CloudKitty Hands-on

Let’s meet your hosts!

Today's speakers

- **Luka Peschke** (Objectif Libre)

  Cloud Consultant / Core CloudKitty

- **Ludovic Logiou** (Objectif Libre)

  Cloud Consultant

- **Christophe Sauthier** (Objectif Libre)

  CEO of Objectif Libre / PTL and co-Father of CloudKitty

Objectif Libre in a Nutshell

100% Open Infrastructure company based in **France** (Toulouse/Paris) and **Sweden** (Stockholm)

By your side all along your cloud/cloud native projects:

- Setting up / Audits
- Management
- Trainings (6 courses on OpenStack by instance)
- Customisation

We operate clouds of our customers world-wide

Objectif Libre an **Innovative and commited** company

- Main developpers of CloudKitty.
- Contribution whenevr possible (for a long time around the Top20 of OpenStack)

Today's tools

Ceilometer

Openstack measurement project

Ceilometer (part of the Telemetry project) collects the usage of all resources in an OpenStack cloud.

It stores **metrics**, like CPU and RAM usage, amount of volume storage used...

Architecture

Ceilometer is composed of several parts. The main ones are:

- `ceilometer-collector` (controller): reads AMQP messages from other components.
- `ceilometer-agent-central` (controller): polls some metrics directly.
- `ceilometer-agent-compute` (compute node): fetches information related to instances.

Gnocchi

Timeseries Database

Gnocchi was initially created as a part of the Telemetry project to address Ceilometer's storage and performance issues. It is independent since March 2017.

It stores and **aggregates** measures for metrics.

Gnocchi has a **resource** notion. Each resource can have several associated metrics. (For example, an `instance` resource has `cpu`, `vcpus` and `memory` metrics associated).

Ceilometer **publishes** measures to Gnocchi (but it does also support other databases).

Architecture

Gnocchi is composed of the following parts:

- An HTTP REST API: Used to push and retrieve data.
- A processing daemon (`gnocchi-metricd`): Performs aggregation, metric cleanup...
- A statsd-compatible daemon (optional): Receives data via TCP rather than the API.

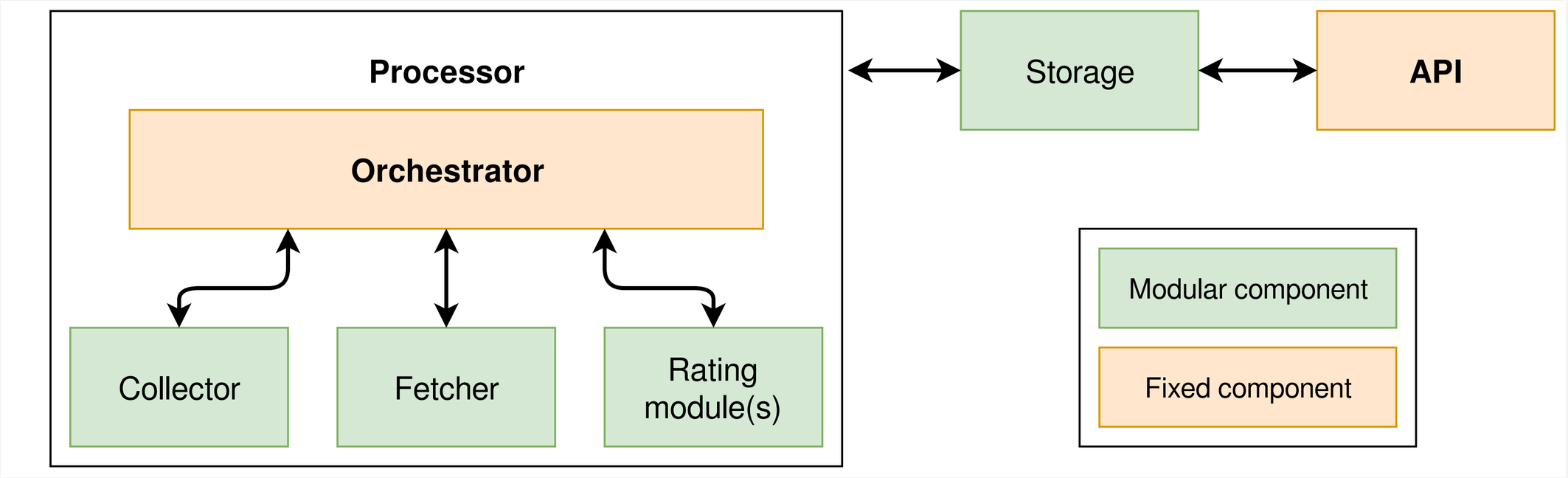
CloudKitty

Rating component for OpenStack and co

CloudKitty was initially created in order to allow rating of Ceilometer metrics.

Today, CloudKitty can be used with Gnocchi and Monasca. Starting with the Rocky release, it will be possible to use it outside of an OpenStack context.

Architecture



CloudKitty works the following way:

- The fetcher fetches scopes on which information should be gathered (these scopes are tenants in the case of OpenStack).
- The collector collects measures from somewhere (gnocchi in our case) for the given scopes.
- The collected data is passed to CloudKitty's rating module(s) (several rating modules can be used simultaneously). The modules apply user-defined rating rules to the data.
- The rated data is pushed to CloudKitty's storage backend (sqlalchmemy in our case).

Installing our components

Get your browser

Slides: https://olib.re/vancouver-ck-handson

Do not open them in your browser or you'll experience copy/paste issues !

Pick an IP: https://olib.re/vancouver-ck-handson-ip

SSH into your instance

Start with SSHing into your instance. The user is « centos » and the password is « v4nc0uv3r ».

Once you're connected, identify yourself:

```
$ source ~/admin.sh
```

Gnocchi and gnocchiclient

```
$ sudo yum -y install openstack-gnocchi-{api,metricd}
```

```
$ sudo yum install -y python-gnocchiclient
```

Adding metric service to keystone

```
$ openstack service create --name gnocchi metric
+---------+----------------------------------+
| Field   | Value                            |
+---------+----------------------------------+
| enabled | True                             |
| id      | fb1e809461964d039f34fdbb0902a394 |
| name    | gnocchi                          |
| type    | metric                           |
+---------+----------------------------------+
```

Creating gnocchi endpoints

```
$ for i in public internal admin; do
      openstack endpoint create --region RegionOne metric $i http://127.0.0.1:8041/
  done
```

Creating the gnocchi user

```
$ openstack user create --project service --password password gnocchi
+---------------------+----------------------------------+
| Field               | Value                            |
+---------------------+----------------------------------+
| default_project_id  | d8017338c4a5452dabe83134daa98741 |
| domain_id           | default                          |
| enabled             | True                             |
| id                  | 2e751de55aa0477095164a29bb496c8c |
| name                | gnocchi                          |
| options             | {}                               |
| password_expires_at | None                             |
+---------------------+----------------------------------+
```

```
$ openstack role add --user gnocchi --project service admin
```

Creating Gnocchi's database

```
$ mysql -uroot -pmysqlpass << EOF
CREATE DATABASE gnocchi;
GRANT ALL PRIVILEGES ON gnocchi.* TO 'gnocchi'@'localhost' \
IDENTIFIED BY 'gnocchidbpassword';
GRANT ALL PRIVILEGES ON gnocchi.* TO 'gnocchi'@'%' \
IDENTIFIED BY 'gnocchidbpassword';
EOF
```

Configuring Gnocchi

```
$ sudo cp ~/handson_files/gnocchi.conf /etc/gnocchi/gnocchi.conf
```

We use gnocchi's file storage:

```
[storage]
driver = file
file_basepath = /var/lib/gnocchi
```

This enables keystone authentication in Gnocchi:

```
[api]
auth_mode = keystone
```

Initialize Gnocchi's storage

```
$ sudo -u gnocchi /usr/bin/gnocchi-upgrade
```

Start Gnocchi's daemons

```
$ sudo systemctl start openstack-gnocchi-api
$ sudo systemctl start openstack-gnocchi-metricd
```

Ceilometer

```
$ sudo yum -y install openstack-ceilometer-{central,notification,compute}
```

Creating the ceilometer user

```
$ openstack user create --project service --password password ceilometer
+---------------------+----------------------------------+
| Field               | Value                            |
+---------------------+----------------------------------+
| default_project_id  | a441c583bfda4f868cf91b6c779c0777 |
| domain_id           | default                          |
| enabled             | True                             |
| id                  | e46f8c6399b1450281f4cc99e1c3c604 |
| name                | ceilometer                       |
| options             | {}                               |
| password_expires_at | None                             |
+---------------------+----------------------------------+
```

```
$ openstack role add --user ceilometer --project service admin
```

Configuring Ceilometer

```
$ sudo cp ~/handson_files/ceilometer.conf /etc/ceilometer/ceilometer.conf
$ sudo cp ~/handson_files/pipeline.yaml /etc/ceilometer/pipeline.yaml
```

Ceilometer's config file is very simple and has only classical OpenStack options: `service_credentials`, `keystone_authtoken`, and a `transport_url`.

`pipeline.yaml` is the default file with all publishers set to gnocchi.

Create ceilometer resource types in Gnocchi

```
$ sudo -u ceilometer /usr/bin/ceilometer-upgrade
```

Starting Ceilometer daemons

```
$ sudo systemctl start openstack-ceilometer-central
$ sudo systemctl start openstack-ceilometer-notification
$ sudo systemctl start openstack-ceilometer-compute
```

CloudKitty, dashboard & client

Daemons:

```
$ sudo yum -y install openstack-cloudkitty-{api,processor}
```

Client:

```
$ sudo yum -y install python-cloudkittyclient
```

Dashboard:

```
$ sudo yum -y install openstack-cloudkitty-ui
$ sudo systemctl restart httpd
```

Adding CloudKitty to Keystone

```
$ openstack service create --name cloudkitty rating
+---------+----------------------------------+
| Field   | Value                            |
+---------+----------------------------------+
| enabled | True                             |
| id      | ef63cce9085443d5b95d9558b6176f90 |
| name    | cloudkitty                       |
| type    | rating                           |
+---------+----------------------------------+
```

Creating the cloudkitty user and the rating role

```
$ openstack user create --project service --password password cloudkitty
+---------------------+----------------------------------+
| Field               | Value                            |
+---------------------+----------------------------------+
| default_project_id  | 8eba65aaa579413fb1dd7ff252caa0a4 |
| domain_id           | default                          |
| enabled             | True                             |
| id                  | faa6b264724946c7bec4fc64376687a4 |
| name                | cloudkitty                       |
| options             | {}                               |
| password_expires_at | None                             |
+---------------------+----------------------------------+
```

```
$ openstack role add --user cloudkitty --project service admin
```

```
$ openstack role create rating
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | None                             |
| id        | 34f04c73b7b640f8bfd78f1faa97eed2 |
| name      | rating                           |
+-----------+----------------------------------+
```

Creating CloudKitty's endpoints

```
$ for i in public internal admin; do
      openstack endpoint create --region RegionOne rating $i http://127.0.0.1:8889/
  done
```

Creating CloudKitty's database

```
$ mysql -uroot -pmysqlpass << EOF
CREATE DATABASE cloudkitty;
GRANT ALL PRIVILEGES ON cloudkitty.* TO 'cloudkitty'@'localhost' \
IDENTIFIED BY 'cloudkittydbpassword';
GRANT ALL PRIVILEGES ON cloudkitty.* TO 'cloudkitty'@'%' \
IDENTIFIED BY 'cloudkittydbpassword';
EOF
```

Configuring CloudKitty [1/2]

```
$ sudo cp ~/handson_files/cloudkitty.conf /etc/cloudkitty/cloudkitty.conf
```

Most options of this file (`keystone_authtoken`, transport_url, database...) are classical OpenStack options. However, there are some specific to cloudkitty.

CloudKitty supports several storage backends (sqlalchemy and hybrid, v2 storage should come with the Rocky release). For this session, we will use the sqlalchemy storage in order to get quicker results (Hybrid storage uses a timeseries backend, which must do some aggregatuion first):

```
[storage]
backend=sqlalchemy
```

As we are using CloudKitty with OpenStack, we use the keystone fetcher. The `tenant_fetcher` section contains general fetcher options (which fetcher to use), and `keystone_fetcher` contains options specific to keystone (authentication):

```
[tenant_fetcher]
backend = keystone

[keystone_fetcher]
auth_section=ks_auth
keystone_version=3
```

Configuring CloudKitty [2/2]

We use the gnocchi collector, so we specify authentication options in the `gnocchi_collector` section.

```
[gnocchi_collector]
auth_section=ks_auth
```

Configuring CloudKitty's metric collection [1/3]

```
$ sudo cp ~/handson_files/metrics.yml /etc/cloudkitty/metrics.yml
```

Here, we declare a processor, called OpenStack. We specify that is should use the `gnocchi` collector, and that it should process periods of one hour (3600s). The `wait_periods` options corresponds to the number of periods, the collector should wait before querying a time frame. If we set this to 2, the collector will collect the metrics from 10AM to 11AM at 12AM. This has been set to 0 for this session in order to get quicker results but IT'S A BAD IDEA. (You should leave time for gnocchi to process the measures it receives).

```
- name: OpenStack

  collector: gnocchi
  period: 3600
  wait_periods: 0
```

Configuring CloudKitty's metric collection [2/3]

Queens is the last release in which CloudKitty still has a « service » notion to group metrics. The best practice for this « pivot » release would be to create services of a single metric. We declare services and their metrics the following way:

```
services:
  - compute
  - image

services_metrics:
  compute:
    - cpu: max
  image:
    - image.size: max
```

The `services` section contains a list of services to use. Their metrics must be defined in the `services_metrics` section. Each entry of this section has the following format: `metrics_name: aggregation_method`.

Configuring CloudKitty's metric collection [3/3]

CloudKitty support unit conversion between any unit with linear scales. The end result is calculated the following way: `result = F * measures + O`. F stands for factor and O for offset.

We specify units and conversion rules the following way:

```
metrics_units:
  compute:
    1:
      unit: instance
  image:
    image.size:
      unit: MiB
      factor: 1/1048576
  default_unit:
    1:
      unit: unknown
```

For each unit of each service, you need to specify a unit (a string which will be in the final result). If you want to do a conversion, you can specify a factor (defaults to 1) and an offset (defaults to 0). Here we convert B to MiB for `image.size`.

Initializing CloudKitty's storage

```
$ sudo -u cloudkitty cloudkitty-storage-init
```

```
$ sudo -u cloudkitty cloudkitty-dbsync upgrade
```

Setup CloudKitty's API

Copy CloudKitty's WSGI config file:

```
$ sudo mkdir -p /var/www/cloudkitty/
$ sudo cp /usr/lib/python2.7/site-packages/cloudkitty/api/app.wsgi /var/www/cloudkitty/app.wsgi
$ sudo chown -R cloudkitty:cloudkitty /var/www/cloudkitty/
$ sudo cp ~/handson_files/cloudkitty-api.conf /etc/httpd/conf.d/cloudkitty-api.conf
```

Restart httpd:

```
$ sudo systemctl restart httpd
```

Pricing policy

Create a project which will be rated

```
$ openstack project create summit
$ openstack user create --password password --project summit summit_user
$ openstack role add --user summit_user --project summit admin
```

Tell CloudKitty to rate your project

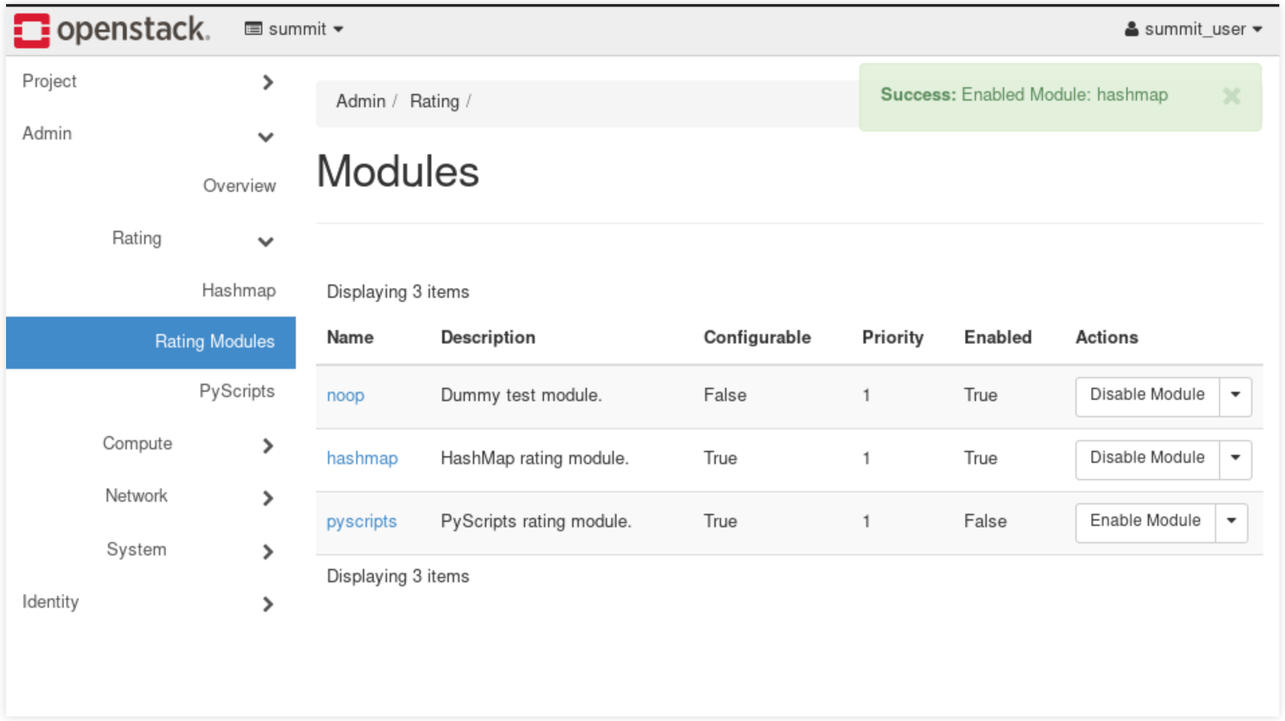
```
$ openstack role add --user cloudkitty --project summit rating
```

Use horizon to define rating rules

Log into horizon at **http://YOUR_IP/dashboard** with the newly created user (summit_user / password).

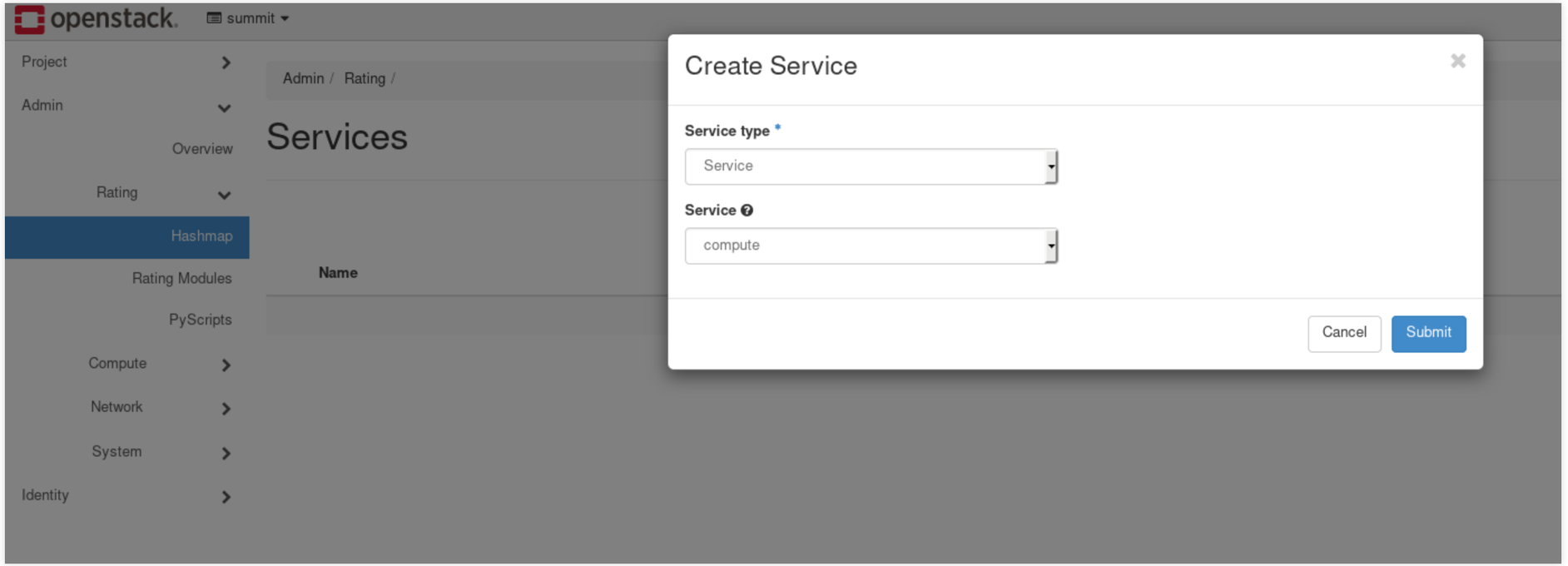
Enable the Hashmap module

We will use CloudKitty's **Hashmap** module to define our rating rules. First of all, enable it. (Admin -> Rating -> Rating Modules)



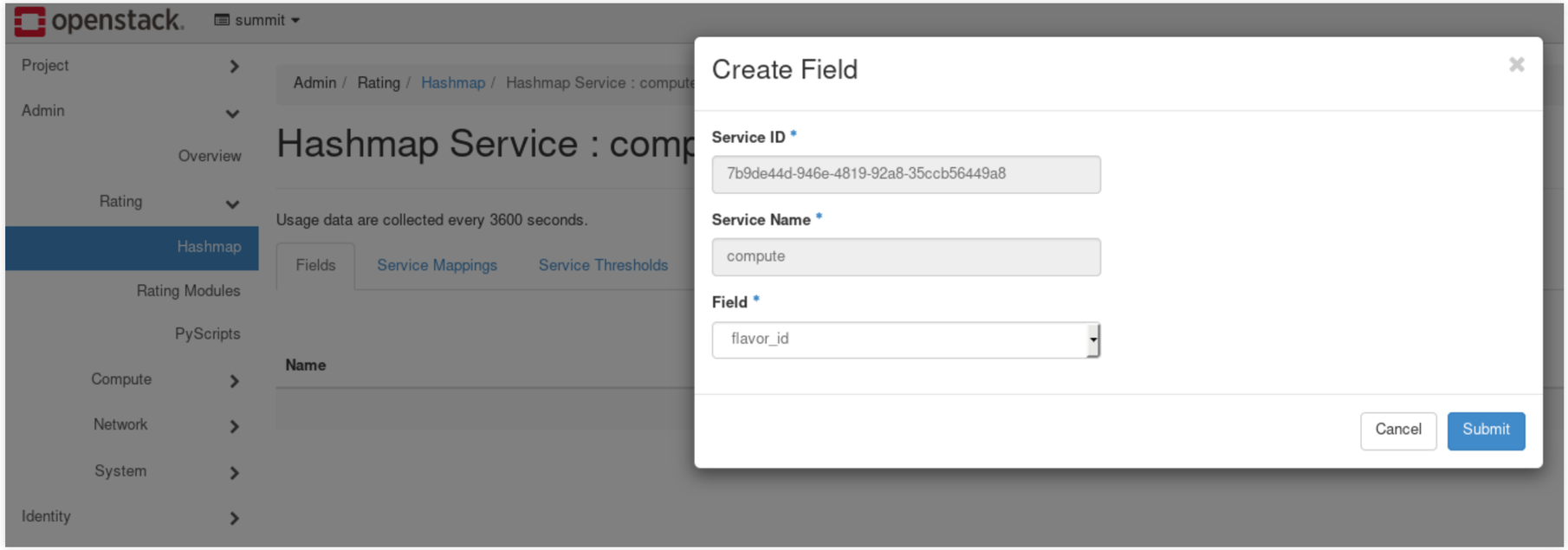
Create the `compute` service

Go to the Hashmap module's configuration panel (Admin -> Rating -> Hashmap) and create a service called `compute`. Once it is created, click on it.



Create a `flavor_id` field

Create a `flavor_id` field in the compute service. This field will be used to match flavors of running instances. Once it is created, click on it.



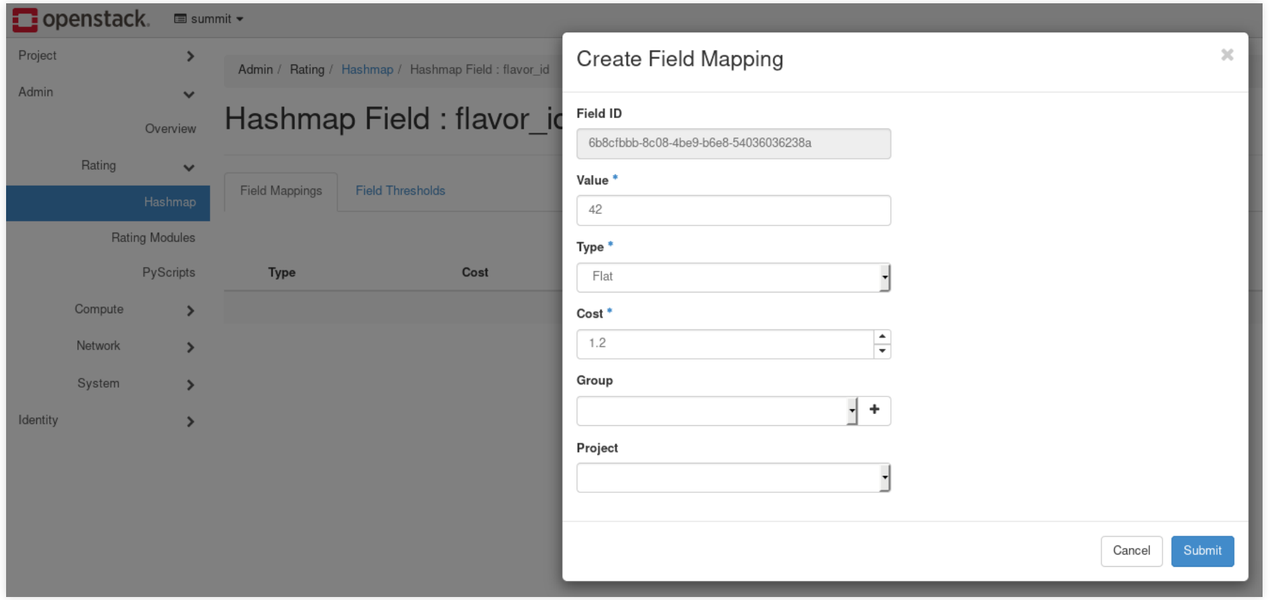
Add Some Field Mappings [1/2]

In order to charge running instances based on their flavor, we need to get the IDs of the flavors we want to use.

```
$ openstack flavor list
+----+----------+-----+------+-----------+-------+-----------+
| ID | Name     | RAM | Disk | Ephemeral | VCPUs | Is Public |
+----+----------+-----+------+-----------+-------+-----------+
| 42 | m1.nano  |  64 |    0 |         0 |     1 | True      |
| 84 | m1.micro | 128 |    0 |         0 |     2 | True      |
+----+----------+-----+------+-----------+-------+-----------+
```

Add Some Field Mappings [2/2]

Create a mapping that matches the `m1.nano` flavor. The specified cost is per instance and collect period (3600 seconds). Pretty expensive, isn't it ? Note that we don't specify a tenant: this means that the mapping will be applied to all tenants on which CloudKitty has the `rating` role.



Once you're done, create a second mapping with a different price for the `m1.micro` flavor.

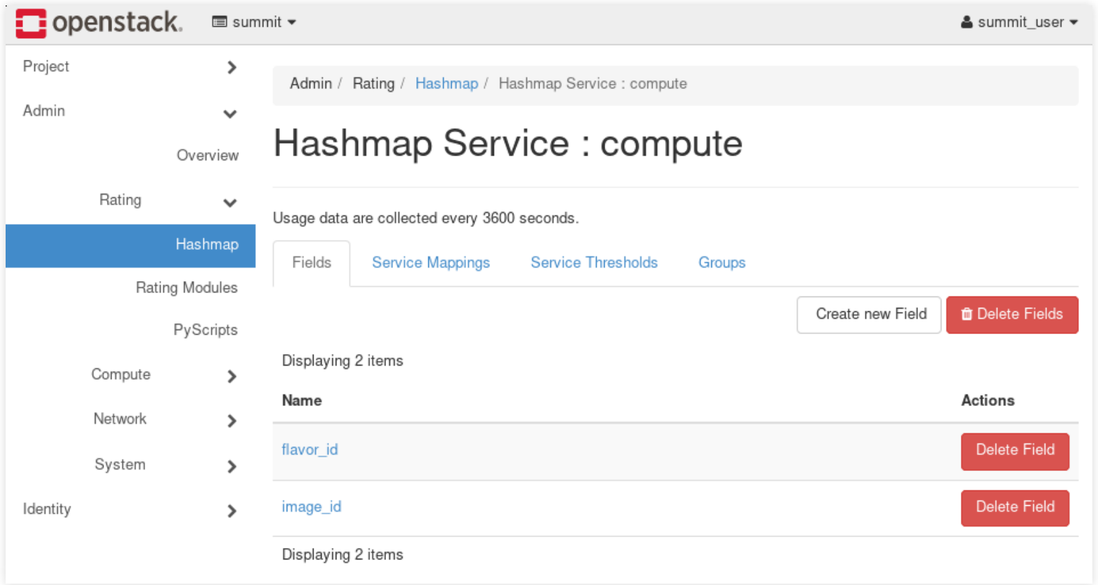
Apply a per-image price [1/3]

First of all, get the ids of the existent images:

```
$ openstack image list
+--------------------------------------+------------+--------+
| ID                                   | Name       | Status |
+--------------------------------------+------------+--------+
| 68c8abc2-295b-4b8b-9c72-5acd8f0b51d1 | cirros-4.0 | active |
| 38cb02be-cca4-4ff7-9a79-ef71607ffa46 | windows    | active |
+--------------------------------------+------------+--------+
```

Apply a per-image price [2/3]

Go back to the `fields` section of the `compute` service, and create an `image_id` field.

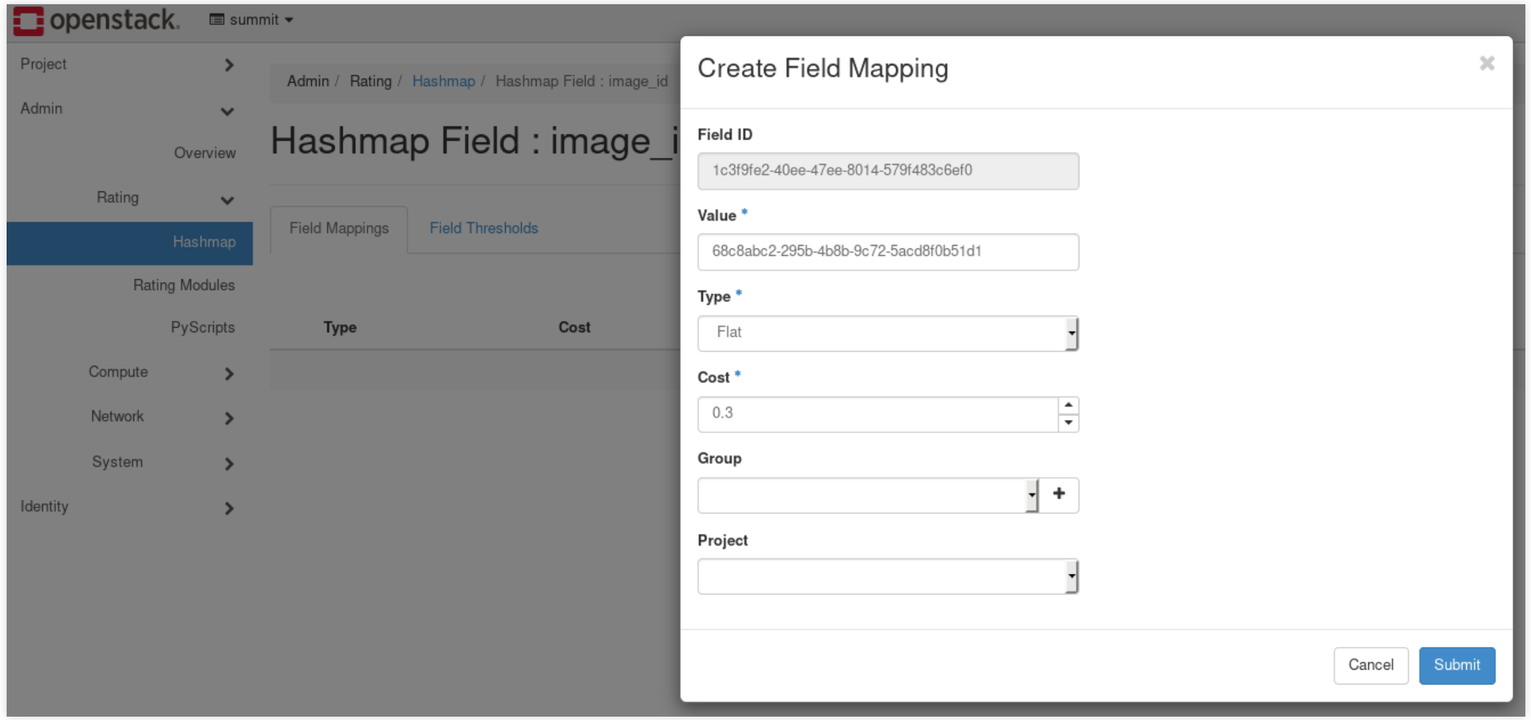


Once it is created, click on it.

Apply a per-image price [3/3]

You can create flat mappings based on `image_id` the same way as you did for `flavor_id`. The specified cost will be charged per instance per collect period.

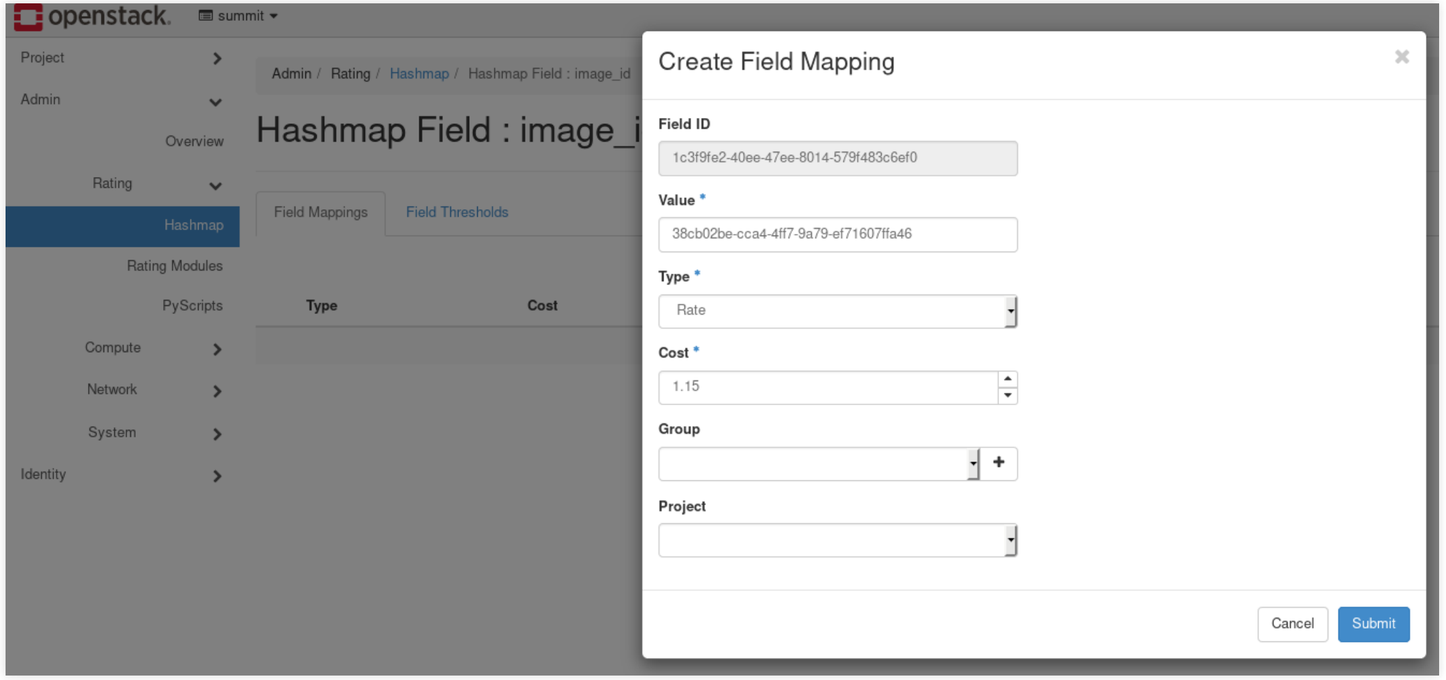
First, create a mapping for the `cirros-4.0` image, using its id:



Once you're done, create mapping for the `windows` image with a different price.

Charge extra for instances running windows

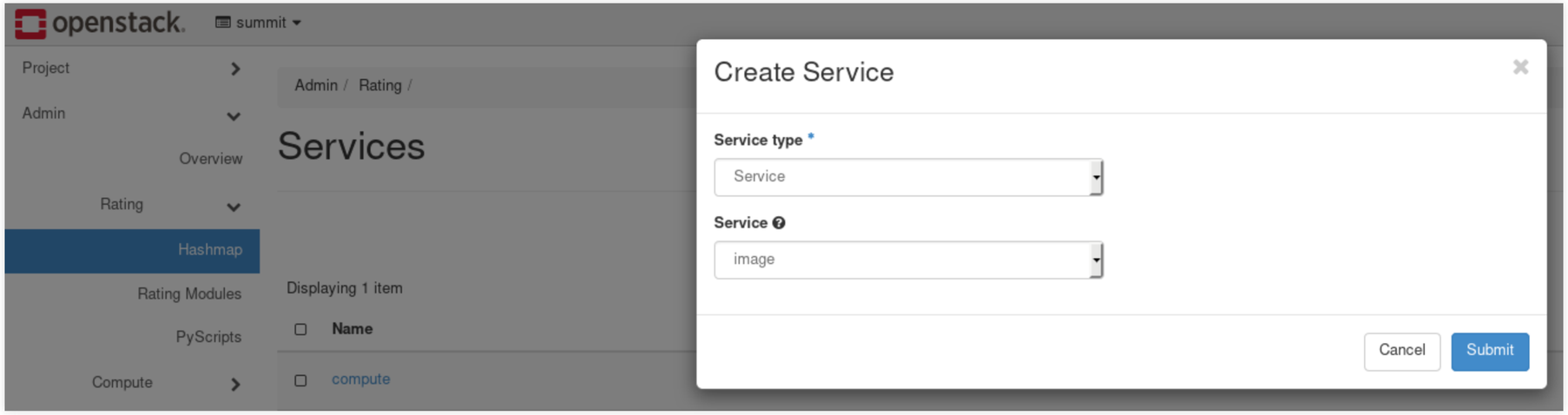
Create the following mapping (use the id of the `windows` image) in the `image_id` field:



Note that this mapping is a `rate`. We want to charge every instance running windows 15% extra, so we set the cost to 1.15. This means that the cost of every instance running windows will be multiplied by 1.15. (You can also apply discounts if the cost is inferior to 1).

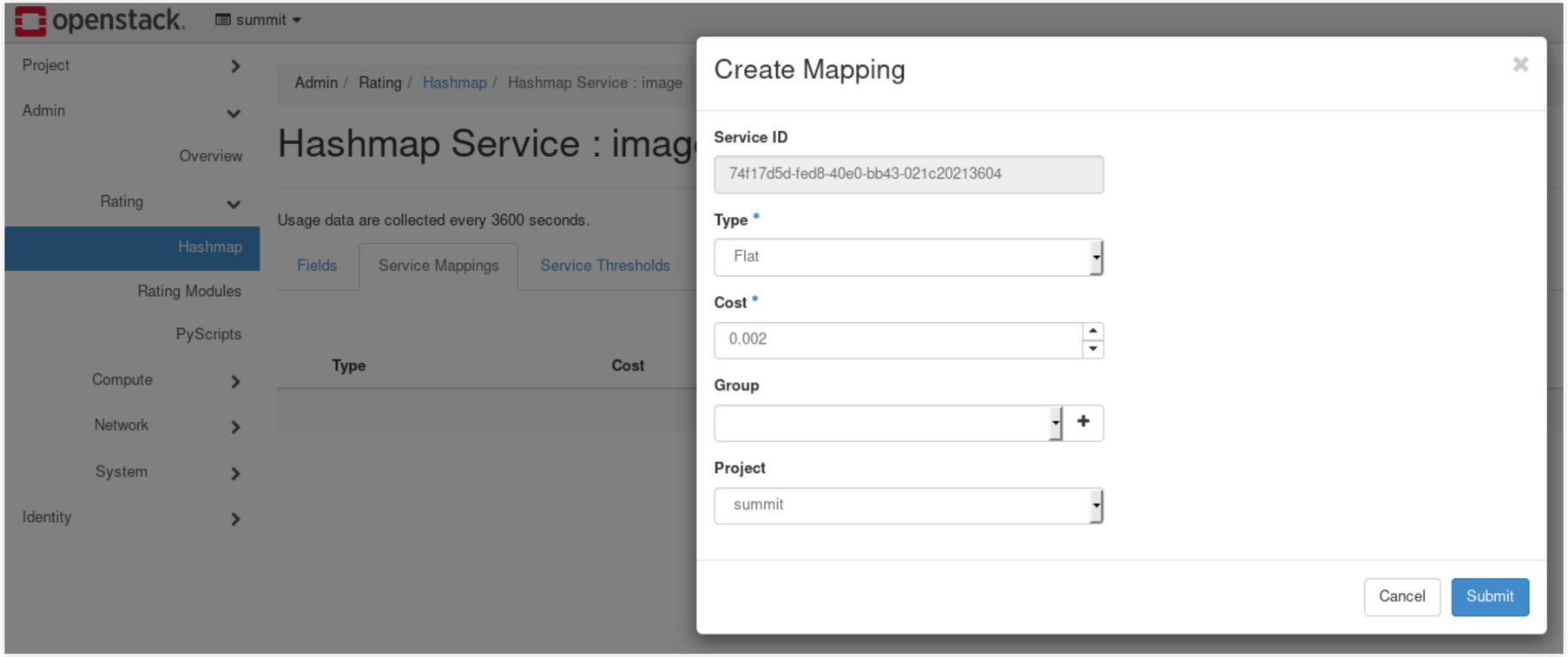
Create the `Image` service

We will now create the required rules to charge glance image creation. Create an `image` service, like you did for `compute`.



Create a service-mapping for image

We will now create a service-mapping for the `image` service. The cost of this mapping will be 0.2 per MiB (per collect period):



Create some data

```
$ source ~/summit.sh
```

```
$ for i in {1..3}; do
      openstack server create --image cirros-4.0 --flavor m1.nano instance${i}
      openstack server create --image windows --flavor m1.nano instance-win${i}
      openstack image create --file ~/cirros-4.0.img image${i}
  done
```

Start CloudKitty's processor

```
$ sudo systemctl start cloudkitty-processor
```

By default, the processor starts at the beginning of the month, so you'll have to wait as few minutes for the processor to catch up with the current day.

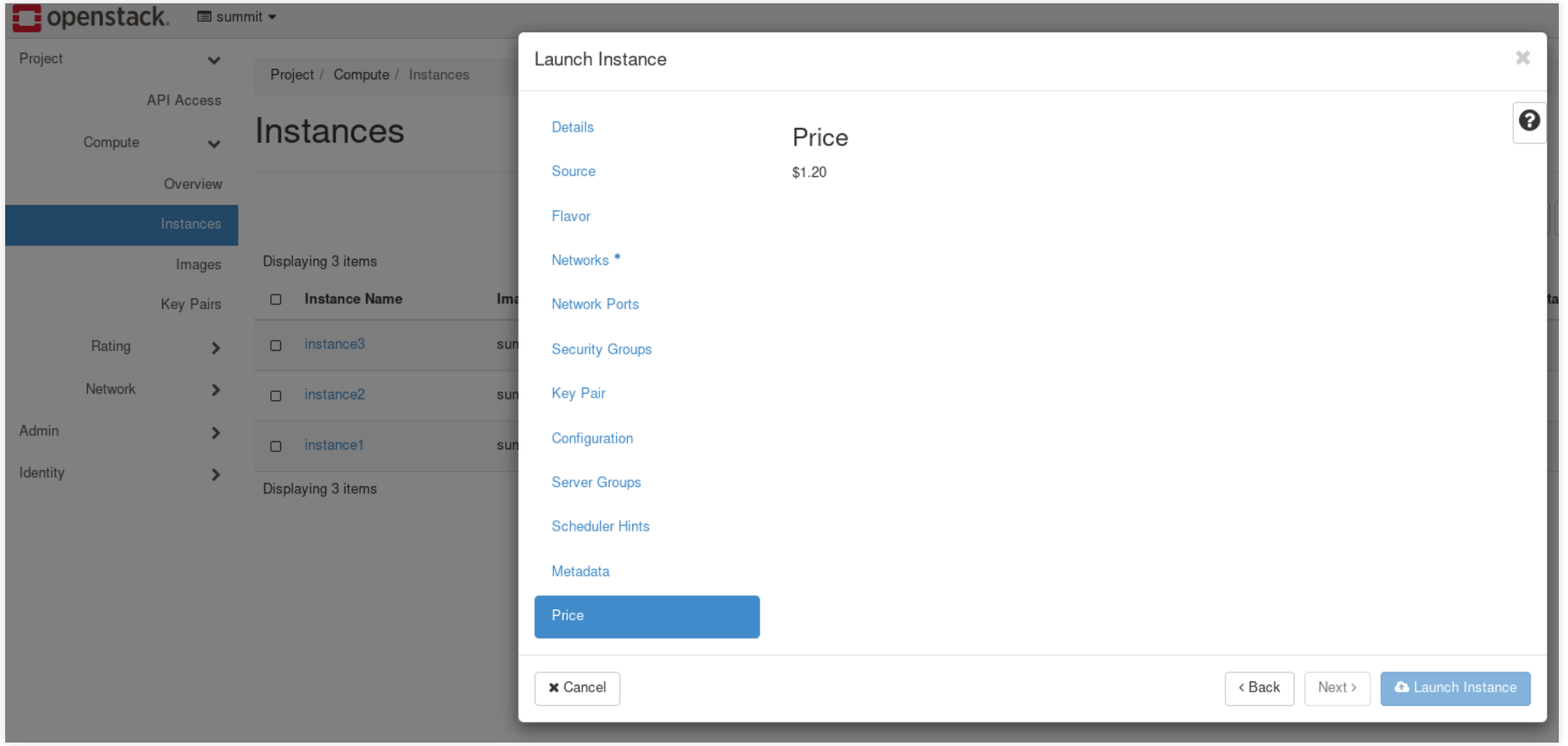
Rating information

You should now have rating information. You can look at Project -> Rating -> Rating/ Reporting.

Given that the current hour has been rated and that we have very few rating information, the charts may look a bit weird. Don’t worry, they are shiny on a regular cloud ;-)

Predictive Pricing

Go to Project -> Compute -> Instances and click on **start an instance**. Select m1.nano flavor and fill out the necessary fields. Once you're done, go to the **Price** tab. You should have something like this:



The price should be the following: HIGHEST(image_price, flavor_price) * rate

It is time for questions!

Or later :

- christophe.sauthier@objectif-libre.com
- luka.peschke@objectif-libre.com
- ludovic.logiou@objectif-libre.com

We'll be happy to send you the latest release of these slides!