Red Hat OpenStack Platform 8
Auto Scaling for Compute

configure Auto Scaling in Red Hat OpenStack Platform

OpenStack Team

Red Hat OpenStack Platform 8 Auto Scaling for Compute

configure Auto Scaling in Red Hat OpenStack Platform

OpenStack Team
rhos-docs@redhat.com

Legal Notice

Abstract

Automatically scale out your Compute instances in response to system usage.

Table of Contents

CHAPTER 1. CONFIGURE AUTO SCALING FOR COMPUTE 3
1.1. ARCHITECTURAL OVERVIEW 3
1.2. EXAMPLE: AUTO SCALING BASED ON CPU USAGE 3
1.3. EXAMPLE: AUTO SCALING APPLICATIONS 9

CHAPTER 1. CONFIGURE AUTO SCALING FOR COMPUTE

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 automatically add and remove additional instances as needed.

1.1. ARCHITECTURAL OVERVIEW

1.1.1. Orchestration

The core component behind auto scaling is Orchestration (heat). Orchestration allows you to define rules using human-readable YAML templates. These rules can assess Telemetry data before deciding to add additional instances. Then, once the activity has subsided, Orchestration can automatically remove any unneeded instances.

1.1.2. 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 when assessing whether to take any pre-defined action.

1.1.3. Key Terms

- **Stack** - A stack comprises 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:

  - **Stack Template** - This is where you define thresholds that Telemetry should respond to, and define the auto scaling group.

  - **Environment Template** - 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.

1.2. 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. Enter the following values in **`/etc/heat/templates/cirros.yaml`**:

```
heat_template_version: 2014-10-16
description: A base Cirros 0.3.4 server

resources:
  server:
    type: OS::Nova::Server
    properties:
      block_device_mapping:
        - device_name: vda
          delete_on_termination: true
          volume_id: { get_resource: volume }
      flavor: m1.nano
      key_name: admin
      networks:
        - port: { get_resource: port }

  port:
    type: OS::Neutron::Port
    properties:
      network: private
      security_groups:
        - all

  floating_ip:
    type: OS::Neutron::FloatingIP
    properties:
      floating_network: public

  floating_ip_assoc:
    type: OS::Neutron::FloatingIPAssociation
    properties:
      floatingip_id: { get_resource: floating_ip }
      port_id: { get_resource: port }

  volume:
    type: OS::Cinder::Volume
    properties:
      image: 'Cirros 0.3.4'
      size: 1
```

2. Register the Orchestration resource in **`/root/environment.yaml`**:

```
resource_registry:

    "OS::Nova::Server::Cirros":
"file:///etc/heat/templates/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.

   Enter the following values in **`/root/example.yaml`**:

```
heat_template_version: 2014-10-16
description: Example auto scale group, policy and alarm
resources:
  scaleup_group:
    type: OS::Heat::AutoScalingGroup
    properties:
      cooldown: 60
      desired_capacity: 1
      max_size: 3
      min_size: 1
      resource:
        type: OS::Nova::Server::Cirros

  scaleup_policy:
    type: OS::Heat::ScalingPolicy
    properties:
      adjustment_type: change_in_capacity
      auto_scaling_group_id: { get_resource: scaleup_group }
      cooldown: 60
      scaling_adjustment: 1

  scaledown_policy:
    type: OS::Heat::ScalingPolicy
    properties:
      adjustment_type: change_in_capacity
      auto_scaling_group_id: { get_resource: scaleup_group }
      cooldown: 60
      scaling_adjustment: -1

  cpu_alarm_high:
    type: OS::Ceilometer::Alarm
    properties:
      meter_name: cpu_util
      statistic: avg
      period: 60
      evaluation_periods: 1
      threshold: 50
      alarm_actions:
        - {get_attr: [scaleup_policy, alarm_url]}
      comparison_operator: gt

  cpu_alarm_low:
    type: OS::Ceilometer::Alarm
    properties:
      meter_name: cpu_util
      statistic: avg
      period: 60
      evaluation_periods: 1
      threshold: 10
      alarm_actions:
        - {get_attr: [scaledown_policy, alarm_url]}
      comparison_operator: lt
```

4. Update the Telemetry collection interval. By default, Telemetry polls instances every 10 minutes for CPU data. For this example, change the interval to 60 seconds in **`/etc/ceilometer/pipeline.yaml`**:

```
- name: cpu_source
interval: 60
meters:
- "cpu"
sinks:
- cpu_sink
```

Note

A polling period of 60 seconds is not recommended for production environments, as a higher polling interval can result in increased load on the control plane.

5. Restart all OpenStack services to apply the updated Telemetry setting:

```
# openstack-service restart
```

Note

This step will result in a brief outage to your OpenStack deployment.

6. Run the Orchestration scripts to build the environment and deploy the instance:

```
# heat stack-create example -f /root/example.yaml -e
/root/environment.yaml
+--------------------------------------+------------+---------
-----------+----------------------+
| id                                   | stack_name |
stack_status       | creation_time        |
+--------------------------------------+------------+---------
-----------+----------------------+
| 6fca513c-25a1-4849-b7ab-909e37f52eca | example    |
CREATE_IN_PROGRESS | 2015-08-31T16:18:02Z |
+--------------------------------------+------------+---------
-----------+----------------------+
```

Orchestration will create the stack and launch a single cirros instance, as set in the **scaleup_group** definition: **min_size**:

```
# nova list
+--------------------------------------+---------------------
----------------------------------+--------+------------+-----
--------+--------------------------------------+
| ID                                   | Name
| Status | Task State | Power State | Networks
|
+--------------------------------------+---------------------
----------------------------------+--------+------------+-----
--------+--------------------------------------+
| 3f627c84-06aa-4782-8c12-29409964cc73 | ex-qeki-3azno6me5gvm-
pqmr5zd6kuhm-server-gieck7uoyrwc | ACTIVE | -          | Running
| private=10.10.1.156, 192.168.122.234 |
+--------------------------------------+---------------------
----------------------------------+--------+------------+-----
--------+--------------------------------------+
```

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**:

```
# ceilometer alarm-list
```

```
+--------------------------------------+---------------------
----------------+-------------------+----------+---------+----
--------+--------------------------------+------------------+
| Alarm ID                             | Name
| State             | Severity | Enabled | Continuous | Alarm
condition                | Time constraints |
+--------------------------------------+---------------------
----------------+-------------------+----------+---------+----
--------+--------------------------------+------------------+
| 04b4f845-f5b6-4c5a-8af0-59e03c22e6fa | example-cpu_alarm_high-
rd5kysmlahvx | ok                | low      | True    | True
| cpu_util > 50.0 during 1 x 60s | None             |
| ac81cd81-20b3-45f9-bea4-e51f00499602 | example-cpu_alarm_low-
6t65kswutupz  | ok                | low      | True    | True
| cpu_util < 10.0 during 1 x 60s | None             |
+--------------------------------------+---------------------
----------------+-------------------+----------+---------+----
--------+--------------------------------+------------------+
```

1.2.1. Test Auto Scaling Instances

Orchestration auto scales instances based on the **`cpu_alarm_high`** threshold. Once CPU utilization is above 50% instances will be scaled up, as set in the **`cpu_alarm_high`** definition: **`threshold: 50`**

To generate CPU load, login to the instance and run the **`dd`** command:

```
$ ssh -i admin.pem cirros@192.168.122.232
$ dd if=/dev/zero of=/dev/null &
$ dd if=/dev/zero of=/dev/null &
$ dd if=/dev/zero of=/dev/null &
```

After running the **`dd`** commands, you can expect to have 100% CPU utilization in the cirros instance. After 60 seconds you should see that Orchestration has auto scaled the group to two instances:

```
# nova list
+--------------------------------------+---------------------------
----------------------------+--------+------------+-------------+--
------------------------------------+
| ID                                   | Name
| Status | Task State | Power State | Networks
|
+--------------------------------------+---------------------------
----------------------------+--------+------------+-------------+--
------------------------------------+
| 3f627c84-06aa-4782-8c12-29409964cc73 | ex-qeki-3azno6me5gvm-
pqmr5zd6kuhm-server-gieck7uoyrwc | ACTIVE | -          | Running     |
private=10.10.1.156, 192.168.122.234 |
| 0f69dfbe-4654-474f-9308-1b64de3f5c18 | ex-qeki-qmvor5rkptj7-
krq7i66h6n7b-server-b4pk3dzjvbpi | ACTIVE | -          | Running     |
private=10.10.1.157, 192.168.122.235 |
+--------------------------------------+---------------------------
----------------------------+--------+------------+-------------+--
------------------------------------+
```

After a further 60 seconds you will observe that Orchestration has auto scaled again to three instances. Since three is the maximum for this configuration, it will not scale any higher (as set in the **scaleup_group** definition: **max_size**)

```
# nova list
+--------------------------------------+---------------------------
----------------------------+--------+------------+-------------+--
-----------------------------------+
| ID                                   | Name
| Status | Task State | Power State | Networks
|
+--------------------------------------+---------------------------
----------------------------+--------+------------+-------------+--
-----------------------------------+
| 3f627c84-06aa-4782-8c12-29409964cc73 | ex-qeki-3azno6me5gvm-
pqmr5zd6kuhm-server-gieck7uoyrwc | ACTIVE | -          | Running     |
private=10.10.1.156, 192.168.122.234 |
| 0e805e75-aa6f-4375-b057-2c173b68f172 | ex-qeki-gajdwmu2cgm2-
vckf4g2gpwis-server-r3smbhtqij76 | ACTIVE | -          | Running     |
private=10.10.1.158, 192.168.122.236 |
| 0f69dfbe-4654-474f-9308-1b64de3f5c18 | ex-qeki-qmvor5rkptj7-
krq7i66h6n7b-server-b4pk3dzjvbpi | ACTIVE | -          | Running     |
private=10.10.1.157, 192.168.122.235 |
+--------------------------------------+---------------------------
----------------------------+--------+------------+-------------+--
-----------------------------------+
```

1.2.2. Automatically Scaling Down Instances

Orchestration automatically scales down instances based on the **cpu_alarm_low** threshold. In this example, the instances are scaled down once CPU utilization is below 10%. Terminate the running **dd** processes and you will observe Orchestration begin to scale the instances back down.

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:

```
# ceilometer alarm-list
+--------------------------------------+---------------------------
----------+-------+----------+---------+------------+---------------
-----------------+------------------+
| Alarm ID                             | Name
| State | Severity | Enabled | Continuous | Alarm condition
| Time constraints |
+--------------------------------------+---------------------------
----------+-------+----------+---------+------------+---------------
-----------------+------------------+
| 04b4f845-f5b6-4c5a-8af0-59e03c22e6fa | example-cpu_alarm_high-
rd5kysmlahvx | ok    | low      | True    | True       | cpu_util >
50.0 during 1 x 60s | None             |
| ac81cd81-20b3-45f9-bea4-e51f00499602 | example-cpu_alarm_low-
6t65kswutupz  | alarm | low      | True    | True       | cpu_util <
10.0 during 1 x 60s | None             |
+--------------------------------------+---------------------------
----------+-------+----------+---------+------------+---------------
-----------------+------------------+
```

After a few minutes you can expect to be back to a single instance, the minimum number of instances allowed in **scaleup_group**: **min_size: 1**

1.3. EXAMPLE: AUTO SCALING APPLICATIONS

The functionality described earlier can also be used to scale up applications; for example, a dynamic web page that be served by one of multiple instances running at a time. In this case, *neutron* can be configured to provide *Load Balancing-as-a-Service*, which works to evenly distribute traffic among instances.

In the following example, Orchestration again examines Telemetry data and increases the number of instances if high CPU usage is detected, or decreases the number of instances if CPU usage returns below a set value.

1. Create the template describing the properties of the *load-balancer* environment. Enter the following values in **/etc/heat/templates/lb-env.yaml**:

```
heat_template_version: 2014-10-16
description: A load-balancer server
parameters:
  image:
    type: string
    description: Image used for servers
  key_name:
    type: string
    description: SSH key to connect to the servers
  flavor:
    type: string
    description: flavor used by the servers
  pool_id:
    type: string
    description: Pool to contact
  user_data:
    type: string
    description: Server user_data
  metadata:
    type: json
  network:
    type: string
    description: Network used by the server

resources:
  server:
    type: OS::Nova::Server
    properties:
      flavor: {get_param: flavor}
      image: {get_param: image}
      key_name: {get_param: key_name}
      metadata: {get_param: metadata}
      user_data: {get_param: user_data}
      networks:
        - port: { get_resource: port }

  member:
    type: OS::Neutron::PoolMember
    properties:
      pool_id: {get_param: pool_id}
      address: {get_attr: [server, first_address]}
      protocol_port: 80

  port:
    type: OS::Neutron::Port
    properties:
      network: {get_param: network}
      security_groups:
        - base

outputs:
  server_ip:
    description: IP Address of the load-balanced server.
    value: { get_attr: [server, first_address] }
  lb_member:
    description: LB member details.
    value: { get_attr: [member, show] }
```

2. Create another template for the instances that will be running the web application. The following template creates a load balancer and uses the existing networks. Be sure to replace the parameters according to your environment, and save the template in a file such as **`/root/lb-webserver-rhel7.yaml`**:

```
heat_template_version: 2014-10-16
description: AutoScaling RHEL 7 Web Application
parameters:
  image:
    type: string
    description: Image used for servers
    default: RHEL 7
  key_name:
    type: string
    description: SSH key to connect to the servers
    default: admin
  flavor:
    type: string
    description: flavor used by the web servers
    default: m2.tiny
  network:
    type: string
    description: Network used by the server
    default: private
  subnet_id:
    type: string
    description: subnet on which the load balancer will be
located
    default: 9daa6b7d-e647-482a-b387-dd5f855b88ef
  external_network_id:
    type: string
    description: UUID of a Neutron external network
    default: db17c885-77fa-45e8-8647-dbb132517960

resources:
  webserver:
    type: OS::Heat::AutoScalingGroup
    properties:
      min_size: 1
      max_size: 3
      cooldown: 60
      desired_capacity: 1
      resource:
        type: file:///etc/heat/templates/lb-env.yaml
        properties:
          flavor: {get_param: flavor}
          image: {get_param: image}
          key_name: {get_param: key_name}
          network: {get_param: network}
          pool_id: {get_resource: pool}
          metadata: {"metering.stack": {get_param:
"OS::stack_id"}}
          user_data:
            str_replace:
              template: |
                #!/bin/bash -v

                yum -y install httpd php
                systemctl enable httpd
                systemctl start httpd
                cat <<EOF > /var/www/html/hostname.php
                <?php echo "Hello, My name is " .
php_uname('n'); ?>
                EOF
              params:
                hostip: 192.168.122.70
                fqdn: sat6.example.com
                shortname: sat6

  web_server_scaleup_policy:
    type: OS::Heat::ScalingPolicy
    properties:
      adjustment_type: change_in_capacity
      auto_scaling_group_id: {get_resource: webserver}
      cooldown: 60
      scaling_adjustment: 1

  web_server_scaledown_policy:
    type: OS::Heat::ScalingPolicy
    properties:
      adjustment_type: change_in_capacity
      auto_scaling_group_id: {get_resource: webserver}
      cooldown: 60
      scaling_adjustment: -1

  cpu_alarm_high:
    type: OS::Ceilometer::Alarm
    properties:
      description: Scale-up if the average CPU > 95% for 1 minute
      meter_name: cpu_util
      statistic: avg
      period: 60
      evaluation_periods: 1
      threshold: 95
      alarm_actions:
        - {get_attr: [web_server_scaleup_policy, alarm_url]}
      matching_metadata: {'metadata.user_metadata.stack':
{get_param: "OS::stack_id"}}
      comparison_operator: gt

  cpu_alarm_low:
    type: OS::Ceilometer::Alarm
    properties:
      description: Scale-down if the average CPU < 15% for 1
minute
      meter_name: cpu_util
      statistic: avg
      period: 60
      evaluation_periods: 1
      threshold: 15
      alarm_actions:
        - {get_attr: [web_server_scaledown_policy, alarm_url]}
      matching_metadata: {'metadata.user_metadata.stack':
{get_param: "OS::stack_id"}}
      comparison_operator: lt

  monitor:
    type: OS::Neutron::HealthMonitor
    properties:
      type: TCP
      delay: 5
      max_retries: 5
      timeout: 5

  pool:
    type: OS::Neutron::Pool
    properties:
      protocol: HTTP
      monitors: [{get_resource: monitor}]
      subnet_id: {get_param: subnet_id}
      lb_method: ROUND_ROBIN
      vip:
        protocol_port: 80

  lb:
    type: OS::Neutron::LoadBalancer
    properties:
      protocol_port: 80
      pool_id: {get_resource: pool}

  lb_floating:
    type: OS::Neutron::FloatingIP
    properties:
      floating_network_id: {get_param: external_network_id}
      port_id: {get_attr: [pool, vip, port_id]}

outputs:
  scale_up_url:
    description: >
      This URL is the webhook to scale up the autoscaling group.
You
      can invoke the scale-up operation by doing an HTTP POST to
this
      URL; no body nor extra headers are needed.
    value: {get_attr: [web_server_scaleup_policy, alarm_url]}
  scale_dn_url:
    description: >
      This URL is the webhook to scale down the autoscaling
group.
      You can invoke the scale-down operation by doing an HTTP
POST to
      this URL; no body nor extra headers are needed.
    value: {get_attr: [web_server_scaledown_policy, alarm_url]}
  pool_ip_address:
    value: {get_attr: [pool, vip, address]}
    description: The IP address of the load balancing pool
  website_url:
    value:
      str_replace:
        template: http://serviceip/hostname.php
        params:
          serviceip: { get_attr: [lb_floating,
floating_ip_address] }
    description: >
      This URL is the "external" URL that can be used to access
the
      website.
  ceilometer_query:
    value:
      str_replace:
        template: >
          ceilometer statistics -m cpu_util
          -q metadata.user_metadata.stack=stackval -p 60 -a avg
        params:
          stackval: { get_param: "OS::stack_id" }
    description: >
      This is a Ceilometer query for statistics on the cpu_util
meter
      Samples about OS::Nova::Server instances in this stack.
The -q
      parameter selects Samples according to the subject's
metadata.
      When a VM's metadata includes an item of the form
metering.X=Y,
      the corresponding Ceilometer resource has a metadata item
of the
      form user_metadata.X=Y and samples about resources so
tagged can
      be queried with a Ceilometer query term of the form
      metadata.user_metadata.X=Y.  In this case the nested stacks
give
      their VMs metadata that is passed as a nested stack
```

```
parameter,
      and this stack passes a metadata of the form
metering.stack=Y,
      where Y is this stack's ID.
```

3. Update the Telemetry collection interval. By default, Telemetry polls instances every 10 minutes for CPU data. For this example, change the interval to 60 seconds in **/etc/ceilometer/pipeline.yaml**:

   ```
   - name: cpu_source
   interval: 60
   meters:
   - "cpu"
   sinks:
   - cpu_sink
   ```

   Note

   A polling period of 60 seconds is not recommended for production environments, as a higher polling interval can result in increased load on the control plane.

4. Restart all OpenStack services to apply the updated Telemetry setting:

   ```
   # openstack-service restart
   ```

   Note

   This step will result in a brief outage to your OpenStack deployment.

5. Run the Orchestration scripts. This will build the environment and use the template to deploy the instance:

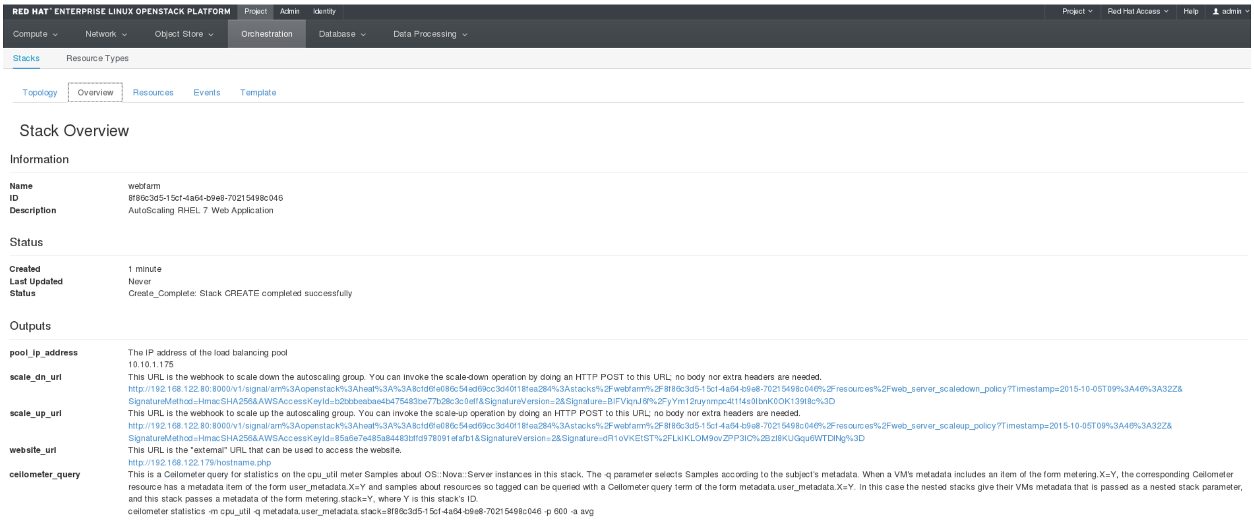
   ```
   # heat stack-create webfarm -f /root/lb-webserver-rhel7.yaml
   ```

   Replace **/root/lb-webserver-rhel7.yaml** with the actual path and file name.

You can monitor the creation of the stack in Dashboard under *Orchestration* → *Stacks* → *Webfarm*. Once the stack has been created, you are presented with multiple useful pieces of information, notably:

- URLs that you can use to trigger manual scale-up or scale-down events.
- The floating IP address, which is the IP address of the website.
- The Telemetry command which shows the CPU load for the whole stack, and which you can use to check whether the scaling is working as expected.

This is what the page looks like in Dashboard:



Stack Overview

Information

Name webfarm
ID 8f86c3d5-15cf-4a64-b9e8-70215498c046
Description AutoScaling RHEL 7 Web Application

Status

Created 1 minute
Last Updated Never
Status Create_Complete: Stack CREATE completed successfully

Outputs

pool_ip_address The IP address of the load balancing pool
10.10.1.175

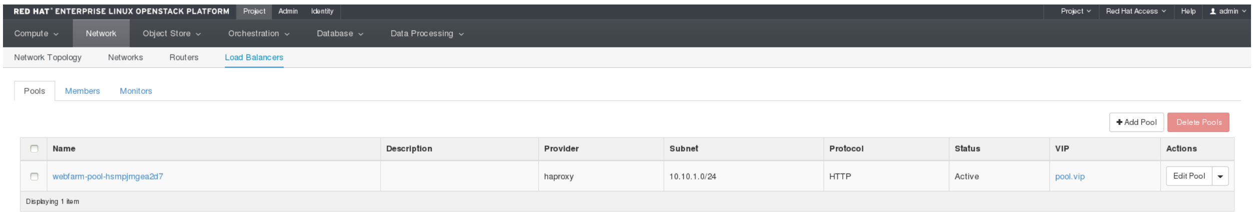
scale_dn_url This URL is the webhook to scale down the autoscaling group. You can invoke the scale-down operation by doing an HTTP POST to this URL; no body nor extra headers are needed.
http://192.168.122.80:8000/v1/signal/arn%3Aopenstack%3Aheat%3A%3A8cfd6fe086c54ed69cc3d40f18fea284%3Astacks%2Fwebfarm%2F8f86c3d5-15cf-4a64-b9e8-70215498c046%2Fresources%2Fweb_server_scaledown_policy?Timestamp=2015-10-05T09%3A46%3A32Z&SignatureMethod=HmacSHA256&AWSAccessKeyId=b2bbbeabae4b475483be77b28c3c0eff&SignatureVersion=2&Signature=BIFViqnJ6f%2FyYm12ruynmpc4t1f4s0IbnK0OK139t8c%3D

scale_up_url This URL is the webhook to scale up the autoscaling group. You can invoke the scale-up operation by doing an HTTP POST to this URL; no body nor extra headers are needed.
http://192.168.122.80:8000/v1/signal/arn%3Aopenstack%3Aheat%3A%3A8cfd6fe086c54ed69cc3d40f18fea284%3Astacks%2Fwebfarm%2F8f86c3d5-15cf-4a64-b9e8-70215498c046%2Fresources%2Fweb_server_scaleup_policy?Timestamp=2015-10-05T09%3A46%3A32Z&SignatureMethod=HmacSHA256&AWSAccessKeyId=85a6e7e485a84483bffd978091efafb1&SignatureVersion=2&Signature=dR1oVKEtST%2FLkIKLOM9ovZPP3IC%2BzI8KUGqu6WTDINg%3D

website_url This URL is the "external" URL that can be used to access the website.
http://192.168.122.179/hostname.php

ceilometer_query This is a Ceilometer query for statistics on the cpu_util meter Samples about OS::Nova::Server instances in this stack. The -q parameter selects Samples according to the subject's metadata. When a VM's metadata includes an item of the form metering.X=Y, the corresponding Ceilometer resource has a metadata item of the form user_metadata.X=Y and samples about resources so tagged can be queried with a Ceilometer query term of the form metadata.user_metadata.X=Y. In this case the nested stacks give their VMs metadata that is passed as a nested stack parameter, and this stack passes a metadata of the form metering.stack=Y, where Y is this stack's ID.
ceilometer statistics -m cpu_util -q metadata.user_metadata.stack=8f86c3d5-15cf-4a64-b9e8-70215498c046 -p 600 -a avg

Open *Network* → *Load Balancers* to view the load balancer:

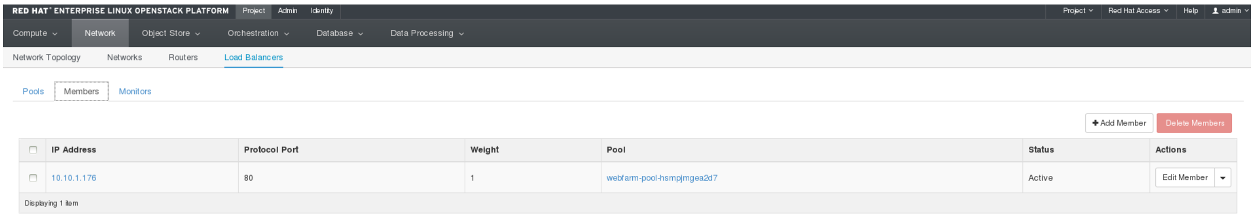


□	Name	Description	Provider	Subnet	Protocol	Status	VIP	Actions
□	webfarm-pool-hsmpjmgea2d7		haproxy	10.10.1.0/24	HTTP	Active	pool.vip	Edit Pool

Displaying 1 item

Click *Members*. This page displays the members of the load balancing pool; these are the instances to which the website traffic can be distributed. Note that a member will not have the *Active* status until the corresponding instance has been created, and Apache has been installed and configured.

When the web server has started, the instance is visible as an active member of the load balancer:



□	IP Address	Protocol Port	Weight	Pool	Status	Actions
□	10.10.1.176	80	1	webfarm-pool-hsmpjmgea2d7	Active	Edit Member

Displaying 1 item

You are now able to access the web application at **`http://IP/hostname.php`**. You can expect to see output similar to the following:

```
Hello, My name is we-zrwm-t4ezkpx34gxu-qbg5d7dqbc4j-server-mzdvigk2jugl
```

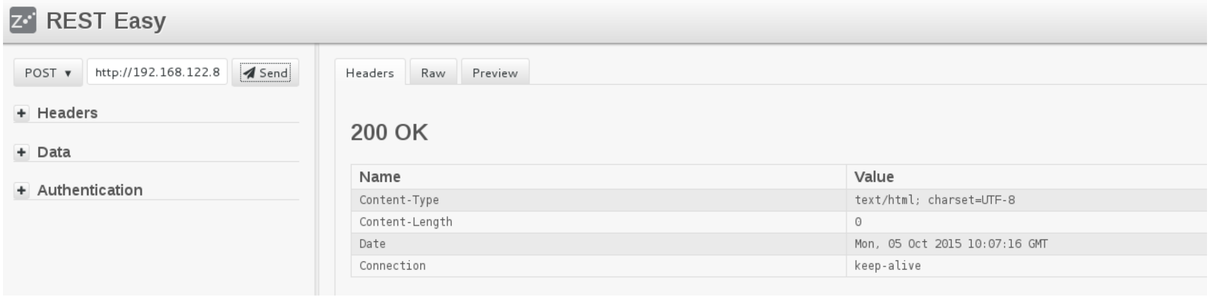
You can now view the stack's CPU performance data by running the Telemetry command from the stack overview in Dashboard. The command looks like the following:

```
# ceilometer statistics -m cpu_util -q
metadata.user_metadata.stack=8f86c3d5-15cf-4a64-b9e8-70215498c046 -p 60
-a avg
```

1.3.1. Test Auto Scaling Applications

To manually trigger application scaling, use the REST *scale-up URL* from the stack overview in Dashboard, or generate load by running a resource-intensive command on the initially deployed instance.

- To use the REST API, you need a tool which can perform **HTTP POST** requests, such as the *REST Easy* Firefox add on or **curl**. Copy the *scale-up URL* and either paste it into the *REST Easy* form:

  200 OK

Name	Value
Content-Type	text/html; charset=UTF-8
Content-Length	0
Date	Mon, 05 Oct 2015 10:07:16 GMT
Connection	keep-alive

  Or use it as a parameter on the **curl** command line:

  ```
  $ curl -X POST "scale-up URL"
  ```

- To artificially generate load, allocate a floating IP to the instance, log in to it with SSH, and run a command which will keep the CPU busy. For example:

  ```
  $ dd if=/dev/zero of=/dev/null &
  ```

Important

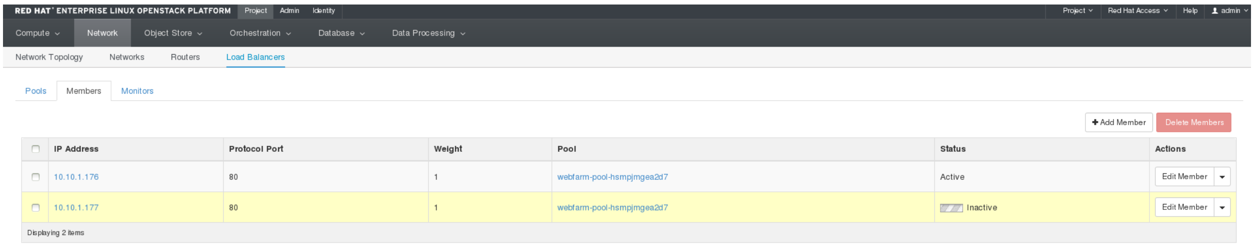
Check whether CPU usage is above 95%, for example, using the **top** command. If the CPU usage is not sufficiently high, run the **dd** command multiple times in parallel, or use another method to keep the CPU busy.

The next time Telemetry collects CPU data from the stack, the scale-up event will trigger and appear at *Orchestration* → *Stacks* → *Webfarm* → *Events*. A new web server instance will be created and added to the load balancer. When this is done, the instance becomes active, and you will notice that the website URL is routed through the load balancer to both instances in the stack.

Note

The creation can take several minutes because the instance must be initialized, Apache installed and configured, and the application deployed. This is monitored by HAProxy, which ensures that the website is available on the instance before it is marked as active.

This is what the list of members of the load balancing pool looks like in the Dashboard while the new instance is being created:



□	IP Address	Protocol Port	Weight	Pool	Status	Actions
□	10.10.1.176	80	1	webfarm-pool-hsmpjmgea2d7	Active	Edit Member ▾
□	10.10.1.177	80	1	webfarm-pool-hsmpjmgea2d7	▨ Inactive	Edit Member ▾

Displaying 2 items

Important

The average CPU usage of the instances in the *heat* stack is taken into account when deciding whether or not an additional instance gets created. Because the second instance will most likely have normal CPU usage, it will balance out the first instance. However, if the second instance becomes busy as well and the average CPU usage of the first and second instance exceeds 95%, another (third) instance will be created.

1.3.2. Automatically Scaling Down Applications

This is similar to Section 1.2.2, “Automatically Scaling Down Instances” in that the scale-down policy is triggered when the average CPU usage for the stack drops below a predefined value, which is 15% in the example described in Section 1.3.1, “Test Auto Scaling Applications”. In addition, when an instance is removed from the stack this way, it is also automatically removed from the load balancer. The website traffic is then automatically distributed among the rest of the instances.