Testing approach for multi-cloud management platforms

Serhiienko Oleksii
Service Prototyping Lab
Zurich University of Applied Sciences
Winterthur, Switzerland
oleksii.serhiienko@gmail.com

***Abstract*—With the growth and evolution of cloud applications, more and more architectures use hybrid clouds to use optimally virtual resources regarding pricing policies and performance. This process led to the creation of management platforms as well as abstract libraries. At the moment, many cloud management platforms(in a further CMP or middlewares) try to cover this requirement. With the growth of such solutions, there is a need for a test environment that could automatically compare existing platforms and help in choosing the most optimal one. This paper focused on the creation of the concept and an actual piece of extendable software which would make experiments repeatable and reusable by other researchers and standardise output and evaluation criteria.**

***Index Terms*—cloud management platform, abstract library, evaluation, test-bed, multi-cloud API**

I. Introduction

Multi-cloud term came from the academic world to essentially business environment recently and the number of cloud management platforms, as well as their demand and functionality, is continuously growing. The goal of this platforms is to assemble various cloud providers for centralised management of virtual resources and a centralised billing system. With the increasing complexity of the applications which are running on cloud deployments, the need for hybrid cloud management system is growing. As for today, different platforms offer different functionality with varying policies of pricing, as well as with varying levels of performance. In this regard, the number of platforms for management is continuously growing. There is no standardisation of the environment for evaluation of centralised cloud management platforms, even though productivity is a critical factor for today's applications.

Cloud management platforms, as well as API abstract libraries currently used by many large groups, such as Redhat [1], Apache [2], Cloud Foundry [3] etc. Depending on the complexity, functionality and architecture of particular middleware, it loads the system in different ways. This work defines three types of middlewares which are aimed to give a certain level of abstraction for cloud and virtual resource management and to centralise providers.

- SaaS - Multi-cloud platforms which are running remotely on the service provider side providing management platform as a service and only one way to access it is via IP address.
- Open-source - Multi-cloud platforms which can be executed by researcher locally or remotely. In most cases, developers provide docker images.
- Library - Multi-cloud API libraries developed for particular language.

It is not correct to compare these middlewares since they fulfil different goals, but evaluation of separately taken functionalities of the systems which are having the same type of management provider brings a complete picture of current state of the art. For example, ManageIQ and MinstIO compare the time of creation and synchronisation of the AWS provider.

This paper will describe the challenges and created an approach for CMPs evaluation. The primary objective of this work is to create a centralised and standardised software solution for testing numerous platforms, to compare results and compress output as table or graphs. Furthermore, this results can be easily extended by other researchers and keep experiments reusable and repeatable by other groups or industry. The testbed focused mostly on:

- Execution time for a specific request
- System Consumption: CPU and Memory

The aim of this work is to systematise the approach of testing and comparing different CMP functionalities. The work is useful for companies that need to choose between existing solutions, and to developers, to see the advantages and disadvantages of the particular function of the platform.

The paper structured as follow:

Related work makes an analyse of existing academic work which focussed on the evaluation of particular platforms to define needs and requirements used in solutions of relevant problems.

Architecture design and implementation describe the classification of management platforms and abstract APIs concerning testbed, an approach of creating extendable software, testbed implementation including high-level architecture and workflow.

Experiments and exemplary results focus on a description of experimental setup and result from the analysis in the form of the table as well as visualised as graphs.

II. Related Work

Most of the relevant works selected from dblp [4], main criteria to choose were relevance, abstract API libraries and cloud management platforms evaluation and testbed design.

This topic has already raised repeatedly in the scientific community, for example, one of the complete works that are engaged in comparing existing abstract libraries is 'An Empirical Study for Evaluating the Performance of jclouds [5]' [6] In this work, the primary focus is on the jclouds and performance results are compared with results from platform-specific libraries. The goal of this study is analysing jclouds API for evaluation concerning its performance against platform-specific APIs from a platform-specific API user interested in enabling cloud portability alternatively, multi-cloud management in the context of a Controlled Experiment. To achieve such a goal a 115 KB file is uploaded to the remote Amazon and Azure endpoints using the above-selected library and also the platform specific ones. For the null hypothesis, it assumed that the download time for a file through the jclouds is the same as the download time through the platform-specific library. As a result, presented dependency graphs of the number of requests and uploading time. In conclusion, pointed that the library's performance depends on the platform, in the experiments jclouds showed itself better compared to the AWS specific API but worse compared to the Azure specific API.

The same authors have expanded the work and called it 'An empirical study for evaluating the performance of multi-cloud APIs' [7]. The goal of this study is to analyse two multi-cloud APIs for evaluation concerning their performance from the platform-specific API users in the context of uploading and downloading files to/from cloud blob storage services. In contrast with the previous work, to the comparison added the libcloud [8] library, file sizes took the more significant range of sizes: 155KB, 310KB, 620KB, 1240KB, 2480KB. To the time evaluation criteria added the CPU time and KB memory criteria, in addition to download time also used uploading file time. This work was much more complete and covered much more problems. In the results, it concluded that the performance of multi-instrument clusters is strongly off-limits from the platform-specific libraries, jclouds is slightly worse in performance compared to platform-specific, while libcloud is better in most experiments. In multi-cloud library selection, the main effort should be on comparing particular attributes depending on the use case.

At the same conference was presented a work 'Critical evaluation on jClouds and Cloudify [9] abstract APIs against EC2, Azure and HP-Cloud' [10], in this document the primary objectives were as follows:

- Analyze the problem and the current literature as well as ask questions that will form the basis for evaluating the abstract APIs.
- Create a tool for analysis of abstract APIs based on questions and criteria that highlighted in the current literature analysis.
- Create prototype tool that will evaluate jClouds and Cloudify.

As a result, presented cloud evaluation tables comparing multi-cloud abstract APIs. Concluded that using abstract interfaces, most of the measured cloud criteria improved.

According to this papers, it is evident that this topic is very relevant and many try to solve it without a standardised test environment. In this work, we will offer our solution and architecture with the help of which it is possible to optimise and automate the evaluation performance tests.

Complete work for designing testbed for Grid [11] describes highly configurable real-life experimental architecture that can be controlled and monitored directly. In this work considered large distributed systems, with numerous parameters and complex interactions between resources, make analytical modelling impractical.

III. Architecture design and implementation

A. Platforms classification and choice

For the first version of the test environment and the demonstration of the workflow process and architecture, three types of platforms were distinguished: SaaS, libraries, and containers

1) Web Platform: For web platforms are understood as a remote access point to which public IP address can only assign, this platform also does not have the opportunity to measure the CPU time and Memory KB, it uses only the time of execution of the query or process. In this paper, as an example, the CloudcheckR [12] is used. CloudCheckr provides an associated cost and security management platform for cost management, AWS inventory, continuous security and compliance auditing across your AWS investment. It also provides comprehensive visibility into a user's cloud environment including billing details, resources, multi-accounts, services, configurations, logs, permissions, changes and more. CloudcheckR is a commercial offer, for the tests used a free 14-day trial. For Access, it is necessary to create an admin access key, which specified further in the test environment configuration file.

2) Libraries: Libraries are aggregators of several cloud platforms for standardised and straightforward access and management. In this case using the Python library, since the code of the test environment written in this language. For evaluation example, using the previously mentioned Libcloud. Libcloud is a Python library for interacting with many of the popular cloud service providers using a unified API. It was created to make it easy for developers to build products that work between any of the services that it supports. Compare to other platforms library is the easiest to use and since it runs on the local machine, it is possible to test the performance of the prototyping python process.

3) Containers: **Composed Containers** Is the platform that is running with the help of the '*docker compose up*' command and consists of some other containers. As an example used MistIO [13] which us managing a mix of public and private clouds, hypervisors, containers, and bare metal trying to optimise costs and policies across platforms also providing visibility and control that makes it easier to govern various infrastructure consistently. Main features are:

- Control hybrid environments
- Enable self-service

- Keep track of usage and cost
- Workflows automation

Such characteristics as CPU time and Memory KB extracted efficiently through the docker open API.

Single Container A single container that runs through the *docker run* command, the platform consists of one single image. ManageIQ [14] is a platform that provides one single image. ManageIQ is an open-source Management Platform that delivers the insight, control, and automation that enterprises need to address the challenges of managing hybrid IT environments. It has the following feature sets:

- **Insight**: Discovery, Monitoring, Utilization, Performance, Reporting, Analytics, Chargeback, and Trending.
- **Control**: Security, Compliance, Alerting, Policy-Based Resource and Configuration Management.
- **Automate**: IT Process, Task and Event, Provisioning, Workload Management and Orchestration.
- **Integrate**: Systems Management, Tools and Processes, Event Consoles, CMDB, RBA, and Web Services.

As well as with composed containers it is easy through the docker to get access to the characteristics of the CPU time and Memory KB.

B. Architecture approach

Designed testbed aims to improve quality of research overall and particularly to create such an environment which would create utterly repeatable set of experiments for management platforms specifying authentication data(credentials, tokens or access keys) with simple YAML file containing instructions in which order, what platforms what amount of times should repeat particular experiments and generate standardised output as raw data, graph or latex table.

While development process every action which supposed to be evaluated should be described with respect of architecture overall and python decorators which are warping methods. In parallel with the platforms, there is also a set of decorators for defining the method and metrics that extracting from the experiments.

1) decorator timing: created for simple time calculation of particular method execution.

2) decorator docker consumption: has an input argument that takes the value of the list of containers, and it will collect the metrics of CPU time and used Memory KB via docker API for each container.

3) decorator docker consumption: also collects the use of the CPU time and Memory KB, but unlike the docker, these values are not the container, but the python's process, also this process is determined automatically.

4) decorator tagging: this decorator has a double benefit, the first is for the more convenient output of information in the form of a dictionary(json) and further easy parsing. And also for registration and mapping all the methods that use it. In the result, a map of all methods and tags recorded in the global variable, which is necessary for the next steps in the workflow.

C. Client implementation

Inside every evaluation client, the method for evaluation should be wrapped by several decorators that are defined above. Each method, regardless of its functionality and classification, has two decorators, the decorator of time should be at the lowest level, so that time is calculated only for the method, and not for other decorators. And the second decorator of the tag, at the highest level, to systematise the output and results of all decorators. When passing a parameter to this decorator, need to be mentioned the correct format to use: '*NameOfProvider:NameOfResource:Action*', where '*' stands for 'any'. For example '**:system:start*' means that this method responsible for the start of the system, and '*aws:provider:create*' defines a method for creating an AWS provider. Between this wrappers, can be as many additional decorators. For evaluation of the libcloud created a specially written decorator for the use of resources by the python, while for the Docker-based platforms written a decorator for the docker resource consumption.

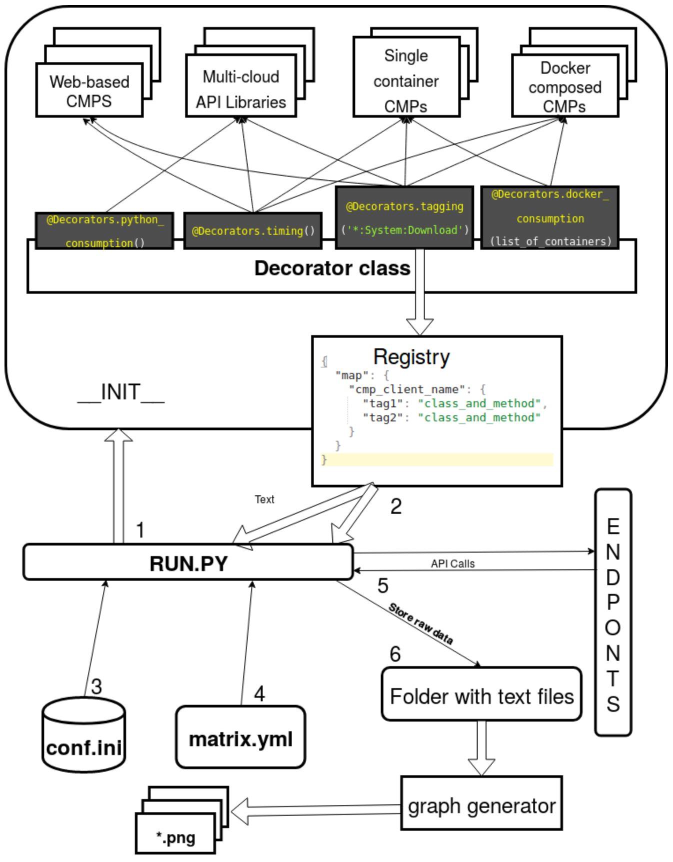
D. Workflow

UML diagrams are presented in the addendum to this work[link to github], here is given the general architecture in Figure 1. When the testbed starts, all the methods that are under this directory classes initialised. All methods with their description saved as a map under the global variable. In the future, the configuration file, as well as the matrix, are loaded. The configuration file contains the information necessary for authorisation (for example, a secret key and an access key for Amazon, an access key for the CloudcheckR, etc.). The matrix is designed to simplify and systematise tests and experiments for multiple platforms. For experiment running, matrix file should have a format in the proposed form:

```
mistio:
  repetitions: 50
  output_dir: /home/ubuntu/experiments
  cmps: [mistio]
  providers: [aws]
  pre_experiment:
    system:
      -start
  post_experiment:
    system:
      -stop
  actions:
    provider:
      - create
      - list
      - delete
```

This YAML interpreted as follow: Create an excerpt under the name of "mistio", which will save all the results in the /home/ubuntu/experiments/mistio directory, perform all the experiments only for the mistio platform and for the AWS provider, start the system before the start of the evaluation, and at the end also stop. Do the experiments over the provider

(AWS) and create it 50 times, show it, delete it. After reading



```
{
  "map": {
    "cmp_client_name": {
      "tag1": "class_and_method",
      "tag2": "class_and_method"
    }
  }
}
```

Fig. 1. Architecture of evaluation application

all the files(configuration and matrix) and creating a common register with the decorator, these data combined and tests produced one by one. Later on, when results are ready, they are dynamically stored in the above folder. In the last step, from the data obtained during the experiment, graphs and matrixes are generated that compare the individual functions.

As it seen the architecture is very flexible and it's easy to add some new platform as well as a characteristic for comparison. With the matrix, it is also possible to create any experiment with the most unusual conditions and the order to evaluate functions and actions

IV. Experiments and Results

A. Experimental setup

All experiments launched on the new virtual machine which is running on OpenStack [15] Platform with following characteristics:

RAM	4GB
VCPUs	2 VCPU with 2500 Mhz
Disk	40GB
OS	Ubuntu 16.04.4 LTS

Tests were conducted with each platform 50 times. As evaluation criterion for demonstrating the test environment, used system tests such as download, start, stop, delete. Since CloudchecR is a web-based platform, it was not tested for these characteristics. Also as a testing function used actions with the provider, in this case, each platform worked with an out-of-order provider on AWS server, with the same region and access keys. All of the platforms tested for creation, listing, deleting provider, also ManageIQ and Cloudcheckr for synchronisation time because of architecture design. The following versions of platforms used in this experiments:

Mist.io	Cloud Management Platform version: 2.0
ManageIQ	gaprindashvili-3
CloudcheckR	last update May 21, 2018
Apache Libcloud	version 2.3.0

For the set of the experiments used one single matrix file, which together with raw and aggregated data, can be found along with the source code for this link(put public link).

B. Results

Architecture of the testbed was designed in that way that overhead is minimal and can be can be neglected. All the information which given below, such as graphics and tables in the latex format, were generated by the test platform itself. In this part, there will be no in-depth analysis of the data, since comparing different types of platforms is not relevant. The purpose of this work is to provide an approach to creating a test platform for multi-cloud management platforms.

Table I shows the results of time evaluation from which concluded that liblcoud performs the fastest system operations since it is an uncomplicated library and it is much simpler than other platforms. Mistio is a multi-image docker platform and because of this loses ManageIQ in the boot time, but shows better results t in time of start and stop the system.

In the results of provider operations, liblcoud was again the fastest, from the graphical interface platforms, good results were also shown by the Mistio, unlike ManageIQ and Cloudcheck R, there is no need for synchronisation time. At the same time, the creation of the provider the fastest results shown by the ManagqIQ platform that can be seen in figure 2.

Table II and III show the characteristics of the CPU time and Memory KB from both results it follows that in docker-base platforms the results are not so stable as seen in figure 3, but it explained by the fact that the systems are complex and they load themselves, while the operations that were carried out were simple and not resource-intensive, which can be seen from the libcloud on figure 4, were the platform itself is not super resource consumption.

V. Conclusion

With the popularity of hybrid cloud systems, the number of platforms trying to centralise their management and the billing system is continuously growing. This Cloud management platform differs depending on the software, and each of them is suitable for specific purposes. A lot of works that compare a separate platform functionality are written and published, which indicates the relevance of this topic. In this paper, we examined the possibility of centralised and standardised testing

TABLE I
TIME CRITERIA EXEMPLARY

src	action	platform	metrics time mu	sigma	median
*-system	download	manageiq	1.06e+05	1.32e+05	8.21e+04
		libcloud	1717.40	120.34	1711.85
		mistio	4.25e+05	7.21e+05	2.58e+05
	start	manageiq	2.00e+05	2455.44	1.99e+05
		libcloud	3430.90	218.90	3445.10
		mistio	7.93e+04	2927.29	7.86e+04
	stop	manageiq	654.38	92.43	645.32
		libcloud	1575.64	125.65	1580.09
		mistio	1.84e+04	483.24	1.83e+04
	remove	manageiq	3670.05	185.47	3669.37
		libcloud	1.99	1.89	1.45
		mistio	6.28e+04	7756.02	6.08e+04
aws-provider	create	cloudcheckr	2411.50	263.09	2339.36
		manageiq	254.98	140.68	225.29
		libcloud	781.10	109.80	732.14
		mistio	1363.78	270.84	1339.77
	list	cloudcheckr	993.05	126.11	953.57
		manageiq	200.26	48.64	187.33
		libcloud	338.55	38.84	328.69
		mistio	20.77	10.16	17.31
	sync	cloudcheckr	4.25e+05	6.15e+04	4.21e+05
		manageiq	6.94e+05	2.06e+06	2.75e+04
	delete	cloudcheckr	1063.39	192.29	1000.20
		manageiq	7482.43	3131.54	7014.02
		libcloud	0.01	0.01	0.01
		mistio	103.77	41.60	88.88

TABLE II
CREATION CPU TIME OF PROVIDER FOR LIBCLOUD

src	action	platform	metrics cpu mu	sigma	median
aws-provider	create	manageiq	4.24e+08	3.67e+08	2.70e+08
		libcloud	0.03	0.01	0.03
		mistio	1.12e+08	1.74e+08	5.00e+07
	list	manageiq	5.88e+08	3.60e+08	4.40e+08
		libcloud	0.01	0.01	0.01
		mistio	9.34e+07	1.79e+08	3.00e+07
	sync	manageiq	6.58e+11	1.93e+12	3.37e+10
	delete	manageiq	7.62e+09	5.42e+09	6.36e+09
		libcloud	0.00	0.00	0.00
		mistio	9.30e+07	1.52e+08	4.00e+07

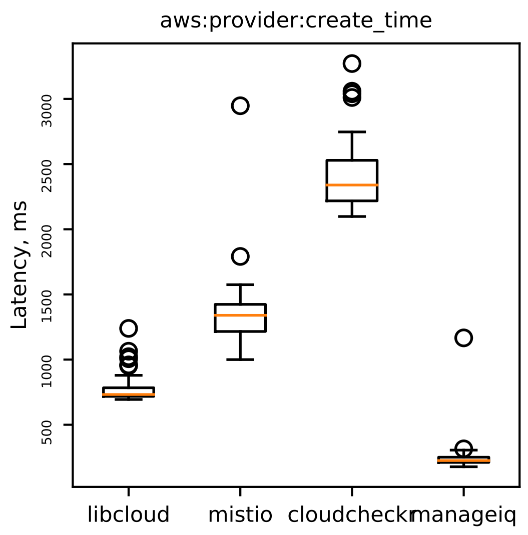


Fig. 2. Creation time of provider

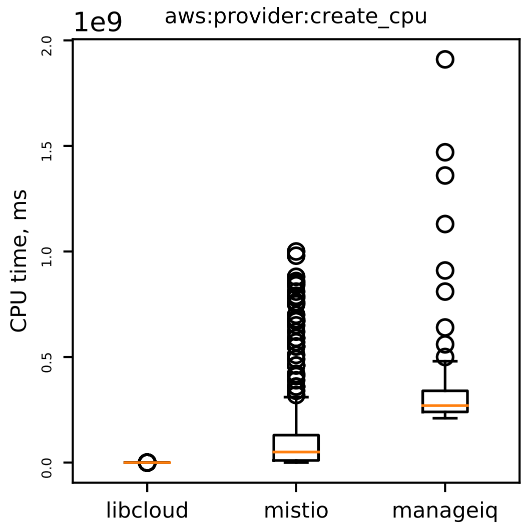


Fig. 3. Creation CPU time of provider

of CMPs based on web platforms, local docker platforms and libraries.

As part of this work written a test environment and created an architecture for multi-platform testing that systematised comparisons and provides the ability to run all tests with just one starting file, which in the future can be used by other researchers to validate the experiments. The architecture is modular and very flexible, which provides the possibility of its constant expansion. The raw result stored in text form, which in the future by the same test-bed generates not only single and combined graphs, but also tables in latex format.

The aim of the work was not to compare the platforms but to create a test environment and, as an example used libcloud (library), mistio(docker composed containers), ManageIQ(single image container), CloudcheckR(website with open API). The results are entirely consistent since libraries are the easiest way to manage platforms and they do not have an overhead, and they showed the best results. In the evaluation of provider management between two local running platforms mistio showed itself better since ManageIQ has the much more full range of functionality and possibilities leading to more significant overhead. All the data and findings published in opensource to keep the studying reusable and repeatable.

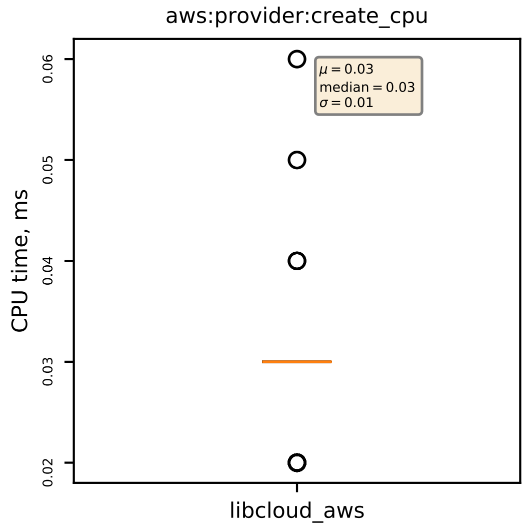


Fig. 4. Creation time of provider

TABLE III
MEMORY KB CRITERIA EVALUATION

src	action	platform		metrics memory	
			mu	sigma	median
aws-provider	create	manageiq	-1.40e+07	7.38e+07	7.37e+04
		libcloud	2.73e+05	4.31e+05	0.00
		mistio	4.04e+05	5.46e+06	0.00
	list	manageiq	2.82e+06	1.56e+07	1.68e+05
		libcloud	1.11e+04	4.64e+04	0.00
		mistio	3.08e+04	6.05e+06	0.00
	sync	manageiq	2.00e+08	9.66e+07	1.75e+08
	delete	manageiq	-1.63e+08	7.67e+07	-1.76e+08
		libcloud	0.00	0.00	0.00
		mistio	-1.47e+05	6.56e+06	0.00

REFERENCES

[1] RedHat LLC. Red Hat Software. https://www.redhat.com/, 1991. Online; accessed 2018-06-06.

[2] Apache Software Foundation. Apache HTTP Server. https://httpd.apache.org/, 1995. Online; accessed 2018-06-06.

[3] Cloud Foundry Foundation. Cloud Foundry. https://www.cloudfoundry.org/, 2011. Online; accessed 2018-06-06.

[4] University of Trier. DBLP. https://dblp.uni-trier.de/, 1993. Online; accessed 2018-06-29.

[5] Apache Software Foundation. Apache jclouds. https://jclouds.apache.org/, 2013. Online; accessed 2018-06-06.

[6] Marcelo Alexandre da Cruz Ismael, César Alberto da Silva, Gabriel Costa Silva, and Reginaldo Ré. An empirical study for evaluating the performance of jclouds. In *7th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2015, Vancouver, BC, Canada, November 30 - December 3, 2015*, pages 115–122. IEEE Computer Society, 2015.

[7] Reginaldo Ré, Rômulo Manciola Meloca, Douglas Nassif Roma Junior, Marcelo Alexandre da Cruz Ismael, and Gabriel Costa Silva. An empirical study for evaluating the performance of multi-cloud apis. *Future Generation Comp. Syst.*, 79:726–738, 2018.

[8] Apache Software Foundation. Apache Libcloud. https://libcloud.apache.org/, 2013. Online; accessed 2018-07-06.

[9] Cloudify. Cloudify. https://cloudify.co/, 2012. Online; accessed 2018-06-06.

[10] Steven Thomas Graham and Xiaodong Liu. Critical evaluation on jclouds and cloudify abstract apis against ec2, azure and hp-cloud. In *IEEE 38th Annual Computer Software and Applications Conference, COMPSAC Workshops 2014, Vasteras, Sweden, July 21-25, 2014*, pages 510–515. IEEE Computer Society, 2014.

[11] Raphael Bolze, Franck Cappello, Eddy Caron, Michel J. Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stéphane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quétier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *IJHPCA*, 20(4):481–494, 2006.

[12] CloudCheckr. CloudcheckR. https://cloudcheckr.com/, 2011. Online; accessed 2018-07-06.

[13] Mistio. mist.io. https://mist.io, 2015. Online; accessed 2018-07-06.

[14] RedHat. ManageIQ. http://manageiq.org/, 2012. Online; accessed 2018-07-06.

[15] OpenStack Foundation. OpenStack. https://www.openstack.org/, 2010. Online; accessed 2018-07-06.