintaining the OpenStack cloud in a healthy state on a daily basis. This includes:

Line of Business Owner

The Line of Business Owner needs to know how "things" are running and if there are any problems that may affect the SLA. This person focuses on marketing and business, not IT, and, thus, is interested in top level indicators to know about services health.

Operational support

Provides support to customers encountering issues. Is generally organized with a service desk and two support levels for problem escalation. The support relies on monitoring solutions to perform diagnostics and also should benefit from preventive alerts.

Subject Matter Expert

Investigates and resolves a domain-specific problem. Validates the resolution. Uses the monitoring system to troubleshoot and observe the cloud infrastructure behavior.

Document scope

This guide is about how to monitor an OpenStack cloud from the perspective of the operations staff with a focus on the infrastructure. As a result, this guide is not directly intended to serve the monitoring needs of a cloud user whether it has access to the administrator role or not because as a cloud user you do not have root access to the servers and host operating systems. The scope therefore includes some hardware monitoring through IPMI, monitoring of the host operating system, monitoring of the cloud management system and processes that are part of its ecosystem.

The processes supporting the cloud management system are roughly of two kind:

- The OpenStack service API endpoints, like nova-api, which receive the user requests.
- The OpenStack service workers connected to the AMQP bus, like nova-scheduler, which process the user requests.

The OpenStack services depend on a number of additional programs that are not part of the OpenStack code base itself but which nonetheless are critically important to monitor as we will see below. This includes but is not limited to Libvirt, MySQL, RabbitMQ, Memcached, HAProxy, Corosync, and Pacemaker.

The scope also includes the host operating systems, the servers and devices such as the disks and network interface cards. Some amount of hardware health checks via IPMI are performed to monitor the status of equipments such as the fans and CPU temperature in an attempt to help with anticipating hardware failures.

The scope of this document does not include the monitoring of the end-user applications as well as the monitoring of the hardware equipments that are vendor-specific or too complex to be practically addressed in this document. This includes but is not limited to the following equipment categories.

The network gears

The monitoring of the network gears such as switches and routers is vendor-specific and too large to be addressed here.

The storage gears

The monitoring of the storage gears like SANs and NASs is vendor-specific and too large to be addressed here.

Common monitoring practices

This chapter describes the common monitoring practices we recommend that you use to design and implement an effective monitoring solution for OpenStack.

Monitoring domains

An effective monitoring solution is comprised of distinct activities aimed at addressing the different problem domains that the operations staff will have to handle. These activities are summarized below.

Availability Monitoring

Availability monitoring, in its broadest sense, is a monitoring activity that is responsible for ensuring that the resources for compute, storage, and networking, as well as the services mediating their access (via the service API endpoints), are effectively available for end-users to consume while meeting the performance requirements of the SLA. In terms of availability monitoring, we use relevant indicators (or metrics); they provide information on how many resources are currently available in the cloud infrastructure as well as the process checks ensuring that the services delivering the access are up and running. Those indicators are obtained from running synthetic transactions, parsing the logs, metrics collectors deployed throughout the system, and so forth.

Performance Monitoring

Performance monitoring is supposes measuring how fast a particular resource can be served by the cloud infrastructure in response to a user request. For example, measuring how much time it takes to create an instance or a volume. Key metrics for performance monitoring can be obtained not only from synthetic transactions simulating an end-user interaction with a service endpoint but also from analysing the logs, instrumenting the code, and extracting performance metrics from the OpenStack notifications.

OpenStack performance and availability monitoring are the two main monitoring issues developed in this document since they directly relate to the SLA.

Resource Usage Monitoring

Resource usage monitoring is only partially addressed here. We view it as a derivative activity by which a cloud operator can retrieve how much resources were consumed by a particular user or tenant during a particular time period for chargeback. Resource usage monitoring supposes measuring consumable resources of the cloud via the APIs. Another key difference between resource usage monitoring and availability monitoring is that resource usage monitoring does not have to be performed in real-time. Readers interested in resource usage monitoring for OpenStack should take a look at the Ceilometer project.

Alerting

Alerting is a process by which the monitoring system notifies the cloud operator about an undesirable situation. The situation is typically described in an alarm like manner, for example, when the value of a key indicator crosses a threshold or unexpectedly changes a value from OK to NOT OK. An unexpected change of state, if not the direct manifestation of a problem, is often a precursor of it. Besides, alerting should have the following properties:

- Provide a comprehensive description of the problem.
- Provide information about which service is affected.

- Provide a severity level.
- Provide the ability to be disabled to avoid false positives during maintenance.
- Provide the ability to combine alarms to express more complex situations.
- Provide the ability to refer to time-series statistics like median, standard deviation and percentiles.

Furthermore, we recommend that the health status of any OpenStack service is expressed using three different values:

- Healthy - when both the HA functions of the controller cluster are still being ensured and no critical errors are being reported by the monitoring system for a service.
- Degraded - when one or more critical errors are reported by the monitoring system for a service but the HA functions of the controller cluster are still being ensured.
- Failed - when both the HA functions of the controller cluster are not being ensured anymore and one or more critical errors are being reported by the monitoring system for a service.

A critical error should always be reported in an alert.

The immediacy of the operations staff's response to an alert depends on the actual status of the HA cluster. It can be any of the following:

- Immediate - when a service is failed. It is a critical situation and so, the alert should be sent to the operations staff for human intervention.
- Deferred - when a service is degraded. While a degraded service may have a negative impact on the quality of service, the nominal function of the cloud service should continue to be ensured by the system and so, the handling of the alert could be safely prioritized through a ticketing system.

Obviously, not all errors are critical. An effective monitoring solution should put a great deal of care at defining the proper level of alerting (smart alerting), in order to avoid flooding the operations staff with benign notifications that are not reflective of a critical situation. This document strives to provide some hints about how to set your alarms with threshold values and status checks but your mileage may vary depending on your particular OpenStack environment. Rally is a load generator for OpenStack that you could use to calibrate the alarms of your monitoring system.

Monitoring activities

As stated earlier, this document is not prescriptive of a particular monitoring system or solution. Instead, it strives to describe common monitoring practices with problem domains and activities to address them that are key to get clear operational insights in order to take actions when problems occur. In this chapter, we try to describe what those activities are as clearly as possible.

Services, processes, and cluster checks

Service checks

Checking the availability of the OpenStack services from the point of view of the user is absolutely necessary in order to make sure that all the services respond to the user requests as expected.

Those service checks should be performed on a regular basis using synthetic transactions that perform various HTTP requests against the OpenStack service endpoints. The service checks should return an availability status that is either pass or failed.

The synthetic transactions should use a dedicated project (tenant) and user so that it is always possible to differentiate between the load generated by the service checks versus the load generated by the genuine activity.

The service checks should be performed against the service endpoints using the HAProxy's Virtual IP (VIP) to ensure that both the service APIs and as the HAProxy, which distributes the load above them, are traversed by the HTTP requests.

The service checks should strive to minimize the observer effect by using lightweight and non-intrusive (read only) requests. The service checks should also avoid using requests that propagate to other services than the service being watched. A general rule of thumb to avoid overloading the system is to use a non-aggressive polling interval as a tradeoff between being alerted of errors quickly and generating a load that is too heavy.

The service checks should also strive to record the response time of synthetic transactions in order to establish a performance baseline and produce statistics (average, mean, percentile, standard deviation) that can be used to detect anomalies.

Process checks

Process checks can be performed either remotely (via SSH) or locally using a monitoring agent running on each of the OpenStack nodes. The goal of process checks is to ensure that all the processes participating in the support of an OpenStack role, such as controller, are indeed up and running. Those processes include the service endpoints (nova-api), the service workers connected to the AMQP bus (nova-scheduler), as well as the various processes supporting the auxiliary functions such as RabbitMQ, MySQL and so forth.

The process checks should also ensure that the OpenStack processes are bound to their respective networking ports.

Cluster checks

Fuel deploys high-availability OpenStack reference architecture that ensures that all components of an OpenStack environment are redundant. For example, the service endpoints are distributed in an active/active HA cluster along with an HAProxy running on top of them on each controller node. There is only one active HAProxy at a time that is responsible for detecting service endpoints failures and distributing the load between them. The high availability of the HAProxy itself is supported by an active/passive HA cluster based on Corosync and Pacemaker. Corosync and Pacemaker are collectively responsible for ensuring the transparent failover of the HAProxy and VIP when the master node is failed. Same thing for the MySQL database. So a correct appraisal of the OpenStack services' availability status depends on the HA cluster healthiness appraisal. That is why we stated earlier that the criticality of an OpenStack error should be evaluated differently depending on whether the HA functions of the cloud infrastructure are still being ensured by the HA cluster or not. In a nutshell, the HA functions are not being ensured anymore when there is no more failover node available in the cluster or when the HA cluster itself is broken.

Metering

Metering is usually done at the source of measurement point with the help of a monitoring agent running on each node being measured. The goal of metering is to collect operational data metrics which are collections of numeric values organized in groups of consecutive, chronologically ordered lists. Each data input consists of a recorded measurement value, a timestamp at which the measurement took place, and a set of properties describing it. When data inputs from a metric are segmented into fixed intervals and summarized by a mathematical transformation in some meaningful way, they can be stored as time series and interpreted on two-dimensional plots.

The benefits of using time series for monitoring are in their ability to accurately illustrate the process of change in the context of historical data. They are indispensable as they answer the question when and what has changed in an OpenStack cloud.

Logs processing

In addition to performing health checks and collecting metrics, a common monitoring practice is to exploit the information that is available in the logs that are produced by the system. Valuable operational data can be extracted from those unstructured messages which should be indexed for search and troubleshooting. The log messages that are produced by the Mirantis OpenStack distribution are sent to syslog at the INFO level by default. Fuel allows to easily configure OpenStack to send all the logs to an external rsyslog server. Those logs contain information about the severity level, the program that issued the log, the service (Nova, Glance, Cinder, ...) that issued the log syslogfacility, metadata info like tenant_id and request_id that are useful for aggregation and correlation, error codes like the HTTP error codes of the service endpoints, performance info like the HTTP response time, and so forth.

Ideally, we should have one syslog facility per service but there are only eight local facilities.

> Note
>
> The HTTP requests response time is meaningful only for synchronous transactions. Asynchronous transactions like those involved in the creation of an instance or volume will only account for the time it takes for the service API endpoint to authenticate and transmit the request to the AMQP bus, and as such, is not reflective of the actual time it takes to process a request end-to-end.

Operations metrics can be derived from those logs for diagnostic and alerting purposes. This includes but is not limited to:

- Operations errors. A sudden spike of errors in the logs, like those found in service API endpoint logs (HTTP return code 5xx), should be monitored since they may be the manifestation of a critical condition.
- HTTP transactions time. The HTTP transactions time should be monitored since it directly affects the end user experience.
- Logs rate. A sudden drop-off of the logs rate should be monitored since it can indicate something went wrong in the system or users do not have access to the cloud any more.

Logs indexing

Another common practice is to index all the logs in a central database like Elasticsearch with Kibana running on top of it to easily search and correlate those logs for root cause analysis and troubleshooting. With complex distributed systems like OpenStack, it is no longer possible to use ad hoc tools like grep and awk to search the logs for troubleshooting.

> Note
>
> The LMA Toolchain provides a Fuel plugin that allows deploying ElasticSearch and Kibana you can use to search and correlate the OpenStack logs and notifications.

OpenStack notifications processing

The OpenStack notifications are another source of valuable operational data information that can be exploited by a monitoring system. The OpenStack notifications, as opposed to logs, are structured messages that are sent to the AMQP bus through the notifications topic with an info priority by default. Some OpenStack services send notifications with a warning and error priority. Notifications contain rich data sets that can be exploited to extract performance metrics and operations status for the service workers at different levels of the stack. Most of the OpenStack services publish notifications to the AMQP bus. See the list of notifications for details.

> Note
>
> Currently, if you enable the notifications, you also need to take care of effectively consuming them otherwise the queue will grow indefinitely.

In addition to the built-in notifications, there is a possibility to configure each service endpoint to emit notifications of type http.request and http.response for all HTTP transactions. This is achieved by adding the notification middleware in the WSGI pipeline.

The LMA Collector plugin, that can be deployed on the controller nodes, taps into the AMQP bus to collect and process the notifications. Out of those notifications, the LMA Collector plugin creates new metrics that can be sent to a time-series database. It includes:

- Nova instance creation time
- Cinder volume creation time

Other asynchronous operations like Glance image creation time or Neutron network/port creation time can be computed the same way.

> Note
>
> The LMA Toolchain provides a Fuel plugin that allows to deploy InfluxDB and Grafana to plot those metrics.

Diagnosing versus alerting

The operational data gathering, that results from the common monitoring practices described above, should serve two different purposes that should not be confused. One is to identify the root cause of a problem. Hereafter, this is referred to as diagnosing. The other one is to send real-time notifications to the operator when something is broken that needs to be repaired. Hereafter, this is referred to as alerting. But those alerts have to be smart. As stated multiple times it is pointless and ineffective to flood the operator with alerts that are not reflective of a situation that requires the operator’s attention whether it is immediate or deferred.

Time synchronization

Lastly, it is utterly important that all the OpenStack nodes and the monitoring system be on the exact same time clock. Without a proper time synchronisation across the system it will be impossible to make any kind of sensible root cause analysis, metrics time-series will be useless, it will cause all sorts of high availability and operations management problems. It is also a good practice to set up the UTC time zone for all the nodes. Usually, the Network Time Protocol(NTP) is used to synchronize the system clocks with remote NTP time servers. The ntpd daemon must run on each node and should be configured to use several external time servers. The Linux distributions provide packages with pre-configured NTP servers but it is necessary to use a pool of geographically closest NTP servers. The monitoring system should check that the ntpd server is alive and kicking to ensure that the OpenStack cloud is time synchronized across all the nodes.

Monitoring activities details

In this chapter we are getting into the details of how the monitoring activities introduced above can be implemented. It is worth restating that a common trait of those monitoring activities is to collect and process the operational data that should increase the operational visibility about how an OpenStack cloud behaves over time. In other words, get the level of insights that is required to make value judgments about what should be done to keep your OpenStack cloud kicking and healthy.

> Note
>
> The LMA Collector, available as a Fuel plugin, does all the heavy lifting work of collecting and processing those operational data for you. Please refer to the LMA Collector documentation to understand how it works and how it can be used. Now, you have the choice between using the LMA Collector directly or build your own solution based on the guidelines described below.

The chapter is organized in sections where each section covers a particular OpenStack service or auxiliary component like RabbitMQ for the AMQP bus or Corosync/Pacemaker for the HA cluster. Then, each section is further divided into subsections describing:

- The process checks, which gives you a list of all processes involved in the support of a particular service function, including details about the role of the node, where the process is running, the incoming connections and a port number and their dependencies. As we have seen above, a failed process check should always be reported as a critical error, but the ensuing alert will not necessarily require an immediate attention.
- The service API checks, which gives you a list of the API endpoints that you should monitor with an example of synthetic transaction you can use to verify that the service responds properly to user requests. A failed service API check should always be reported as a critical error and the ensuing alert should call for immediate attention.
- The operational data metrics, which aims to increase your operational visibility along with a simple method to retrieve their values. Those metrics can be used for both diagnosis and alerting purposes.

Keystone

Keystone is an OpenStack service that provides identity, token, catalog, and policy services to users and to other OpenStack services. As such, the availability of all the OpenStack services depends on the availability of Keystone.

Process Checks

Process name	Incoming connections	Role	Dependencies	HA mode
keystone-all	HTTP 5000 (public) and 35357 (admin)	controller	db, memcached, Apache	active/active

API checks

Check the proper functioning of authentication and token revocation operations:

- POST /v2.0/tokens
- DELETE /v2.0/tokens/{token-id} or DELETE /v3/auth/tokens

Collected Metrics

Metrics	Source	Purpose
authentication errors	log: POST /v2.0/tokens HTTP/1.1" 401 330 0.205647 401 error code indicates an authentication error	alert: When a sudden spike of errors is detected. A high authentication errors rate can be the symptom of a brute-force attack.
authentication response time	log: POST /v2.0/tokens HTTP/1.1" 200 4199 0.092479 where 0.092479 is the response time in seconds	alert: When the value is beyond standard deviation or top percentiles threshold depending on the data-points distribution.
token validation errors	logs GET /v3/auth/tokens HTTP/1.1" 404 7317 0.071319 #404 indicates a token validation error	alert: When a sudden spike of errors is detected.
token validation response time	logs: GET /v3/auth/tokens HTTP/1.1" 200 7317 0.071319 # 0.071319 is the response time in seconds	alert: When the value is beyond standard deviation or top percentiles threshold depending on the data-points distribution.
number of users	poll API: GET /v2.0/users	diag
number of tenants	poll API: GET /v2.0/tenants	diag
API errors	Logs or HAProxy: All HTTP 500 error code.	alert: When a sudden spike of errors is detected.

Nova

Nova is the OpenStack service for Compute, a cloud computing fabric controller, the main part of a cloud system. Nova is composed of several processes, each assuring a particular function. The Nova processes are distributed on the controller node(s) and compute nodes.

Process checks

Process name	Incoming connections	Role	Dependencies	HA mode
nova-api	HTTP 8774 (Nova API), 8773 (Nova EC2 API)	controller	amqp	active/active
nova-scheduler	RPC	controller	amqp	active/active
nova-conductor	RPC	controller	db, amqp	active/active
nova-consoleauth	RPC	controller	amqp	active/active
nova-console	RPC	controller	amqp	active/active
nova-novncproxy	RPC	controller	amqp	active/active
nova-cert	RPC	controller	amqp	active/active
nova-compute	RPC	controller	libvirt	not available

API checks

Check the proper functioning of the API with a read operation. Example:

- list of flavors
    - GET /v2/<tenant-id>/flavors

A more intrusive operation checks if it’s possible to create and delete a keypair:

- create and delete a keypair
    - POST /v2/<tenant-id>os-keypairs '{"keypair": {"name": "test-mon"}}'
    - DELETE /v2/<tenant-id>os-keypairs/test-mon

Collected Metrics

Metrics	Source	Purpose
Total number of instances in error state	poll SQL: select count(*) from instances where vm_state='error' and deleted=0 or poll API: /v2/{tenant_id}/servers/detail/?all_tenant=1	diag That should probably not trigger an alert, since those errors may be due to the user mistakes.
Total number of instances in running state	poll SQL: select count(*) from instances where deleted=0 and vm_state='active' poll API: /v2/{tenant_id}/servers/detail/?all_tenant=1	Alert: Too few running instances could be a symptom of a deeper problem.

Total number of instances per state, where state can be: deleted, paused, resumed, rescued, resized, shelved_offloaded, or suspended.	poll SQL: select instances.vm_state, count(instances.id) from instances where deleted=0 group by vm_state or poll API: GET /v2/{tenant_id}/servers/detail/?all_tenant=1	diag
Number of compute nodes in operational state	poll API GET /v2/{tenant_id}/os-services or SQL: select count(services.id) from services where disabled=0 and deleted=0 and services.binary = 'nova-compute' and timestampdiff(SECOND,updated_at,utc_timestamp())<60;	Alert: Too few running instances could a symptom of a deeper problem.
Number of compute nodes in not operational state	poll API: GET /v2/{tenant_id}/os-services or SQL: select count(services.id) from services where disabled=0 and deleted=0 and services.binary = 'nova-compute' and timestampdiff(SECOND,updated_at,utc_timestamp())>60;	diag
Number of services offline	poll API: GET /v2/{tenant_id}/os-services or poll SQL: select count(*) from services where disabled=1 and deleted=0 and timestampdiff(SECOND,updated_at,utc_timestamp())>60	diag
Number of services available per function: conductor, scheduler	poll API: GET /v2/{tenant_id}/os-services poll SQL: select services.binary, count(services.id) from services where disabled=0 and deleted=0 and timestampdiff (SECOND,updated_at,utc_timestamp())>60 group by services.binary;	diag
Number of running instances per compute node	API: GET /v2/{tenant-id}/os-hypervisors/detail	diag
Total number of VCPUs	poll SQL: select ifnull(sum(vcpus), 0) from compute_nodes where deleted=0 poll API: /v2/{tenant_id}/os-hypervisors/statistics	diag

Total number of VCPUs used	poll SQL: select ifnull(sum(vcpus), 0) from instances where deleted=0 and vm_state='active' poll API: /v2/{tenant_id}/os-hypervisors/statistics	diag
Total number of free VCPUs	calculated from previous metrics poll API: /v2/{tenant_id}/os-hypervisors/statistics	Alert: Too few running instances could be a symptom of a deeper problem.
Total memory available	poll SQL: select ifnull(sum(memory_mb), 0) from compute_nodes where deleted=0 poll API: /v2/{tenant_id}/os-hypervisors/statistics	diag
Total memory used by instances	poll SQL: select ifnull(sum(memory_mb), 0) from instances where deleted=0 and vm_state='active' poll API: /v2/{tenant_id}/os-hypervisors/statistics	diag
Total free memory	calculated from previous metrics or poll API: /v2/{tenant_id}/os-hypervisors/statistics	diag
API response time	Logs: see examples below	Alert: When the value is beyond standard deviation or top percentiles threshold depending on the data-points distribution.
API errors	Logs or HAProxy: All HTTP 500 error code. Log example: POST /v2/{tenant-id}/os-volumes_boot HTTP/1.1" status: 500 len: 354 time: 32.3032150 #where status: 500 indicates error	Alert: When a sudden spike of errors is detected.

Note

The main advantage of using SQL queries versus using API checks is execution speed and lower overhead. The disadvantage of using SQL queries is that your checks won’t work anymore when the SQL schema changes.

Examples of log entries containing response times:

Synchronous operations response time is logged in nova-api.log. Example of nova key pair creation log entry:

2015-03-02 12:33:59.898 6819 INFO nova.osapi_compute.wsgi.server [req-c0391ca2-e0e2-41bf-af64-0df222654620 None] 192.168.0.5 "POST /v2/{tenant-id}/os-keypairs HTTP/1.1" status: 200 len: 2473 time: 1.4112680

HTTP response code is logged in nova-api.log. Example of an instance creation log entry:

2015-03-02 12:43:59.898 6819 INFO nova.osapi_compute.wsgi.server [req-c0391ca2-e0e2-41bf-af64-0df222654620 None] 192.168.0.5 "POST /v2/{tenant-id}/servers HTTP/1.1" status: 202 len: 780 time: 2.4308009

202 (ACCEPTED) return code indicates the request has been accepted for processing.

Network

Neutron

Neutron is the OpenStack service providing network connectivity as a service between network interfaces (vNICs) managed by other OpenStack services like Nova.

> Note
>
> Neutron plugins load balancer, firewall, and ipsec monitoring are not covered in this version of the document.

Process Checks

Process name	Incoming connections	Role	Dependencies	HA mode
neutron-server	HTTP 9696	controller	db, amqp	active/active
neutron-dhcp-agent	RPC	active controller	amqp, dnsmasq	active/passive
neutron-l3-agent	RPC	active controller	db, iptables	active/passive
neutron-metadata-agent	RPC	compute	amqp, db	active/active
neutron-ns-metadata-proxy	RPC	controller	amqp, db	active/passive
neutron-openvswitch-agent	RPC	all nodes	amqp, ovs	active/passive
dnsmasq	UDP port 67	controller		active/passive, see DHCP agent below

API checks

Check the proper functioning of the API with a read operation. Example:

- list subnets
    - GET /v2.0/subnets

Collected Metrics

Metric	Source	Purpose
Number of networks	poll API GET /v2.0/networks	Alert: Too few active networks could be the symptom of a deeper problem.
Number of subnets	poll API GET /v2.0/subnets	diag
Number of routers	poll API GET /v2.0/routers	diag
Number of ports	poll API GET /v2.0/ports	diag
API errors	Logs or HAProxy: All HTTP 500 error code. Log example: INFO neutron.wsgi [{req-id} None] 192.168.0.1 - - [11/Mar/2015 19:17:22] "POST /v2.0/networks.json HTTP/1.1" 500 324 0.178729 # where 500 indicates an error	Alert: When a sudden spike of errors is detected.

DHCP agent

The neutron-dhcp-agent relies on dnsmasq to handle the DHCP requests which provide network configuration to instances.

We recommend you perform the following checks to detect anomalies:

- There must be at least one dnsmasq process per tenant network when DHCP is enabled.
- Too many DHCPNAK entries in dnsmasq logs could be the symptom of connectivity issues with the instances.

Open vSwitch

Open vSwitch is a central component of tenant networking. You should check that the following processes are up and running.

Process name	Role	HA mode

ovsdb-server	all nodes	not available
ovs-vswitchd	all nodes	not available

Also, a good practice for diagnosis is to collect the number of dropped packets and packets in the error state per interface. We recommend, however, not to consider those errors as critical ones since they do not necessarily represent a service failure.

```
# ovs-vsctl get Interface br-tun statistics
{collisions=0,
rx_bytes=648,
rx_crc_err=0,
rx_dropped=0,
rx_errors=0,
rx_frame_err=0,
rx_over_err=0,
rx_packets=8,
tx_bytes=0,
tx_dropped=0,
tx_errors=0,
tx_packets=0}
```

The Open vSwitch logs are stored in the /var/log/openvswitch/ directory.

Glance

Glance is the OpenStack service allowing users to upload and discover data assets that are meant to be used with other services. This currently includes images and metadata definitions used by the nova service.

Process checks

Process name	Incoming connections	Role	Dependencies	HA mode
glance-api	9292	controller	db, amqp	active/active
glance-registry	9191	controller	db, amqp, storage	active/active

Glance can use different storage backends: Swift or Ceph that are used by default in HA deployments.

API checks

Check the proper functioning of the API with a read operation. Example:

- list images
    - GET /v1/images

A more intrusive test checks the complete creation of an image:

- create an image by uploading a small image (a few megabytes in size)

    - POST /v1/images
- get image details
    - GET /v1/images/{image_id}
- delete an image
    - DELETE /v1/images/{image_id}

Collected Metrics

Metrics	Source	Purpose
Number of active images public/private	poll API GET /v2/images	diag and/or alert: Too few active images could be the symptom of a deeper problem.
Number of images per status (active, queued, saving)	poll API GET /v2/images	diag
Total size of active images	poll API GET /v2/images?visibility=public&status=active	diag
API errors	Logs or HAProxy: All HTTP 500 error code. Examples of a log entry containing a failed image upload error: 2015-03-02 12:44:12.438 1212 INFO glance.wsgi.server [{request-id} {user-id} {tenant-id} - - -] 192.168.0.1 - - [04/ Mar/2015 12:38:55] "POST /v1/images HTTP/1.1" 500 877 49.117649	Alert: When a sudden spike of errors is detected.

Cinder

Cinder is the OpenStack service for block storage. It allows users to manage block storage resources that could be attached to the instances.

Process checks

Process name	Incoming connections	Role	Dependencies	HA mode
cinder-api	8776	controller	db, amqp	active/active
cinder-scheduler	RPC	controller	amqp	active/active

cinder-volume	RPC	storage-cinder	db, amqp, storage	active/active (Ceph backend) n/a (LVM backend)

Cinder can use different block storage backends like LVM or Ceph.

API checks

Check the proper functioning of the API with a read operation. For example:

- list volume
    - GET /v2/{tenant_id}/volumes

A more intrusive test may want to check the complete creation of a volume:

- create a volume
    - POST /v2/{tenant_id}/volumes
- get volume details
    - GET /v2/{tenant_id}/volumes/{volume_id}
- delete a volume
    - DELETE /v2/{tenant_id}/volumes/{volume_id}

Collected Metrics

Metrics	Source	Purpose
number of volumes in error state	poll SQL: select count(*) from volumes where status=’error’	Alert: When the value of error state ratio is beyond a certain threshold.
number of volumes deleting	poll SQL: select count(*) from volumes where status='deleting'	diag
number of snapshots in progress	poll SQL: select count(*) from snapshots where progress NOT LIKE '100%'	diag
number of snapshots deleting	poll SQL: select count(*) from snapshots where status='deleting'	diag
total number of volumes	poll SQL: select count(*) from volumes where deleted != 1;	diag

total size of active volumes	poll SQL: select sum(size) from volumes where deleted != 1 and status = 'available';	diag
API errors	Logs or HAProxy: All with HTTP 500 error code.	alert: When a sudden spike of errors is detected.

Horizon

Horizon is a canonical implementation of the OpenStack dashboard, which provides a web-based user interface to OpenStack services including Nova, Swift, Keystone, and others. Horizon is often the main, if not the only, interface to the OpenStack services, and it is critical to ensure that it is always available and responsive to users.

The Apache HTTP server hosts the Horizon dashboard that is implemented as a WSGI application. The HTTP server running the Horizon dashboard is deployed behind the HAProxy load balancer which distributes the load across the controller nodes cluster. This application does not use the OpenStack database. It is simply a web interface facade for the OpenStack services API endpoints.

Process checks

Process name	Incoming connections	Role	Dependencies	HA mode
apache2 or httpd	HTTP 80	controller	n/a	active/active

Interface checks

- A synthetic HTTP transaction process performing login/logout sequences against the Horizon's VIP should be executed on a regular basis to ensure it is responding properly to user requests.

Collected Horizon Metrics

Horizon metrics should be extracted from the Apache server(s) logs in order to detect errors.

Metrics	Source	Purpose
number of logins	logs: dashboard-openstack_auth.forms : INFO Login successful for user "admin".	alert: The absence of logins during a certain period may indicate that users do not have access to Horizon anymore.
login errors	logs: dashboard-openstack_auth.forms : WARNING Login failed for user "xxx".	alert: When a sudden spike of errors is detected. Could indicate a brute force attack situation.

Collected Apache Metrics

Apache server should be sized and configured according to the expected load. Here are some metrics we suggest that you collect:

Metrics	Source	Purpose
number of requests/sec	server-status	diag
number of bytes served/sec	server-status	diag
number of busy workers	server-status	diag
number of idle workers	server-status	alert: The continuous observation of zero idle workers may be the symptom of a server that is too busy or improperly configured.

Note

Server-status must be enabled in the Apache's configuration to provide information on the server's activity and performance.

Heat

Heat is the OpenStack service to orchestrate the deployment of multiple composite cloud applications using the Heat Orchestration Template (HOT) and also compatible with the AWS CloudFormation template format through both an OpenStack-native ReST API and a CloudFormation-compatible Query API.

Process checks

Process name	Incoming connections	Role	Dependencies	HA mode
heat-api	8004	controller	db, amqp	active/active
heat-engine	RPC	controller	db, amqp, other OpenStack services	active/active
heat-api-cfn	8000	controller	db, amqp	active/active

Note

The heat-api-cloudwatch service is not addressed here as it is deprecated by the Heat team.

Collected Metrics

Metric	Source	Purpose
number of active stacks	API GET /v1/{tenant_id}/stacks	diag
number of stacks in error	API GET /v1/{tenant_id}/stacks	diag
number of stacks in progress	API GET /v1/{tenant_id}/stacks	diag
API errors	Logs or HAProxy: All HTTP 500 error code.	Alert: When a sudden spike of errors is detected.

Ceilometer

Ceilometer is the OpenStack telemetry project which aims to provide a unique point of information to acquire all of the resource usage measurements that operators need for chargeback and billing.

Process checks

Process name	Incoming connections	Role	Dependencies	HA mode
ceilometer-api	HTTP 8777	controller	storage	active/active
ceilometer-agent-central	RPC	controller	amqp	active/passive
ceilometer-agent-compute	RPC	compute	amqp	active/passive
ceilometer-agent-notification	RPC	controller	amqp	active/active
ceilometer-collector	RPC	controller	amqp,storage	active/active
ceilometer-alarm-evaluator	RPC	controller	ceilometer api, storage	active/active
ceilometer-alarm-notifier	RPC	controller	amqp, external system	active/active

ceilometer-agent-central and ceilometer-agent-compute are replaced by a single process named ceilometer-polling.

> Note
>
> Since several storage backends can be used by Ceilometer, the monitoring of these backends is not addressed in this document. See Ceilometer backends list.

API checks

Check the proper functioning of the API with a read operation like listing of samples:

- GET /v2/samples

> Note
>
> When requesting the API, you must use the limit option not to overload the service by retrieving too much data.

Collected Metrics

Metrics	Source	Purpose
API errors	Logs or HAProxy: All HTTP 500 error code.	Alert: When a sudden spike of errors is detected.

Sahara

Sahara is the OpenStack Data processing service. Sahara performs two types of activities in OpenStack:

- Hadoop cluster provisioning
- Executing Elastic Data Processing (EDP) jobs

Sahara runs as a single process service on each controller in MOS 6.x. This, however, will change in MOS 7.x.

Process checks

Process name	Incoming connections	Role	Dependencies	HA mode
sahara-all	HTTP 8386	controller	db	active/active

API checks

Check the proper functioning of the API with a read operation. For example:

- list provisioning plugins
    - GET /v1.1/<tenant-id>/plugins

Collected Metrics for Cluster related activities

Metrics	Source	Purpose

Total number of Sahara clusters in Active state	poll SQL: select count(*) from clusters where status="Active";	diag
Total number of Sahara clusters in Error state	poll SQL: select count(*) from clusters where status="Error";	Alert: Too many clusters in Error state may be a sign of the provisioning engine failure or failures in other OpenStack services.
Total number of Sahara clusters doing provisioning at the moment	poll SQL: select count(*) from clusters where status not in ("Active", "Error");	Alert: Too many clusters in provisioning state may be a sign of a slow network throughput or even missing network connectivity between VMs. It may also be a sign of a corrupted image being used for the clusters.
Total number of VMs used in Sahara clusters	poll SQL select count(*) from instances;	diag
Total number of Cinder Volumes attached to Sahara VMs	poll SQL (cross project) select count(*) from cinder.volumes where instance_uuid in (select instance_id from sahara.instances) and attach_status="attached";	diag
Total number of Security Group Rules generated by Sahara	poll SQL (neutron) select count(*) from neutron.securitygrouprules where security_group_id in (select id from neutron.securitygroups where description LIKE "%Auto security group created by Sahara%");	diag
Total number of Floating IPs attached to Sahara VMs	poll SQL: select count(*) from instances where management_ip is not NULL;	diag

Total number of Glance Images uploaded for Sahara clusters	poll SQL (glance): select count(*) from glance.image_properties where name="_sahara_username" and deleted=0;	diag

Collected Metrics for Elastic Data Processing (EDP) related activities

Metrics	Source	Purpose
Total number of EDP jobs that have finished successfully	poll SQL: select count(*) from job_executions where info like '%"status": "SUCCEEDED"%';	diag
Total number of EDP jobs that have failed	poll SQL: select count(*) from sahara.job_executions where info like '%"status": "KILLED"%' or info like '%"status": "Error"%';	Alert: Too many errors in EDP may be a sign of deeper problems.
Total number of EDP jobs that are running at the moment	poll SQL: select count(*) from sahara.job_executions where not info like '%"status": "KILLED"%' and not info like '%"status": "Error"%' and not info like '%"status": "SUCCEEDED"%';	diag

Murano

Murano is an OpenStack service which provides a catalog of applications that can be readily deployed in an OpenStack cloud. Murano orchestrates the deployment of those applications automatically using other OpenStack services.

Process checks

Process name	Incoming connections	Role	Dependencies	HA mode
murano-api	HTTP 8082	controller	db, amqp	active/active
murano-engine	RPC	controller	murano-api, heat, neutron, amqp	active/active

In MOS 7.0, Neutron will be an optional dependency. If not found in Keystone, Murano will fall back to use Nova Network.

May be configured to use an additional instance of RabbitMQ. Note that the RabbitMQ instance used by Murano resides on the primary controller node and listens on port 55572 on the public network.

API checks

Check the proper functioning of the API with a read operation. For example:

- list available packages:

  /v1/catalog/packages

- check if the package with Core Murano library is registered:

  /v1/catalog/packages?fqn=io.murano

  This API call should return a JSON object with “packages” property set to a json-array containing at least one object.

Collected Metrics

Metrics	Source	Purpose
API errors	Logs or HAProxy: All HTTP 500 error code.	Alert: When a sudden spike of errors is detected.
Number of environments	Poll SQL: select count(id) from environment;	diag
Number of successful deployments	Poll SQL: select count(id) from session where state='deployed'	diag
Number of running deployments	Poll SQL: select count(id) from session where state='deploying'	diag
Number of deployments which failed to complete or failed to be deleted	Poll SQL: select count(id) from session where state like '%failure'	diag
Number of running deployments which have not been updated for more than 2 hours	Poll SQL: select count(id) from session where state='deploying' and updated < (now() - INTERVAL 2 HOUR)	Alert: Deployment in deploying state which has not been updated for a long period of time likely indicates a job which has hung up and needs some attention.
Total number of application packages in catalog	Poll SQL: select count(id) from package;	diag

Murano RabbitMQ instance

Process checks:

Murano RabbitMQ instance runs as a named instance - murano@localhost.

Process name	Incoming connections	Role	HA mode
beam	TCP 41056, 55572	controller	no

The command below returns the status of the Murano RabbitMQ instance:

```
# rabbitmqctl -n murano@localhost status
```

And the following command returns the pid of the Murano RabbitMQ instance if it exists:

```
# ps axf | grep beam | grep  'murano@localhost' | grep -oP '(?<=(^\s))(\d+)'
```

Libvirt

Libvirt provides a common layer on top of hypervisors or containers like KVM and LXC. Nova uses libvirt to manage instances. The libvirt daemon must be started on all compute nodes, otherwise no instances can be spawned.

Process name	Incoming connections	Role	HA mode
libvirtd	internal RPC protocol, XML format	compute	n/a

Collected Metrics

Collecting statistics about hypervisors can be done either by requesting them directly from libvirt or by using the following Nova API /os-hypervisors/detail and /os-hypervisors/statistics, which is the recommended approach. These metrics are described above in the Nova section.

Furthermore, libvirt provides per instance statistics like CPU, Disk, and Network IO which are further discussed in appendix. These statistics are mainly useful if they are associated with their respective user, tenant, and cloud resource ID. To associate them, it is necessary to either request Nova API and perform the mapping between libvirt instance ID and OpenStack ID (prior to Juno), or more effectively, by using the libvirt instance metadata set by Nova and providing all the necessary information:

```
# virsh edit instance-00000002

<name>instance-00000002</name>
<uuid>01c2d829-e480-4568-92c2-7dc0432a2549</uuid>
<metadata>
  <nova:instance xmlns:nova="http://openstack.org/xmlns/libvirt/nova/1.0"
    <nova:package version="2014.2.2"/>
    <nova:name>z</nova:name>
    <nova:creationTime>2015-06-05 08:52:12</nova:creationTime>
```

```
    <nova:flavor name="m1.micro">
      <nova:memory>64</nova:memory>
      <nova:disk>0</nova:disk>
      <nova:swap>0</nova:swap>
      <nova:ephemeral>0</nova:ephemeral>
      <nova:vcpus>1</nova:vcpus>
    </nova:flavor>
    <nova:owner/>
    <nova:root type="image" uuid="fff4ed5c-1ec6-4263-bc6a-0c5bfb9e9f62"/>
  </nova:instance>
 </metadata>
...
```

Libvirt logs are under the /var/log/libvirt/ directory.

HAProxy

HAproxy is the HTTP load balancer in front of all OpenStack services endpoints and a TCP load balancer for MySQL.

Process checks

Process name	Incoming connections	Role	HA mode
haproxy	All API requests through HTTP port and MySQL requests through TCP port	active controller	active/passive

Performing checks against the HAProxy process through the VIP requires one to know which node is the active (master) controller node in the Corosync/Pacemaker cluster. This is detailed below in the Corosync/Pacemaker HA cluster section.

The active controller handles all the OpenStack services requests through HAProxy which in turn, distributes the load across the API endpoints of the controller cluster.

Collected Metrics

It is critical to monitor the status of the backend from the point of view of HAProxy. A backend is in down state when all the API endpoints behind the load-balancer are failed, and, therefore, should be reported immediately to the operator in an alert.

Metrics	Source	Purpose
backend connections errors	haproxy socket	Alert: When a sudden spike of errors is detected.
number of current sessions	haproxy socket	diag

number of denied requests	haproxy socket	diag Can be useful for security audit
number of denied responses	haproxy socket	diag Can be useful for security audit
bytes in/out	haproxy socket	diag
max queued requests	haproxy socket	diag
number of queued requests	haproxy socket	Alert: Beyond a certain threshold, the number of queued requests can be a symptom of a performance bottleneck in the workflow
queue limit	haproxy socket	diag
request errors	haproxy socket	Alert: When a sudden spike of errors is detected

HAProxy provides a CLI to collect statistics for its frontends and backends. Several statistics are available.

For example, you can use the command below to detect that the backend is in down state. Here, the nova-api stopped responding:

```
echo "show stat" | socat /var/lib/haproxy/stats stdio | grep BACKEND \
| awk -F , '{print $1, $2, $18}' | grep DOWN
nova-api node-10 DOWN
```

As another example, you can use the command below to get a list of the API endpoints with their respective status:

```
echo "show stat" | socat /var/lib/haproxy/stats stdio | grep BACKEND \
| awk -F , '{print $1, $2, $18}'
horizon BACKEND UP
keystone-1 BACKEND UP
keystone-2 BACKEND UP
nova-api-1 BACKEND DOWN
nova-api-2 BACKEND UP
nova-metadata-api BACKEND UP
cinder-api BACKEND UP
glance-api BACKEND UP
neutron BACKEND UP
glance-registry BACKEND UP
mysql BACKEND UP
swift BACKEND UP
heat-api BACKEND UP
heat-api-cfn BACKEND UP
heat-api-cloudwatch BACKEND UP
nova-novncproxy BACKEND UP
```

With the command below, you can see that the glance-api endpoints on node-7 and node-10 are down while the backend is still up. However, the HA status of the Glance service as a whole is no longer ensured and therefore should be reported immediately to the operator in an alert.

```
echo "show stat" | socat /var/lib/haproxy/stats stdio | awk -F , \
'{print $1, $2, $18}' | grep glance-api

glance-api FRONTEND OPEN
glance-api node-6 UP
glance-api node-7 DOWN
glance-api node-10 DOWN
glance-api BACKEND UP
```

Finally, the command below can be used to collect all the HAProxy statistics:

```
echo "show stat" | socat /var/lib/haproxy/stats stdio
```

RabbitMQ

All OpenStack services depend on RabbitMQ message queues to communicate and distribute the workload across workers. Therefore, it is critical to monitor the healthiness of this component to ensure there are no communication issues or performance bottlenecks especially between the API endpoints and the workers. Furthermore, in order to appraise correctly the availability status of the message queues you need to take into account that RabbitMQ operates in a cluster of highly available queues.

Process checks

RabbitMQ is composed of two processes which run in pairs located on each controller node of the HA cluster.

Process name	Incoming connections	Role	HA mode
epmd beam	TCP 4369 TCP 41055, 5673 HTTP 15672 (management port used to monitor servers)	controller	active/active

> Note
>
> In order to enable the monitoring of RabbitMQ, the management plugin must be installed to expose RabbitMQ's management Rest API. As for the service checks, a dedicated user should be used to query the Rest API or use rabbitmqctl command line. For example, the following command returns the status of the cluster:

```
rabbitmqctl cluster_status
```

RabbitMQ Cluster status

Check	Source	Purpose
Unmirror queues	API management: In response from resource /queues, check for each queue with x-ha-policy arguments that synchronised_slave_nodes is more than 0.	Alert: Slaves are not synchronized.
Missing nodes in cluster	API management: Check the running status for each node, resource /nodes.	Alert: One or more nodes are not being viewed as running. This should not happen unless they are in maintenance.
Number of queues without consumer	API management: The number of consumers is directly accessible within the response from resources /queues/<name>.	Alert: Queues without consumers should not happen. This could be a symptom of a resource leak situation.

Collected Metrics

Metric	Source	Purpose
Total number of nodes in cluster	API management	diag
Number of connections	API management	diag
Number of consumers	API management	Alert: Zero consumers should never happen. Something is probably deeply broken.
Number of exchanges	API management	Alert: Zero exchanges should never happen. Something is probably deeply broken.
Number of queues	API management	Alert: Zero queues should never happen. Something is probably deeply broken.

Metrics per queue

Metric	Source	Purpose

Number of ready messages	API management	Alert: When the value is beyond standard deviation or top percentiles threshold depending on the data-points distribution. This could be the symptom of a resource congestion situation.
Number of consumers	API management	diag
Number of published messages	API management	diag
Number of delivered messages	API management	diag
Number of acked messages	API management	diag
Number of memory used	API management	diag
Errors	/var/log/rabbitmq/*.log	Alert: When a sudden spike of errors is detected.

MySQL

The MySQL database running on the OpenStack controller nodes is a central component because it is used by almost all the OpenStack components as their primary data persistence storage. Therefore, it is critical to monitor the healthiness of this component on each of the controller nodes in the cluster.

Process checks

Process name	Incoming connections	Role	Dependencies	HA mode
mysqld	TCP 3306	controller	storage	active/passive

In addition to checking the existence of the process, it is necessary to check the availability status of the MySQL database. This can be verified using the command:

```
# mysqladmin ping
```

Collected Metrics

Metrics	Source	Purpose
bytes received (bytes/sec)	poll SQL	diag
bytes sent (bytes/sec)	poll SQL	diag
begin operations	poll SQL	diag
commit operations	poll SQL	diag
delete operations	poll SQL	diag

insert operations	poll SQL	diag
rollback operations	poll SQL	diag
select operations	poll SQL	Alert: The absence of select operations could indicate that the connection to the database is broken unless the MySQL server is in maintenance.
update operations	poll SQL	diag
number of queries	poll SQL	diag
slow queries	poll SQL	diag
Database physical size (Mbyte)	poll SQL: SELECT table_schema "database", sum(data_length + index_length) / 1024 / 1024 "size_mb" FROM information_schema.TABLES GROUP BY table_schema order by 2 desc;	diag

These poll SQL metrics can be collected using the following SQL command:

```
SHOW GLOBAL STATUS WHERE Variable_name=<NAME>;
```

See MySQL server status variables for details.

> Note
>
> You should pay attention to MySQL logs to detect slow queries for diagnostic purposes. You can activate slow queries log with the following configuration parameters: slow_query_log=1, long_query_time=5, and slow_query_log_file=<filename>.

Metrics related to the MySQL cluster

The high availability of the MySQL database is supported in active/passive mode with one master and several slave nodes. To ensure that the MySQL cluster remains highly available, you should continuously monitor that slave nodes are ready to take over in case of a master node failure.

Metrics	Source	Purpose
wsrep_ready	SQL possible values: ON/OFF	Alert: node not ready if OFF
wsrep_cluster_size	SQL: number of nodes	diag

wsrep_replicated_bytes	SQL: bytes sent to other nodes	diag
wsrep_received_bytes	SQL: bytes received from other nodes	diag
wsrep_cluster_status	SQL: Primary/Non-Primary/Disconnected	Alert: A node is disconnected from the cluster.
wsrep_local_commits	SQL number of commit	diag
errors	/var/log/mysqld.log	Alert: When a sudden spike of errors is detected.

These metrics can be collected with the SQL command:

```
SHOW STATUS WHERE Variable_name REGEXP 'wsrep.*';
```

Memcached

Memcache is an in-memory storage server. It is mainly used by Keystone to store tokens. The availability of memcache is therefore critical to ensure that the authentication requests performed by the OpenStack services can be satisfied.

The consoleauth Nova service also uses memcache to share authorization tokens and to ensure the high availability of the service.

Process checks

Memcached process checks should be performed for each controller node.

Process name	Incoming connections	Role	HA mode
memcached	TCP port 11211	controller	active/active

Memcache statistics per server can be collected with the command:

```
echo -e "<command>\nquit" | nc 127.0.0.1 11211

# where command is one of “stats” or “stats items”
```

Refer to the memcached documentation for the complete list of stats available. Below is a selected list of metrics:

Metrics	Source	Purpose
curr_item	command stats: number of current items	diag
total_item	command stats items: total number of items	diag

cmd_get	command stats: number of get	diag
cmd_set	command stats: number of set	diag
get_hits	command stats: number of hits	diag
get_misses	command stats: number of get misses	Alert: high number of misses indicates a misconfiguration somewhere (TTL too short or memory starvation)
curr_connections	command stats: number of current connections	diag
total_connections	command stats: counter total of connections	diag
evictions	command stats: number of valid items removed from cache to free memory for new items	Alert: should never happen. Requires to increase the memory size
bytes_read	command stats: bytes read from cache	diag
bytes_written	command stats: bytes write in cache	diag
limit_maxbytes	command stats: max bytes to use for storage	diag
threads	command stats: number of threads	diag
conn_yields	command stats: connection yield	diag when > 0 consider increasing the connection limit
maxbytes	command stats: maximum memory bytes to use	diag
maxconns	command stats: maximum connection	diag
evicted	command stats: counter of evicted items	diag
outofmemory	command stats: number of times the server fails to store a new item due to a lack of memory available	Alert: Unable to store item should never happen
errors	/var/log/memcached.log	Alert: When a sudden spike of errors is detected

HA cluster

Corosync/Pacemaker

The Mirantis OpenStack HA cluster for the controller nodes is implemented using Pacemaker and Corosync. Corosync provides messaging and membership services while Pacemaker performs resources management. Resources management includes detecting and recovering the nodes and resources under its control from a failure.

For Corosync you should perform regular checks to verify that the ring is in the active with no faults state as shown below:

```
# corosync-cfgtool -s
Printing ring status.
Local node ID 33597632
RING ID 0
id= 192.168.0.2
status= ring 0 active with no faults
```

Pacemaker is responsible for handling the failover of some of OpenStack services in the HA cluster in case of a hardware or software failure. To achieve this goal, Pacemaker monitors the state of the resources under its control (every 30 seconds by default) to return a health status for each of those resources.

Free space monitoring

Pacemaker provides a free space monitoring alarm that enables you to plan storage capacity expansion and prevent the out-of-memory errors on the controller node.

> Note
>
> The Pacemaker free space monitoring alarm does not replace a fully-fledged monitoring solution and only provides basic notification capabilities.

When a controller node runs out of disk space, all services managed by Pacemaker on that node stop and the node becomes unavailable. If you have a highly-available configuration, the cloud continues to operate using remaining controller nodes without noticeable data plane downtime.

Pacemaker automatically attempts to stop the services before the disks are completelly full. However, if all controller nodes fill their disks, then Pacemaker performs a graceful shutdown of the controller nodes which results in the control plane downtime, as well as inavailability of virtual instances. The shutdwon controller nodes will come back online automatically after the cloud administrator addresses the disk issues. However, the cloud administrator must manually clean the alarms on the affected controller nodes using the following command: # crm node status-attr <hostname> delete "#health_disk".

State verification

With Pacemaker, you should perform regular checks to verify that its resources are in the started state on at least one node when HA is handled in active/passive mode, and that at least one resource is in the started state on each of the cluster nodes when HA is handled in active/active mode. You can make that verification with the crm status command as shown below:

```
# crm status
```

Example of the output for the above command:

```
============
Last updated: Tue Jun 23 08:47:23 2015
Last change: Mon Jun 22 17:24:32 2015
Stack: corosync
Current DC: node-1.domain.tld (1) - partition with quorum
Version: 1.1.12-561c4cf
3 Nodes configured
43 Resources configured
============

Online: [ node-1.domain.tld node-2.domain.tld node-3.domain.tld ]

Clone Set: clone_p_vrouter [p_vrouter]
    Started: [ node-1.domain.tld node-2.domain.tld node-3.domain.tld ]
vip__management (ocf::fuel:ns_IPaddr2): Started node-1.domain.tld
vip__public_vrouter (ocf::fuel:ns_IPaddr2): Started node-1.domain.tld
vip__management_vrouter (ocf::fuel:ns_IPaddr2): Started node-1.domain.tld
vip__public (ocf::fuel:ns_IPaddr2): Started node-2.domain.tld
Master/Slave Set: master_p_conntrackd [p_conntrackd]
    Masters: [ node-1.domain.tld ]
    Slaves: [ node-2.domain.tld node-3.domain.tld ]
Clone Set: clone_p_haproxy [p_haproxy]
    Started: [ node-1.domain.tld node-2.domain.tld node-3.domain.tld ]
Clone Set: clone_p_dns [p_dns]
    Started: [ node-1.domain.tld node-2.domain.tld node-3.domain.tld ]
Clone Set: clone_p_mysql [p_mysql]
    Started: [ node-1.domain.tld node-2.domain.tld node-3.domain.tld ]
Master/Slave Set: master_p_rabbitmq-server [p_rabbitmq-server]
    Masters: [ node-1.domain.tld ]
    Slaves: [ node-2.domain.tld node-3.domain.tld ]
Clone Set: clone_p_heat-engine [p_heat-engine]
    Started: [ node-1.domain.tld node-2.domain.tld node-3.domain.tld ]
Clone Set: clone_p_neutron-plugin-openvswitch-agent [p_neutron-plugin-openvswitch-agent]
    Started: [ node-1.domain.tld node-2.domain.tld node-3.domain.tld ]
Clone Set: clone_p_neutron-dhcp-agent [p_neutron-dhcp-agent]
    Started: [ node-1.domain.tld node-2.domain.tld node-3.domain.tld ]
Clone Set: clone_p_neutron-metadata-agent [p_neutron-metadata-agent]
    Started: [ node-1.domain.tld node-2.domain.tld node-3.domain.tld ]
```

```
Clone Set: clone_p_neutron-l3-agent [p_neutron-l3-agent]
    Started: [ node-1.domain.tld node-2.domain.tld node-3.domain.tld ]
Clone Set: clone_p_ntp [p_ntp]
    Started: [ node-1.domain.tld node-2.domain.tld node-3.domain.tld ]
Clone Set: clone_ping_vip__public [ping_vip__public]
    Started: [ node-1.domain.tld node-2.domain.tld node-3.domain.tld ]
```

Here, the crm status command provides an easy method to inform the monitoring system that the HA cluster is comprised of three controller nodes (node-1, node-2, and node-3), and that node-1 is the actual master node. It also tells the monitoring system that the VIPs for the public and management interfaces are started on the master node and that the HAProxy is started on all the nodes of the HA cluster.

The crm_resource command can also be used to verify on which node a particular resource is active. Execution of the command below, for example, tells the monitoring system that the Neutron DHCP agent is active on node-1:

```
# crm_resource  --locate --quiet --resource p_neutron-dhcp-agent
node-1
```

Use the --resource vip_public option to find out on which node the public VIP is active:

```
# crm_resource  --locate --quiet --resource vip__public
node-2
```

Storage clusters

Ceph

Ceph is a unified and distributed storage system that can be used as a storage backend for Cinder volumes and Glance images.

> Note
>
> Ceph Filesystem monitoring is not covered in this document since its capabilities are not natively supported by Mirantis OpenStack. Mirantis OpenStack uses a single Ceph cluster named ceph.

Process Checks

Process name	Incoming connections	Role	Dependencies	HA mode
ceph-mon	n/a	controller		active/active
ceph-osd	n/a	storage		data replication mechanisms
apache/httpd	HTTP 6780	controller	apache mod_fastcgi	active/active

Collected Metrics

Metrics	Details	Purpose
cluster health	command: ceph health	alert: When the status is something else than HEALTH_OK
cluster total space available (Mbyte)	command: ceph df	diag
cluster space used (Mbyte)	command: ceph df	diag
cluster free space	calculated from previous metrics	alert: Not enough disk space before more storage capacity can be physically provisioned
total number of monitor	command: ceph mon dump	diag
number of monitor in Quorum	command: ceph mon dump	diag

number of OSD daemons per state	command: ceph osd dump, where states are UP or DOWN and IN or OUT	diag
rate Kbytes read/write (per pool)	command: ceph osd pool	diag
operation/second (per pool)	command: ceph osd pool	diag
number of object (per pool)	command: ceph osd pool	diag
total number of Placement Groups (PG) per status	command: ceph pg dump	diag
filesystem commit latency (per OSD daemon)	command: ceph pg dump	diag
filesystem apply latency (per OSD daemon)	command: ceph pg dump	diag
KByte used (per OSD daemon)	command: ceph pg dump	diag
cluster write latency	command: rados bench	diag

Ceph Cluster Health Checks

ceph health

The monitoring system should perform regular checks to verify that the Ceph cluster is healthy. This can be achieved using the ceph health command:

```
# ceph health
HEALTH_OK
```

Anything else than HEALTH_OK should be reported in an alert like the following:

```
HEALTH_WARN clock skew detected on mon.node-16, mon.node-17
```

ceph df

```
# ceph df
GLOBAL:
  SIZE     AVAIL     RAW USED     %RAW USED
  380G      368G       12560M         3.22
POOLS:
  NAME         ID     USED      %USED     MAX AVAIL     OBJECTS
  data         0         0         0         184G           0
  metadata     1         0         0         184G           0
  rbd          2         0         0         184G           0
```

```
 images      3     13696k        0        184G         5
 volumes     4          0        0        184G         0
 compute     5          0        0        184G         0
```

ceph mon dump

```
# ceph mon dump --format json
{
   "created": "0.000000",
   "epoch": 3,
   "fsid": "bbec22eb-b852-4f6f-89f8-9d7fcceb062a",
   "modified": "2015-03-19 14:41:32.374329",
   "mons": [
      {
         "addr": "192.168.0.3:6789/0",
         "name": "node-28",
         "rank": 0
      },
      {
         "addr": "192.168.0.4:6789/0",
         "name": "node-29",
         "rank": 1
      },
      {
         "addr": "192.168.0.5:6789/0",
         "name": "node-30",
         "rank": 2
      }
   ],
   "quorum": [
      0,
      1,
      2
   ]
}
```

ceph osd dump

```
# ceph osd dump (output truncated)
...
osd.0 up   in  weight 1 up_from 7 up_thru 23 down_at 0   ...
osd.1 up   in  weight 1 up_from 10 up_thru 23 down_at 0   ...
osd.2 up   in  weight 1 up_from 15 up_thru 23 down_at 0   ...
osd.3 up   in  weight 1 up_from 18 up_thru 23 down_at 0   ...
osd.4 up   in  weight 1 up_from 23 up_thru 23 down_at 0   ...
osd.5 up   in  weight 1 up_from 23 up_thru 23 down_at 0   ...
```

ceph osd pool

```
# ceph osd pool stats -f json
[
   {
      "client_io_rate": {
         "op_per_sec": 1,
         "read_bytes_sec": 242,
         "write_bytes_sec": 2982616
      },
      "pool_id": 4,
      "pool_name": "volumes",
      "recovery": {},
      "recovery_rate": {}
   },
...
```

ceph pg dump

```
# ceph pg dump -f json (output truncated)
{
   "full_ratio": "0.950000",
   "last_osdmap_epoch": 25,
   "last_pg_scan": 4,
   "near_full_ratio": "0.850000",
   "osd_stats": [
      {
         "fs_perf_stat": {
            "apply_latency_ms": 3,
            "commit_latency_ms": 2
         },
         "hb_in": [
            1,
            2,
            3,
            4,
            5
         ],
         "hb_out": [],
         "kb": 66497820,
         "kb_avail": 64344180,
         "kb_used": 2153640,
         "num_snap_trimming": 0,
         "op_queue_age_hist": {
            "histogram": [],
            "upper_bound": 1
         },
         "osd": 0,
         "snap_trim_queue_len": 0
```

```
    },
...
```

rados bench

The write latency can be obtained with the rados command. It writes objects in different pools. You should keep the frequency of these checks lightweight to avoid overwhelming the cluster.

For example:

```
rados -p data  bench 5 write -t 2  --run-name monit_perf
# where command options are:
# -p data: use the pool named ‘data’
# bench: the rados command ‘bench’
# 5 : run the bench for 5 seconds
# write: perform write operations
# -t 2 : number of concurrent thread
```

> Note
>
> Another way to collect metrics related to OSD daemons is to grab from each node the OSD daemon's socket. This command retrieves all metrics available, but the output is really verbose and not all metrics are useful to monitor:
>
> echo '{"prefix": "perf dump"}\0' | socat /var/run/\ ceph/<cluster>-osd.<ID>.asok stdio

Swift

Swift is the OpenStack project, which provides highly available, distributed and eventually consistent storage services for objects. It is used by default as a storage backend to store Glance images.

Process Checks

Process name	Incoming connection	Role	HA mode
swift-proxy-server	HTTP 8080	controller	active/active
swift-object-replicator	n/a	controller	active/active
swift-object-server	HTTP 6000	controller	active/active
swift-container-server	HTTP 6001	controller	active/active
swift-container-replicator	n/a	controller	active/active
swift-account-server	HTTP 6002	controller	active/active

swift-account-replicator	n/a	controller	active/active

API Checks

Check the availability of the service through synthetic HTTP transactions against the Swift API endpoint:

- create a container
- upload a small (few kilobytes) object
- delete the container and object

Collected Metrics

The Swift project is made to collect metrics natively. Indeed, it is the only OpenStack project that is natively instrumented to send metrics to statsd or any statsd enabled data acquisition service like Heka. It provides real-time operational data about the object storage cluster activity and errors across all components. Please check the documentation for further information about how to enable statsd metrics in the Swift developer documentation.

Hardware and system monitoring

An effective monitoring solution for OpenStack should also check the host operating system and the underlying infrastructure on top of which your cloud is running.

As we stated earlier in this document, we are not going to address the monitoring of the network and storage equipments, because these topics are too broad and vendor specific to be addressed here. However, we recommend that you perform some amount of server monitoring to anticipate hardware failures using IPMI since it is a relatively standard interface that is supported by most hardware vendors.

IPMI

IPMI is a standard driven by Intel that has been widely adopted by the server manufacturers. It provides hardware Sensors Data Records to collect information such as:

- Components temperature
- Fan rotation
- Components voltage
- Power supply status (redundancy check)
- Power status (on or off)

The IPMI System Event Log provides a timed journal of all events that occurred in the server. Each threshold crossing of previous sensors is logged with a severity level that can be one of the following: recoverable, non-critical, critical, unrecoverable. See the IPMI specifications for further details.

Other events can also be logged, such as:

- Memory Error-Correcting-Code memory (ECC) detection that can be reported in an alert if they happen too often.
- Chassis intrusion detections that can be reported in an alert for security reasons.

Most Linux distributions provide the ipmitool package that allows to interact with the IPMI interface.

Retrieve the SDR records for voltage by running:

```
# /usr/bin/ipmitool -I lan -L operator -U root -H <ip> \
-P <password> sdr type "Voltage" list

VTT             | 30h | ok  |  7.10 | 0.99 Volts
CPU1 Vcore      | 21h | ok  |  3.3 | 0.83 Volts
CPU2 Vcore      | 22h | ns  |  3.4 | Disabled
VDIMM AB        | 61h | ok  | 32.1 | 1.49 Volts
VDIMM CD        | 62h | ok  | 32.2 | 1.49 Volts
VDIMM EF        | 63h | ns  | 32.3 | Disabled
VDIMM GH        | 64h | ns  | 32.4 | Disabled
+1.1 V          | 31h | ok  |  7.11 | 1.09 Volts
+1.5 V          | 32h | ok  |  7.12 | 1.47 Volts
```

```
3.3V          | 33h | ok  |  7.13 | 3.26 Volts
+3.3VSB       | 34h | ok  |  7.14 | 3.36 Volts
5V            | 35h | ok  |  7.15 | 5.06 Volts
+5VSB         | 36h | ok  |  7.16 | 5.06 Volts
12V           | 37h | ok  |  7.17 | 12.30 Volt
VBAT          | 38h | ok  |  7.18 | 3.22 Volts
```

To get the system events logs, run:

```
# /usr/bin/ipmitool -I lan -L operator -U root -H <ip> -P <password> sel list

17 | 01/27/2015 | 11:31:21 | OS Boot | C: boot completed | Asserted
18 | 01/27/2015 | 11:41:08 | Memory | Correctable ECC | Asserted | CPU 0 DIMM 8
19 | 01/27/2015 | 12:07:14 | Physical Security #0x51 \
| General Chassis intrusion | Asserted
1a | 01/27/2015 | 17:37:46 | Memory | Correctable ECC | Asserted | CPU 0 DIMM 8
1b | 01/28/2015 | 06:27:27 | Memory | Correctable ECC | Asserted | CPU 0 DIMM 8
1c | 01/28/2015 | 12:03:13 | Memory | Correctable ECC | Asserted | CPU 0 DIMM 8
1d | 01/28/2015 | 17:39:00 | Memory | Correctable ECC | Asserted | CPU 0 DIMM 8
1e | 01/28/2015 | 23:14:46 | Memory | Correctable ECC | Asserted | CPU 0 DIMM 8
1f | 01/29/2015 | 04:50:33 | Memory | Correctable ECC | Asserted | CPU 0 DIMM 8
20 | 01/29/2015 | 10:26:19 | Memory | Correctable ECC | Asserted | CPU 0 DIMM 8
3e | 02/01/2015 | 17:14:54 | VBAT   | 38h | lcr |  7.18 | 2.54 Volts
```

Get the power status:

```
# /usr/bin/ipmitool -I lan -L operator -U root -H <ip> \
-P <password> power status
Chassis Power is on
```

> Note
>
> When coupled with the Ironic project, Ceilometer has the ability to collect IPMI sensors.

Disks monitoring

The disks are often the primary cause of server failures. You should monitor the disks of your servers to detect and whenever possible anticipate the occurrence of those failures. First signs of problems with your disks are generally reported by the kernel. Usually, disks errors are logged in /var/log/kern.log or /var/log/messages, but the location may differ depending on your Linux distribution.

A recommended approach is to watch your system logs for bad drives using programs like Logstash or Heka or even dedicated tools like swift-drive-audit that can be run using Cron.

Another approach to check the status of your disks is to rely on the S.M.A.R.T interface when it is supported although some differences may be found between Hard Disk Drives (HDD) and Solid State Disks (SSD) devices.

Many attributes/counters are available through the S.M.A.R.T interface but your mileage may vary depending on the disk manufacturer.

It is hard to anticipate disk failures in a deterministic way. The handling of disk failures is generally addressed on a case-by-case basis using S.M.A.R.T attributes that can be indicative of a dysfunctioning and as such conductive of future problems.

For HDD:

- Uncorrectable sector/event count
- Reallocated sector/event count
- Spin retry count
- Temperature: should usually be lower than 50°C

And for SDD:

- Media Wearout Indicator: indicator of the cells health, where 0 is the worst value. When the value reaches around 20 you should consider changing the disk.
- Temperature: should usually be lower than 50°C

Linux distributions provide the smartmontools package to play with the S.M.A.R.T interface.

To read the attributes of your sda disk run:

```
# smartctl -a /dev/sda
```

The smartctl command displays a health status or pending alerts when used with the -H option.

```
# smartctl -H /dev/sda
=== START OF READ SMART DATA SECTION ===
SMART overall-health self-assessment test result: PASSED
```

Operating system monitoring

Getting access to an operating system's health status and key metrics is largely supported by all Linux distributions through a variety of tools and monitoring applications including Nagios, Zabbix, Collectd, Diamond, Ganglia, Sensu, and others.

Below is the list of key metrics that we think are critical to collect in the context of OpenStack monitoring. Whenever possible we try to provide alerting criteria that should be applied to these metrics, but they largely depend on the node's role (workload characterisation) and hardware characteristics of the servers.

Host monitoring

Metrics	Unit	Purpose

node uptime check	second	diag
OS version	string	diag
kernel version	string	diag
host is alive (simple ping)	bool	alert: When host is down

Disk usage monitoring

Metrics	Unit	Purpose
read	bytes/sec	diag
write	bytes/sec	diag
operation read	operation/sec	diag
read time	millisecond	diag
write time	millisecond	diag

Soft RAID monitoring

Checks	Alert criteria
pool state	missing member
synchronization	synchronization running

Filesystem usage monitoring

Metrics	Purpose
free space	Alert: static thresholds like <10% or <5% free space in the file system can generate false positives. It is instead recommended to set a smarter alarm that is based on the trend observed from historical data so that you are alerted only when it is projected that the file system becomes full within the next 24 hours for example.
used space	diag
free inodes	Alert: Below 10% free inodes may indicate too many small files or zero sized files on disk. An exhaustion of inodes raises the error no space left regardless of whether there is still plenty of free space on the file system or not.
used inodes	diag

CPU usage monitoring

Metric	Purpose

% CPU user	diag
% CPU system	diag
% CPU wait	Alert: Above 10% of CPU wait could be suspicious depending on the node’s role which calls for further investigations. For example, a CPU wait above around 10% on the compute nodes is probably not a desirable situation because it means that the local disk(s) are a performance bottleneck for the hypervisor.
% CPU idle	Alert: Below 20% CPU idle could be an issue depending on the node’s role which calls for further investigations. For example, below 20% CPU idle on controller nodes is probably not a desirable situation because it means that the cloud management system is overloaded. However, below 20% CPU idle on the compute nodes may be considered as normal and even expected depending on the operator’s overcommitment policies.
system load (5, 10, 15)	diag
context switches	diag: It is important to take a closer look at the rate of context switches. A rate that is too high should be interpreted as an anomaly that may result from having too many processes running on a node or from running poorly parallelized applications that are too heavily competing for shared resources.

RAM usage monitoring

Setting alarms for RAM usage is not necessarily appropriate, because it could generate false positives. This is due to the fact that some applications allocate more memory than they currently need, for example, to support caches. And so, in order to correctly identify a condition of memory shortage in an alarm, you would have to take into account how the applications actually use the memory, which is not really possible in practice.

Metrics	Unit	Purpose
free	megabytes	diag
used	megabytes	diag
cached	megabytes	diag
buffered	megabytes	diag

Swap usage monitoring

Same thing for the swap usage. Swap usage may be an indication of a memory shortage situation when you observe a steady increase of swap space usage over a relatively long period of time. But, probably, not in terms of usage percentage, because files may stay in swap for a long period of time without any further access to them.

Metrics	Unit	Purpose
free	megabytes	diag
used	megabytes	diag
cached	megabytes	diag
io in/out	megabytes	diag

Process statistics monitoring

It is generally a good idea to collect process statics to be stored in a time-series database for trend analysis and anomaly detection using statistical models.

Metrics	Purpose
number of processes running	diag
number of processes paging	diag
number of processes blocked	diag
number of processes sleeping	diag
number of processes zombies	diag
number of processes stopped	diag
fork rate megabytes	diag

More fine-grained statistics could be collected for key processes like those supporting the OpenStack services:

Metrics	Purpose
number of threads	diag
memory usage (Mbytes)	diag
cpu usage (user/system)	diag

Network Interface Card (NIC) monitoring

Collected metrics

Metrics	Purpose
bandwidth	Alert: When bandwidth is consumed steadily approaching the nominal bandwidth of the network link.
errors	Alert: When errors rate is too high.

Status checks

Checks	Purpose
link status	diag
bonding status	diag

Note

Bonding can be achieved with Linux bonding or Open vSwitch.

Linux bonding status information is found in /sys/class/net/<bondX>/operate. See as an example how Nagios performs linux bonding checks.

Open vSwitch bonding status information is displayed with the ovs-appctl bond/show.

Firewall (iptables) monitoring

Checks	Source	Purpose
status	iptables -L Alert:	When firewall is not enabled

It is generally a good idea to collect firewall metrics for diagnostic. The iptstate command allows the number of connections and TCP sessions metrics collection.

Metrics	Purpose
dropped packets	diag
number of connection TCP	diag
number of connection UDP	diag
number of connection ICMP	diag
number of TCP sessions SYN	diag
number of TCP sessions TIME_WAIT	diag
number of TCP sessions ESTABLISHED	diag
number of TCP sessions CLOSE	diag

Appendix

We put additional materials in this appendix that are out of the scope but which could, nonetheless, be of interest.

Virtual machine monitoring

It is possible to collect guests statistics from libvirt, see libvirt-domain for details.

1. Block IO
    - read_reqs
    - read_bytes
    - write_reqs
    - write_bytes
2. Network IO
    - rx_bytes
    - rx_packets
    - rx_errors
    - rx_drops
    - tx_bytes
    - tx_packets
    - tx_errors
    - tx_drops
3. CPU
    - cputime
    - vcputime
    - systemtime
    - usertime

> Note
>
> The VCPU time is global and cumulative and is reported in nanoseconds since the last boot. To calculate a VCPU usage percentage you need to divide vcputime by the number of VCPUS divided by the wallclock time of the sampling interval.

Guest agent

A guest agent allows running scripts or applications inside an instance while it runs. Unfortunately, there is no support of the guest agent with KVM hypervisor at the moment, only XEN driver supports it.

VM network traffic

The traffic across VMs can be monitored from the virtual switches by enabling monitoring sampling with sFlow on each Open vSwitch server.

The basic principle is to sample network traffic and send all the samples to the sFlow collector for analysis. Known open source software that supports sFlow include pmacct, Ganglia, and Ntop.

NetFlow is a commercial standard embedded in many physical devices with the main difference that it does not sampling network traffic, which is more resource-intensive than sFlow but also more accurate.