Rally Documentation

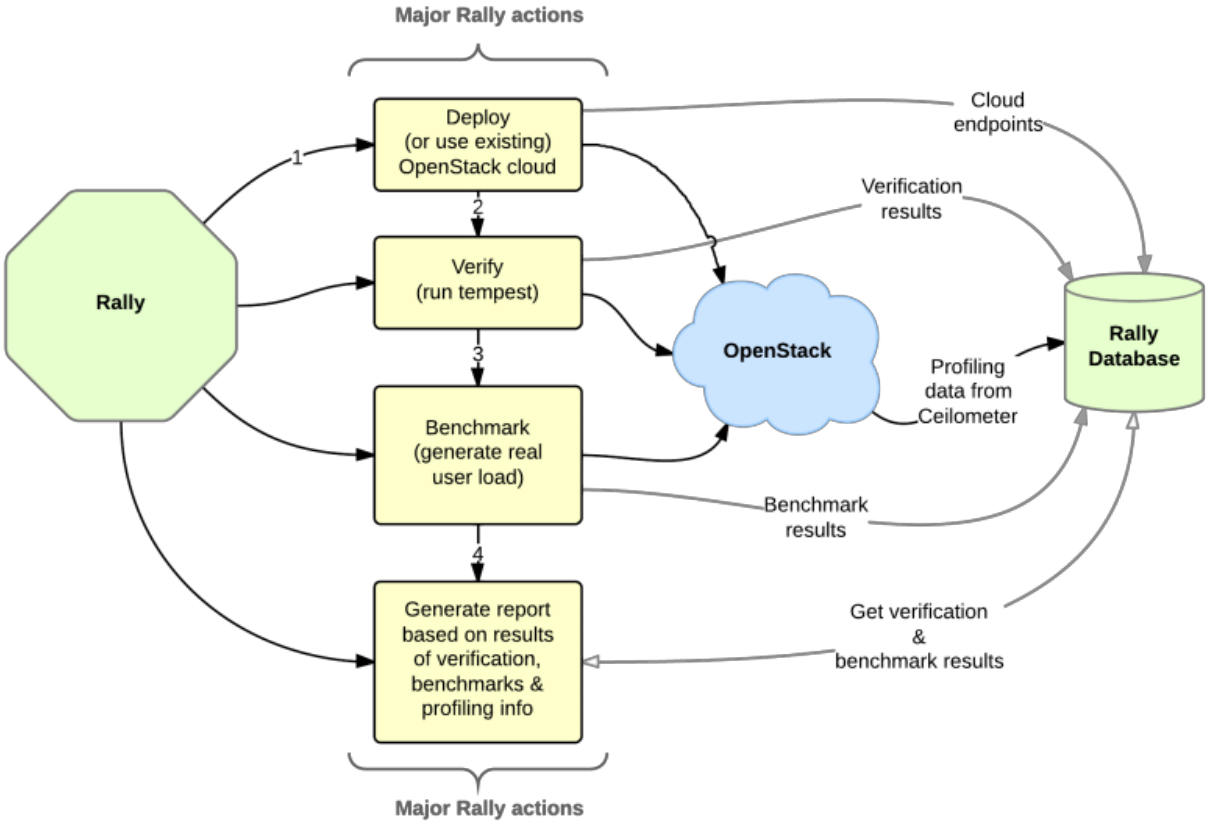
Release 0.0.3

OpenStack Foundation

April 14, 2015

Contents

OpenStack is, undoubtedly, a really *huge* ecosystem of cooperative services. **Rally** is a **benchmarking tool** that answers the question: **"How does OpenStack work at scale?"**. To make this possible, Rally **automates** and **unifies** multi-node OpenStack deployment, cloud verification, benchmarking & profiling. Rally does it in a **generic** way, making it possible to check whether OpenStack is going to work well on, say, a 1k-servers installation under high load. Thus it can be used as a basic tool for an *OpenStack CI/CD system* that would continuously improve its SLA, performance and stability.



CHAPTER 1

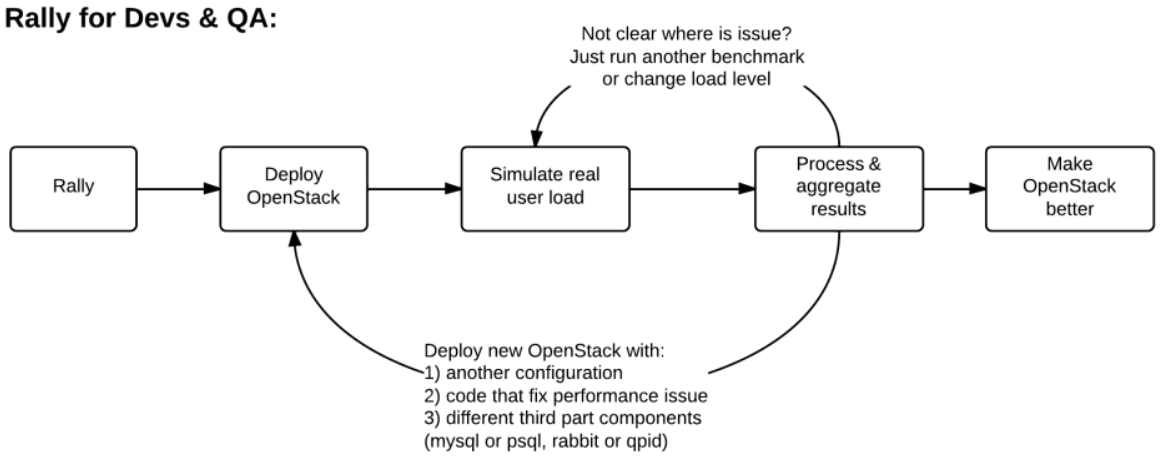
Contents

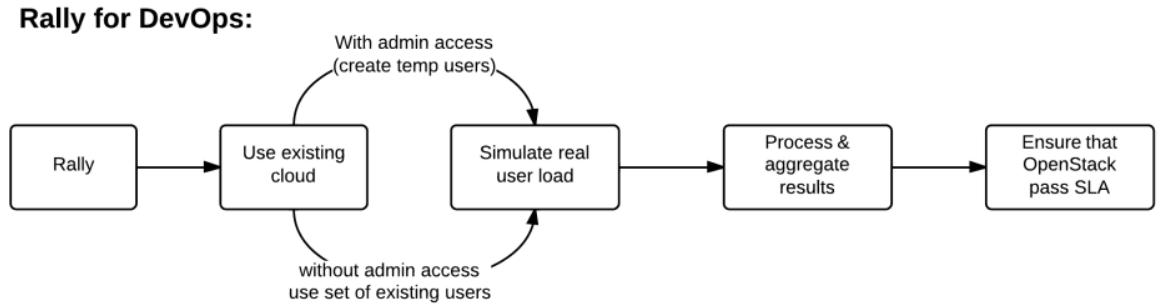
1.1 Overview

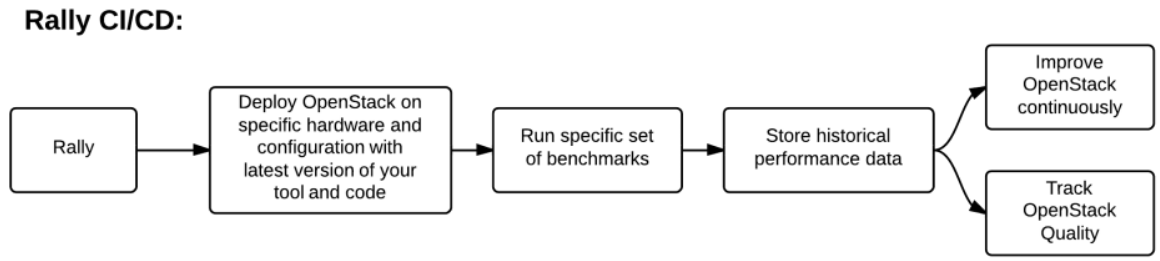
Rally is a **benchmarking tool** that **automates** and **unifies** multi-node OpenStack deployment, cloud verification, benchmarking & profiling. It can be used as a basic tool for an *OpenStack CI/CD system* that would continuously improve its SLA, performance and stability.

1.1.1 Use Cases

Let's take a look at 3 major high level Use Cases of Rally:







Generally, there are a few typical cases where Rally proves to be of great use:

1. Automate measuring & profiling focused on how new code changes affect the OS performance;
2. Using Rally profiler to detect scaling & performance issues;
3. Investigate how different deployments affect the OS performance:
   - Find the set of suitable OpenStack deployment architectures;
   - Create deployment specifications for different loads (amount of controllers, swift nodes, etc.);
4. Automate the search for hardware best suited for particular OpenStack cloud;
5. Automate the production cloud specification generation:
   - Determine terminal loads for basic cloud operations: VM start & stop, Block Device create/destroy & various OpenStack API methods;

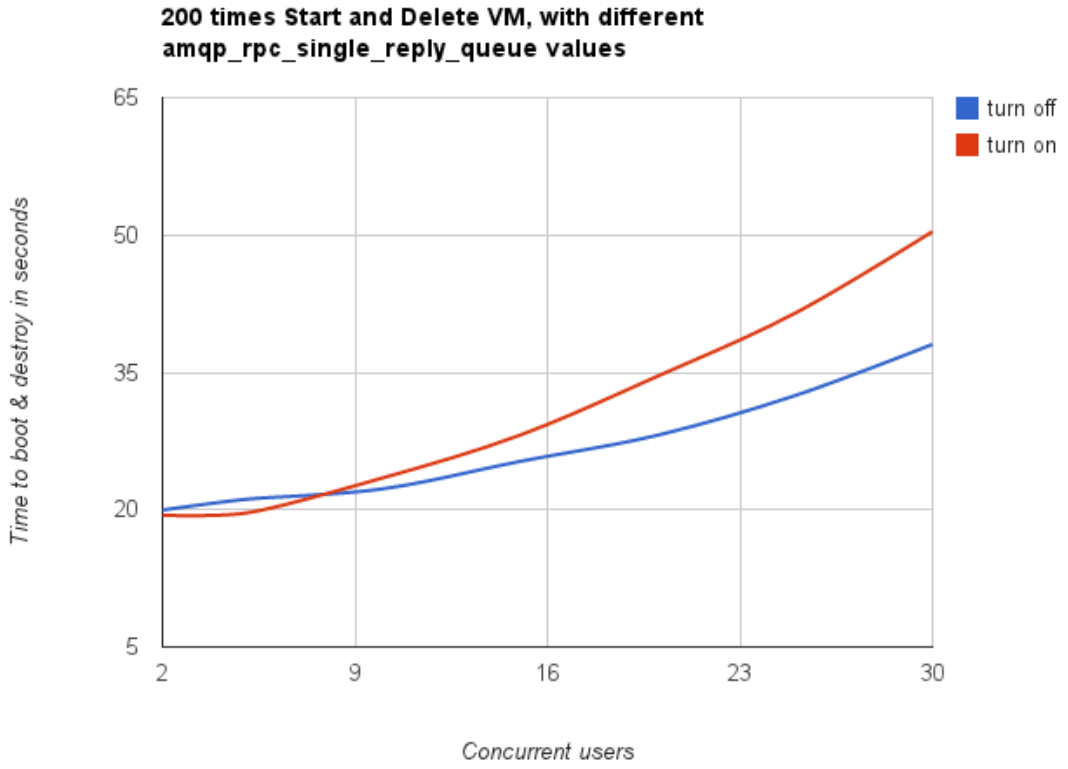
- Check performance of basic cloud operations in case of different loads.

1.1.2 Real-life examples

To be substantive, let's investigate a couple of real-life examples of Rally in action.

How does amqp_rpc_single_reply_queue affect performance?

Rally allowed us to reveal a quite an interesting fact about **Nova**. We used *NovaServers.boot_and_delete* benchmark scenario to see how the *amqp_rpc_single_reply_queue* option affects VM bootup time (it turns on a kind of fast RPC). Some time ago it was shown that cloud performance can be boosted by setting it on, so we naturally decided to check this result with Rally. To make this test, we issued requests for booting and deleting VMs for a number of concurrent users ranging from 1 to 30 with and without the investigated option. For each group of users, a total number of 200 requests was issued. Averaged time per request is shown below:



So Rally has unexpectedly indicated that setting the *amqp_rpc_single_reply_queue* option apparently affects the cloud performance, but in quite an opposite way rather than it was thought before.

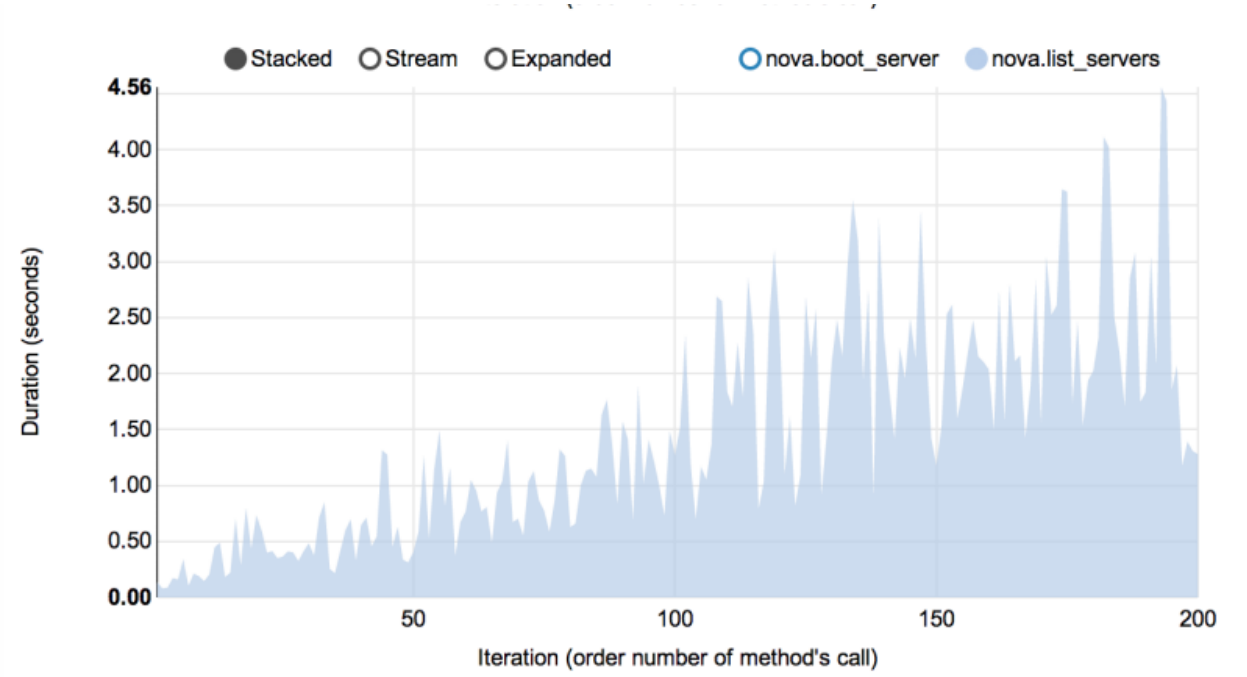
Performance of Nova list command

Another interesting result comes from the *NovaServers.boot_and_list_server* scenario, which enabled us to we launched the following benchmark with Rally:

- **Benchmark environment** (which we also call **"Context"**): 1 temporary OpenStack user.

- **Benchmark scenario**: boot a single VM from this user & list all VMs.
- **Benchmark runner** setting: repeat this procedure 200 times in a continuous way.

During the execution of this benchmark scenario, the user has more and more VMs on each iteration. Rally has shown that in this case, the performance of the **VM list** command in Nova is degrading much faster than one might expect:

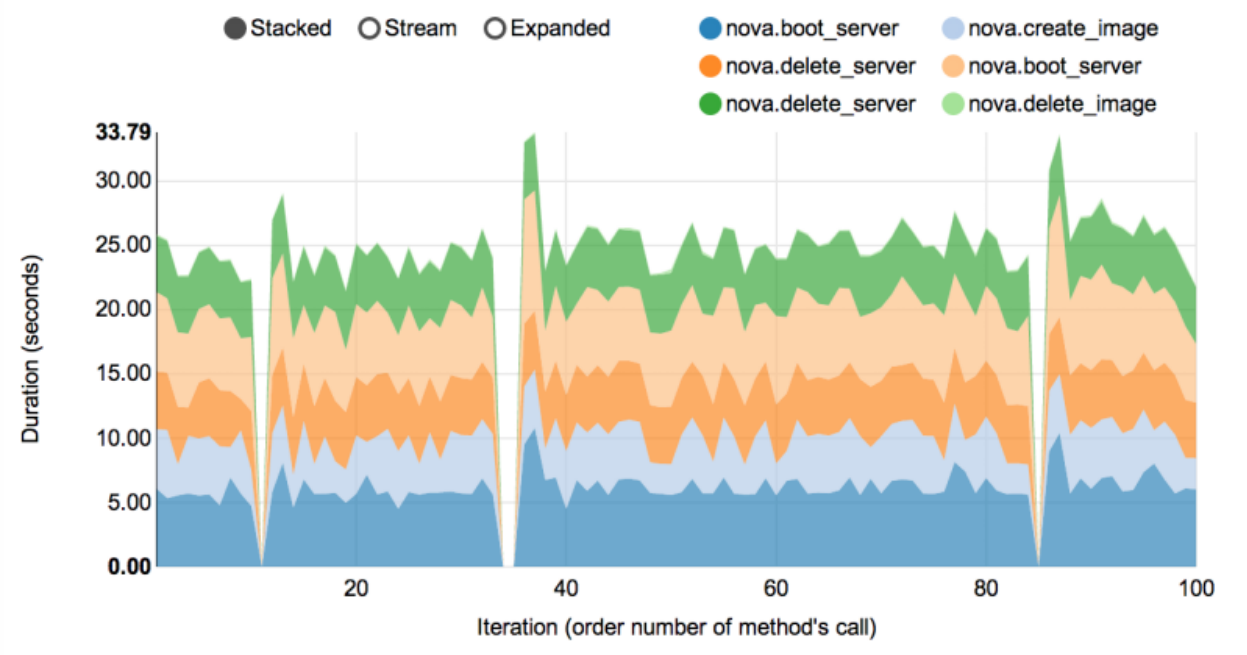


Complex scenarios

In fact, the vast majority of Rally scenarios is expressed as a sequence of **"atomic" actions**. For example, *NovaServers.snapshot* is composed of 6 atomic actions:

1. boot VM
2. snapshot VM
3. delete VM
4. boot VM from snapshot
5. delete VM
6. delete snapshot

Rally measures not only the performance of the benchmark scenario as a whole, but also that of single atomic actions. As a result, Rally also plots the atomic actions performance data for each benchmark iteration in a quite detailed way:

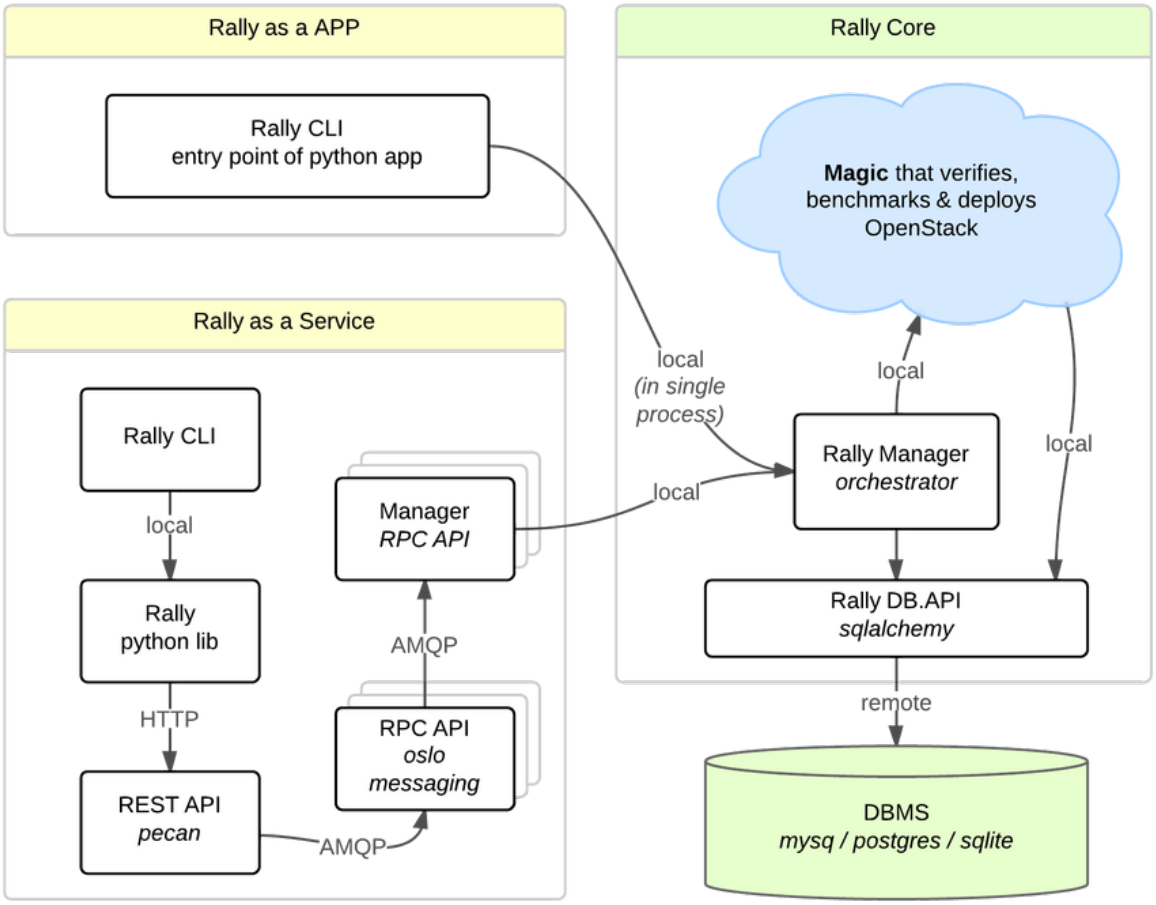


1.1.3 Architecture

Usually OpenStack projects are implemented *"as-a-Service"*, so Rally provides this approach. In addition, it implements a *CLI-driven* approach that does not require a daemon:

1. **Rally as-a-Service**: Run rally as a set of daemons that present Web UI *(work in progress)* so 1 RaaS could be used by a whole team.
2. **Rally as-an-App**: Rally as a just lightweight and portable CLI app (without any daemons) that makes it simple to use & develop.

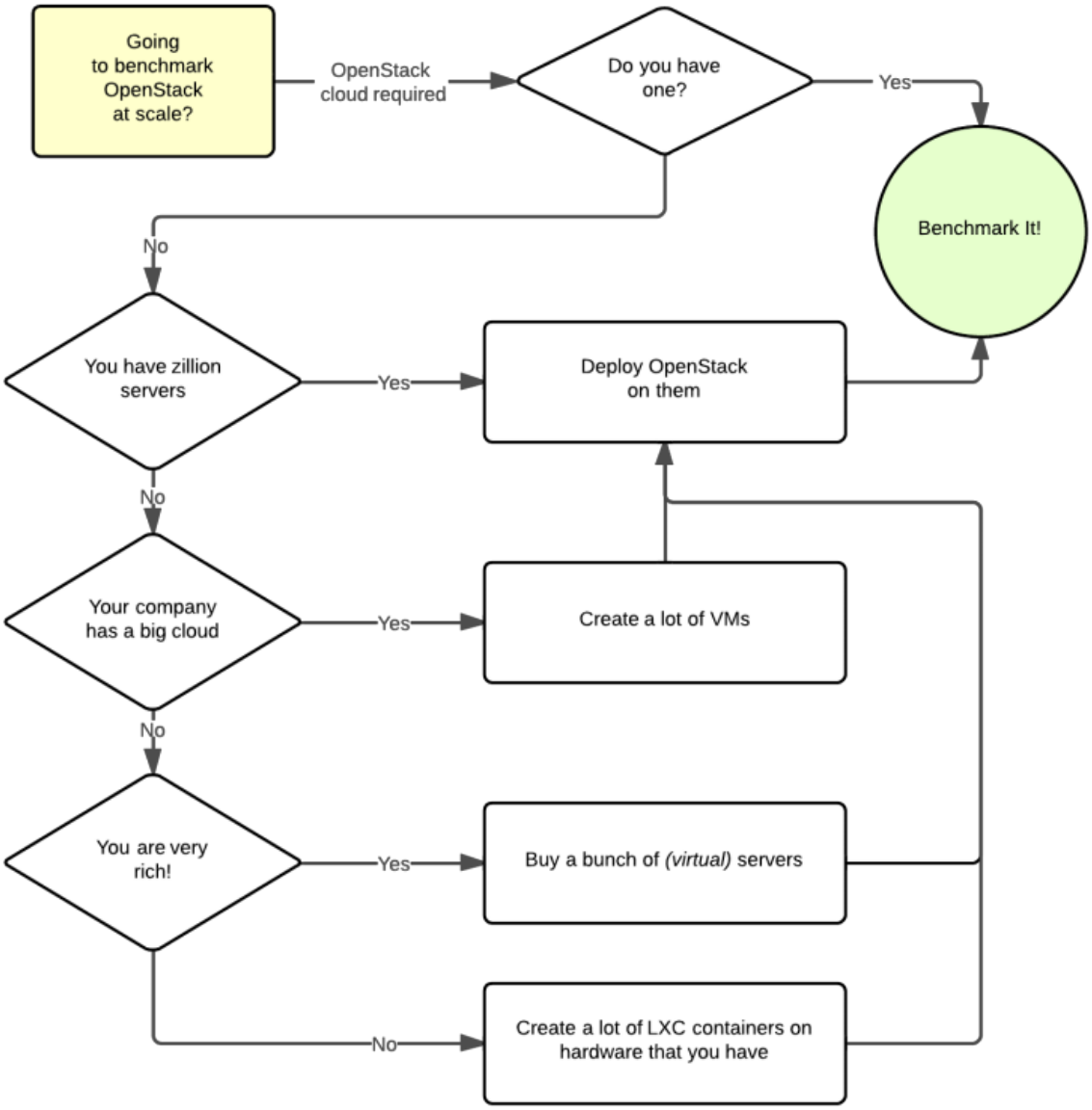
The diagram below shows how this is possible:



The actual **Rally core** consists of 4 main components, listed below in the order they go into action:

1. **Server Providers** - provide a **unified interface** for interaction with different **virtualization technologies** (*LXS*, *Virsh* etc.) and **cloud suppliers** (like *Amazon*): it does so via *ssh* access and in one *L3 network*;
2. **Deploy Engines** - deploy some OpenStack distribution (like *DevStack* or *FUEL*) before any benchmarking procedures take place, using servers retrieved from Server Providers;
3. **Verification** - runs *Tempest* (or another specific set of tests) against the deployed cloud to check that it works correctly, collects results & presents them in human readable form;
4. **Benchmark Engine** - allows to write parameterized benchmark scenarios & run them against the cloud.

It should become fairly obvious why Rally core needs to be split to these parts if you take a look at the following diagram that visualizes a rough **algorithm for starting benchmarking OpenStack at scale**. Keep in mind that there might be lots of different ways to set up virtual servers, as well as to deploy OpenStack to them.



1.2 Installation

1.2.1 Automated installation

```
git clone https://git.openstack.org/stackforge/rally
./rally/install_rally.sh
```

Notes: The installation script should be run as root or as a normal user using **sudo**. Rally requires either the Python 2.6 or the Python 2.7 version.

Alternatively, you can install Rally in a **virtual environment**:

```
git clone https://git.openstack.org/stackforge/rally
./rally/install_rally.sh -v
```

You also have to set up the **Rally database** after the installation is complete:

```
rally-manage db recreate
```

1.2.2 Rally with DevStack all-in-one installation

It is also possible to install Rally with DevStack. First, clone the corresponding repositories:

```
git clone https://git.openstack.org/openstack-dev/devstack
git clone https://github.com/stackforge/rally
```

Then, configure DevStack to run Rally:

```
cp rally/contrib/devstack/lib/rally devstack/lib/
cp rally/contrib/devstack/extras.d/70-rally.sh devstack/extras.d/
cd devstack
echo "enable_service rally" >> localrc
```

Finally, run DevStack as usually:

```
./stack.sh
```

1.2.3 Rally & Docker

First you need to install docker. Installing docker in ubuntu may be done by following:

```
$ sudo apt-get update
$ sudo apt-get install docker.io
$ sudo usermod -a -G docker `id -u -n` # add yourself to docker group
```

NOTE: re-login is required to apply users groups changes and actually use docker.

Pull docker image with rally:

```
$ docker pull rallyforge/rally
```

Or you may want to build rally image from source:

```
# first cd to rally source root dir
docker build -t myrally .
```

Since rally stores local settings in user's home dir and the database in /var/lib/rally/database, you may want to keep this directories outside of container. This may be done by the following steps:

```
cd
mkdir rally_home
sudo chown 65500 rally_home
docker run -t -i -v ~/rally_home:/home/rally rallyforge/rally
```

You may want to save last command as an alias:

```
echo 'alias dock_rally="docker run -t -i -v ~/rally_home:/home/rally rallyforge/rally"' >> ~/.bashrc
```

After executing `dock_rally` alias, or `docker run` you got bash running inside container with rally installed. You may do anytnig with rally, but you need to create db first:

```
user@box:~/rally$ dock_rally
rally@1cc98e0b5941:~$ rally-manage db recreate
rally@1cc98e0b5941:~$ rally deployment list
There are no deployments. To create a new deployment, use:
rally deployment create
rally@1cc98e0b5941:~$
```

More about docker: https://www.docker.com/

1.3 Rally step-by-step

In the following tutorial, we will guide you step-by-step through different use cases that might occur in Rally, starting with the easy ones and moving towards more complicated cases.

1.3.1 Step 0. Installation

Installing Rally is very simple. Just execute the following commands:

```
git clone https://git.openstack.org/stackforge/rally
./rally/install_rally.sh
```

Notes: The installation script should be run as root or as a normal user using **sudo**. Rally requires either the Python 2.6 or the Python 2.7 version.

There are also other installation options that you can find *here*.

Now that you have rally installed, you are ready to start *benchmarking OpenStack with it*!

1.3.2 Step 1. Setting up the environment and running a benchmark from samples

In this demo, we will show how to perform some basic operations in Rally, such as registering an OpenStack cloud, benchmarking it and generating benchmark reports.

We assume that you have a *Rally installation* and an already existing OpenStack deployment with Keystone available at *<KEYSTONE_AUTH_URL>*.

Registering an OpenStack deployment in Rally

First, you have to provide Rally with an Openstack deployment it is going to benchmark. This should be done either through OpenRC files or through deployment configuration files. In case you already have an *OpenRC*, it is extremely simple to register a deployment with the *deployment create* command:

```
$ . openrc admin admin
$ rally deployment create --fromenv --name=existing
+--------------------------------------+----------------------------+------------+------------------
| uuid                                 | created_at                 | name       | status
+--------------------------------------+----------------------------+------------+------------------
| 28f90d74-d940-4874-a8ee-04fda59576da | 2015-01-18 00:11:38.059983 | devstack_2 | deploy->finished
+--------------------------------------+----------------------------+------------+------------------
Using deployment : <Deployment UUID>
...
```

Alternatively, you can put the information about your cloud credentials into a JSON configuration file (let's call it existing.json). The *deployment create* command has a slightly different syntax in this case:

```
$ rally deployment create --file=existing.json --name=existing
+--------------------------------------+----------------------------+------------+------------------
| uuid                                 | created_at                 | name       | status
+--------------------------------------+----------------------------+------------+------------------
| 28f90d74-d940-4874-a8ee-04fda59576da | 2015-01-18 00:11:38.059983 | devstack_2 | deploy->finished
+--------------------------------------+----------------------------+------------+------------------
Using deployment : <Deployment UUID>
...
```

Note the last line in the output. It says that the just created deployment is now used by Rally; that means that all the benchmarking operations from now on are going to be performed on this deployment. Later we will show how to switch between different deployments.

Finally, the *deployment check* command enables you to verify that your current deployment is healthy and ready to be benchmarked:

```
$ rally deployment check
keystone endpoints are valid and following services are available:
+----------+----------------+-----------+
| services | type           | status    |
+----------+----------------+-----------+
| cinder   | volume         | Available |
| cinderv2 | volumev2       | Available |
| ec2      | ec2            | Available |
| glance   | image          | Available |
| heat     | orchestration  | Available |
| heat-cfn | cloudformation | Available |
| keystone | identity       | Available |
| nova     | compute        | Available |
| novav21  | computev21     | Available |
| s3       | s3             | Available |
+----------+----------------+-----------+
```

Benchmarking

Now that we have a working and registered deployment, we can start benchmarking it. The sequence of benchmarks to be launched by Rally should be specified in a *benchmark task configuration file* (either in *JSON* or in *YAML* format). Let's try one of the sample benchmark tasks available in samples/tasks/scenarios, say, the one that boots and deletes multiple servers (*samples/tasks/scenarios/nova/boot-and-delete.json*):

```
{
    "NovaServers.boot_and_delete_server": [
        {
            "args": {
                "flavor": {
                    "name": "m1.nano"
                },
                "image": {
                    "name": "^cirros.*uec$"
                },
                "force_delete": false
            },
            "runner": {
                "type": "constant",
                "times": 10,
```

```
                "concurrency": 2
            },
            "context": {
                "users": {
                    "tenants": 3,
                    "users_per_tenant": 2
                }
            }
        }
    ]
}
```

To start a benchmark task, run the task start command (you can also add the *-v* option to print more logging information):

```
$ rally task start samples/tasks/scenarios/nova/boot-and-delete.json
--------------------------------------------------------------------------------
 Preparing input task
--------------------------------------------------------------------------------

Input task is:
<Your task config here>

--------------------------------------------------------------------------------
 Task  6fd9a19f-5cf8-4f76-ab72-2e34bb1d4996: started
--------------------------------------------------------------------------------

Benchmarking... This can take a while...

To track task status use:

  rally task status
  or
  rally task detailed

--------------------------------------------------------------------------------
 Task 6fd9a19f-5cf8-4f76-ab72-2e34bb1d4996: finished
--------------------------------------------------------------------------------

test scenario NovaServers.boot_and_delete_server
args position 0
args values:
{u'args': {u'flavor': {u'name': u'm1.nano'},
           u'force_delete': False,
           u'image': {u'name': u'^cirros.*uec$'}},
 u'context': {u'users': {u'project_domain': u'default',
                         u'resource_management_workers': 30,
                         u'tenants': 3,
                         u'user_domain': u'default',
                         u'users_per_tenant': 2}},
 u'runner': {u'concurrency': 2, u'times': 10, u'type': u'constant'}}
+--------------------+-----------+-----------+-----------+---------------+---------------+---------+
| action             | min (sec) | avg (sec) | max (sec) | 90 percentile | 95 percentile | success |
+--------------------+-----------+-----------+-----------+---------------+---------------+---------+
| nova.boot_server   | 7.99      | 9.047     | 11.862    | 9.747         | 10.805        | 100.0%  |
| nova.delete_server | 4.427     | 4.574     | 4.772     | 4.677         | 4.725         | 100.0%  |
| total              | 12.556    | 13.621    | 16.37     | 14.252        | 15.311        | 100.0%  |
+--------------------+-----------+-----------+-----------+---------------+---------------+---------+
```

```
Load duration: 70.1310448647
Full duration: 87.545541048


HINTS:
* To plot HTML graphics with this data, run:
    rally task plot2html 6fd9a19f-5cf8-4f76-ab72-2e34bb1d4996 --out output.html

* To get raw JSON output of task results, run:
    rally task results 6fd9a19f-5cf8-4f76-ab72-2e34bb1d4996

Using task: 6fd9a19f-5cf8-4f76-ab72-2e34bb1d4996
```

Note that the Rally input task above uses *regular expressions* to specify the image and flavor name to be used for server creation, since concrete names might differ from installation to installation. If this benchmark task fails, then the reason for that might a non-existing image/flavor specified in the task. To check what images/flavors are available in the deployment you are currently benchmarking, you might use the *rally show* command:

```
$ rally show images
+--------------------------------------+-----------------------+------------+
|                 UUID                 |          Name         |  Size (B)  |
+--------------------------------------+-----------------------+------------+
| 8dfd6098-0c26-4cb5-8e77-1ecb2db0b8ae |  CentOS 6.5 (x86_64)  | 344457216 |
| 2b8d119e-9461-48fc-885b-1477abe2edc5 | CirrOS 0.3.1 (x86_64) |  13147648 |
+--------------------------------------+-----------------------+------------+

$ rally show flavors
+--------------------+-----------+-------+----------+-----------+-----------+
| ID                 |    Name   | vCPUs | RAM (MB) | Swap (MB) | Disk (GB) |
+--------------------+-----------+-------+----------+-----------+-----------+
| 1                  |  m1.tiny  |   1   |   512    |           |     1     |
| 2                  |  m1.small |   1   |   2048   |           |     20    |
| 3                  | m1.medium |   2   |   4096   |           |     40    |
| 4                  |  m1.large |   4   |   8192   |           |     80    |
| 5                  | m1.xlarge |   8   |  16384   |           |    160    |
+--------------------+-----------+-------+----------+-----------+-----------+
```

Report generation

One of the most beautiful things in Rally is its task report generation mechanism. It enables you to create illustrative and comprehensive HTML reports based on the benchmarking data. To create and open at once such a report for the last task you have launched, call:

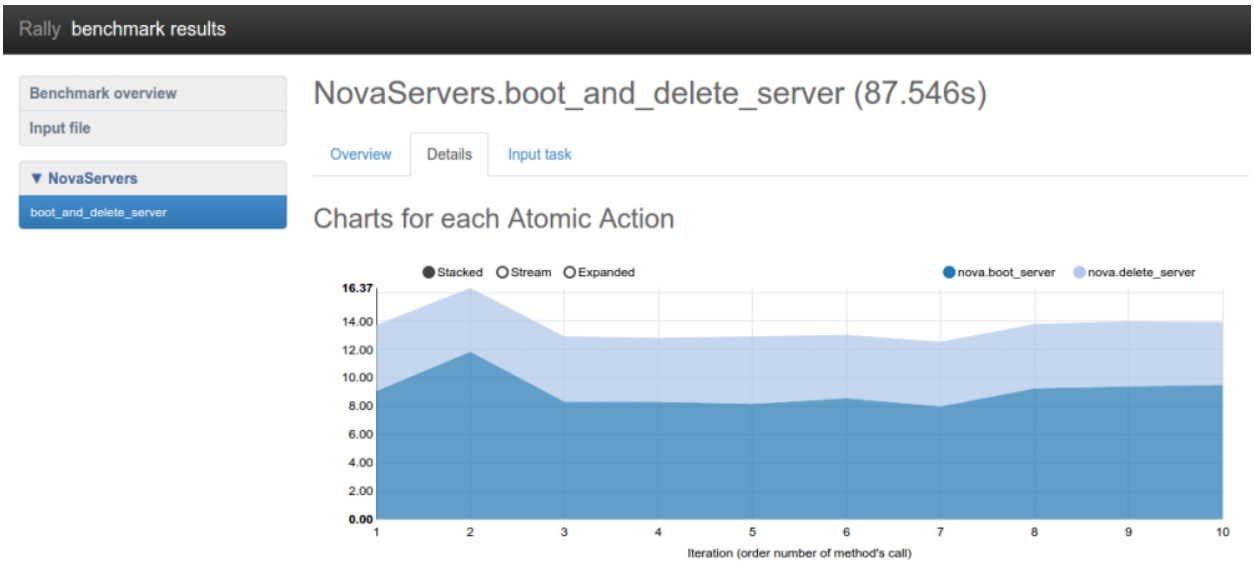
```
$ rally task report --out=report1.html --open
```

This will produce an HTML page with the overview of all the scenarios that you've included into the last benchmark task completed in Rally (in our case, this is just one scenario, and we will cover the topic of multiple scenarios in one task in *the next step of our tutorial*):

This aggregating table shows the duration of the load produced by the corresponding scenario (*"Load duration"*), the overall benchmark scenario execution time, including the duration of environment preparation with contexts (*"Full duration"*), the number of iterations of each scenario (*"Iterations"*), the type of the load used while running the scenario (*"Runner"*), the number of failed iterations (*"Errors"*) and finally whether the scenario has passed certain Success Criteria (*"SLA"*) that were set up by the user in the input configuration file (we will cover these criteria in *one of the next steps*).

By navigating in the left panel, you can switch to the detailed view of the benchmark results for the only scenario we included into our task, namely **NovaServers.boot_and_delete_server**:

This page, along with the description of the success criteria used to check the outcome of this scenario, shows some more detailed information and statistics about the duration of its iterations. Now, the *"Total durations"* table splits the duration of our scenario into the so-called **"atomic actions"**: in our case, the **"boot_and_delete_server"** scenario consists of two actions - **"boot_server"** and **"delete_server"**. You can also see how the scenario duration changed throughout is iterations in the *"Charts for the total duration"* section. Similar charts, but with atomic actions detalization, will arise if you switch to the *"Details"* tab of this page:



Note that all the charts on the report pages are very dynamic: you can change their contents by clicking the switches above the graph and see more information about its single points by hovering the cursor over these points.

Take some time to play around with these graphs and then move on to *the next step of our tutorial*.

1.3.3 Step 2. Running multiple benchmarks in a single task

Rally input task syntax

Rally comes with a really great collection of *benchmark scenarios* and in most real-world scenarios you will use multiple scenarios to test your OpenStack cloud. Rally makes it very easy to run **different benchmarks defined in a single benchmark task**. To do so, use the following syntax:

```
{
    "<ScenarioName1>": [<benchmark_config>, <benchmark_config2>, ...]
    "<ScenarioName2>": [<benchmark_config>, ...]
}
```

where <*benchmark_config*>, as before, is a dictionary:

```
{
    "args": { scenario-specific arguments },
    "runner": {"type": ..., }
    ...
}
```

Multiple benchmarks in a single task

As an example, let's edit our configuration file from *step 1* so that it prescribes Rally to launch not only the **NovaServers.boot_and_delete_server** scenario, but also the **KeystoneBasic.create_delete_user** scenario. All we have to do is to append the configuration of the second scenario as yet another top-level key of our json file:

multiple-scenarios.json

```
{
    "NovaServers.boot_and_delete_server": [
```

```
        {
            "args": {
                "flavor": {
                    "name": "m1.nano"
                },
                "image": {
                    "name": "^cirros.*uec$"
                },
                "force_delete": false
            },
            "runner": {
                "type": "constant",
                "times": 10,
                "concurrency": 2
            },
            "context": {
                "users": {
                    "tenants": 3,
                    "users_per_tenant": 2
                }
            }
        }
    ],
    "KeystoneBasic.create_delete_user": [
        {
            "args": {
                "name_length": 10
            },
            "runner": {
                "type": "constant",
                "times": 10,
                "concurrency": 3
            }
        }
    ]
}
```

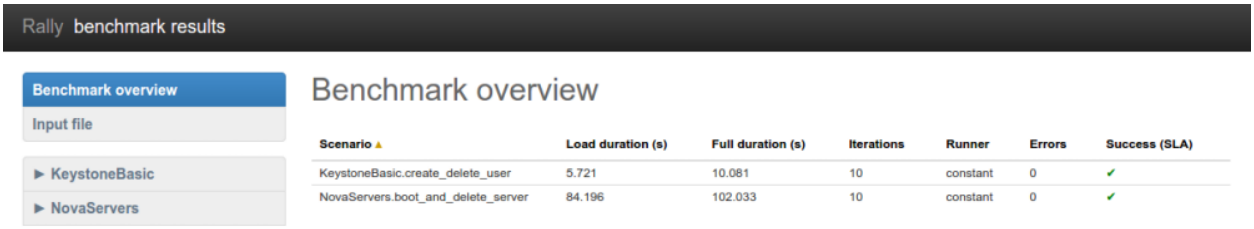
Now you can start this benchmark task as usually:

```
$ rally task start multiple-scenarios.json
...
+--------------------+-----------+-----------+-----------+---------------+---------------+---------+
| action             | min (sec) | avg (sec) | max (sec) | 90 percentile | 95 percentile | success |
+--------------------+-----------+-----------+-----------+---------------+---------------+---------+
| nova.boot_server   | 8.06      | 11.354    | 18.594    | 18.54         | 18.567        | 100.0%  |
| nova.delete_server | 4.364     | 5.054     | 6.837     | 6.805         | 6.821         | 100.0%  |
| total              | 12.572    | 16.408    | 25.396    | 25.374        | 25.385        | 100.0%  |
+--------------------+-----------+-----------+-----------+---------------+---------------+---------+
Load duration: 84.1959171295
Full duration: 102.033041
-------------------------------------------------------------------------------

...

+----------------------+-----------+-----------+-----------+---------------+---------------+---------
| action               | min (sec) | avg (sec) | max (sec) | 90 percentile | 95 percentile | success
+----------------------+-----------+-----------+-----------+---------------+---------------+---------
| keystone.create_user | 0.676     | 0.875     | 1.03      | 1.02          | 1.025         | 100.0%
```

```
| keystone.delete_user | 0.407     | 0.647     | 0.84      | 0.739         | 0.79          | 100.0%
| total                | 1.082     | 1.522     | 1.757     | 1.724         | 1.741         | 100.0%
+----------------------+-----------+-----------+-----------+---------------+---------------+--------
Load duration: 5.72119688988
Full duration: 10.0808410645

...
```

Note that the HTML reports you can generate by typing **rally task report –out=report_name.html** after your benchmark task has completed will get richer as your benchmark task configuration file includes more benchmark scenarios. Let's take a look at the report overview page for a task that covers all the scenarios available in Rally:

```
$ rally task report --out=report_multiple_scenarios.html --open
```



Multiple configurations of the same scenario

Yet another thing you can do in Rally is to launch **the same benchmark scenario multiple times with different configurations**. That's why our configuration file stores a list for the key *"NovaServers.boot_and_delete_server"*: you can just append a different configuration of this benchmark scenario to this list to get it. Let's say, you want to run the **boot_and_delete_server** scenario twice: first using the *"m1.nano"* flavor and then using the *"m1.tiny"* flavor:

multiple-configurations.json

```
{
    "NovaServers.boot_and_delete_server": [
        {
            "args": {
                "flavor": {
                    "name": "m1.nano"
                },
                "image": {
                    "name": "^cirros.*uec$"
                },
                "force_delete": false
            },
            "runner": {...},
            "context": {...}
        },
        {
            "args": {
                "flavor": {
                    "name": "m1.tiny"
                },
                "image": {
                    "name": "^cirros.*uec$"
```

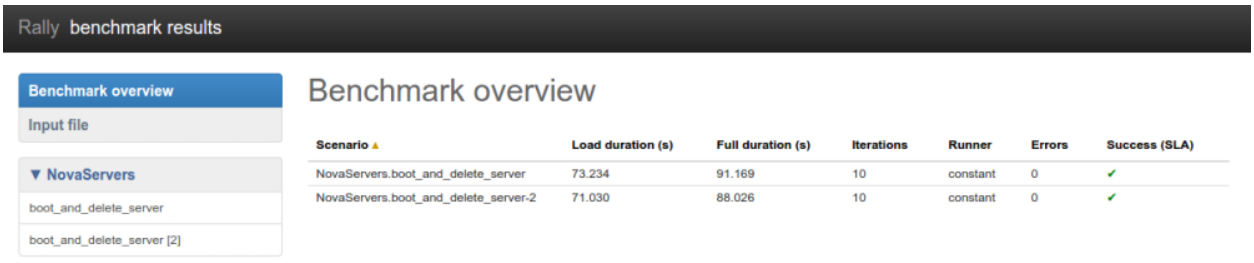
```
                },
                "force_delete": false
            },
            "runner": {...},
            "context": {...}
        }
    ]
}
```

That's it! You will get again the results for each configuration separately:

```
$ rally task start --task=multiple-configurations.json
...
+--------------------+-----------+-----------+-----------+---------------+---------------+---------+
| action             | min (sec) | avg (sec) | max (sec) | 90 percentile | 95 percentile | success |
+--------------------+-----------+-----------+-----------+---------------+---------------+---------+
| nova.boot_server   | 7.896     | 9.433     | 13.14     | 11.329        | 12.234        | 100.0%  |
| nova.delete_server | 4.435     | 4.898     | 6.975     | 5.144         | 6.059         | 100.0%  |
| total              | 12.404    | 14.331    | 17.979    | 16.72         | 17.349        | 100.0%  |
+--------------------+-----------+-----------+-----------+---------------+---------------+---------+
Load duration: 73.2339417934
Full duration: 91.1692159176
--------------------------------------------------------------------------------

...

+--------------------+-----------+-----------+-----------+---------------+---------------+---------+
| action             | min (sec) | avg (sec) | max (sec) | 90 percentile | 95 percentile | success |
+--------------------+-----------+-----------+-----------+---------------+---------------+---------+
| nova.boot_server   | 8.207     | 8.91      | 9.823     | 9.692         | 9.758         | 100.0%  |
| nova.delete_server | 4.405     | 4.767     | 6.477     | 4.904         | 5.691         | 100.0%  |
| total              | 12.735    | 13.677    | 16.301    | 14.596        | 15.449        | 100.0%  |
+--------------------+-----------+-----------+-----------+---------------+---------------+---------+
Load duration: 71.029528141
Full duration: 88.0259010792
...
```

The HTML report will also look similar to what we have seen before:

```
$ rally task report --out=report_multiple_configuraions.html --open
```



1.3.4 Step 3. Adding success criteria (SLA) for benchmarks

SLA - Service-Level Agreement (Success Criteria)

Rally allows you to set success criteria (also called *SLA - Service-Level Agreement*) for every benchmark. Rally will automatically check them for you.

To configure the SLA, add the *"sla"* section to the configuration of the corresponding benchmark (the check name is a key associated with its target value). You can combine different success criteria:

```
{
    "NovaServers.boot_and_delete_server": [
        {
            "args": {
                ...
            },
            "runner": {
                ...
            },
            "context": {
                ...
            },
            "sla": {
                "max_seconds_per_iteration": 10,
                "failure_rate": {
                    "max": 25
                }
            }
        }
    ]
}
```

Such configuration will mark the **NovaServers.boot_and_delete_server** benchmark scenario as not successful if either some iteration took more than 10 seconds or more than 25% iterations failed.

Checking SLA

Let us show you how Rally SLA work using a simple example based on **Dummy benchmark scenarios**. These scenarios actually do not perform any OpenStack-related stuff but are very useful for testing the behavious of Rally. Let us put in a new task, *test-sla.json*, 2 scenarios – one that does nothing and another that just throws an exception:

```
{
    "Dummy.dummy": [
        {
            "args": {},
            "runner": {
                "type": "constant",
                "times": 5,
                "concurrency": 2
            },
            "context": {
                "users": {
                    "tenants": 3,
                    "users_per_tenant": 2
                }
            },
            "sla": {
```

```
                "failure_rate": {"max": 0.0}
            }
        }
    ],
    "Dummy.dummy_exception": [
        {
            "args": {},
            "runner": {
                "type": "constant",
                "times": 5,
                "concurrency": 2
            },
            "context": {
                "users": {
                    "tenants": 3,
                    "users_per_tenant": 2
                }
            },
            "sla": {
                "failure_rate": {"max": 0.0}
            }
        }
    ]
}
```

Note that both scenarios in these tasks have the **maximum failure rate of 0%** as their **success criterion**. We expect that the first scenario will pass this criterion while the second will fail it. Let's start the task:

```
$ rally task start test-sla.json
...
```

After the task completes, run *rally task sla_check* to check the results again the success criteria you defined in the task:

```
$ rally task sla_check
+-----------------------+-----+--------------+--------+-----------------------------------------
| benchmark             | pos | criterion    | status | detail
+-----------------------+-----+--------------+--------+-----------------------------------------
| Dummy.dummy           | 0   | failure_rate | PASS   | Maximum failure rate percent 0.0% failures, r
| Dummy.dummy_exception | 0   | failure_rate | FAIL   | Maximum failure rate percent 0.0% failures, r
+-----------------------+-----+--------------+--------+-----------------------------------------
```

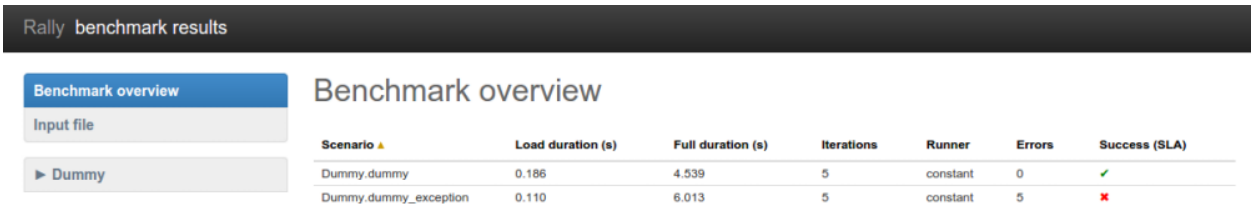
Exactly as expected.

SLA in task report

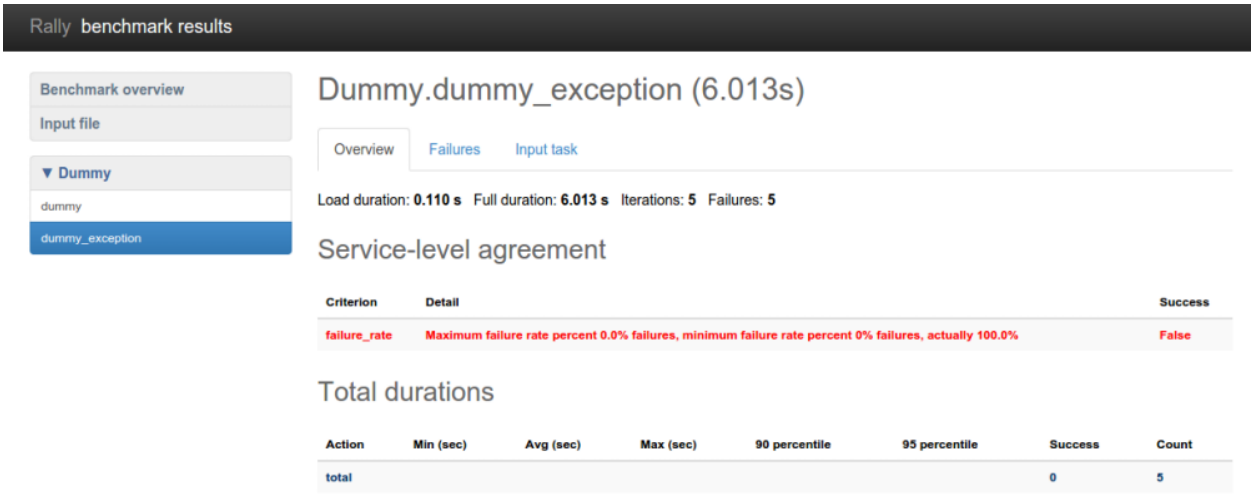
SLA checks are nicely visualized in task reports. Generate one:

```
$ rally task report --out=report_sla.html --open
```

Benchmark scenarios that have passed SLA have a green check on the overview page:



Somewhat more detailed information about SLA is displayed on the scenario pages:



Success criteria present a very useful concept that enables not only to analyze the outcome of your benchmark tasks, but also to control their execution. In the *the next section of our tutorial*, we will show how to use SLA to abort the load generation before your OpenStack goes wrong.

1.3.5 Step 4. Aborting load generation on success criteria failure

Benchmarking pre-production and production OpenStack clouds is not a trivial task. From the one side it's important to reach the OpenStack cloud's limits, from the other side the cloud shouldn't be damaged. Rally aims to make this task as simple as possible. Since the very beginning Rally was able to generate enough load for any OpenStack cloud. Generating to big load was the major issue for production clouds, because Rally didn't know how to stop the load until it was to late. Finally I am happy to say that we solved this issue.

With the **"stop on SLA failure"** feature, however, things are much better.

This feature can be easily tested in real life by running one of the most important and plain benchmark scenario called *"KeystoneBasic.authenticate"*. This scenario just tries to authenticate from users that were pre-created by Rally. Rally input task looks as follows (*auth.yaml*):

```
---
  Authenticate.keystone:
  -
    runner:
      type: "rps"
```

```
    times: 6000
    rps: 50
  context:
    users:
      tenants: 5
      users_per_tenant: 10
  sla:
    max_avg_duration: 5
```

In human-readable form this input task means: *Create 5 tenants with 10 users in each, after that try to authenticate to Keystone 6000 times performing 50 authentications per second (running new authentication request every 20ms). Each time we are performing authentication from one of the Rally pre-created user. This task passes only if max average duration of authentication takes less than 5 seconds.*

Note that this test is quite dangerous because it can DDoS Keystone. We are running more and more simultaneously authentication requests and things may go wrong if something is not set properly (like on my DevStack deployment in Small VM on my laptop).

Let's run Rally task with **an argument that prescribes Rally to stop load on SLA failure**:

```
$ rally task start --abort-on-sla-failure auth.yaml
```

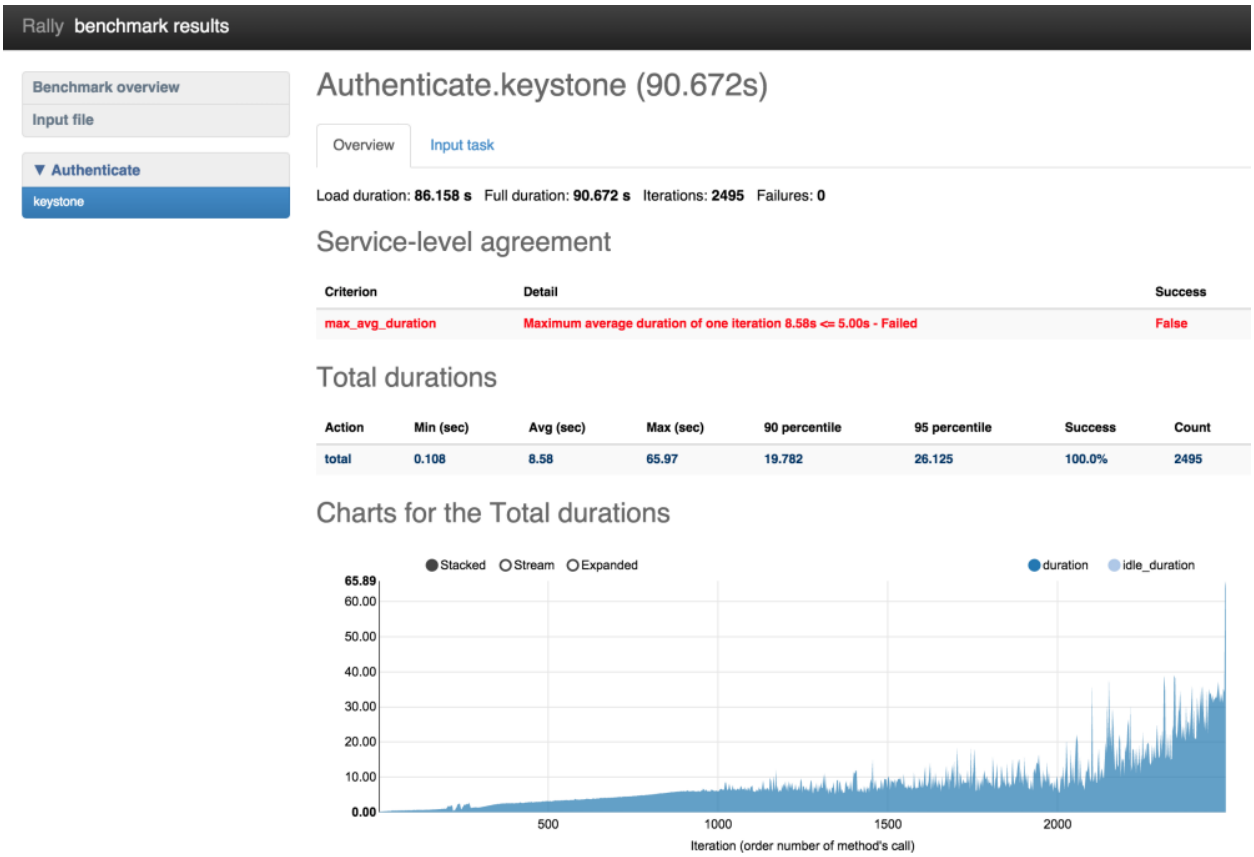
```
....
+--------+-----------+-----------+-----------+---------------+---------------+---------+-------+
| action | min (sec) | avg (sec) | max (sec) | 90 percentile | 95 percentile | success | count |
+--------+-----------+-----------+-----------+---------------+---------------+---------+-------+
| total  | 0.108     | 8.58      | 65.97     | 19.782        | 26.125        | 100.0%  | 2495  |
+--------+-----------+-----------+-----------+---------------+---------------+---------+-------+
```

On the resulting table there are 2 interesting things:

1. Average duration was 8.58 sec which is more than 5 seconds
2. Rally performed only 2495 (instead of 6000) authentication requests

To understand better what has happened let's generate HTML report:

```
$ rally task report --out auth_report.html
```



On the chart with durations we can observe that the duration of authentication request reaches 65 seconds at the end of the load generation. **Rally stopped load at the very last moment just before the mad things happened. The reason why it runs so many attempts to authenticate is because of not enough good success criteria.** We had to run a lot of iterations to make average duration bigger than 5 seconds. Let's chose better success criteria for this task and run it one more time.

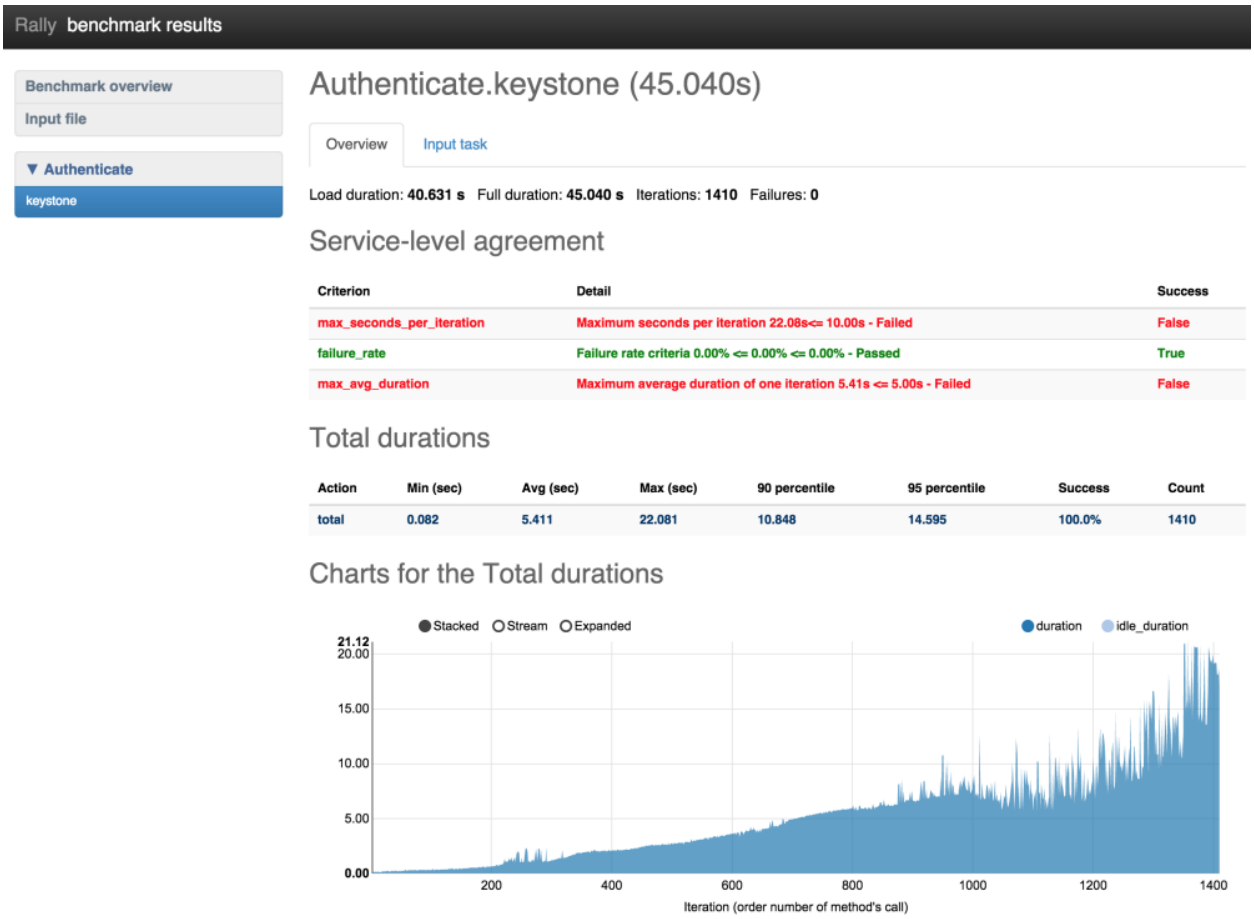
```
---
  Authenticate.keystone:
  -
    runner:
      type: "rps"
      times: 6000
      rps: 50
    context:
      users:
        tenants: 5
        users_per_tenant: 10
    sla:
      max_avg_duration: 5
      max_seconds_per_iteration: 10
      failure_rate:
        max: 0
```

Now our task is going to be successful if the following three conditions hold:

1. maximum average duration of authentication should be less than 5 seconds
2. maximum duration of any authentication should be less than 10 seconds
3. no failed authentication should appear

Let's run it!

```
$ rally task start --abort-on-sla-failure auth.yaml

...
+--------+-----------+-----------+-----------+---------------+---------------+---------+-------+
| action | min (sec) | avg (sec) | max (sec) | 90 percentile | 95 percentile | success | count |
+--------+-----------+-----------+-----------+---------------+---------------+---------+-------+
| total  | 0.082     | 5.411     | 22.081    | 10.848        | 14.595        | 100.0%  | 1410  |
+--------+-----------+-----------+-----------+---------------+---------------+---------+-------+
```



This time load stopped after 1410 iterations versus 2495 which is much better. The interesting thing on this chart is that first occurence of "> 10 second" authentication happened on 950 iteration. The reasonable question: "Why Rally run 500 more authentication requests then?". This appears from the math: During the execution of **bad** authentication (10 seconds) Rally performed about 50 request/sec * 10 sec = 500 new requests as a result we run 1400 iterations instead of 950.

(based on: http://boris-42.me/rally-tricks-stop-load-before-your-openstack-goes-wrong/)

1.3.6 Step 5. Working with multiple OpenStack clouds

Rally is an awesome tool that allows you to work with multiple clouds and can itself deploy them. We already know how to work with *a single cloud*. Let us now register 2 clouds in Rally: the one that we have access to and the other that we know is registered with wrong credentials.

```
$ . openrc admin admin  # openrc with correct credentials
$ rally deployment create --fromenv --name=cloud-1
+--------------------------------------+----------------------------+------------+-------------------
| uuid                                 | created_at                 | name       | status
+--------------------------------------+----------------------------+------------+-------------------
| 4251b491-73b2-422a-aecb-695a94165b5e | 2015-01-18 00:11:14.757203 | cloud-1    | deploy->finished
+--------------------------------------+----------------------------+------------+-------------------
Using deployment: 4251b491-73b2-422a-aecb-695a94165b5e
~/.rally/openrc was updated
...

$ . bad_openrc admin admin  # openrc with wrong credentials
$ rally deployment create --fromenv --name=cloud-2
+--------------------------------------+----------------------------+------------+-------------------
| uuid                                 | created_at                 | name       | status
+--------------------------------------+----------------------------+------------+-------------------
| 658b9bae-1f9c-4036-9400-9e71e88864fc | 2015-01-18 00:38:26.127171 | cloud-2    | deploy->finished
+--------------------------------------+----------------------------+------------+-------------------
Using deployment: 658b9bae-1f9c-4036-9400-9e71e88864fc
~/.rally/openrc was updated
...
```

Let us now list the deployments we have created:

```
$ rally deployment list
+--------------------------------------+----------------------------+------------+-------------------
| uuid                                 | created_at                 | name       | status
+--------------------------------------+----------------------------+------------+-------------------
| 4251b491-73b2-422a-aecb-695a94165b5e | 2015-01-05 00:11:14.757203 | cloud-1    | deploy->finished
| 658b9bae-1f9c-4036-9400-9e71e88864fc | 2015-01-05 00:40:58.451435 | cloud-2    | deploy->finished
+--------------------------------------+----------------------------+------------+-------------------
```

Note that the second is marked as **"active"** because this is the deployment we have created most recently. This means that it will be automatically (unless its UUID or name is passed explicitly via the *–deployment* parameter) used by the commands that need a deployment, like *rally task start ...* or *rally deployment check*:

```
$ rally deployment check
Authentication Issues: wrong keystone credentials specified in your endpoint properties. (HTTP 401).

$ rally deployment check --deployment=cloud-1
keystone endpoints are valid and following services are available:
+----------+----------------+-----------+
| services | type           | status    |
+----------+----------------+-----------+
| cinder   | volume         | Available |
| cinderv2 | volumev2       | Available |
| ec2      | ec2            | Available |
| glance   | image          | Available |
| heat     | orchestration  | Available |
| heat-cfn | cloudformation | Available |
| keystone | identity       | Available |
| nova     | compute        | Available |
| novav21  | computev21     | Available |
| s3       | s3             | Available |
+----------+----------------+-----------+
```

You can also switch the active deployment using the **rally deployment use** command:

```
$ rally deployment use cloud-1
Using deployment: 658b9bae-1f9c-4036-9400-9e71e88864fc
~/.rally/openrc was updated
...

$ rally deployment check
keystone endpoints are valid and following services are available:
+----------+----------------+-----------+
| services | type           | status    |
+----------+----------------+-----------+
| cinder   | volume         | Available |
| cinderv2 | volumev2       | Available |
| ec2      | ec2            | Available |
| glance   | image          | Available |
| heat     | orchestration  | Available |
| heat-cfn | cloudformation | Available |
| keystone | identity       | Available |
| nova     | compute        | Available |
| novav21  | computev21     | Available |
| s3       | s3             | Available |
+----------+----------------+-----------+
```

Note the first two lines of the CLI output for the *rally deployment use* command. They tell you the UUID of the new active deployment and also say that the *~/.rally/openrc* file was updated – this is the place where the "active" UUID is actually stored by Rally.

One last detail about managing different deployments in Rally is that the *rally task list* command outputs only those tasks that were run against the currently active deployment, and you have to provide the *–all-deployments* parameter to list all the tasks:

```
$ rally task list
+--------------------------------------+-----------------+-----------------------------+--------------
| uuid                                 | deployment_name | created_at                  | duration
+--------------------------------------+-----------------+-----------------------------+--------------
| c21a6ecb-57b2-43d6-bbbb-d7a827f1b420 | cloud-1         | 2015-01-05 01:00:42.099596 | 0:00:13.41922
| f6dad6ab-1a6d-450d-8981-f77062c6ef4f | cloud-1         | 2015-01-05 01:05:57.653253 | 0:00:14.16049
+--------------------------------------+-----------------+-----------------------------+--------------
$ rally task list --all-deployment
+--------------------------------------+-----------------+-----------------------------+--------------
| uuid                                 | deployment_name | created_at                  | duration
+--------------------------------------+-----------------+-----------------------------+--------------
| c21a6ecb-57b2-43d6-bbbb-d7a827f1b420 | cloud-1         | 2015-01-05 01:00:42.099596 | 0:00:13.41922
| f6dad6ab-1a6d-450d-8981-f77062c6ef4f | cloud-1         | 2015-01-05 01:05:57.653253 | 0:00:14.16049
| 6fd9a19f-5cf8-4f76-ab72-2e34bb1d4996 | cloud-2         | 2015-01-05 01:14:51.428958 | 0:00:15.04226
+--------------------------------------+-----------------+-----------------------------+--------------
```

1.3.7 Step 6. Discovering more benchmark scenarios in Rally

Scenarios in the Rally repository

Rally currently comes with a great collection of benchmark scenarios that use the API of different OpenStack projects like **Keystone**, **Nova**, **Cinder**, **Glance** and so on. The good news is that you can combine multiple benchmark scenarios in one task to benchmark your cloud in a comprehensive way.

First, let's see what scenarios are available in Rally. One of the ways to discover these scenario is just to inspect their source code.

Rally built-in search engine

A much more convenient way to learn about different benchmark scenarios in Rally, however, is to use a special **search engine** embedded into its Command-Line Interface, which, for a given **search query**, prints documentation for the corresponding benchmark scenario (and also supports other Rally entities like SLA).

To search for some specific benchmark scenario by its name or by its group, use the **rally info find <query>** command:

```
$ rally info find create_meter_and_get_stats
--------------------------------------------------------------------------------
 CeilometerStats.create_meter_and_get_stats (benchmark scenario)
--------------------------------------------------------------------------------

Create a meter and fetch its statistics.

Meter is first created and then statistics is fetched for the same
using GET /v2/meters/(meter_name)/statistics.

Parameters:
    - kwargs: contains optional arguments to create a meter

$ rally info find some_non_existing_benchmark
Failed to find any docs for query: 'some_non_existing_benchmark'
```

You can also get the list of different benchmark scenario groups available in Rally by typing **rally info BenchmarkScenarios** command:

```
$ rally info BenchmarkScenarios
--------------------------------------------------------------------------------
 Rally - Benchmark scenarios
--------------------------------------------------------------------------------

Benchmark scenarios are what Rally actually uses to test the performance of an OpenStack deployment.
Each Benchmark scenario implements a sequence of atomic operations (server calls) to simulate
interesing user/operator/client activity in some typical use case, usually that of a specific OpenSta
project. Iterative execution of this sequence produces some kind of load on the target cloud.
Benchmark scenarios play the role of building blocks in benchmark task configuration files.

Scenarios in Rally are put together in groups. Each scenario group is concentrated on some specific
OpenStack functionality. For example, the "NovaServers" scenario group contains scenarios that employ
several basic operations available in Nova.

 List of Benchmark scenario groups:
------------------------------------------------------------------------------------------------
 Name                        Description
------------------------------------------------------------------------------------------------
 Authenticate                Benchmark scenarios for the authentication mechanism.
 CeilometerAlarms            Benchmark scenarios for Ceilometer Alarms API.
 CeilometerMeters            Benchmark scenarios for Ceilometer Meters API.
 CeilometerQueries           Benchmark scenarios for Ceilometer Queries API.
 CeilometerResource          Benchmark scenarios for Ceilometer Resource API.
 CeilometerStats             Benchmark scenarios for Ceilometer Stats API.
 CinderVolumes               Benchmark scenarios for Cinder Volumes.
 DesignateBasic              Basic benchmark scenarios for Designate.
 Dummy                       Dummy benchmarks for testing Rally benchmark engine at scale.
 GlanceImages                Benchmark scenarios for Glance images.
 HeatStacks                  Benchmark scenarios for Heat stacks.
 KeystoneBasic               Basic benchmark scenarios for Keystone.
 NeutronNetworks             Benchmark scenarios for Neutron.
```

```
 NovaSecGroup               Benchmark scenarios for Nova security groups.
 NovaServers                Benchmark scenarios for Nova servers.
 Quotas                     Benchmark scenarios for quotas.
 Requests                   Benchmark scenarios for HTTP requests.
 SaharaClusters             Benchmark scenarios for Sahara clusters.
 SaharaJob                  Benchmark scenarios for Sahara jobs.
 SaharaNodeGroupTemplates   Benchmark scenarios for Sahara node group templates.
 TempestScenario            Benchmark scenarios that launch Tempest tests.
 VMTasks                    Benchmark scenarios that are to be run inside VM instances.
 ZaqarBasic                 Benchmark scenarios for Zaqar.
-------------------------------------------------------------------------------

To get information about benchmark scenarios inside each scenario group, run:
  $ rally info find <ScenarioGroupName>
```

1.3.8 Step 7. Deploying OpenStack from Rally

Along with supporting already existing OpenStack deployments, Rally itself can **deploy OpenStack automatically** by using one of its *deployment engines*. Take a look at other deployment configuration file samples. For example, *devstack-in-existing-servers.json* is a deployment configuration file that tells Rally to deploy OpenStack with **Devstack** on the existing servers with given credentials:

```
{
    "type": "DevstackEngine",
    "provider": {
        "type": "ExistingServers",
        "credentials": [{"user": "root", "host": "10.2.0.8"}]
    }
}
```

You can try to deploy OpenStack in your Virtual Machine using this script. Edit the configuration file with your IP address/user name and run, as usual:

```
$ rally deployment create --file=samples/deployments/devstack-in-existing-servers.json.json --name=ne
+--------------------------+--------------------------+--------------+------------------+
|            uuid          |         created_at       |     name     |      status      |
+--------------------------+--------------------------+--------------+------------------+
|     <Deployment UUID>    | 2015-01-10 22:00:28.270941 | new-devstack | deploy->finished |
+--------------------------+--------------------------+--------------+------------------+
Using deployment : <Deployment UUID>
```

1.3.9 Step 8. Rally task templates

Basic template syntax

A nice feature of the input task format used in Rally is that it supports the **template syntax** based on Jinja2. This turns out to be extremely useful when, say, you have a fixed structure of your task but you want to parameterize this task in some way. For example, imagine your input task file (*task.yaml*) runs a set of Nova scenarios:

```
---

  NovaServers.boot_and_delete_server:
    -
      args:
        flavor:
```

```
            name: "m1.tiny"
        image:
            name: "^cirros.*uec$"
      runner:
        type: "constant"
        times: 2
        concurrency: 1
      context:
        users:
          tenants: 1
          users_per_tenant: 1

  NovaServers.resize_server:
    -
      args:
        flavor:
            name: "m1.tiny"
        image:
            name: "^cirros.*uec$"
        to_flavor:
            name: "m1.small"
      runner:
        type: "constant"
        times: 3
        concurrency: 1
      context:
        users:
          tenants: 1
          users_per_tenant: 1
```

In all the three scenarios above, the *"^cirros.*uec$"* image is passed to the scenario as an argument (so that these scenarios use an appropriate image while booting servers). Let's say you want to run the same set of scenarios with the same runner/context/sla, but you want to try another image while booting server to compare the performance. The most elegant solution is then to turn the image name into a template variable:

```
---

  NovaServers.boot_and_delete_server:
    -
      args:
        flavor:
            name: "m1.tiny"
        image:
            name: {{image_name}}
      runner:
        type: "constant"
        times: 2
        concurrency: 1
      context:
        users:
          tenants: 1
          users_per_tenant: 1

  NovaServers.resize_server:
    -
      args:
        flavor:
            name: "m1.tiny"
```

```
        image:
            name: {{image_name}}
        to_flavor:
            name: "m1.small"
      runner:
        type: "constant"
        times: 3
        concurrency: 1
      context:
        users:
          tenants: 1
          users_per_tenant: 1
```

and then pass the argument value for **{{image_name}}** when starting a task with this configuration file. Rally provides you with different ways to do that:

1. Pass the argument values directly in the command-line interface (with either a JSON or YAML dictionary):

```
$ rally task start task.yaml --task-args '{"image_name": "^cirros.*uec$"}'
$ rally task start task.yaml --task-args 'image_name: "^cirros.*uec$"'
```

2. Refer to a file that specifies the argument values (JSON/YAML):

```
$ rally task start task.yaml --task-args-file args.json
$ rally task start task.yaml --task-args-file args.yaml
```

where the files containing argument values should look as follows:

args.json:

```
{
    "image_name": "^cirros.*uec$"
}
```

args.yaml:

```
---

  image_name: "^cirros.*uec$"
```

Passed in either way, these parameter values will be substituted by Rally when starting a task:

```
$ rally task start task.yaml --task-args "image_name: "^cirros.*uec$""
--------------------------------------------------------------------------------
 Preparing input task
--------------------------------------------------------------------------------

Input task is:
---

  NovaServers.boot_and_delete_server:
    -
      args:
        flavor:
            name: "m1.tiny"
        image:
            name: ^cirros.*uec$
      runner:
        type: "constant"
        times: 2
        concurrency: 1
```

```
      context:
        users:
          tenants: 1
          users_per_tenant: 1

  NovaServers.resize_server:
    -
      args:
        flavor:
            name: "m1.tiny"
        image:
            name: ^cirros.*uec$
        to_flavor:
            name: "m1.small"
      runner:
        type: "constant"
        times: 3
        concurrency: 1
      context:
        users:
          tenants: 1
          users_per_tenant: 1

--------------------------------------------------------------------------------
 Task  cbf7eb97-0f1d-42d3-a1f1-3cc6f45ce23f: started
--------------------------------------------------------------------------------

Benchmarking... This can take a while...
```

Using the default values

Note that the Jinja2 template syntax allows you to set the default values for your parameters. With default values set, your task file will work even if you don't parameterize it explicitly while starting a task. The default values should be set using the *{% set ... %}* clause (*task.yaml*):

```
{% set image_name = image_name or "^cirros.*uec$" %}
---

  NovaServers.boot_and_delete_server:
    -
      args:
        flavor:
            name: "m1.tiny"
        image:
            name: {{image_name}}
      runner:
        type: "constant"
        times: 2
        concurrency: 1
      context:
        users:
          tenants: 1
          users_per_tenant: 1

    ...
```

If you don't pass the value for *{{image_name}}* while starting a task, the default one will be used:

```
$ rally task start task.yaml
--------------------------------------------------------------------------------
 Preparing input task
--------------------------------------------------------------------------------

Input task is:
---

  NovaServers.boot_and_delete_server:
    -
      args:
        flavor:
            name: "m1.tiny"
        image:
            name: ^cirros.*uec$
      runner:
        type: "constant"
        times: 2
        concurrency: 1
      context:
        users:
          tenants: 1
          users_per_tenant: 1

    ...
```

Advanced templates

Rally makes it possible to use all the power of Jinja2 template syntax, including the mechanism of **built-in functions**. This enables you to construct elegant task files capable of generating complex load on your cloud.

As an example, let us make up a task file that will create new users with increasing concurrency. The input task file (*task.yaml*) below uses the Jinja2 **for-endfor** construct to accomplish that:

```
---

  KeystoneBasic.create_user:
  {% for i in range(2, 11, 2) %}
    -
      args:
        name_length: 10
      runner:
        type: "constant"
        times: 10
        concurrency: {{i}}
      sla:
        failure_rate:
          max: 0
  {% endfor %}
```

In this case, you don't need to pass any arguments via *–task-args/–task-args-file*, but as soon as you start this task, Rally will automatically unfold the for-loop for you:

```
$ rally task start task.yaml
--------------------------------------------------------------------------------
 Preparing input task
--------------------------------------------------------------------------------
```

```
Input task is:
---

  KeystoneBasic.create_user:

    -
      args:
        name_length: 10
      runner:
        type: "constant"
        times: 10
        concurrency: 2
      sla:
        failure_rate:
          max: 0

    -
      args:
        name_length: 10
      runner:
        type: "constant"
        times: 10
        concurrency: 4
      sla:
        failure_rate:
          max: 0

    -
      args:
        name_length: 10
      runner:
        type: "constant"
        times: 10
        concurrency: 6
      sla:
        failure_rate:
          max: 0

    -
      args:
        name_length: 10
      runner:
        type: "constant"
        times: 10
        concurrency: 8
      sla:
        failure_rate:
          max: 0

    -
      args:
        name_length: 10
      runner:
        type: "constant"
        times: 10
        concurrency: 10
      sla:
        failure_rate:
```

```
          max: 0


--------------------------------------------------------------------------------
 Task  ea7e97e3-dd98-4a81-868a-5bb5b42b8610: started
--------------------------------------------------------------------------------

Benchmarking... This can take a while...
```

As you can see, the Rally task template syntax is a simple but powerful mechanism that not only enables you to write elegant task configurations, but also makes them more readable for other people. When used appropriately, it can really improve the understanding of your benchmarking procedures in Rally when shared with others.

1.4 User stories

Many users of Rally were able to make interesting discoveries concerning their OpenStack clouds using our benchmarking tool. Numerous user stories presented below show how Rally has made it possible to find performance bugs and validate improvements for different OpenStack installations.

1.4.1 4x performance increase in Keysone inside Apache using the token creation benchmark

(Contributed by Neependra Khare, Red Hat)

Below we describe how we were able to get and verify a 4x better performance of Keysone inside Apache. To do that, we ran a Keystone token creation benchmark with Rally under different load (this benchmark scenario essentially just authenticates users with keystone to get tokens).

Goal

- Get the data about performance of token creation under different load.
- Ensure that keystone with increased public_workers/admin_workers values and under Apache works better than the default setup.

Summary

- As the concurrency increases, time to authenticate the user gets up.
- Keystone is CPU bound process and by default only one thread of keystone-all process get started. We can increase the parallelism by :- 1. increasing public_workers/admin_workers values in keystone.conf file 2. running keystone inside Apache
- We configured Keystone with 4 public_workers and ran Keystone inside Apache. In both cases we got upto 4x better performance as compared to default keystone configuration.

Setup

Server : Dell PowerEdge R610

CPU make and model : Intel(R) Xeon(R) CPU X5650 @ 2.67GHz

CPU count: 24

RAM : 48 GB

Devstack - Commit#d65f7a2858fb047b20470e8fa62ddaede2787a85

Keystone - Commit#455d50e8ae360c2a7598a61d87d9d341e5d9d3ed

Keystone API - 2

To increase public_workers - Uncomment line with public_workers and set public_workers to 4. Then restart keystone service.

To run keystone inside Apache - Added *APACHE_ENABLED_SERVICES=key* in localrc file while setting up OpenStack environment with devstack.

Results

1. Concurrency = 4

```
{'context': {'users': {'concurrent': 30,
                       'tenants': 12,
                       'users_per_tenant': 512}},
                       'runner': {'concurrency': 4, 'times': 10000, 'type': 'constant'}}
```

action	min (sec)	avg (sec)	max (sec)	90 percentile	95 percentile	success	count	apache enabled keystone	public_workers
total	0.537	0.998	4.553	1.233	1.391	100.0%	10000	N	1
total	0.189	0.296	5.099	0.417	0.474	100.0%	10000	N	4
total	0.208	0.299	3.228	0.437	0.485	100.0%	10000	Y	NA

2. Concurrency = 16

```
{'context': {'users': {'concurrent': 30,
                       'tenants': 12,
                       'users_per_tenant': 512}},
                       'runner': {'concurrency': 16, 'times': 10000, 'type': 'constant'}}
```

action	min (sec)	avg (sec)	max (sec)	90 percentile	95 percentile	success	count	apache enabled keystone	public_workers
total	1.036	3.905	11.254	5.258	5.700	100.0%	10000	N	1
total	0.187	1.012	5.894	1.61	1.856	100.0%	10000	N	4
total	0.515	0.970	2.076	1.113	1.192	100.0%	10000	Y	NA

3. Concurrency = 32

```
{'context': {'users': {'concurrent': 30,
                       'tenants': 12,
                       'users_per_tenant': 512}},
                       'runner': {'concurrency': 32, 'times': 10000, 'type': 'constant'}}
```

action	min (sec)	avg (sec)	max (sec)	90 percentile	95 percentile	success	count	apache enabled keystone	public_workers
total	1.493	7.752	16.007	10.428	11.183	100.0%	10000	N	1
total	0.198	1.967	8.54	3.223	3.701	100.0%	10000	N	4
total	1.115	1.986	6.224	2.133	2.244	100.0%	10000	Y	NA

1.4.2 Finding a Keystone bug while benchmarking 20 node HA cloud performance at creating 400 VMs

(Contributed by Alexander Maretskiy, Mirantis)

Below we describe how we found a bug in keystone and achieved 2x average performance increase at booting Nova servers after fixing that bug. Our initial goal was to benchmark the booting of a significant amount of servers on a cluster (running on a custom build of Mirantis OpenStack v5.1) and to ensure that this operation has reasonable performance and completes with no errors.

Goal

- Get data on how a cluster behaves when a huge amount of servers is started
- Get data on how good the neutron component is good in this case

Summary

- Creating 400 servers with configured networking
- Servers are being created simultaneously - 5 servers at the same time

Hardware

Having a real hardware lab with 20 nodes:

Vendor	SUPERMICRO SUPERSERVER
CPU	12 cores, Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
RAM	32GB (4 x Samsung DDRIII 8GB)
HDD	1TB

Cluster

This cluster was created via Fuel Dashboard interface.

Rally

Version

For this benchmark, we use custom rally with the following patch:

https://review.openstack.org/#/c/96300/

Deployment

Rally was deployed for cluster using ExistingCloud type of deployment.

Server flavor

```
$ nova flavor-show ram64
+----------------------------+--------------------------------------+
| Property                   | Value                                |
+----------------------------+--------------------------------------+
| OS-FLV-DISABLED:disabled   | False                                |
| OS-FLV-EXT-DATA:ephemeral  | 0                                    |
| disk                       | 0                                    |
| extra_specs                | {}                                   |
| id                         | 2e46aba0-9e7f-4572-8b0a-b12cfe7e06a1 |
| name                       | ram64                                |
| os-flavor-access:is_public | True                                 |
```

```
| ram                          | 64                                       |
| rxtx_factor                  | 1.0                                      |
| swap                         |                                          |
| vcpus                        | 1                                        |
+------------------------------+------------------------------------------+
```

Server image

```
$ nova image-show TestVM
+----------------------------+-----------------------------------------------------+
| Property                   | Value                                               |
+----------------------------+-----------------------------------------------------+
| OS-EXT-IMG-SIZE:size       | 13167616                                            |
| created                    | 2014-08-21T11:18:49Z                                |
| id                         | 7a0d90cb-4372-40ef-b711-8f63b0ea9678                |
| metadata murano_image_info | {"title": "Murano Demo", "type": "cirros.demo"}     |
| minDisk                    | 0                                                   |
| minRam                     | 64                                                  |
| name                       | TestVM                                              |
| progress                   | 100                                                 |
| status                     | ACTIVE                                              |
| updated                    | 2014-08-21T11:18:50Z                                |
+----------------------------+-----------------------------------------------------+
```

Task configuration file (in JSON format):

```
{
   "NovaServers.boot_server": [
       {
           "args": {
               "flavor": {
                   "name": "ram64"
               },
               "image": {
                   "name": "TestVM"
               }
           },
           "runner": {
               "type": "constant",
               "concurrency": 5,
               "times": 400
           },
           "context": {
               "neutron_network": {
                   "network_ip_version": 4
               },
               "users": {
                   "concurrent": 30,
                   "users_per_tenant": 5,
                   "tenants": 5
               },
               "quotas": {
                   "neutron": {
                       "subnet": -1,
                       "port": -1,
                       "network": -1,
                       "router": -1
                   }
               }
```

```
                }
            }
        ]
}
```

The only difference between first and second run is that runner.times for first time was set to 500

Results

First time - a bug was found:

Starting from 142 server, we have error from novaclient: Error <class 'novaclient.exceptions.Unauthorized'>: Unauthorized (HTTP 401).

That is how a bug in keystone was found.

action	min (sec)	avg (sec)	max (sec)	90 percentile	95 percentile	success	count
nova.boot_server	6.507	17.402	100.303	39.222	50.134	26.8%	500
total	6.507	17.402	100.303	39.222	50.134	26.8%	500

Second run, with bugfix:

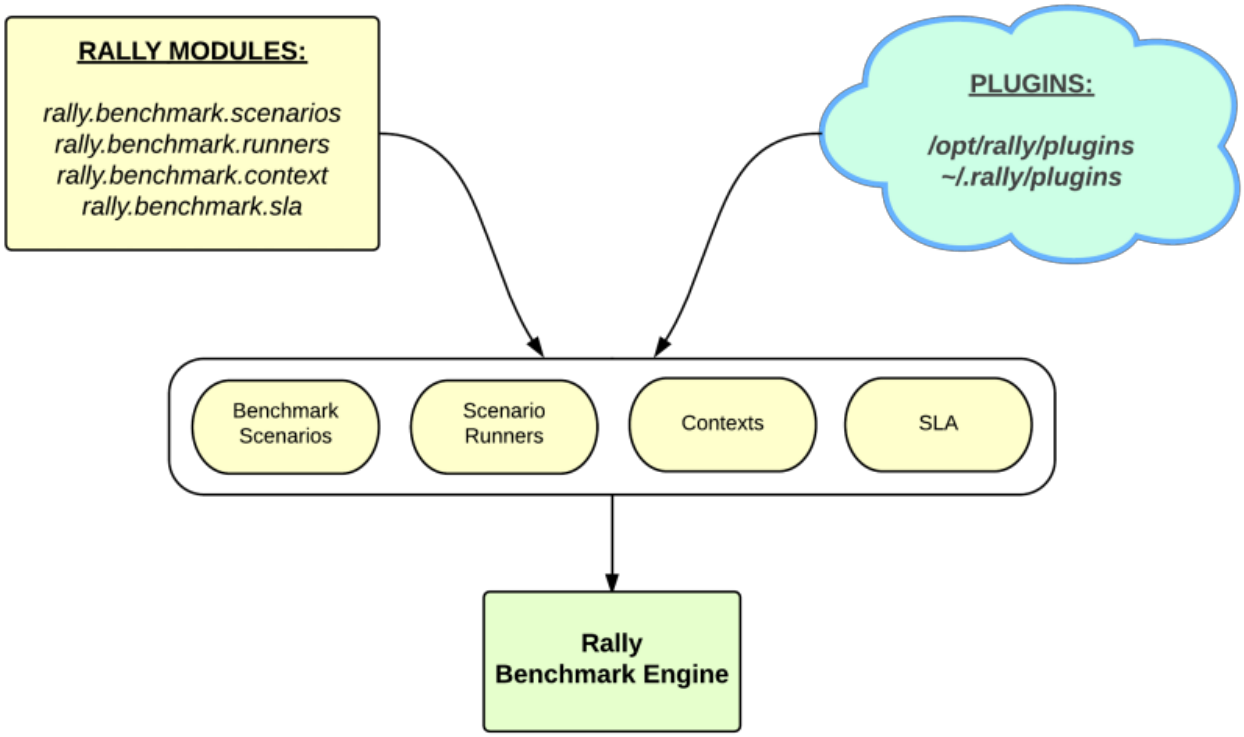
After a patch was applied (using RPC instead of neutron client in metadata agent), we got **100% success and 2x improved average perfomance**:

action	min (sec)	avg (sec)	max (sec)	90 percentile	95 percentile	success	count
nova.boot_server	5.031	8.008	14.093	9.616	9.716	100.0%	400
total	5.031	8.008	14.093	9.616	9.716	100.0%	400

1.5 Rally Plugins

1.5.1 How plugins work

Rally provides an opportunity to create and use a **custom benchmark scenario, runner or context** as a **plugin**:



Plugins can be quickly written and used, with no need to contribute them to the actual Rally code. Just place a python module with your plugin class into the **/opt/rally/plugins** or **~/.rally/plugins** directory (or it's subdirectories), and it will be autoloaded.

1.5.2 Example: Benchmark scenario as a plugin

Let's create as a plugin a simple scenario which lists flavors.

Creation

Inherit a class for your plugin from the base *Scenario* class and implement a scenario method inside it as usual. In our scenario, let us first list flavors as an ordinary user, and then repeat the same using admin clients:

```
from rally.benchmark.scenarios import base


class ScenarioPlugin(base.Scenario):
    """Sample plugin which lists flavors."""

    @base.atomic_action_timer("list_flavors")
    def _list_flavors(self):
        """Sample of usage clients - list flavors

        You can use self.context, self.admin_clients and self.clients which are
        initialized on scenario instanse creation"""
        self.clients("nova").flavors.list()

    @base.atomic_action_timer("list_flavors_as_admin")
    def _list_flavors_as_admin(self):
        """The same with admin clients"""
```

```
        self.admin_clients("nova").flavors.list()

    @base.scenario()
    def list_flavors(self):
        """List flavors."""
        self._list_flavors()
        self._list_flavors_as_admin()
```

Placement

Put the python module with your plugin class into the **/opt/rally/plugins** or **~/.rally/plugins** directory or it's subdirectories and it will be autoloaded. You can also use a script **unpack_plugins_samples.sh** from **samples/plugins** which will automatically create the **~/.rally/plugins** directory.

Usage

You can refer to your plugin scenario in the benchmark task configuration files just in the same way as to any other scenarios:

```
{
    "ScenarioPlugin.list_flavors": [
        {
            "runner": {
                "type": "serial",
                "times": 5,
            },
            "context": {
                "create_flavor": {
                    "ram": 512,
                }
            }
        }
    ]
}
```

This configuration file uses the *"create_flavor"* context which we'll create as a plugin below.

1.5.3 Example: Context as a plugin

Let's create as a plugin a simple context which adds a flavor to the environment before the benchmark task starts and deletes it after it finishes.

Creation

Inherit a class for your plugin from the base *Context* class. Then, implement the Context API: the *setup()* method that creates a flavor and the *cleanup()* method that deletes it.

```
from rally.benchmark.context import base
from rally.common import log as logging
from rally import consts
from rally import osclients

LOG = logging.getLogger(__name__)
```

```
@base.context(name="create_flavor", order=1000)
class CreateFlavorContext(base.Context):
    """This sample create flavor with specified options before task starts and
    delete it after task completion.

    To create your own context plugin, inherit it from
    rally.benchmark.context.base.Context
    """

    CONFIG_SCHEMA = {
        "type": "object",
        "$schema": consts.JSON_SCHEMA,
        "additionalProperties": False,
        "properties": {
            "flavor_name": {
                "type": "string",
            },
            "ram": {
                "type": "integer",
                "minimum": 1
            },
            "vcpus": {
                "type": "integer",
                "minimum": 1
            },
            "disk": {
                "type": "integer",
                "minimum": 1
            }
        }
    }

    def setup(self):
        """This method is called before the task start"""
        try:
            # use rally.osclients to get nessesary client instance
            nova = osclients.Clients(self.context["admin"]["endpoint"]).nova()
            # and than do what you need with this client
            self.context["flavor"] = nova.flavors.create(
                # context settings are stored in self.config
                name=self.config.get("flavor_name", "rally_test_flavor"),
                ram=self.config.get("ram", 1),
                vcpus=self.config.get("vcpus", 1),
                disk=self.config.get("disk", 1)).to_dict()
            LOG.debug("Flavor with id '%s'" % self.context["flavor"]["id"])
        except Exception as e:
            msg = "Can't create flavor: %s" % e.message
            if logging.is_debug():
                LOG.exception(msg)
            else:
                LOG.warning(msg)

    def cleanup(self):
        """This method is called after the task finish"""
        try:
            nova = osclients.Clients(self.context["admin"]["endpoint"]).nova()
            nova.flavors.delete(self.context["flavor"]["id"])
```

```
            LOG.debug("Flavor '%s' deleted" % self.context["flavor"]["id"])
        except Exception as e:
            msg = "Can't delete flavor: %s" % e.message
            if logging.is_debug():
                LOG.exception(msg)
            else:
                LOG.warning(msg)
```

Placement

Put the python module with your plugin class into the **/opt/rally/plugins** or **~/.rally/plugins** directory or it's subdirectories and it will be autoloaded. You can also use a script **unpack_plugins_samples.sh** from **samples/plugins** which will automatically create the **~/.rally/plugins** directory.

Usage

You can refer to your plugin context in the benchmark task configuration files just in the same way as to any other contexts:

```
{
    "Dummy.dummy": [
        {
            "args": {
                "sleep": 0.01
            },
            "runner": {
                "type": "constant",
                "times": 5,
                "concurrency": 1
            },
            "context": {
                "users": {
                    "tenants": 1,
                    "users_per_tenant": 1
                },
                 "create_flavor": {
                    "ram": 1024
                }
            }
        }
    ]
}
```

1.5.4 Example: SLA as a plugin

Let's create as a plugin an SLA (success criterion) which checks whether the range of the observed performance measurements does not exceed the allowed maximum value.

Creation

Inherit a class for your plugin from the base *SLA* class and implement its API (the *check()* method):

```
from rally.benchmark.sla import base


class MaxDurationRange(base.SLA):
    """Maximum allowed duration range in seconds."""
    OPTION_NAME = "max_duration_range"
    CONFIG_SCHEMA = {"type": "number", "minimum": 0.0,
                     "exclusiveMinimum": True}

    @staticmethod
    def check(criterion_value, result):
        durations = [r["duration"] for r in result if not r.get("error")]
        durations_range = max(durations) - min(durations)
        success = durations_range <= criterion_value
        msg = (_("Maximum duration range per iteration %ss, actual %ss")
               % (criterion_value, durations_range))
        return base.SLAResult(success, msg)
```

Placement

Put the python module with your plugin class into the **/opt/rally/plugins** or **~/.rally/plugins** directory or it's subdirectories and it will be autoloaded. You can also use a script **unpack_plugins_samples.sh** from **samples/plugins** which will automatically create the **~/.rally/plugins** directory.

Usage

You can refer to your SLA in the benchmark task configuration files just in the same way as to any other SLA:

```
{
    "Dummy.dummy": [
        {
            "args": {
                "sleep": 0.01
            },
            "runner": {
                "type": "constant",
                "times": 5,
                "concurrency": 1
            },
            "context": {
                "users": {
                    "tenants": 1,
                    "users_per_tenant": 1
                }
            },
            "sla": {
                "max_duration_range": 2.5
            }
        }
    ]
}
```

1.5.5 Example: Scenario runner as a plugin

Let's create as a plugin a scenario runner which runs a given benchmark scenario for a random number of times (chosen at random from a given range).

Creation

Inherit a class for your plugin from the base *ScenarioRunner* class and implement its API (the *_run_scenario()* method):

```
import random

from rally.benchmark.runners import base
from rally import consts


class RandomTimesScenarioRunner(base.ScenarioRunner):
    """Sample of scenario runner plugin.

    Run scenario random number of times, which is choosen between min_times and
    max_times.
    """

    __execution_type__ = "random_times"

    CONFIG_SCHEMA = {
        "type": "object",
        "$schema": consts.JSON_SCHEMA,
        "properties": {
            "type": {
                "type": "string"
            },
            "min_times": {
                "type": "integer",
                "minimum": 1
            },
            "max_times": {
                "type": "integer",
                "minimum": 1
            }
        },
        "additionalProperties": True
    }

    def _run_scenario(self, cls, method_name, context, args):
        # runners settings are stored in self.config
        min_times = self.config.get('min_times', 1)
        max_times = self.config.get('max_times', 1)

        for i in range(random.randrange(min_times, max_times)):
            run_args = (i, cls, method_name,
                        base._get_scenario_context(context), args)
            result = base._run_scenario_once(run_args)
            # use self.send_result for result of each iteration
            self._send_result(result)
```

Placement

Put the python module with your plugin class into the **/opt/rally/plugins** or **~/.rally/plugins** directory or it's subdirectories and it will be autoloaded. You can also use a script **unpack_plugins_samples.sh** from **samples/plugins** which will automatically create the **~/.rally/plugins** directory.

Usage

You can refer to your scenario runner in the benchmark task configuration files just in the same way as to any other runners. Don't forget to put you runner-specific parameters to the configuration as well (*"min_times"* and *"max_times"* in our example):

```
{
    "Dummy.dummy": [
        {
            "runner": {
                "type": "random_times",
                "min_times": 10,
                "max_times": 20,
            },
            "context": {
                "users": {
                    "tenants": 1,
                    "users_per_tenant": 1
                }
            }
        }
    ]
}
```

Different plugin samples are available here.

1.6 Contribute to Rally

1.6.1 Where to begin

Please take a look our Roadmap to get information about our current work directions.

In case you have questions or want to share your ideas, be sure to contact us at the `#openstack-rally` IRC channel on **irc.freenode.net**.

If you are going to contribute to Rally, you will probably need to grasp a better understanding of several main design concepts used throughout our project (such as **benchmark scenarios**, **contexts** etc.). To do so, please read *this article*.

1.6.2 How to contribute

1. You need a Launchpad account and need to be joined to the Openstack team. You can also join the Rally team if you want to. Make sure Launchpad has your SSH key, Gerrit (the code review system) uses this.
2. Sign the CLA as outlined in the account setup section of the developer guide.
3. Tell git your details:

```
git config --global user.name "Firstname Lastname"
git config --global user.email "your_email@youremail.com"
```

4. Install git-review. This tool takes a lot of the pain out of remembering commands to push code up to Gerrit for review and to pull it back down to edit it. It is installed using:

```
pip install git-review
```

Several Linux distributions (notably Fedora 16 and Ubuntu 12.04) are also starting to include git-review in their repositories so it can also be installed using the standard package manager.

5. Grab the Rally repository:

```
git clone git@github.com:stackforge/rally.git
```

6. Checkout a new branch to hack on:

```
git checkout -b TOPIC-BRANCH
```

7. Start coding
8. Run the test suite locally to make sure nothing broke, e.g. (this will run py26/py27/pep8 tests):

```
tox
```

(NOTE: you should have installed tox<=1.6.1)

If you extend Rally with new functionality, make sure you have also provided unit and/or functional tests for it.

9. Commit your work using:

```
git commit -a
```

Make sure you have supplied your commit with a neat commit message, containing a link to the corresponding blueprint / bug, if appropriate.

10. Push the commit up for code review using:

```
git review -R
```

That is the awesome tool we installed earlier that does a lot of hard work for you.

11. Watch your email or review site, it will automatically send your code for a battery of tests on our Jenkins setup and the core team for the project will review your code. If there are any changes that should be made they will let you know.
12. When all is good the review site will automatically merge your code.

(This tutorial is based on: http://www.linuxjedi.co.uk/2012/03/real-way-to-start-hacking-on-openstack.html)

1.6.3 Testing

Please, don't hesitate to write tests ;)

Unit tests

*Files: /tests/unit/**

The goal of unit tests is to ensure that internal parts of the code work properly. All internal methods should be fully covered by unit tests with a reasonable mocks usage.

About Rally unit tests:

- All unit tests are located inside /tests/unit/*
- Tests are written on top of: *testtools*, *fixtures* and *mock* libs
- Tox is used to run unit tests

To run unit tests locally:

```
$ pip install tox
$ tox
```

To run py26, py27 or pep8 only:

```
$ tox -e <name>

#NOTE: <name> is one of py26, py27 or pep8
```

To get test coverage:

```
$ tox -e cover

#NOTE: Results will be in /cover/index.html
```

To generate docs:

```
$ tox -e docs

#NOTE: Documentation will be in doc/source/_build/html/index.html
```

Functional tests

*Files: /tests/functional/**

The goal of functional tests is to check that everything works well together. Fuctional tests use Rally API only and check responses without touching internal parts.

To run functional tests locally:

```
$ source openrc
$ rally deployment create --fromenv --name testing
$ tox -e cli

#NOTE: openrc file with OpenStack admin credentials
```

Output of every Rally execution will be collected under some reports root in directiry structure like: reports_root/ClassName/MethodName_suffix.extension This functionality implemented in tests.functional.utils.Rally.__call__ method. Use 'gen_report_path' method of 'Rally' class to get automaticaly generated file path and name if you need. You can use it to publish html reports, generated during tests. Reports root can be passed throw environment variable 'REPORTS_ROOT'. Default is 'rally-cli-output-files'.

Rally CI scripts

*Files: /tests/ci/**

This directory contains scripts and files related to the Rally CI system.

Rally Style Commandments

Files: /tests/hacking/

This module contains Rally specific hacking rules for checking commandments.

For more information about Style Commandments, read the OpenStack Style Commandments manual.

1.7 Rally OS Gates

1.7.1 Gate jobs

The **Openstack CI system** uses the so-called **"Gate jobs"** to control merges of patched submitted for review on Gerrit. These **Gate jobs** usually just launch a set of tests – unit, functional, integration, style – that check that the proposed patch does not break the software and can be merged into the target branch, thus providing additional guarantees for the stability of the software.

1.7.2 Create a custom Rally Gate job

You can create a **Rally Gate job** for your project to run Rally benchmarks against the patchsets proposed to be merged into your project.

To create a rally-gate job, you should create a **rally-jobs/** directory at the root of your project.

As a rule, this directory contains only **{projectname}.yaml**, but more scenarios and jobs can be added as well. This yaml file is in fact an input Rally task file specifying benchmark scenarios that should be run in your gate job.

To make *{projectname}.yaml* run in gates, you need to add *"rally-jobs"* to the "jobs" section of *projects.yaml* in *openstack-infra/project-config*.

1.7.3 Example: Rally Gate job for Glance

Let's take a look at an example for the Glance project:

Edit *jenkins/jobs/projects.yaml:*

```
- project:
    name: glance
    node: 'bare-precise || bare-trusty'
    tarball-site: tarballs.openstack.org
    doc-publisher-site: docs.openstack.org

    jobs:
      - python-jobs
      - python-icehouse-bitrot-jobs
      - python-juno-bitrot-jobs
      - openstack-publish-jobs
      - translation-jobs
      - rally-jobs
```

Also add *gate-rally-dsvm-{projectname}* to *zuul/layout.yaml*:

```
- name: openstack/glance
  template:
    - name: merge-check
```

```
    - name: python26-jobs
    - name: python-jobs
    - name: openstack-server-publish-jobs
    - name: openstack-server-release-jobs
    - name: periodic-icehouse
    - name: periodic-juno
    - name: check-requirements
    - name: integrated-gate
    - name: translation-jobs
    - name: large-ops
    - name: experimental-tripleo-jobs
  check:
    - check-devstack-dsvm-cells
    - gate-rally-dsvm-glance
  gate:
    - gate-devstack-dsvm-cells
  experimental:
    - gate-grenade-dsvm-forward
```

To add one more scenario and job, you need to add *{scenarioname}.yaml* file here, and *gate-rally-dsvm-{scenarioname}* to *projects.yaml*.

For example, you can add *myscenario.yaml* to *rally-jobs* directory in your project and then edit *jenkins/jobs/projects.yaml* in this way:

```
- project:
    name: glance
    github-org: openstack
    node: bare-precise
    tarball-site: tarballs.openstack.org
    doc-publisher-site: docs.openstack.org

    jobs:
      - python-jobs
      - python-havana-bitrot-jobs
      - openstack-publish-jobs
      - translation-jobs
      - rally-jobs
      - 'gate-rally-dsvm-{name}':
        name: myscenario
```

Finally, add *gate-rally-dsvm-myscenario* to *zuul/layout.yaml*:

```
- name: openstack/glance
  template:
    - name: python-jobs
    - name: openstack-server-publish-jobs
    - name: periodic-havana
    - name: check-requirements
    - name: integrated-gate
  check:
    - check-devstack-dsvm-cells
    - check-tempest-dsvm-postgres-full
    - gate-tempest-dsvm-large-ops
    - gate-tempest-dsvm-neutron-large-ops
    - gate-rally-dsvm-myscenario
```

It is also possible to arrange your input task files as templates based on jinja2. Say, you want to set the image names used throughout the *myscenario.yaml* task file as a variable parameter. Then, replace concrete image names in this file with a variable:

```
...

NovaServers.boot_and_delete_server:
  -
    args:
      image:
          name: {{image_name}}
    ...

NovaServers.boot_and_list_server:
  -
    args:
      image:
          name: {{image_name}}
    ...
```

and create a file named *myscenario_args.yaml* that will define the parameter values:

```
---

  image_name: "^cirros.*uec$"
```

this file will be automatically used by Rally to substitute the variables in *myscenario.yaml*.

1.7.4 Plugins & Extras in Rally Gate jobs

Along with scenario configs in yaml, the **rally-jobs** directory can also contain two subdirectories:

- **plugins**: *Plugins* needed for your gate job;
- **extra**: auxiliary files like bash scripts or images.

Both subdirectories will be copied to *~/.rally/* before the job gets started.

1.8 Request New Features

To request a new feature, you should create a document similar to other feature requests and then contribute it to the **doc/feature_request** directory of the Rally repository (see the *How-to-contribute tutorial*).

If you don't have time to contribute your feature request via gerrit, please contact Boris Pavlovic (boris@pavlovic.me)

Active feature requests:

1.8.1 Support benchmarking clouds that are using LDAP

Use Case

A lot of production clouds are using LDAP with read only access. It means that load can be generated only by existing in system users and there is no admin access.

Problem Description

Rally is using admin access to create temporary users that will be used to produce load.

Possible Solution

- Drop admin requirements
- Add way to pass already existing users

1.8.2 Ability to compare results between task

Use case

During the work on performance it's essential to be able to compare results of similar task before and after change in system.

Problem description

There is no command to compare two or more tasks and get tables and graphs.

Possible solution

- Add command that accepts 2 tasks UUID and prints graphs that compares result

1.8.3 Distributed load generation

Use Case

Some OpenStack projects (Marconi, MagnetoDB) require a real huge load, like 10-100k request per second for benchmarking.

To generate such huge load Rally have to create load from different servers.

Problem Description

- Rally can't generate load from different servers
- Result processing can't handle big amount of data
- There is no support for chunking results

1.8.4 Historical performance data

Use case

OpenStack is really rapidly developed. Hundreds patches are merged daily and it's really hard to track how performance is changed during time. It will be nice to have a way to track performance of major functionality of OpenStack running periodically rally task and building graphs that represent how performance of specific method is changed during the time.

Problem description

There is no way to bind tasks

Possible solution

- Add grouping for tasks
- Add command that creates historical graphs

1.8.5 Using multi scenarios to generate load

Use Case

Rally should be able to generate real life load. Simultaneously create load on different components of OpenStack, e.g. simultaneously booting VM, uploading image and listing users.

Problem Description

At the moment Rally is able to run only 1 scenario per benchmark. Scenario are quite specific (e.g. boot and delete VM for example) and can't actually generate real life load.

Writing a lot of specific benchmark scenarios that will produce more real life load will produce mess and a lot of duplication of code.

Possible solution

- Extend Rally task benchmark configuration in such way to support passing multiple benchmark scenarios in singe benchmark context
- Extend Rally task output format to support results of multiple scenarios in single benchmark separately.
- Extend rally task plot2html and rally task detailed to show results separately for every scenario.

1.8.6 Add support of persistence benchmark environment

Use Case

To benchmark many of operations like show, list, detailed you need to have already these resource in cloud. So it will be nice to be able to create benchmark environment once before benchmarking. The run some amount of benchmarks that are using it and at the end just delete all created resources by benchmark environment.

Problem Description

Fortunately Rally has already a mechanism for creating benchmark environment, that is used to create load. Unfortunately it's atomic operation: (create environment, make load, delete environment). This should be split to 3 separated steps.

Possible solution

- Add new CLI operations to work with benchmark environment: (show, create, delete, list)
- Allow task to start against benchmark environment (instead of deployment)

1.8.7 Production read cleanups

Use Case

Rally should delete in any case all resources that it created during benchmark.

Problem Description

- (implemented) Deletion rate limit

  You can kill cloud by deleting too many objects simultaneously, so deletion rate limit is required

- (implemented) Retry on failures

  There should be few attempts to delete resource in case of failures

- (implemented) Log resources that failed to be deleted

  We should log warnings about all non deleted resources. This information should include UUID of resource, it's type and project.

- (implemented) Pluggable

  It should be simple to add new cleanups adding just plugins somewhere.

- Disaster recovery

  Rally should use special name patterns, to be able to delete resources in such case if something went wrong with server that is running rally. And you have just new instance (without old rally db) of rally on new server.

1.9 Project Info

1.9.1 Useful links

- Source code
- Rally road map
- Project space
- Bugs
- Patches on review
- Meeting logs (server: **irc.freenode.net**, channel: **#openstack-meeting**)
- IRC logs (server: **irc.freenode.net**, channel: **#openstack-rally**)

1.9.2 Where can I discuss and propose changes?

- Our IRC channel: **#openstack-rally** on **irc.freenode.net**;
- Weekly Rally team meeting (in IRC): **#openstack-meeting** on **irc.freenode.net**, held on Tuesdays at 17:00 UTC;
- Openstack mailing list: **openstack-dev@lists.openstack.org** (see subscription and usage instructions);
- Rally team on Launchpad: Answers/Bugs/Blueprints.

1.10 Release Notes

1.10.1 All release notes

Rally v0.0.1

Information

Commits	**1039**
Bug fixes	**0**
Dev cycle	**547 days**
Release date	**26/Jan/2015**

Details

Rally is awesome tool for testing verifying and benchmarking OpenStack clouds.

A lot of people started using Rally in their CI/CD so Rally team should provide more stable product with clear strategy of deprecation and upgrades.

Rally v0.0.2

Information

Commits	**100**
Bug fixes	**18**
Dev cycle	**45 days**
Release date	**12/Mar/2015**

Details

This release contains new features, new benchmark plugins, bug fixes, various code and API improvements.

New Features

- rally task start **–abort-on-sla-failure**

  Stopping load before things go wrong. Load generation will be interrupted if SLA criteria stop passing.

- Rally verify command supports multiple Tempest sources now.
- python34 support
- postgres DB backend support

API changes

- [new] **rally [deployment | verify | task] use** subcommand

  It should be used instead of root command **rally use**

- [new] Rally as a Lib API

  To avoid code duplication between Rally as CLI tool and Rally as a Service we decide to make Rally as a Lib as a common part between these 2 modes.

  Rally as a Service will be a daemon that just maps HTTP request to Rally as a Lib API.

- [deprecated] **rally use** CLI command
- [deprecated] Old Rally as a Lib API

  Old Rally API was quite mixed up so we decide to deprecate it

Plugins

- **Benchmark Scenario Runners**:

  [improved] Improved algorithm of generation load in **constant runner**

  Before we used processes to generate load, now it creates pool of processes (amount of processes is equal to CPU count) after that in each process use threads to generate load. So now you can easily generate load of 1k concurrent scenarios.

  [improved] Unify code of **constant** and **rps** runners

  [interface] Added **abort()** to runner's plugin interface

  New method **abort()** is used to immediately interrupt execution.

- **Benchmark Scenarios**:

  [new] DesignateBasic.create_and_delete_server

  [new] DesignateBasic.create_and_list_servers

  [new] DesignateBasic.list_servers

  [new] MistralWorkbooks.list_workbooks

  [new] MistralWorkbooks.create_workbook

  [new] Quotas.neutron_update

  [new] HeatStacks.create_update_delete_stack

  [new] HeatStacks.list_stacks_and_resources

  [new] HeatStacks.create_suspend_resume_delete_stac

  [new] HeatStacks.create_check_delete_stack

  [new] NeutronNetworks.create_and_delete_routers

  [new] NovaKeypair.create_and_delete_keypair

  [new] NovaKeypair.create_and_list_keypairs

[new] NovaKeypair.boot_and_delete_server_with_keypair

[new] NovaServers.boot_server_from_volume_and_live_migrate

[new] NovaServers.boot_server_attach_created_volume_and_live_migrate

[new] CinderVolumes.create_and_upload_volume_to_image

[fix] CinderVolumes.create_and_attach_volume

Pass optional **kwargs only to create server command

[fix] GlanceImages.create_image_and_boot_instances

Pass optional **kwargs only to create server command

[fix] TempestScenario.* removed stress cleanup.

Major issue is that tempest stress cleanup cleans whole OpenStack. This is very dangerous, so it's better to remove it and leave some extra resources.

[improved] NovaSecGroup.boot_and_delete_server_with_secgroups

Add optional **kwargs that are passed to boot server comment

- **Benchmark Context**:

[new] **stacks**

Generates passed amount of heat stacks for all tenants.

[new] **custom_image**

Prepares images for benchmarks in VMs.

To Support generating workloads in VMs by existing tools like: IPerf, Blogbench, HPCC and others we have to have prepared images, with already installed and configured tools.

Rally team decide to generate such images on fly from passed to avoid requirements of having big repository with a lot of images.

This context is abstract context that allows to automate next steps:

1. runs VM with passed image (with floating ip and other stuff)
2. execute abstract method that has access to VM
3. snapshot this image

In future we are going to use this as a base for making context that prepares images.

[improved] **allow_ssh**

Automatically disable it if security group are disabled in neutron.

[improved] **keypair**

Key pairs are stored in "users" space it means that accessing keypair from scenario is simpler now:

self.context["user"]["keypair"]["private"]

[fix] **users**

Pass proper EndpointType for newly created users

[fix] **sahara_edp**

The Job Binaries data should be treated as a binary content

- **Benchmark SLA**:

  [interface] SLA calculations is done in additive way now

  Resolves scale issues, because now we don't need to have whole array of iterations in memory to process SLA.

  This is required to implement **–abort-on-sla-failure** feature

  [all] SLA plugins were rewritten to implement new interface

Bug fixes 18 bugs were fixed, the most critical are:

- Fix **rally task detailed –iterations-data**

  It didn't work in case of missing atomic actions. Such situation can occur if scenario method raises exceptions

- Add user-friendly message if the task cannot be deleted

  In case of trying to delete task that is not in "finished" status users get traces instead of user-friendly message try to run it with –force key.

- Network context cleanups networks properly now

Documentation

- Image sizes are fixed
- New tutorial in "Step by Step" relate to **–abort-on-sla-failure**
- Various fixes

Rally v0.0.3

Information

Commits	**53**
Bug fixes	**14**
Dev cycle	**33 days**
Release date	**14/Apr/2015**

Details

This release contains new features, new benchmark plugins, bug fixes, various code and API improvements.

New Features & API changes

- Add the ability to specify versions for clients in benchmark scenarios

  You can call self.clients("glance", "2") and get any client for specific version.

- Add API for tempest uninstall

  $ rally-manage tempest uninstall # removes fully tempest for active deployment

- Add a –uuids-only option to rally task list

  $ rally task list –uuids-only # returns list with only task uuids

- Adds endpoint to –fromenv deployment creation

  $ rally deployment create –fromenv # recognizes standard OS_ENDPOINT environment variable

- Configure SSL per deployment

  Now SSL information is deployment specific not Rally specific and rally.conf option is deprecated

  Like in this sample https://github.com/stackforge/rally/blob/14d0b5ba0c75ecefdb6a6c121d9cf2810571f77/samples/deployr
  L12

Specs

- [spec] Proposal for new task input file format

  This spec describes new task input format that will allow us to generate multi scenario load which is crucial for HA and more real life testing:

  https://github.com/stackforge/rally/blob/master/doc/specs/in-progress/new_rally_input_task_format.rst

Plugins

- **Benchmark Scenario Runners**:
  - Add a maximum concurrency option to rps runner

    To avoid running to heavy load you can set 'concurrency' to configuration and in case if cloud is not able to process all requests it won't start more parallel requests then 'concurrency' value.

- **Benchmark Scenarios**:

  [new] CeilometerAlarms.create_alarm_and_get_history

  [new] KeystoneBasic.get_entities

  [new] EC2Servers.boot_server

  [new] KeystoneBasic.create_and_delete_service

  [new] MuranoEnvironments.list_environments

  [new] MuranoEnvironments.create_and_delete_environment

  [new] NovaServers.suspend_and_resume_server

  [new] NovaServers.pause_and_unpause_server

  [new] NovaServers.boot_and_rebuild_server

  [new] KeystoneBasic.create_and_list_services

  [new] HeatStacks.list_stacks_and_events

  [improved] VMTask.boot_runcommand_delete

  restore ability to use fixed IP and floating IP to connect to VM via ssh

  [fix] NovaServers.boot_server_attach_created_volume_and_live_migrate

  Kwargs in nova scenario were wrongly passed

- **Benchmark SLA**:
  - [new] aborted_on_sla

  This is internal SLA criteria, that is added if task was aborted

  - [new] something_went_wrong

This is internal SLA criteria, that is added if something went wrong, context failed to create or runner raised some exceptions

Bug fixes **14 bugs were fixed, the most critical are**:

- Set default task uuid to running task. Before it was set only after task was fully finished.
- The "rally task results" command showed a disorienting "task not found" message for a task that is currently running.
- Rally didn't know how to reconnect to OpenStack in case if token expired.

Documentation

- New tutorial **task templates**

https://rally.readthedocs.org/en/latest/tutorial/step_8_task_templates.html

- Various fixes

1.10.2 Rally v0.0.3

Information

Commits	**53**
Bug fixes	**14**
Dev cycle	**33 days**
Release date	**14/Apr/2015**

Details

This release contains new features, new benchmark plugins, bug fixes, various code and API improvements.

New Features & API changes

- Add the ability to specify versions for clients in benchmark scenarios

  You can call self.clients("glance", "2") and get any client for specific version.

- Add API for tempest uninstall

  $ rally-manage tempest uninstall # removes fully tempest for active deployment

- Add a –uuids-only option to rally task list

  $ rally task list –uuids-only # returns list with only task uuids

- Adds endpoint to –fromenv deployment creation

  $ rally deployment create –fromenv # recognizes standard OS_ENDPOINT environment variable

- Configure SSL per deployment

  Now SSL information is deployment specific not Rally specific and rally.conf option is deprecated

  Like in this sample https://github.com/stackforge/rally/blob/14d0b5ba0c75ecefdb6a6c121d9cf2810571f77/samples/deployr
  L12

Specs

- [spec] Proposal for new task input file format

  This spec describes new task input format that will allow us to generate multi scenario load which is crucial for HA and more real life testing:

  https://github.com/stackforge/rally/blob/master/doc/specs/in-progress/new_rally_input_task_format.rst

Plugins

- **Benchmark Scenario Runners**:
  - Add a maximum concurrency option to rps runner

    To avoid running to heavy load you can set 'concurrency' to configuration and in case if cloud is not able to process all requests it won't start more parallel requests then 'concurrency' value.
- **Benchmark Scenarios**:

  [new] CeilometerAlarms.create_alarm_and_get_history

  [new] KeystoneBasic.get_entities

  [new] EC2Servers.boot_server

  [new] KeystoneBasic.create_and_delete_service

  [new] MuranoEnvironments.list_environments

  [new] MuranoEnvironments.create_and_delete_environment

  [new] NovaServers.suspend_and_resume_server

  [new] NovaServers.pause_and_unpause_server

  [new] NovaServers.boot_and_rebuild_server

  [new] KeystoneBasic.create_and_list_services

  [new] HeatStacks.list_stacks_and_events

  [improved] VMTask.boot_runcommand_delete

  restore ability to use fixed IP and floating IP to connect to VM via ssh

  [fix] NovaServers.boot_server_attach_created_volume_and_live_migrate

  Kwargs in nova scenario were wrongly passed
- **Benchmark SLA**:
  - [new] aborted_on_sla

  This is internal SLA criteria, that is added if task was aborted
  - [new] something_went_wrong

  This is internal SLA criteria, that is added if something went wrong, context failed to create or runner raised some exceptions

Bug fixes

14 bugs were fixed, the most critical are:

- Set default task uuid to running task. Before it was set only after task was fully finished.
- The "rally task results" command showed a disorienting "task not found" message for a task that is currently running.
- Rally didn't know how to reconnect to OpenStack in case if token expired.

Documentation

- New tutorial **task templates**

https://rally.readthedocs.org/en/latest/tutorial/step_8_task_templates.html

- Various fixes