Red Hat OpenStack Platform 16.0

Auto Scaling for Instances

Configure Auto Scaling in Red Hat OpenStack Platform
Red Hat OpenStack Platform 16.0 Auto Scaling for Instances

Configure Auto Scaling in Red Hat OpenStack Platform

OpenStack Team
rhos-docs@redhat.com
Abstract

Automatically scale out your Compute instances in response to system usage.
# Table of Contents

**CHAPTER 1. ABOUT THIS GUIDE** ................................................................. 3

**CHAPTER 2. CONFIGURE AUTO SCALING FOR COMPUTE INSTANCES** .................. 4
  2.1. ARCHITECTURAL OVERVIEW ......................................................... 4  
    2.1.1. Orchestration ................................................................. 4  
    2.1.2. Telemetry ................................................................. 4  
    2.1.3. Key Terms ................................................................. 4  
  2.2. EXAMPLE: AUTO SCALING BASED ON CPU USAGE .................................. 4  
    2.2.1. Test Automatic Scaling Up Instances ...................................... 8  
    2.2.2. Automatically Scaling Down Instances ................................... 10  
    2.2.3. Troubleshooting the setup .................................................. 10
CHAPTER 1. ABOUT THIS GUIDE

WARNING

Red Hat is currently reviewing the information and procedures provided in this guide for this release.


If you require assistance for the current Red Hat OpenStack Platform release, please contact Red Hat support.
CHAPTER 2. CONFIGURE AUTO SCALING FOR COMPUTE INSTANCES

This guide describes how to automatically scale out your Compute instances in response to heavy system usage. By using pre-defined rules that consider factors such as CPU or memory usage, you can configure Orchestration (heat) to add and remove additional instances automatically, when they are needed.

2.1. ARCHITECTURAL OVERVIEW

2.1.1. Orchestration

The core component providing automatic scaling is Orchestration (heat). Orchestration allows you to define rules using human-readable YAML templates. These rules are applied to evaluate system load based on Telemetry data to find out whether there is need to add more instances into the stack. Once the load has dropped, Orchestration can automatically remove the unused instances again.

2.1.2. Telemetry

Telemetry does performance monitoring of your OpenStack environment, collecting data on CPU, storage, and memory utilization for instances and physical hosts. Orchestration templates examine Telemetry data to assess whether any pre-defined action should start.

2.1.3. Key Terms

- **Stack** - A stack stands for all the resources necessary to operate an application. It can be as simple as a single instance and its resources, or as complex as multiple instances with all the resource dependencies that comprise a multi-tier application.

- **Templates** - YAML scripts that define a series of tasks for Heat to execute. For example, it is preferable to use separate templates for certain functions:
  - **Template File** - This is where you define thresholds that Telemetry should respond to, and define the auto scaling group.
  - **Environment File** - Defines the build information for your environment: which flavor and image to use, how the virtual network should be configured, and what software should be installed.

2.2. EXAMPLE: AUTO SCALING BASED ON CPU USAGE

In this example, Orchestration examines Telemetry data, and automatically increases the number of instances in response to high CPU usage. A stack template and environment template are created to define the needed rules and subsequent configuration. This example makes use of existing resources (such as networks), and uses names that are likely to differ in your own environment.

1. Create the environment template, describing the instance flavor, networking configuration, and image type and save it in the template `/home/<user>/stacks/example1/cirros.yaml` file.

   Please, replace the `<user>` variable with a real user name:

   ```yaml
   heat_template_version: 2016-10-14
   description: Template to spawn an cirros instance.
   ```
parameters:
metadata:
    type: json
image:
    type: string
description: image used to create instance
default: cirros
flavor:
    type: string
description: instance flavor to be used
default: m1.tiny
key_name:
    type: string
description: keypair to be used
default: mykeypair
network:
    type: string
description: project network to attach instance to
default: internal1
external_network:
    type: string
description: network used for floating IPs
default: external_network
resources:
server:
    type: OS::Nova::Server
properties:
    block_device_mapping:
        - device_name: vda
declete_on_termination: true
          volume_id: { get_resource: volume }
    flavor: {get_param: flavor}
    key_name: {get_param: key_name}
    metadata: {get_param: metadata}
    networks:
        - port: { get_resource: port }
port:
    type: OS::Neutron::Port
properties:
    network: {get_param: network}
    security_groups:
        - default
floating_ip:
    type: OS::Neutron::FloatingIP
properties:
    floating_network: {get_param: external_network}
floating_ip_assoc:
    type: OS::Neutron::FloatingIPAssociation
properties:
    floatingip_id: { get_resource: floating_ip }
    port_id: { get_resource: port }
2. Register the Orchestration resource in ~/stacks/example1/environment.yaml:

```
resource_registry:
  "OS::Nova::Server::Cirros": ~/stacks/example1/cirros.yaml
```

3. Create the stack template, describing the CPU thresholds to watch for, and how many instances should be added. An instance group is also created, defining the minimum and maximum number of instances that can participate in this template.

**NOTE**

The `granularity` parameter needs to be set according to `gnocchi cpu_util` metric granularity. For more information, refer to this solution article.

Save the following values in ~/stacks/example1/template.yaml:

```yaml
heat_template_version: 2016-10-14
description: Example auto scale group, policy and alarm
resources:
scaleup_group:
  type: OS::Heat::AutoScalingGroup
  properties:
    cooldown: 300
    desired_capacity: 1
    max_size: 3
    min_size: 1
  resource:
    type: OS::Nova::Server::Cirros
    properties:
      metadata: {"metering.server_group": {get_param: "OS::stack_id"}}

scaleup_policy:
  type: OS::Heat::ScalingPolicy
  properties:
    adjustment_type: change_in_capacity
    auto_scaling_group_id: { get_resource: scaleup_group }
    cooldown: 300
    scaling_adjustment: 1

scaledown_policy:
  type: OS::Heat::ScalingPolicy
  properties:
    adjustment_type: change_in_capacity
    auto_scaling_group_id: { get_resource: scaleup_group }
    cooldown: 300
    scaling_adjustment: -1
```
cpu_alarm_high:
  type: OS::Aodh::GnocchiAggregationByResourcesAlarm
  properties:
    description: Scale up if CPU > 80%
    metric: cpu_util
    aggregation_method: mean
    granularity: 300
    evaluation_periods: 1
    threshold: 80
    resource_type: instance
    comparison_operator: gt
    alarm_actions:
      - str_replace:
          template: trust+url
          params:
            url: {get_attr: [scaleup_policy, signal_url]}
    query:
      str_replace:
        template: '{"=": {"server_group": "stack_id"}}'
        params:
          stack_id: {get_param: "OS::stack_id"}

cpu_alarm_low:
  type: OS::Aodh::GnocchiAggregationByResourcesAlarm
  properties:
    metric: cpu_util
    aggregation_method: mean
    granularity: 300
    evaluation_periods: 1
    threshold: 5
    resource_type: instance
    comparison_operator: lt
    alarm_actions:
      - str_replace:
          template: trust+url
          params:
            url: {get_attr: [scaledown_policy, signal_url]}
    query:
      str_replace:
        template: '{"=": {"server_group": "stack_id"}}'
        params:
          stack_id: {get_param: "OS::stack_id"}

outputs:
  scaleup_policy_signal_url:
    value: {get_attr: [scaleup_policy, signal_url]}
  scaledown_policy_signal_url:
    value: {get_attr: [scaledown_policy, signal_url]}

4. Run the following OpenStack command to build the environment and deploy the instance:

    $ openstack stack create -t template.yaml -e environment.yaml example

    +---------------------+--------------------------------------------+
    | Field               | Value                                      |
    +---------------------+--------------------------------------------+
Red Hat OpenStack Platform 16.0 Auto Scaling for Instances

5. Orchestration will create the stack and launch a defined minimum number of cirros instances, as defined in the `min_size` parameter of the `scaleup_group` definition. Verify that the instances were created successfully:

```bash
$ openstack server list
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Status</th>
<th>Task State</th>
<th>Power State</th>
<th>Networks</th>
</tr>
</thead>
<tbody>
<tr>
<td>15505a5f-5be6-49e4-8501-e56e5d8212612</td>
<td>ex-3gax-5f3a40g5cwn2-png47w3u2vjd-server-vaaajuv4mj3j</td>
<td>ACTIVE</td>
<td>Running</td>
<td>Running</td>
<td>internal1=10.10.10.9, 192.168.122.8</td>
</tr>
</tbody>
</table>

6. Orchestration also creates two cpu alarms which are used to trigger scale-up or scale-down events, as defined in `cpu_alarm_high` and `cpu_alarm_low`. Verify that the triggers exist:

```bash
$ openstack alarm list
```

<table>
<thead>
<tr>
<th>alarm_id</th>
<th>type</th>
<th>name</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0227070d-46cc-4d39-a0b2-adf2fc7ab86a</td>
<td>gnocchi_aggregation_by_resources_threshold</td>
<td>example-cpu_alarm_high-odj77q9t7d7</td>
<td>insufficient data</td>
</tr>
<tr>
<td>46ed2c50-e05a-44d8-b6f6-1eb683afa913</td>
<td>gnocchi_aggregation_by_resources_threshold</td>
<td>example-cpu_alarm_low-m37jvnm56x2t</td>
<td>insufficient data</td>
</tr>
</tbody>
</table>

2.2.1. Test Automatic Scaling Up Instances

Orchestration can scale instances automatically based on the `cpu_alarm_high` threshold definition. Once the CPU utilization reaches a value defined in the `threshold` parameter, another instance is started to balance the load. The `threshold` value in the above `template.yaml` file is set to 80%.

1. Login to the instance and run several `dd` commands to generate the load:

```bash
$ ssh -i ~/mykey.pem cirros@192.168.122.8
$ sudo dd if=/dev/zero of=/dev/null &
$ sudo dd if=/dev/zero of=/dev/null &
$ sudo dd if=/dev/zero of=/dev/null &
```
2. Having run the `dd` commands, you can expect to have 100% CPU utilization in the cirros instance. Verify that the alarm has been triggered:

```bash
$ openstack alarm list
+--------------------------------------+--------------------------------------------+--------------------------
| alarm_id                             | type                                       | name                                | state | severity | enabled |
+--------------------------------------+--------------------------------------------+--------------------------
| 022f707d-46cc-4d39-a0b2-afd2fc7ab86a | gnocchi_aggregation_by_resources_threshold | example-cpu_alarm_high-odj77qpbld7 | alarm  | low      | True    |
| 46ed2c50-e05a-44d8-b616-f1ebd83af913 | gnocchi_aggregation_by_resources_threshold | example-cpu_alarm_low-m37jvnm56x2t | ok     | low      | True    |
+--------------------------------------+--------------------------------------------+--------------------------
```

3. After some time (approximately 60 seconds), Orchestration will start another instance and add it into the group. You can verify this with the `nova list` command:

```bash
$ openstack server list
+--------------------------------------+-------------------------------------------------------+--------+------+
| ID                                   | Name                                                  | Status | Task State | Power State | Networks                              |
+--------------------------------------+-------------------------------------------------------+--------+------+
| 477ee1af-096c-477c-9a3f-b95b0e2d4ab5 | ex-3gax-4urpik5koff-yrrxk3zxfmpf-server-2hde4tp4trmk | ACTIVE | -          | Running     | internal1=10.10.10.13, 192.168.122.17 |
| e1524f65-5be6-49e4-8501-e5e5d812c612 | ex-3gax-5f3a4og5cwn2-png47w3u2vjd-server-vaa7huu4mj3 | ACTIVE | -          | Running     | internal1=10.10.10.9, 192.168.122.8   |
+--------------------------------------+-------------------------------------------------------+--------+------+
```
2.2.2. Automatically Scaling Down Instances

Orchestration can also automatically scale down instances based on the `cpu_alarm_low` threshold. In this example, the instances are scaled down once CPU utilization is below 5%.

1. Terminate the running `dd` processes and you will observe Orchestration begin to scale the instances back down.

```
$ killall dd
```

2. Stopping the `dd` processes causes the `cpu_alarm_low` event to trigger. As a result, Orchestration begins to automatically scale down and remove the instances. Verify, that the corresponding alarm has been triggered.

```
$ openstack alarm list
```

<table>
<thead>
<tr>
<th>alarm_id</th>
<th>type</th>
<th>name</th>
<th>state</th>
<th>severity</th>
<th>enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>022f707d-46cc-4d39-a0b2-afd2fc7ab86a</td>
<td>gnocchi_aggregation_by_resources_threshold</td>
<td>example-cpu_alarm_high-odj77qpld77</td>
<td>ok</td>
<td>low</td>
<td>True</td>
</tr>
<tr>
<td>46ed2c50-e05a-44d8-b6f6-f1ebd83af913</td>
<td>gnocchi_aggregation_by_resources_threshold</td>
<td>example-cpu_alarm_low-m37vnm56x2t</td>
<td>alarm</td>
<td>low</td>
<td>True</td>
</tr>
</tbody>
</table>

After a few minutes, Orchestration continually reduce the number of instances to the minimum value defined in the `min_size` parameter of the `scaleup_group` definition. In this scenario, the `min_size` parameter is set to 1.

2.2.3. Troubleshooting the setup

If your environment is not working properly, you can look for errors in the log files and history records.

1. To get information on state transitions, you can list the stack event records:

```
$ openstack stack event list example
```

```
2017-03-06 11:12:43Z [example]: CREATE_IN_PROGRESS  Stack CREATE started
2017-03-06 11:12:43Z [example.scaleup_group]: CREATE_IN_PROGRESS  state changed
2017-03-06 11:13:04Z [example.scaleup_group]: CREATE_COMPLETE  state changed
2017-03-06 11:13:04Z [example.scaledown_policy]: CREATE_IN_PROGRESS  state changed
2017-03-06 11:13:05Z [example.scaleup_policy]: CREATE_IN_PROGRESS  state changed
2017-03-06 11:13:05Z [example.scaleup_policy]: CREATE_COMPLETE  state changed
2017-03-06 11:13:05Z [example.cpu_alarm_low]: CREATE_IN_PROGRESS  state changed
2017-03-06 11:13:05Z [example.cpu_alarm_high]: CREATE_IN_PROGRESS  state changed
2017-03-06 11:13:06Z [example.cpu_alarm_low]: CREATE_COMPLETE  state changed
2017-03-06 11:13:07Z [example.cpu_alarm_high]: CREATE_COMPLETE  state changed
2017-03-06 11:13:07Z [example]: CREATE_COMPLETE  Stack CREATE completed successfully
2017-03-06 11:19:34Z [example.scaleup_policy]: SIGNAL_COMPLETE  alarm state changed from alarm to alarm (Remaining as alarm due to 1 samples outside threshold, most
2. To read the alarm history log:

```bash
$ openstack alarm-history show 022f707d-46cc-4d39-a0b2-afd2fc7ab86a
```

```bash
+----------------------------+----------------+--------------------------------------------------------------
| timestamp                  | type            | detail                                                        |
| event_id                             |                |                                                              |
```
To see the records of scale-out or scale-down operations that heat collects for the existing stack, you can use `awk` to parse the `heat-engine.log`:

```
$ awk '/Stack UPDATE started/,/Stack CREATE completed successfully/ {print $0}' /var/log/heat/heat-engine.log
```

To see the `aodh` related information, examine the `evaluator.log`:

```
$ grep -i alarm /var/log/aodh/evaluator.log | grep -i transition
```