OpenStack Monitoring
As private cloud deployments become more and more widespread, the need for a thorough and efficient method of monitoring has become increasingly important. The main features of data center monitoring include:

Understanding the current state of an environment, with the ability to act in near real-time

Monitoring and metering resource usage and utilization, which enables capacity planning

Attaining transparency in order to analyze historical events for an optimized IT environment

These features hold true when monitoring any IT environment. However, with the private cloud, new methods and tools have been introduced. In this article, we’ll provide a brief overview of what the OpenStack private cloud has to offer in terms of monitoring approaches and tools, then link you to external resources that can help you take the next step. Lastly, we’ll provide details of how to utilize ELK to conduct basic troubleshooting of your OpenStack cloud.

OpenStack Monitoring Projects

Three major projects have been created to monitor the OpenStack private cloud:

Ceilometer

This telemetry project within OpenStack provides a framework to meter and collect infrastructure metrics such as CPU, network, and storage utilization. This project aims to provide the infrastructure needed to collect measurements within OpenStack, such as physical and virtual resource utilization. The measurements are then transformed into meters that can be retrieved by external applications via APIs.

Ceilometer can also process events that are generated based on objects in an OpenStack system, such as a newly created instance image. Ideally, events would be created from notifications that are sent by different OpenStack components via the notifications system. In practice, this is not supported by most components and Ceilometer had to add polling agents for APIs or a local hypervisor to collect information at a regular interval.

Additionally, Ceilometer provides alarms that are set when a metric crosses a threshold triggering an action such as sending the alarm information to an external server. The existing Alarm Evaluator
periodically queries/polls the databases in order to check the changes in alarm states independently of other OpenStack components. This is appropriate for evaluating an alarm related to meters stored within a certain period. However, this is not efficient when evaluating an alarm for events that are only emitted by other OpenStack servers every once in a while. As a result, event driven alarms need to be added to OpenStack.

**Monasca**

Defined as “monitoring as a service,” Monasca is a large framework that incorporates all aspects of monitoring, including alarms, statistics, measurements and more, for all OpenStack components. Monasca largely focuses on allowing tenants to define what should be measured, what statistics should be collected, what should trigger an alarm, and the best notification method. The relationship between Monasca and Ceilometer is currently being defined.

**Heat**

As the OpenStack orchestration project, Heat enables you to define templates, including template resources, and implement them in the cloud. A resource can be a server, a block storage unit, a network and so on. It works by defining the number of resources that you need and how they are related. Heat provides an Event API through which you can get events pertaining to orchestration operations (such as start, stop and terminate). Ceilometer provides Heat with alarms that can trigger it to perform actions such as autoscaling.
Missing Pieces

Data Correlation

The information above is good, but it’s not enough. As an administrator, you need to know more about how components influence one another. So when one failure triggers multiple events, you want to be able to find the root cause, and fast. For example, when your system tries to start a new VM and there are no available CPU resources in the cloud, you might get an event from Heat indicating that a VM failed to start, and get an alarm from Ceilometer indicating that your CPU utilization is over a certain threshold.

You then need to quickly make the connection between the two in order to understand what happened, fix the problem and prevent it from happening again in the future. This is usually done with external applications, such as Nagios and Zabbix, that process raw data that is collected from alarms and events.
Log Collection

Different components of the cloud need to be able to generate meaningful logs to be processed, stored and accessed by an administrator for further analysis. Ideally, logs should also be provided from the infrastructure itself, however there is currently no standard API to collect logs from OpenStack.

Physical Infrastructure Awareness

OpenStack is layered over the physical components of your cloud. This is where the cloud’s operating system comes into play. You need to know that a disk has failed in order to understand that it is the cause of your storage over-utilization alarm, and therefore the decrease in your storage capacity. An alarm, in and of itself, is not enough.

Unlike with the AWS public cloud, when running a private cloud, your IT team is responsible for the physical infrastructure. It’s important to have a connection to your actual physical infrastructure and to receive alarms/events regarding the physical aspects of your cloud. In a private cloud deployment, the interaction between the cloud’s operating system and the physical layer needs to be an integral part of the monitoring process.

A smart cloud solution will not only enable you to automatically plug-in new resources, but will also provide self-healing mechanisms that allow the administrator more time to react to any underlying physical infrastructure issues.

Monitoring is a key building block toward achieving this goal. This article is just the tip of the OpenStack monitoring iceberg. Monitoring any IT system is not a simple task and in the highly dynamic world of the cloud, it is even more complex. In addition to the components and tools mentioned above, there are third-party open source tools such as the ELK stack (Elasticsearch, Logstash and Kibana) as well as multiple modern commercial ones such
A variety of infrastructure and application monitoring and telemetry solutions are available; today we are going to focus on a particular solution. Sensu is an open-source monitoring framework that is designed to help solve monitoring challenges in cloud environments. It is not a SaaS solution, but is instead installed at the infrastructure level monitoring servers, services, application health and also providing custom analysis metrics.

Sensu relies on number of open source components and is quite simple to install. The documentation is quite extensive and easy to follow (Install Sensu in 5 minutes). Once Sensu is installed you can manage and view your alerts with another open source product. Uchiwa aggregates the results of one or multiple Sensu API’s and displays these in a real-time dashboard, with customizable filters.

Sensu has a modern design, relying on local agents to run checks and push results to an AMQP broker. You can write your checks in almost any programming language you like, Sensu is very flexible. A number of enterprises are using Sensu at an extremely large scale.

To assure that your environment is healthy, we recommend performing monitoring at three different layers which could be compared to peeling the layers off an onion. The three layers are covered in each of the following sections:

- API Availability
- Basic API Flows
- Full Stack Development
API Availability

Each and every service within OpenStack has an API endpoint. The purpose of these endpoints are to provide an entry point into the service behind it. It is very easy to see what the endpoints are, provided you have provided the appropriate credentials beforehand all you have to do is run the following command:

```
keystone endpoint-list: the output will be a list of your endpoints, for this post we will choose one – OpenStack Keystone. You will find a public URL similar to this example, but the IP address will be appropriate for your own environment:
http://192.168.16.100:5000/v2.0
```

The first level of monitoring checks to see if the port is actually available. For an example of a Sensu check for Keystone port availability here’s the link on github.

The script above polls the API listening on port 5000 for Keystone and will return a status code based on the result of the request.
But this does not mean that Keystone is actually functioning properly – it just means that the API is up. Which brings us to our next layer.

Basic API Flows

Keystone provides Identity, Token, Catalog and Policy services for use specifically by projects in the OpenStack family. What better way to test if Keystone is functioning in proper manner than by actually asking it to what it is supposed to do – authenticate and supply a token. Here’s a link to a bash script, showing how easy it is to customize these checks.

This script will try and receive a token – i.e. test that the Keystone API is actually doing its job. If a token is not acquired in a timely manner – or is not acquired at all – then a warning or an alert will be triggered (respectively).

Full Stack Development

The third layer in the onion deploys a VM that is utilizing as many as possible of the OpenStack services, to verify that all of them are working. The services we would like to check are:

1. Keystone
2. Neutron
3. Nova
4. Glance
5. Swift
6. Cinder
7. Heat
Of course, if there are other services within OpenStack that you want to verify then you can add them to this list. The easiest way to do this would be to deploy a Heat stack. The stack tests the following:

1. Running the command tests authentication and if the orchestration engine is functioning (Keystone & Heat).
2. Spin up an instance and test it to see if you can retrieve an image (Glance), start an instance (Nova) and get an IP address (Neutron).
3. Attach a volume which will verify your volume storage (Cinder).
4. Run a script in the instance from an Object store (Swift).

All of the above can be packaged into a simple Heat template, here’s the link on github.

**Tying it all together as a Sensu Check**

The Sensu community has a substantial number of checks that you can use to monitor your environment. To delve into more information about how checks work and where you can find more information – the Sensu documentation site is your friend.

To tie the whole process together you will need to wrap this into a script.

1. Save the Heat template above as check-openstack.yaml, here’s the link on github
2. Save the bash script as check-openstack.sh, here’s the link on github
3. Add this as a check to your Sensu server – passing the correct parameters for OS_AUTH_URL, OS_USERNAME and OS_PASSWORD.

This is done by adding a check which is usually located under /etc/sensu/plugins/ and will be structured similar to the syntax below:

```
"checks": {
   "openstack_check": {
      "command": "check-openstack.sh",
      "interval": 300,
      "subscribers": [ "linux" ]
   }
}
```

The expected result of this script will be a stack that is created with a CREATE_COMPLETE result, indicating that the provisioning is functioning correctly. If the result you received differs from this result, you will have identified an error with one of the steps in the process. Identifying which of the steps is problematic requires further discussion.
How-to Monitor and Log OpenStack Clouds with ELK

While much information is available in OpenStack it is not always easily accessible. If we know how to make the proper use of the log data, it can be very helpful in monitoring and troubleshooting our OpenStack environment. ELK is a useful tool that helps clarify what is working, what is wrong and what is happening in our cloud, both in real time and for events that have occurred previously. We'll discuss how to forward log information from your OpenStack Controller into ELK and how to conduct basic troubleshooting of your OpenStack cloud.

Installing Logstash on your OpenStack Controller

We'll use LogStash to forward the logs on the control nodes to a Central Elasticsearch server. The exact details of how to install an ELK server is beyond the scope of this article, but Elasticsearch has comprehensive documentation to help you on your way. You might also look at an alternate next generation tool, Beats.

Note: This article assumes that OpenStack is running on a Centos/Redhat system.

To begin, on your control node(s) you will need to install Java.

First, install the prerequisite for Logstash, which is Java, by running the following:

```
yum -y localinstall jdk-8u73-linux-x64.rpm
rm jdk-8u73-linux-x64.rpm
```

Once that is complete, enable the logstash repository in YUM:

```
cat << EOF > /etc/yum.repos.d/logstash.repo
[logstash-2.2]
name=logstash repository for 2.2 packages
baseurl=http://packages.elasticsearch.org/logstash/2.2/centos
gpgcheck=1
gpgkey=http://packages.elasticsearch.org/GPG-KEY-elasticsearch
enabled=1
EOF
```

Then install and enable Logstash:

```
yum -y install logstash
chkconfig logstash on
```
Log collection and forwarding

We need to configure Logstash to collect the appropriate logs, parse them and send them to an Elasticsearch server.
We are going to take a tested Logstash configuration from GoDaddy, and define the configuration in /etc/logstash/conf.d/logstash.conf
The file is very large so let’s take an example of each section to explain what it does:

```plaintext
file {
    path => ['/var/log/keystone/keystone-all.log']
    tags => ['keystone', 'oslofmt', 'keystoneapi']
    type => "keystone"
}
```

The declaration above defines that logstash will monitor the file in /var/log/keystone/keystone-all.log and will apply the tags keystone, oslofmt, and keystoneapi to each of the entries in the logs and define it as a type of keystone. Next we will do some massaging and shaping of the logs so that we can better understand the information.

```plaintext
if "keystoneapi" in [tags] {
    mutate {
        gsub => ['logmessage','"\"","\"]
    }
    grok {
        match => { "logmessage" => "\[\-\] %{NOTSPACE:requesterip} \- \- \\
                    [%{NOTSPACE:req_date} %{NOTSPACE:req_time}] %{NOTSPACE:method} %{NOTSPACE:url_path} %{NOTSPACE:http_ver} %{NUMBER:response} %{NUMBER:bytes} %{NUMBER:seconds}" }
        add_field => ["api", "keystone"]
        add_tag => ["apimetrics"]
    }
}
```

By using the mutate plugin in Logstash, we use the gsub filter to convert a string field by applying a regular expression and a replacement. Then we use grok to parse the text and structure it, so that ELK can present the data in an appropriate fashion:

```plaintext
output {
    elasticsearch {
        hosts => ["elk.mydomain.com:9200"]
    }
}
```

The last part is the output – the destination to where to send the information. Define the URL of your ELK endpoint as this destination, then start the logstash client on the controller:

```plaintext
service logstash start
```
Identifying a Failed Login using Kibana

The graphic interface that organizes and clarifies the information flowing into ELK is Kibana. If this is a new ELK installation you will not have any information in Kibana.

As soon as the logs start flowing Kibana will start showing more information.

Now that you have logs flowing from your OpenStack Controller into ELK, you can make use of that information.

A common example would check for failed logins to the OpenStack API. A sudden influx of failed logins could point to a potential attack on your OpenStack infrastructure. This is something we would want to identify and with additional tools, alert the appropriate admins of a potential breach.
Here is an example of such a message:

<table>
<thead>
<tr>
<th>_index</th>
<th>Q</th>
<th>logstash-2016.04.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>_score</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>_type</td>
<td>Q</td>
<td>keystone</td>
</tr>
<tr>
<td>host</td>
<td>Q</td>
<td>rdo.vagrant.dev</td>
</tr>
<tr>
<td>logdate</td>
<td>Q</td>
<td>2016-04-25 20:20:37.115</td>
</tr>
<tr>
<td>loglevel</td>
<td>Q</td>
<td>WARNING</td>
</tr>
<tr>
<td>logmessage</td>
<td>Q</td>
<td>[req-f72976e8-a307-4bf8-8c31-10effc541910 - - - -] Authorization failed. The request you have made requires authentication. from 192.168.0.42</td>
</tr>
<tr>
<td>message</td>
<td>Q</td>
<td>2016-04-25 20:20:37.115 4854 WARNING keystone.common.wsgi [req-f72976e8-a307-4bf8-8c31-10effc541910 - - - -] Authorization failed. The request you have made requires authentication. from 192.168.0.42</td>
</tr>
<tr>
<td>module</td>
<td>Q</td>
<td>keystone.common.wsgi</td>
</tr>
<tr>
<td>path</td>
<td>Q</td>
<td>/var/log/keystone/keystone.log</td>
</tr>
<tr>
<td>pid</td>
<td>Q</td>
<td>4854</td>
</tr>
<tr>
<td>received_at</td>
<td>Q</td>
<td>April 25th 2016, 22:20:38.738</td>
</tr>
<tr>
<td>tags</td>
<td>Q</td>
<td>keystone, oslofmt</td>
</tr>
<tr>
<td>type</td>
<td>Q</td>
<td>keystone</td>
</tr>
</tbody>
</table>

The original message from Keystone looked like this:

As we see, there are number of additional items that Logstash added to the file and the information is indexed in multiple fields.

We will now create a Dashboard to present the number of failed logins. To do so, we click on the Visualize tab in Kibana:

We are going to choose a Line Chart:

Select ‘From a new search’ and select your index pattern (in this case logstash-*).

The Y-Axis will be selected as ‘Count’ by default.
For the X-Axis we will choose ‘Date Histogram’:

Finally, we identify the type of data that we are looking for. Here we are looking for logs that have a tag keystone (because this is a Keystone authentication failure) and we are looking for the word “failed.”

The search syntax that we will enter in the search box is:

tag: keystone AND failed

Select the timeframe that you are looking for (Last 1 hour – with a 30 second refresh):

You will start to see how many failed logins have occurred:
Summary

Monitoring your OpenStack infrastructure is imperative to ensure that everything is functioning correctly; that you are proactively managing your environment. Thus you are aware of issues in your infrastructure long before you receive complaints from your customers and end users.

Monitoring your environment using a multi-layer approach can help you identify problems as they occur and provide a better, more robust service all round.

About

Stratoscale is the cloud infrastructure company, providing comprehensive cloud infrastructure software solutions for service providers, enterprise IT and development teams. The company’s comprehensive cloud data center software, Stratoscale Symphony, can be deployed in minutes on commodity x86 servers, providing an Amazon Web Services (AWS) experience with the ability to augment aging VMware infrastructure. Stratoscale was named a “Cool Vendor in Servers and Virtualization” by Gartner and is backed by over $70M from leading investors including: Battery Ventures, Bessemer Venture Partners, Cisco, Intel, Qualcomm Ventures, SanDisk and Leslie Ventures.

For more information, visit http://www.stratoscale.com/ and follow us on Twitter @stratoscale.