RAX-AutoScaler Documentation

Release 0.2.29

Simon Mirco, Simone Soldateschi, Suraj Thapa, Teddy Schmitz, J

January 09, 2015

Contents

RAX-AutoScaler use the rackspace APIs to allow for scaling based on aggregate metrics across a cluster. Can be used and installed on the auto-scale group members or on a dedicated management instance.

- GitHub repository: https://github.com/boxidau/rax-autoscaler
- PyPI package: https://pypi.python.org/pypi/rax-autoscaler
- Stories in Ready: https://badge.waffle.io/boxidau/rax-autoscaler.svg?label=ready&title=Ready)](http://waffle.io/boxidau/rax-autoscaler

Contents:

CHAPTER 1

Quick Start

1.1 Installation

```
pip install RAX-AutoScaler
```

1.2 Configuration

Edit config.json adding the following: * API username * API key * Region name * Autoscaling group section should contain:

- AutoScale Group UUID
- Scale Up Policy UUID
- Scale Down Policy UUID
- Check Type (agent.cpu, agent.load_average...)
- Metric Name (depends on the check type)
- Scale Up Threshold
- Scale Down Threshold
- Webhooks Url (Pre & Post commit url(s) for scale up/down)

1.3 Usage

Once configured you can invoke the autoscaler.py script.

–cluster option should be used when this script actually runs on auto-scale group members. Otherwise if it is running on a dedicated management instance you do not require this option.

–as-group option should be used when you have multiple groups listed in the config.json file.

–config-file option should be used if config.json file does not exists in current directory or in '/etc/rax-autoscaler' path.

Once tested you should configure this script to run as a cron job either on a management instance or on all cluster members

1.4 Cloud Init

You can use the cloud-config file to auto-install RAX-Autoscaler on new servers. For example to do so on Rackspace cloud using supernova

> *supernova <region> boot –user-data ./cloud-config –image <image id/name> –flavor <flavor id/name> –config-drive=true <server name>*

To use this with autoscale you would want to set the userdata of your launch configuration to the base64 encoded string of the file:

```
"launchConfiguration": {
      "args": {
          "server": {
              "config_drive" : true,
              "flavorRef": "general1-1",
              "imageRef": "CentOS 6.5 (PVHVM)",
              "key_name" : "MY_SSH_KEY",
              "user_data" : "I2Nsb3VkLWNvbmZpZwoKcGFja2FnZXM6CiAgLSBweXRob24tcGlwCgpydW5jbWQ6CiAgLSBb
              "name": "test-autoscale"
          }
      },
      "type": "launch_server"
  }
```

This has been tested on these images:

- Ubuntu 14.04 LTS (PVHVM)
- CentOS 6.5 (PVHVM)

In the example the value of user_data contains the base64 encoded version of the following script:

```
echo -n "I2Nsb3VkLWNvbmZpZwoKcGFja2FnZXM6CiAgLSBweXRob24tcGlwCgpydW5jbWQ6CiAgLSBbIHBpcCwgaW5zdGFsbCw
```

To base64 encode a script :: cat /path/to/USER_DATA | base64

Size of USER_DATA can be reduced with gzip: :: cat /path/to/USER_DATA | gzip | base64

1.5 config.json

There are a few things to configure both on the Rackspace API side (essentially creating the autoscale group and setting the launch configuration) and on our side.

First up, create the autoscaling group in the portal and create 2 policies, 1 to scale up and another to scale down. The policy scale trigger should be set to webhook url.

The things we need now on the Rackspace side aren't available in the portal so you'll have to make use of https://cloud-api.info/ to quickly interact with the API. To login, you'll need your API key which is available from the portal under the "Account Settings" screen.

I'm going to quote the config/config-template.json file to make it easier to explain

```
"autoscale_groups": {
    "group0": {
```

"group0" is just a name used internally by autoscale.py to make it easier for the user to reference the scaling group. It can stay as group0 or be changed to match what you've called the group in the portal

```
"group_id": "group id",
```

"group_id" is the UUID of the scaling group, this can be seen in the portal or in pitchfork by calling the "List Scaling Groups" API method https://cloud-api.info/autoscale/#list_scaling_groups-autoscale

```
"scale_up_policy": "scale up policy id",
"scale_down_policy": "scale down policy id",
```

scale_up_policy and scale_down_policy requires the UUID of the policies configured as webhooks in the portal. You can retrieve the UUID in pitchfork by calling the Get Policies List method https://cloud-api.info/autoscale/#get_policies_list-autoscale

```
"check_type": "agent.load_average",
"check_config": {},
"metric_name": "1m",
```

currently, the only check type we have is load_average (ram utilisation is on the backlog)

Webhook urls are an optional URL to call when we scale up or down. Pre is called before we call the Rackspace API, post is called after. You can configure multiple urls (or just a single url) to call at each stage.

1.6 Note

RAX-AutoScaler depends on Rackspace Monitoring Agent to get the data from nodes in scaling group.

If the agent is not installed please read: Install the Cloud Monitoring Agent: http://www.rackspace.com/knowledge_center/article/install-the-cloud-monitoring-agent

1.7 Contributing

- Fork it
- Create your feature branch (git checkout -b my-new-feature)
- Commit your changes (git commit -am 'Add some feature')
- Push to the branch (git push origin my-new-feature)
- Create new Pull Request

CHAPTER 2

Plugins

2.1 Monitoring Plugins

These plugins are used to provide different ways to determing whether to scale up or down. Currently there are 2 monitoring plugins available.

2.1.1 Raxmon

Plugin for Rackspace monitoring.

Uses Rackspace cloud monitoring to check a server statstic and use it to make a scaling decision. The statistic is averaged over all currently active servers in the scaling group. The plugin will create the check on the servers if it does not currently exist. The Rackspace monitoring agent must be installed on all servers in the scaling group and should be part of your launch_configuration.

This is used to help smooth out fluctuations in the connection count so we do not scale on small fluctuations.

Config should look like this:

```
"raxmon":{
    "scale_up_threshold": 0.6,
    "scale_down_threshold": 0.4,
    "check_config": {},
    "metric_name": "1m",
    "check_type": "agent.load_average"
}
```

scale_up_threshold - Set this to a value that makes sense for the check you are performing. If we go over this number we will scale up.

scale_down_threshold - Set this to a value that makes sense for the check you are performing. If we go under this number we will scale down.

check_config (optional) - configuration options for the check we will be performing. Used when we are creating the check on servers that don't have it.

check_type - What type of check to perform. Default is agent.load_average (check servers load average)

metric_name - Name of metric checked. We are checking the load_average over 1 minute periods so the metric name could be 1m. Default is 1m

2.1.2 Raxclb

Plugin for Rackspace cloud load balancer.

This checks the load balancer connection counts and uses it to make a scaling decision. The algorithm is:

(current_connection * 1.5 + historical_connection) / 2

This is used to help smooth out fluctuations in the connection count so we do not scale on small fluctuations.

Config should look like this:

```
"raxclb":{
    "scale_up_threshold": 100,
    "scale_down_threshold": 10,
    "check_type": "SSL"
    "loadbalancers":[]
}
```

scale_up_threshold - How many connections per server you want to handle. We will multiply this number by the number of servers currently active in the group. If we go over this number we will scale up. Default is 50

scale_down_threshold - How many connections per server you want to handle. We will multiply this number by the number of servers currently active in the group. If we go under this number we will scale down. Default is 1

check_type - set this to SSL if you want to check SSL connection counts instead of regular http. Default is to not check SSL.

loadbalancers - provide a list of loadbalancer ids to check. If you do not provide this we will detect the loadbalancer(s) in use by the scaling group and check all of them and aggregate results. Otherwise we will only check the loadbalancer ids you provide here. Default is an empty list (Auto-detect loadbalancers).

2.2 Creating Plugins

All monitoring plugins should inherit from raxas.core_plugins.base. You must implement a make_decision function that returns a 1 for scale up, -1, for scale down, or 0 for do nothing.:

```
from raxas.core_plugins.base import PluginBase
class Yourplugin(PluginBase):
    def __init__(self, scaling_group, config, args):
    super(Yourplugin, self).__init__(scaling_group, config, args)
```

common.**check_file**(*fname*)

This function checks if file exists and is readable.

Parameters **fname** – file name

Returns file name with absolute path

common.**exit_with_error**(*msg*)

This function prints error message and exit with error.

Parameters **msg** – error message

Returns 1 (int) – the return code

common.**get_auth_value**(*args*, *config*, *key*)

This function returns value associated with the key if its available in user arguments else in json config file.

Parameters

- **args** – user arguments
- **config** – json configuration data
- **key** – key name

Returns value associated with key

common.**get_config**(*config_file*)
This function read and returns jsons configuration data

Parameters **config_file** – json configuration file name

Returns json data

common.**get_logger**()
This function instantiate the logger.

Returns logger

common.**get_machine_uuid**(*scaling_group*)
This function will search for the server's UUID and return it.

First it searches in a rax-autoscaler cache, followed by cloud-init cache. If it cannot find a cached UUID, it will get the details of each server in the scaling group in turn and attempt to match the local IP address with that of the server object returned from the API

Parameters **scaling_group** – raxas.scaling_group.ScalingGroup object

Returns None if no UUID could be matched against a cache file or the API. UUID as a string

common.**get_server**(*server_id*)
It gets Cloud server object by server_id

common.**is_ipv4**(*address*)
It checks if address is valid IP v4

class auth.**Auth**(*username*, *apikey*, *region*, *identity_type='rackspace'*, *token_filename='/home/docs/.rax-autoscaler-token'*)
This class implements Rackspace cloud account authentication

authenticate()
This method loads a token from a file, authenticate with it, and if it fails then tries to authenticate with credentials

Returns True or False (Boolean)

authenticate_credentials()
This method try to authenticate with available credentials

Returns True or False (Boolean)

authenticate_token()
This authenticate with Rackspace cloud using existing token.

Returns True or False (Boolean)

force_unauthenticate()
This unauthenticate and delete token file

load_token()
This loads token from a file

Returns True or False (Boolean)

save_token()
This saves token to a file

Returns True or False (Boolean)

static status()
This queries pyrax for values

Returns list – the return value

token_filename
This returns token filename

Returns _token_filename

class colouredconsolehandler.**ColouredConsoleHandler**(*stream=None*)
console handler with ANSI colours support Usage example:

def white_bold_underlined(self, msg):

return self.decorate(self.BOLD + self.UNDERLINE + self.COLOR['white'], msg)

ref: http://cwoebker.com/posts/ansi-escape-codes

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

CHAPTER 4

License

a

c

A

C

E

F

G

I

L

S

T