Red Hat OpenStack Services on OpenShift 18.0

Autoscaling for instances

Configuring autoscaling in Red Hat OpenStack Services on OpenShift

Last Updated: 2024-09-04

Red Hat OpenStack Services on OpenShift 18.0 Autoscaling for instances

Configuring autoscaling in Red Hat OpenStack Services on OpenShift

OpenStack Team
rhos-docs@redhat.com

Legal Notice

Abstract

Use Red Hat OpenStack Services on OpenShift telemetry components and heat templates to automatically launch instances for workloads.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your input on our documentation. Tell us how we can make it better.

Providing documentation feedback in Jira

Use the Create Issue form to provide feedback on the documentation for Red Hat OpenStack Services on OpenShift (RHOSO) or earlier releases of Red Hat OpenStack Platform (RHOSP). When you create an issue for RHOSO or RHOSP documents, the issue is recorded in the RHOSO Jira project, where you can track the progress of your feedback.

To complete the Create Issue form, ensure that you are logged in to Jira. If you do not have a Red Hat Jira account, you can create an account at https://issues.redhat.com.

1. Click the following link to open a **Create Issue** page: Create Issue
2. Complete the **Summary** and **Description** fields. In the **Description** field, include the documentation URL, chapter or section number, and a detailed description of the issue. Do not modify any other fields in the form.
3. Click **Create**.

CHAPTER 1. INTRODUCTION TO AUTOSCALING COMPONENTS

Use telemetry components to collect data about your Red Hat OpenStack Services on OpenShift (RHOSO) environment, such as CPU, storage, and memory usage. You can launch and scale instances in response to workload demand and resource availability. You can define the upper and lower bounds of telemetry data that control the scaling of instances in your Orchestration service (heat) templates.

Control automatic instance scaling with the following telemetry components:

- **Data collection**: Telemetry uses the data collection service (Ceilometer) to gather metric and event data.
- **Storage**: Telemetry stores metrics data in the time-series database service (Prometheus).
- **Alarm**: Telemetry uses the Alarming service (aodh) to trigger actions based on rules against metrics or event data collected by Ceilometer.

1.1. DATA COLLECTION SERVICE (CEILOMETER) FOR AUTOSCALING

You can use Ceilometer to collect data about metering information for Red Hat OpenStack Services on OpenShift (RHOSO) components.

The Ceilometer service uses three agents to collect data from RHOSO components:

- **A compute agent (ceilometer-agent-compute)**: Runs on each Compute node and polls for resource use statistics.
- **A central agent (ceilometer-agent-central)**: Runs on the control plane to poll for resource use statistics for resources that are not provided by Compute nodes.
- **A notification agent (ceilometer-agent-notification)**: Runs on the control plane and consumes messages from the message queues to build event and metering data.

1.1.1. Publishers

In Red Hat OpenStack Services on OpenShift (RHOSO), **ceilometer-agent-compute** and **ceilometer-agent-central** metrics use the RabbitMQ bus for transporting metrics to the **ceilometer-agent-notification**. **Ceilometer-agent-notification** uses the TCP publisher to transport metrics to the **sg-core** container, which then exposes them in Prometheus format on the **ceilometer-internal.<namespace>.svc:3000/metrics** endpoint. From this endpoint, the metrics are either scraped by a Prometheus instance provided by the **MetricStorage**, or by any external user-provided Prometheus-compatible system.

1.2. TIME-SERIES DATABASE SERVICE FOR AUTOSCALING

Prometheus is a time-series database that you can use for storing metrics. The Alarming service (aodh) and Orchestration service (heat) use the data that is stored in Prometheus for autoscaling. Prometheus uses the query language, PromQL.

1.3. ALARMING SERVICE (AODH)

You can configure the Alarming service (aodh) to trigger actions based on rules against the metrics data that is collected in Prometheus.

Alarms can be in one of the following states:

- **Ok**: The metric or event is in an acceptable state.
- **Alarm**: The metric or event is outside of the defined **Ok** state.
- **insufficient data**: The alarm state is unknown, for example, if there is no data, or the check has not been executed yet, and so on.

1.4. ORCHESTRATION SERVICE (HEAT) FOR AUTOSCALING

The Orchestration service (heat) uses templates in YAML format. The purpose of a template is to define and create a stack, which is a collection of resources that the Orchestration service creates, and the configuration of the resources. Resources are objects in Red Hat OpenStack Services on OpenShift (RHOSO) and can include compute resources, network configuration, security groups, scaling rules, and custom resources.

CHAPTER 2. MIGRATING AUTOSCALING FROM EARLIER RHOSP DEPLOYMENTS

There are important differences between the autoscaling heat templates on Red Hat OpenStack Services on OpenShift (RHOSO) and previous versions of Red Hat OpenStack Platform (RHOSP):

- Updated alarm types and parameters.
- Groups of instances.
- Alarm queries.

2.1. UPDATING ALARM TYPES AND OBSOLETE PARAMETERS

Previous versions of Red Hat OpenStack Platform (RHOSP) used Gnocchi for metric storage. Because Red Hat OpenStack Services on OpenShift (RHOSO) uses Prometheus for metric storage instead, you need to edit the alarm types in your existing template files.

Procedure

1. Update the value of the **resources.cpu_alarm_high.type** parameter from **OS::Aodh::GnocchiAggregationByResourcesAlarm** to **OS::Aodh::PrometheusAlarm**.
2. Update the value of the **resources.cpu_alarm_low.type** parameter from **OS::Aodh::GnocchiAggregationByResourcesAlarm** to **OS::Aodh::PrometheusAlarm**.
3. Delete the following parameters from the alarm definitions section of the template:
   - metric
   - aggregation_method
   - granularity
   - evaluation_periods
   - resource_type

2.2. GROUPING INSTANCES

For efficient autoscaling, the Alarming service (aodh) must know which instances belong to their respective stacks. This was previously achieved with the **metering.server_group** parameter in the instance metadata. Red Hat OpenStack Services on OpenShift (RHOSO) uses Prometheus, therefore the instance grouping is managed by instance name.

Procedure

1. Include a regex with the instance name in the Prometheus queries for the alarms.
2. Create a stack from your template without alarms.
3. Check the names of instances in the stack.
4. Define the alarms and the queries to work with the names of the instances in the stack.

5. You can also create a custom resource for the instance and make the **name** parameter unique for the Prometheus queries.

The following examples uses the **server_name_prefix** field to propagate instance names in the **instance.yaml** and **autoscaling.yaml** files:

instance.yaml

```
6a7,10
>   server_name_prefix:
>   type: string
>   description: a prefix for each server name
>   default: ""
37a37,38
>     name:
>       list_join: ["", [{get_param: server_name_prefix}, {get_param: OS::stack_name}]]
```

autoscaling.yaml

```
2c2,7
> parameters:
>   server_name_prefix:
>   description: A prefix for servers created by this stack. Can be used in queries.
>   type: string
>   default: autoscaling_server_
11a19
>         server_name_prefix: { get_param: server_name_prefix }
```

2.3. QUERYING THE ALARMS

In Red Hat OpenStack Services on OpenShift (RHOSO), ensure that the queries in the alarm definitions are compatible with PromQL.

Procedure

1. Include **resource_name=~<instance_name_regex>** in the queries to select only the instances in the stack.

2. The **ceilometer_cpu** metric is stored as an accumulated value recording the CPU time total since the instance was created. You can use the PromQL **rate()** function to query only for a specified CPU time or a percentage value of CPU time for a certain duration. In previous Red Hat OpenStack Platform (RHOSP) versions, this was not automatic, but now you can automatically compute the percentage as part of the query.

3. You can use the PromQL **rate()** function like this: **rate(ceilometer_cpu{resource_name=~'server_name_prefix.*'}[<duration>])**. To avoid unstable query results, calculate the <duration> by adding <ceilometer polling interval> to <prometheus scrape interval>.

4. Optional: You can use the **$ openstack metric query** command to check what the new queries return. Ensure that you test your queries. You can check that only instances from the relevant stack are queried, take note of the metric values returned by the queries, and ensure that you set the thresholds in the alarm definitions accordingly.

autoscaling.yaml

```
47,49c50,53
<       list_join:
<         - ''
<         - - {'=': {server_group: {get_param: "OS::stack_id"}}}
---
>       str_replace:
>         template: "(rate(ceilometer_cpu{resource_name=~'server_name_prefix.*'}[150s]))/10000000"
>         params:
>           server_name_prefix: {get_param: server_name_prefix}
68,70c67,70
<       list_join:
<         - ''
<         - - {'=': {server_group: {get_param: "OS::stack_id"}}}
---
>       str_replace:
>         template: "(rate(ceilometer_cpu{resource_name=~'server_name_prefix.*'}[150s]))/10000000"
>         params:
>           server_name_prefix: {get_param: server_name_prefix}
```

CHAPTER 3. CONFIGURING AND DEPLOYING AUTOSCALING

To configure and deploy autoscaling, you must complete the following prerequisites:

1. Create environment templates and a resource registry for autoscaling services before you deploy the cloud for autoscaling.

2. Deploy the cloud.

3.1. ENABLING THE CONTROL PLANE RESOURCES FOR AUTOSCALING

Use the following procedure to enable the control plane resources that you need for autoscaling.

Prerequisites

- A deployed Red Hat OpenStack Services on OpenShift (RHOSO) environment.

Procedure

- Edit the **OpenStackControlPlane** custom resource to enable the Orchestration (heat) and telemetry services. Inside the telemetry section of the .yaml file, enable the Ceilometer, Autoscaling, and MetricStorage objects:

```
$ oc edit oscp
...
  heat:
    enabled: true
...
  telemetry:
    apiOverride: {}
    enabled: true
    template:
      autoscaling:
        enabled: true
        heatInstance: heat
        ...
      ceilometer:
        enabled: true
        ...
      metricStorage:
        enabled: true
        ...
...
```

Verification

1. Verify that the required custom resources are ready:

```
$ oc get heat heat -o jsonpath='{.status.conditions[?(@.type=="Ready")].status}{"\n"}'
True
$ oc get ceilometer ceilometer -o jsonpath='{.status.conditions[?(@.type=="Ready")].status}
{"\n"}'
True
```

```
$ oc get autoscaling autoscaling -o jsonpath='{.status.conditions[?
(@.type=="Ready")].status}{"\n"}'
True
$ oc get metricstorage metric-storage -o jsonpath='{.status.conditions[?
(@.type=="Ready")].status}{"\n"}'
True
```

2. Verify that there is an Alarming service endpoint:

```
$ oc rsh openstackclient openstack endpoint list --service alarming
+----------------------------------+-----------+--------------+--------------+---------+-----------+---------
--------------------------------------+
| ID                               | Region    | Service Name | Service Type | Enabled | Interface | URL
|
+----------------------------------+-----------+--------------+--------------+---------+-----------+---------
--------------------------------------+
| 62d20961a3354daa8adf52db62cc809d | regionOne | aodh         | alarming     | True    |
public    | https://aodh-public-openstack.apps.example.testing |
| 8e526119b01248aba8d304de627a6267 | regionOne | aodh         | alarming     | True    |
internal  | http://aodh-internal.openstack.svc:8042        |
+----------------------------------+-----------+--------------+--------------+---------+-----------+---------
--------------------------------------+
```

3. Verify that there is an Orchestration service endpoint:

```
$ oc rsh openstackclient openstack endpoint list --service orchestration
+----------------------------------+-----------+--------------+---------------+---------+-----------+--------
-------------------------------------------------------------+
| ID                               | Region    | Service Name | Service Type  | Enabled | Interface | URL
|
+----------------------------------+-----------+--------------+---------------+---------+-----------+--------
-------------------------------------------------------------+
| 77190103ae3b479290138ebeca4a54d7 | regionOne | heat         | orchestration | True    |
internal  | http://heat-api-internal.openstack.svc:8004/v1/%(tenant_id)s        |
| c302ee5ff5cc45709d20b2d4a872d08f | regionOne | heat         | orchestration | True    |
public    | https://heat-api-public-openstack.apps.example.testing/v1/%(tenant_id)s |
+----------------------------------+-----------+--------------+---------------+---------+-----------+--------
-------------------------------------------------------------+
```

4. Verify that Prometheus is enabled and can scrape each endpoint. Confirm that the value in the output is 1 for each row, except for the Prometheus container, which is 0 when TLS is enabled:

```
$ oc rsh openstackclient openstack metric query up --disable-rbac -c instance -c container -c
value
+---------------------+---------------------+-------+
| instance            | container           | value |
+---------------------+---------------------+-------+
| 10.217.1.27:9093    | alertmanager        | 1     |
| 10.217.1.27:9093    | prometheus          | 0     |
| 10.217.1.52:3000    | proxy-httpd         | 1     |
| 10.217.0.169:3000   |                     | 1     |
| 192.168.122.100:9100 |                    | 1     |
| 192.168.122.101:9100 |                    | 1     |
+---------------------+---------------------+-------+
```

CHAPTER 4. USING THE ORCHESTRATION SERVICE (HEAT) FOR AUTOSCALING

After you deploy the services required to provide autoscaling, you must configure the environment so that the Orchestration service (heat) can manage instances for autoscaling.

Prerequisites

- A deployed Red Hat OpenStack Services on OpenShift (RHOSO) environment.
- Ceilometer, Autoscaling, MetricStorage, and the Orchestration service are enabled on the control plane.

4.1. CONFIGURING A HEAT TEMPLATE FOR AUTOSCALING

Procedure

You can configure an Orchestration service (heat) template to create the instances, and configure alarms that create and scale instances when triggered. This procedure uses example values that might differ from your environment.

Prerequisites

- You have deployed the cloud with the autoscaling services.

Procedure

1. Access the **openstackclient** pod:

```
$ oc rsh openstackclient
```

2. Create a directory for your templates, for example **/tmp/templates/**:

```
$ mkdir -p /tmp/templates
```

3. Create an instance configuration template, for example **/tmp/templates/instance.yaml**:
4. Add the following configuration to your **instance.yaml** file:

```
$ cat <<EOF > /tmp/templates/instance.yaml
heat_template_version: wallaby
description: Template to control scaling of VNF instance

parameters:
  metadata:
    type: json
  server_name_prefix:
    type: string
    description: a prefix for each server name
    default: ""
  image:
    type: string
    description: image used to create instance
```

```
    default: cirros
  flavor:
    type: string
    description: instance flavor to be used
    default: m1.small
  network:
    type: string
    description: project network to attach instance to
    default: private
  external_network:
    type: string
    description: network used for floating IPs
    default: public

resources:
  vnf:
    type: OS::Nova::Server
    properties:
      flavor: {get_param: flavor}
      image: { get_param: image }
      metadata: { get_param: metadata }
      networks:
        - port: { get_resource: port }
      name:
        list_join: ["", [{get_param: server_name_prefix}, {get_param: OS::stack_name}]]

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

  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 }
EOF
```

5. Create the resource to reference in the heat template:

```
$ cat <<EOF > /tmp/templates/resources.yaml
resource_registry:
  "OS::Nova::Server::VNF": /tmp/templates/instance.yaml
EOF
```

6. Create the deployment template for the Orchestration service to control instance scaling:

```
$ cat <<EOF > /tmp/templates/autoscaling.yaml
```

```
heat_template_version: wallaby
description: Example auto scale group, policy and alarm
parameters:
  server_name_prefix:
    description: A prefix for servers created by this stack. Can be used in queries.
    type: string
    default: autoscaling_server_
resources:
  autoscalinggroup:
    type: OS::Heat::AutoScalingGroup
    properties:
      cooldown: 300
      desired_capacity: 1
      max_size: 3
      min_size: 1
      resource:
        # Configure the resource to be autoscaled
        type: OS::Nova::Server::VNF
        properties:
          server_name_prefix: { get_param: server_name_prefix }
          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: autoscalinggroup }
      cooldown: 300
      scaling_adjustment: 1

  scaledown_policy:
    type: OS::Heat::ScalingPolicy
    properties:
      adjustment_type: change_in_capacity
      auto_scaling_group_id: { get_resource: autoscalinggroup }
      cooldown: 300
      scaling_adjustment: -1

  cpu_alarm_high:
    type: OS::Aodh::PrometheusAlarm
    properties:
      description: Scale up if CPU > 80%
      threshold: 80 # 80%
      comparison_operator: gt
      alarm_actions:
        - str_replace:
            template: trust+url
            params:
              url: {get_attr: [scaleup_policy, signal_url]}
      query:
        str_replace:
          # ceilometer_cpu metric is in ns. Divide the rate by 10000000 to get percentage
          # The time duration in [] should be higher than ceilometer polling interval
          # the recommended value is {ceilometer polling} + {prometheus scrape interval}. This
          # is 150s by default.
          template: "(rate(ceilometer_cpu{resource_name=~'server_name_prefix.*'}
```

```
[150s]))/10000000"
          params:
            server_name_prefix: {get_param: server_name_prefix}

  cpu_alarm_low:
    type: OS::Aodh::PrometheusAlarm
    properties:
      description: Scale up if CPU < 20%
      threshold: 20 # 20%
      comparison_operator: lt
      alarm_actions:
        - str_replace:
            template: trust+url
            params:
              url: {get_attr: [scaledown_policy, signal_url]}
      query:
        str_replace:
          # ceilometer_cpu metric is in ns. Divide the rate by 10000000 to get percentage
          # The time duration in [] should be higher than ceilometer polling interval
          # the recommended value is {ceilometer polling} + {prometheus scrape interval}. This
          # is 150s by default.
          template: "(rate(ceilometer_cpu{resource_name=~'server_name_prefix.*'}
[150s]))/10000000"
          params:
            server_name_prefix: {get_param: server_name_prefix}

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

  scaledown_policy_signal_url:
    value: {get_attr: [scaledown_policy, signal_url]}
EOF
```

4.2. CREATING THE STACK DEPLOYMENT FOR AUTOSCALING

Create the stack deployment for autoscaling. This procedure uses example values that might differ from your environment.

Prerequisites

- You have configured a heat template for automatically scaling instances.

Procedure

1. Access the openstack client pod:

```
$ oc rsh openstackclient
```

2. Create the stack:

```
$ openstack stack create -t /tmp/templates/autoscaling.yaml -e
/tmp/templates/resources.yaml stack1
```

Verification

1. Verify that you created the stack successfully:

```
$ openstack stack show stack1 -c id -c stack_status
+--------------+--------------------------------------+
| Field        | Value                                |
+--------------+--------------------------------------+
| id           | 9ac44c45-4917-4cd5-a713-9ef7828d8457 |
| stack_status | CREATE_COMPLETE                      |
+--------------+--------------------------------------+
```

2. Verify that you created the stack resources, including alarms, scaling policies, and the autoscaling group:

```
$ export STACK_ID=$(openstack stack show stack1 -c id -f value)

$ openstack stack resource list $STACK_ID
+------------------+--------------------------------------+---------------------------+-----------------+----
------------------+
| resource_name    | physical_resource_id                 | resource_type             |
resource_status | updated_time         |
+------------------+--------------------------------------+---------------------------+-----------------+----
------------------+
| scaleup_policy   | 3cfb2a746dcf4fb6b3284b6c164e4ff5     | OS::Heat::ScalingPolicy   |
CREATE_COMPLETE | 2024-03-26T09:17:54Z |
| scaledown_policy | ef60360ae7564abda088e67b3cc542f4     | OS::Heat::ScalingPolicy   |
CREATE_COMPLETE | 2024-03-26T09:17:54Z |
| cpu_alarm_low    | 95503054-aada-457c-b586-f98a88b00962 |
OS::Aodh::PrometheusAlarm  | CREATE_COMPLETE | 2024-03-26T09:17:54Z |
| cpu_alarm_high   | 65be604f-1262-4567-851d-ac28ae1bb178 |
OS::Aodh::PrometheusAlarm  | CREATE_COMPLETE | 2024-03-26T09:17:54Z |
| autoscalinggroup | 8c4fea31-3beb-47b7-841d-fe82fb284628 | OS::Heat::AutoScalingGroup
| CREATE_COMPLETE | 2024-03-26T09:17:54Z |
+------------------+--------------------------------------+---------------------------+-----------------+----
------------------+
```

3. Verify that an instance was launched by the stack creation:

```
$ openstack server list --long | grep $STACK_ID
| 9e2e9266-d69e-4de1-8b3a-e75f7111d1d9 | autoscaling_server_stack1-autoscalinggroup-
oqmxiukze6dg-hdtuapbjxcow-q366nn4k5utr | ACTIVE | None       | Running     |
private=192.168.0.209, 192.168.122.205 | cirros     | 239959f7-3d16-4744-8eca-
435cdfc1a4cf | m1.small | nova               | edpm-compute-0.ctlplane.example.com |
metering.server_group='9ac44c45-4917-4cd5-a713-9ef7828d8457' | UP          |
```

4. Verify that you created the alarms for the stack:

   a. List the alarm IDs. If the **state** of the alarm is **insufficient data**, try checking again after the data collects:

```
$ openstack alarm list
+--------------------------------------+------------+----------------------------------+-------+-------
---+---------+
| alarm_id                             | type       | name                             | state | severity |
```

```
enabled |
+--------------------------------------+-----------+---------------------------------+-------+-------
---+---------+
| 65be604f-1262-4567-851d-ac28ae1bb178 | prometheus | stack1-cpu_alarm_high-
q5yw2fg4h2vh | ok    | low       | True    |
| 95503054-aada-457c-b586-f98a88b00962 | prometheus | stack1-cpu_alarm_low-
hjmpszetdpsw  | alarm | low       | True    |
+--------------------------------------+-----------+---------------------------------+-------+-------
---+---------+
```

b. List the resources for the stack and note the **physical_resource_id** values for the **cpu_alarm_high** and **cpu_alarm_low** resources:

```
$ openstack stack resource list $STACK_ID
+-----------------+--------------------------------------+--------------------------+----------------
+---------------------+
| resource_name    | physical_resource_id                 | resource_type            |
resource_status | updated_time         |
+-----------------+--------------------------------------+--------------------------+----------------
+---------------------+
| scaleup_policy   | 3cfb2a746dcf4fb6b3284b6c164e4ff5     | OS::Heat::ScalingPolicy    |
CREATE_COMPLETE | 2024-03-26T09:17:54Z |
| scaledown_policy | ef60360ae7564abda088e67b3cc542f4     | OS::Heat::ScalingPolicy
| CREATE_COMPLETE | 2024-03-26T09:17:54Z |
| cpu_alarm_low    | 95503054-aada-457c-b586-f98a88b00962 |
OS::Aodh::PrometheusAlarm  | CREATE_COMPLETE | 2024-03-26T09:17:54Z |
| cpu_alarm_high   | 65be604f-1262-4567-851d-ac28ae1bb178 |
OS::Aodh::PrometheusAlarm  | CREATE_COMPLETE | 2024-03-26T09:17:54Z |
| autoscalinggroup | 8c4fea31-3beb-47b7-841d-fe82fb284628 |
OS::Heat::AutoScalingGroup | CREATE_COMPLETE | 2024-03-26T09:17:54Z |
+-----------------+--------------------------------------+--------------------------+----------------
+---------------------+
```

5. Verify that metric resources and measurements exist for the stack:

```
$ openstack metric query "ceilometer_cpu{resource_name=~'autoscaling_server.*'}"
+---------------+---------+--------------------------------------+---------------------------------+--------
-----------------------------+------------------------------------------------------------------------------
-------------------+-----------+------+------+---------------------------------+-------------------------
------------------------------+-------------+
| __name__       | counter | cpu                                  | project                         | resource
| resource_name                                                                | service    | type | unit
| user                           | vm_instance                                  | value       |
+---------------+---------+--------------------------------------+---------------------------------+--------
-----------------------------+------------------------------------------------------------------------------
-------------------+-----------+------+------+---------------------------------+-------------------------
------------------------------+-------------+
| ceilometer_cpu | cpu     | 9e2e9266-d69e-4de1-8b3a-e75f7111d1d9 |
f680a9fa467d47449fc84aae3ef1f911 | 9e2e9266-d69e-4de1-8b3a-e75f7111d1d9 |
autoscaling_server_stack1-autoscalinggroup-oqmxiukze6dg-hdtuapbjxcow-
q366nn4k5utr:instance-00000037 | ceilometer | cpu  | ns   |
057899f318f7496bb4fd1acc1c0a553c |
e2a6916584b9a7b6c711f2563842f78205af88bd7e5abaaa84f7b49b | 26660000000 |
+---------------+---------+--------------------------------------+---------------------------------+--------
```

```
------------------------------+--------------------------------------------------------------------------------
-------------------+------------+------+------+---------------------------------+--------------------------
------------------------------+------------+
```

CHAPTER 5. TESTING AND TROUBLESHOOTING AUTOSCALING

Use the Orchestration service (heat) to automatically scale instances up and down based on threshold definitions. To troubleshoot your environment, you can look for errors in the log files and history records.

5.1. TESTING AUTOMATIC SCALING UP OF INSTANCES

You can use the Orchestration service (heat) to scale instances automatically based on the **cpu_alarm_high** threshold definition. When the CPU use reaches a value defined in the **threshold** parameter, another instance starts up to balance the load. The **threshold** value in the **template.yaml** file is set to 80%. This procedure uses example values that might differ from your environment.

Procedure

1. Access the client pod:

```
$ oc rsh openstackclient
```

2. Log in to the instance. The default password for cirros images is "gocubsgo":

```
$ ssh cirros@192.168.122.8
```

3. Run multiple **dd** commands to generate the load:

```
[instance ~]$ sudo dd if=/dev/zero of=/dev/null &
[instance ~]$ sudo dd if=/dev/zero of=/dev/null &
[instance ~]$ sudo dd if=/dev/zero of=/dev/null &
```

4. Exit from the running instance and return to the host.

5. After you run the **dd** commands, you can expect to have 100% CPU use in the instance after some time. Verify that the alarm has been triggered:

```
$ openstack alarm list
+--------------------------------------+------------+----------------------------------+-------+----------+-
--------+
| alarm_id                             | type       | name                             | state | severity | enabled |
+--------------------------------------+------------+----------------------------------+-------+----------+-
--------+
| 3000766d-a7fc-4187-98ea-cafd1239c160 | prometheus | stack1-cpu_alarm_low-
n3iy7oaffq4i  | ok    | low      | True    |
| 899f8242-96dd-4bad-a7f9-0b9526ea6032 | prometheus | stack1-cpu_alarm_high-
riua4sthkvcs | alarm | low      | True    |
+--------------------------------------+------------+----------------------------------+-------+----------+-
--------+
```

6. After approximately 60 seconds, Orchestration starts another instance and adds it to the group. To verify that an instance has been created, enter the following command:

```
$ openstack server list
+--------------------------------------+-----------------------------------------------------------------------
-----------+--------+-----------------------------------------+--------+----------+
```

```
| ID                                   | Name                                                                  | Status |
Networks                             | Image  | Flavor   |
+--------------------------------------+-----------------------------------------------------------------------
----------+--------+--------------------------------------+--------+----------+
| 53ffb3c5-a425-4929-b01d-50d653fe23f8 | autoscaling_server_stack1-autoscalinggroup-
oqmxiukze6dg-qn7drottgjek-3oiecx4gkowi | ACTIVE | private=192.168.0.116,
192.168.122.204 | cirros | m1.small |
| 9e2e9266-d69e-4de1-8b3a-e75f7111d1d9 | autoscaling_server_stack1-autoscalinggroup-
oqmxiukze6dg-hdtuapbjxcow-q366nn4k5utr | ACTIVE | private=192.168.0.209,
192.168.122.205 | cirros | m1.small |
+--------------------------------------+-----------------------------------------------------------------------
----------+--------+--------------------------------------+--------+----------+
```

7. (Optional) After another short period of time, observe if the Orchestration service autoscales to three instances. The configuration is set to a maximum of three instances. Verify there are three instances:

```
$ openstack server list
+--------------------------------------+-------------------------------------------------------+--------+------
------+-------------+--------------------------------------+
| ID                                   | Name                                                  | Status | Task State | Power
State | Networks                             |
+--------------------------------------+-------------------------------------------------------+--------+------
------+-------------+--------------------------------------+
| 477ee1af-096c-477c-9a3f-b95b0e2d4ab5 | ex-3gax-4urpikl5koff-yrxk3zxzfmpf-server-
2hde4tp4trnk | ACTIVE | -          | Running     | internal1=10.10.10.13, 192.168.122.17 |
| e1524f65-5be6-49e4-8501-e5e5d812c612 | ex-3gax-5f3a4og5cwn2-png47w3u2vjd-server-
vaajhuv4mj3j | ACTIVE | -          | Running     | internal1=10.10.10.9, 192.168.122.8   |
| 6c88179e-c368-453d-a01a-555eae8cd77a | ex-3gax-fvxz3tr63j4o-36fhftuja3bw-server-
rhl4sqkjuy5p | ACTIVE | -          | Running     | internal1=10.10.10.5, 192.168.122.5   |
+--------------------------------------+-------------------------------------------------------+--------+------
------+-------------+--------------------------------------+
```

5.2. TESTING AUTOMATIC SCALING DOWN OF INSTANCES

You can use the Orchestration service (heat) to automatically scale down instances based on the **cpu_alarm_low** threshold. In this example, the instances are scaled down when CPU usage is below 20%.

Procedure

1. From within the workload instance, terminate the running **dd** processes and observe Orchestration begin to scale the instances back down.

```
$ sudo killall dd
```

2. Exit from the running instance and return to the host.

3. When you stop the **dd** processes, this triggers the **cpu_alarm_low event** alarm. As a result, Orchestration begins to automatically scale down and remove the instances. Verify that the corresponding alarm has triggered:

```
$ openstack alarm list
+--------------------------------------+--------------------------------------------+-------------------------
```

```
----------+-------+----------+---------+
| alarm_id                             | type                                        | name                          | state |
severity | enabled |
+--------------------------------------+---------------------------------------------+-------------------------
----------+-------+----------+---------+
| 022f707d-46cc-4d39-a0b2-afd2fc7ab86a | gnocchi_aggregation_by_resources_threshold |
example-cpu_alarm_high-odj77qpbld7j | ok    | low      | True    |
| 46ed2c50-e05a-44d8-b6f6-f1ebd83af913 | gnocchi_aggregation_by_resources_threshold |
example-cpu_alarm_low-m37jvnm56x2t  | alarm | low      | True    |
+--------------------------------------+---------------------------------------------+-------------------------
----------+-------+----------+---------+
```

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**.

5.3. TROUBLESHOOTING FOR AUTOSCALING

If necessary, you can look for errors in the log files and history records.

Procedure

1. Access the openstack client pod:

```
$ oc rsh openstackclient
```

2. To retrieve information on state transitions, 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.scaledown_policy]: CREATE_COMPLETE  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
recent: 95.4080102993)
2017-03-06 11:25:43Z [example.scaleup_policy]: SIGNAL_COMPLETE  alarm state
changed from alarm to alarm (Remaining as alarm due to 1 samples outside threshold, most
recent: 95.8869217299)
2017-03-06 11:33:25Z [example.scaledown_policy]: SIGNAL_COMPLETE  alarm state
changed from ok to alarm (Transition to alarm due to 1 samples outside threshold, most
recent: 2.73931707966)
2017-03-06 11:39:15Z [example.scaledown_policy]: SIGNAL_COMPLETE  alarm state
changed from alarm to alarm (Remaining as alarm due to 1 samples outside threshold, most
recent: 2.78110858552)
```

3. Read the alarm history log:

```
$ openstack alarm-history show 022f707d-46cc-4d39-a0b2-afd2fc7ab86a
+--------------------------+-----------------+-------------------------------------------------------------
-------------------------------------+----------------------------------+
| timestamp                | type            | detail
| event_id                             |
+--------------------------+-----------------+-------------------------------------------------------------
-------------------------------------+----------------------------------+
| 2017-03-06T11:32:35.510000 | state transition | {"transition_reason": "Transition to ok due
to 1 samples inside threshold, most recent:              | 25e0e70b-3eda-466e-abac-
42d9cf67e704 |
|                          |                  | 2.73931707966", "state": "ok"}
|                                    |
| 2017-03-06T11:17:35.403000 | state transition | {"transition_reason": "Transition to alarm
due to 1 samples outside threshold, most recent:          | 8322f62c-0d0a-4dc0-9279-
435510f81039 |
|                          |                  | 95.0964497325", "state": "alarm"}
|                                    |
| 2017-03-06T11:15:35.723000 | state transition | {"transition_reason": "Transition to ok due
to 1 samples inside threshold, most recent:              | 1503bd81-7eba-474e-b74e-
ded8a7b630a1 |
|                          |                  | 3.59330523447", "state": "ok"}
|                                    |
| 2017-03-06T11:13:06.413000 | creation         | {"alarm_actions":
["trust+http://fca6e27e3d524ed68abdc0fd576aa848:delete@192.168.122.126:8004/v1/fd |
224f15c0-b6f1-4690-9a22-0c1d236e65f6 |
|                          |                  |
1c345135be4ee587fef424c241719d/stacks/example/d9ef59ed-b8f8-4e90-bd9b-
|                                    |
|                          |                  | ae87e73ef6e2/resources/scaleup_policy/signal"], "user_id":
"a85f83b7f7784025b6acdc06ef0a8fd8",      |                                  |
|                          |                  | "name": "example-cpu_alarm_high-odj77qpbld7j", "state":
"insufficient data", "timestamp":           |                                  |
|                          |                  | "2017-03-06T11:13:06.413455", "description": "Scale up if
CPU > 80%", "enabled": true,               |                                  |
|                          |                  | "state_timestamp": "2017-03-06T11:13:06.413455", "rule":
{"evaluation_periods": 1, "metric":        |                                  |
|                          |                  | "cpu_util", "aggregation_method": "mean", "granularity": 300,
"threshold": 80.0, "query": "{\"=\":   |                                  |
|                          |                  | {\"server_group\": \"d9ef59ed-b8f8-4e90-bd9b-
ae87e73ef6e2\"}}", "comparison_operator": "gt",        |                                  |
|                          |                  | "resource_type": "instance"}, "alarm_id": "022f707d-46cc-
4d39-a0b2-afd2fc7ab86a",                   |                                  |
|                          |                  | "time_constraints": [], "insufficient_data_actions": null,
"repeat_actions": true, "ok_actions":    |                                  |
|                          |                  | null, "project_id": "fd1c345135be4ee587fef424c241719d",
"type":                                  |                                  |
|                          |                  | "gnocchi_aggregation_by_resources_threshold", "severity":
"low"}                                   |                                  |
+--------------------------+-----------------+-------------------------------------------------------------
-------------------------------------+----------------------------------
```

4. Exit the openstack client pod:

```
$ exit
```

5. To view the records of scale-out or scale-down operations that the Orchestration service (heat) collects for the existing stack, you can use the **awk** command to parse the heat pod logs.

```
$ oc logs -l component=engine,service=heat --tail=-1 | awk '/Updating stack/,/Stack CREATE completed successfully/ {print $0}'
```

6. To view information about the Alarming service (aodh), examine the logs of the **aodh-evaluator** container.

```
$ oc logs aodh-0 -c aodh-evaluator | grep -i alarm | grep -i transition | grep -v DEBUG
```

5.4. USING CPU TELEMETRY VALUES FOR AUTOSCALING THRESHOLD WHEN USING THE RATE FUNCTION

When using the **OS::Heat::Autoscaling** heat orchestration template (HOT) and setting a threshold value for CPU, the value is expressed in nanoseconds of CPU time, which is a dynamic value based on the number of virtual CPUs allocated to the instance workload. You can calculate and express the CPU nanosecond value as a percentage when using the **rate** function when querying prometheus.

5.4.1. Calculating CPU telemetry values as a percentage

CPU telemetry is stored in Prometheus as CPU utilization in nanoseconds. When using CPU telemetry to define autoscaling thresholds, it is useful to express the values as a percentage of CPU utilization since that is more natural when defining the threshold values. When defining the scaling policies used as part of an autoscaling group, you can define thresholds as percentages and calculate the CPU utilization as a percentage, instead of in nanoseconds. * Divide the rate of increase of the metric by 1,000,000,000. * Divide the answer by the number of vCPUs * Multiply the answer by 100 to get a percentage.

Use this formula to calculate this value in the PromQL queries: "(rate(ceilometer_cpu{resource_name=~<server_name_prefix>.*'}[150s])) / 1000000000 / <number_of_vCPUs> * 100".

- Replace <server_name_prefix> with the name of your server name prefix.
- Replace <number_of_vCPUs> with the the number of vCPUs.

5.4.2. Displaying instance workload vCPU as a percentage

You can display the CPU telemetry data as a percentage rather than the nanosecond values for instances by using the **openstack metric query** command.

Prerequisites

- Create a heat stack using the autoscaling group resource that results in an instance workload.

Procedure

1. Access the openstack client pod:

```
$ oc rsh openstackclient
```

2. Retrieve the **server_name_prefix** of the autoscaling group heat stack:

```
$ openstack stack show stack1 -c parameters -c stack_status
+--------------+-----------------------------------------------------+
| Field        | Value                                               |
+--------------+-----------------------------------------------------+
| stack_status | CREATE_COMPLETE                                     |
| parameters   | OS::project_id: f680a9fa467d47449fc84aae3ef1f911    |
|              | OS::stack_id: 0d6e19a9-2e1c-4728-b9ed-320da60c3c9d |
|              | OS::stack_name: stack1                              |
|              | server_name_prefix: autoscaling_server_             |
|              |                                                     |
+--------------+-----------------------------------------------------+
```

3. Return the metrics rate of increase with the value calculated as a percentage. The rate of increase is returned as a value of nanoseconds of CPU time by which the metric increased in our given window.You divide that number by 1000000000 to get the value in seconds. That value is then converted to a percentage by multiplying by 100. Finally, you divide the total value by the number of vCPU provided by the flavor assigned to the instance, in this example a value of 1 vCPU, providing a value expressed as a percentage of CPU time. Use **server_name_prefix** for the value of the 'resource_name' option in the query to get only the results from your stack:

```
$ openstack metric query "rate(ceilometer_cpu{resource_name=~'autoscaling_server.*'}
[150s])/1000000000*100/1" -c resource_name -c value
+-----------------------------------------------------------------------------------------------------+---------
----------+
| resource_name                                                                                       | value             |
+-----------------------------------------------------------------------------------------------------+---------
----------+
| autoscaling_server_stack1-autoscalinggroup-5yzjn6ow4x6v-ijlwqqozjuva-
gedovjdujsqf:instance-0000003a | 92.60000000000001  |
| autoscaling_server_stack1-autoscalinggroup-5yzjn6ow4x6v-gmu2apozwluw-
t7nqx63tybxh:instance-0000003c | 0.6333333333333333 |
| autoscaling_server_stack1-autoscalinggroup-5yzjn6ow4x6v-i3ml7xq7wznl-
n2ke5abj45ai:instance-0000003b | 93.38333333333334  |
+-----------------------------------------------------------------------------------------------------+---------
----------+
```

4. You can look at historical values by adding the offset keyword to the query. The following gives similar output as the step before, but with data 10 minutes in the past:

```
$ openstack metric query "rate(ceilometer_cpu{resource_name=~'autoscaling_server.*'}
[150s] offset 10m)/1000000000*100/1" -c resource_name -c value
+-----------------------------------------------------------------------------------------------------+---------
----------+
| resource_name                                                                                       | value             |
+-----------------------------------------------------------------------------------------------------+---------
----------+
| autoscaling_server_stack1-autoscalinggroup-5yzjn6ow4x6v-ijlwqqozjuva-
gedovjdujsqf:instance-0000003a | 45.525             |
| autoscaling_server_stack1-autoscalinggroup-5yzjn6ow4x6v-gmu2apozwluw-
t7nqx63tybxh:instance-0000003c | 0.6083333333333333 |
| autoscaling_server_stack1-autoscalinggroup-5yzjn6ow4x6v-i3ml7xq7wznl-
```

```
n2ke5abj45ai:instance-0000003b | 44.79166666666667  |
+-----------------------------------------------------------------------------------------------+---------
----------+
```

5.4.3. Retrieving available telemetry for an instance workload

Retrieve the available telemetry for an instance workload and express the vCPU utilization as a percentage.

Prerequisites

- Create a heat stack using the autoscaling group resource that results in an instance workload.

Procedure

1. Access the openstack client pod:

```
$ oc rsh openstackclient
```

2. Retrieve the ID of the autoscaling group heat stack:

```
$ openstack stack show stack1 -c id -c stack_status
+--------------+--------------------------------------+
| Field        | Value                                |
+--------------+--------------------------------------+
| id           | 0d6e19a9-2e1c-4728-b9ed-320da60c3c9d |
| stack_status | CREATE_COMPLETE                      |
+--------------+--------------------------------------+
```

3. Set the value of the stack ID to an environment variable:

```
$ export STACK_ID=$(openstack stack show stack1 -c id -f value)
```

4. Retrieve the ID of the workload instance that you want to return data for. You can use the server list long form and filter for instances that are part of the autoscaling group:

```
$ openstack server list --long --fit-width -c ID -c Name -c Properties | grep
"metering.server_group='$STACK_ID'"
| 94a776a6-ab74-41ec-9c21-26d3ca0a0138 | autoscaling_server_stack1-autoscalinggroup-
l3qthrzpad67-nonm7balfxrn-sx6oebhyi3qo | metering.server_group='168364ae-afe2-4275-
af28-aae01ee63b00' |
```

5. Set the instance ID for one of the returned instance workload names:

```
$ INSTANCE_ID=94a776a6-ab74-41ec-9c21-26d3ca0a0138
```

6. Verify that the metrics are stored for the instance resource ID. If no metrics are available, not enough time has elapsed since the instance was created. If enough time has elapsed, you can check the logs for the data collection service by executing **podman logs ceilometer_agent_compute** on the Compute node where the instance is running, and you can check the logs for the time-series database service Prometheus by executing **oc logs prometheus-metric-storage-0 -c prometheus**:

```
$ openstack metric query "ceilometer_cpu{resource='$INSTANCE_ID'}" -c resource -c
resource_name -c value -c unit
+--------------------------------------+----------------------------------------------------------------------------
----------------------------+------+-------------+
| resource                             | resource_name
| unit | value       |
+--------------------------------------+----------------------------------------------------------------------------
----------------------------+------+-------------+
| a19f11ba-7834-4143-bf83-766af8199f04 | autoscaling_server_stack1-autoscalinggroup-
5yzjn6ow4x6v-gmu2apozwluw-t7nqx63tybxh:instance-0000003c | ns   | 204290000000 |
+--------------------------------------+----------------------------------------------------------------------------
----------------------------+------+-------------+
```

7. Display gathered ceilometer metrics:

```
$ openstack metric list | grep ceilometer
| ceilometer_cpu                          |
| ceilometer_disk_device_allocation       |
| ceilometer_disk_device_capacity         |
| ceilometer_disk_device_read_bytes       |
| ceilometer_disk_device_read_latency     |
| ceilometer_disk_device_read_requests    |
| ceilometer_disk_device_usage            |
| ceilometer_disk_device_write_bytes      |
| ceilometer_disk_device_write_latency    |
| ceilometer_disk_device_write_requests   |
| ceilometer_image_size                   |
| ceilometer_memory_usage                 |
| ceilometer_network_incoming_bytes       |
| ceilometer_network_incoming_bytes_delta |
| ceilometer_network_incoming_packets     |
| ceilometer_network_incoming_packets_drop |
| ceilometer_network_incoming_packets_error |
| ceilometer_network_outgoing_bytes       |
| ceilometer_network_outgoing_bytes_delta |
| ceilometer_network_outgoing_packets     |
| ceilometer_network_outgoing_packets_drop |
| ceilometer_network_outgoing_packets_error |
```

8. Retrieve the number of vCPU cores applied to the workload instance by reviewing the configured flavor for the instance workload:

```
$ openstack server show $INSTANCE_ID -cflavor -f value
m1.small (692533fe-0912-417e-b706-5d085449db53)

$ openstack flavor show 692533fe-0912-417e-b706-5d085449db53 -c vcpus -f value
1
```

9. Return the metrics rate of increase with the value calculated as a percentage. The rate of increase is returned as a value of nanoseconds of CPU time by which the metric increased in our given window. You divide that number by 1000000000 to get the value in seconds. That value is then converted to a percentage by multiplying by 100. Finally, you divide the total value by the number of vCPU provided by the flavor assigned to the instance, in this example a value of 1 vCPU, providing a value expressed as a percentage of CPU time:

```
$ openstack metric query "rate(ceilometer_cpu{resource=~'$INSTANCE_ID'}
[150s])/1000000000*100/1" -c resource_name -c value -c resource
+--------------------------------------+----------------------------------------------------------------------------
----------------------------+--------------------+
| resource                             | resource_name
| value              |
+--------------------------------------+----------------------------------------------------------------------------
----------------------------+--------------------+
| a19f11ba-7834-4143-bf83-766af8199f04 | autoscaling_server_stack1-autoscalinggroup-
5yzjn6ow4x6v-gmu2apozwluw-t7nqx63tybxh:instance-0000003c | 0.6166666666666667 |
+--------------------------------------+----------------------------------------------------------------------------
----------------------------+--------------------+
```

10. Add the offset keyword to the query to view historical results. The following command provides similar output to the step before, but with data 10 minutes in the past:

```
$ openstack metric query "rate(ceilometer_cpu{resource=~'$INSTANCE_ID'}[150s] offset
10m)/1000000000*100/1" -c resource_name -c value -c resource
+--------------------------------------+----------------------------------------------------------------------------
----------------------------+--------------------+
| resource                             | resource_name
| value              |
+--------------------------------------+----------------------------------------------------------------------------
----------------------------+--------------------+
| a19f11ba-7834-4143-bf83-766af8199f04 | autoscaling_server_stack1-autoscalinggroup-
5yzjn6ow4x6v-gmu2apozwluw-t7nqx63tybxh:instance-0000003c | 0.6083333333333333 |
+--------------------------------------+----------------------------------------------------------------------------
----------------------------+--------------------+
```