View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ZHAW digitalcollection

Systematic and recomputable comparison of multi-cloud management platforms

Oleksii Serhiienko and Josef Spillner
Service Prototyping Lab
Zurich University of Applied Sciences
Winterthur, Switzerland
oleksii.serhiienko@gmail.com, josef.spillner@zhaw.ch

***Abstract*—With the growth and evolution of cloud applications, more and more architectures use hybrid cloud bindings to optimally use virtual resources regarding pricing policies and performance. This process has led to the creation of multi-cloud management platforms as well as abstraction libraries. At the moment, many (multi-)cloud management platforms (CMPs) are designed to cover the functional requirements. Along with growing adoption and industrial impact of such solutions, there is a need for a comparison and test environment which automatically assesses and compares existing platforms and helps in choosing the optimal one. This paper focuses on the creation of a suitable testbed concept and an actual extensible software prototype which makes multi-cloud experiments repeatable and reusable by other researchers. The work is evaluated by an exemplary comparison of 4 CMPs bound to AWS, showcasing standardised output formats and evaluation criteria.**

***Index Terms*—cloud management platform, multi-cloud, testbed, recomputation**

I. Introduction

Managing bindings to multiple cloud services for one or many software applications is a crucial task for companies in order to forecast subscription spendings, exploit volume discounts, and remain auditable as data processing is outsourced to a multitude of software and platform providers [1]. Many concepts of the resulting *hybrid-cloud management* (HCM) and the generalised form of *multi-cloud management* (MCM) transitioned from academia into the business environment just recently [2]–[4]. Already, the number of widely deployed *cloud management platforms* (CMPs) is continuously growing. Similarly, their functional scope increases with brokering, access unification, metrics aggregation and effective policy enforcement being among the features. The perceived business benefit of these platforms is to abstract one or more cloud provider interfaces for centralised management of virtual resources. Moreover, from a financial management perspective, they offer a centralised billing system capable of multi-tenant charging within companies.

With the increasing complexity of distributed applications which are running on cloud deployments, the practical aspects of operating multi-cloud management topologies involving CMPs still overwhelm many businesses [5]. Different platforms offer diverging functionality with varying policies of pricing, as well as with varying levels of performance. There is no standard benchmark and no standardisation of the environment for the evaluation of centralised CMPs, even though productivity is a critical factor for today's applications.

Research in education linked to funded project Amysta-SaaS/18371.1 by Innosuisse - Swiss Innovation Agency - and an AWS Cloud Credits for Research grant.

CMPs as well as simpler application programming interface (API) abstraction libraries with multi-cloud support are currently produced and maintained by many middleware vendors, such as Red Hat, Apache and Cloud Foundry, as well as by specialised CMP vendors such as CloudcheckR and RightScale. Depending on the complexity, functionality and architecture of a particular middleware, it induces load into the system in different ways. This work defines three types of middleware which are all delivering a certain level of abstraction for cloud and virtual resource management as well as pooling of multiple providers and their offerings.

- SaaS - Multi-cloud platforms which are running remotely on the service provider side, providing management platform as a service. The only way to access them is via their public endpoints.
- Open-source/installable - Multi-cloud platforms which can be executed by operators locally or remotely. In most cases, the vendors provide executable container images.
- Library - Multi-cloud API libraries developed for particular language for direct integration in applications.

CMPs encompass both SaaS and open source platforms, where the platform functionality ranges from simple protocol translation proxies to user-friendly web-based management tools. In most cases, more than one role is defined. For instance, operators run the solution, developers access the translation and pooling functionality, and company executives read generated reports about incurred cost.

It is not trivial to compare CMPs since many of them are still in early development and they fulfil different goals. However, the systematic evaluation of common functionalities of the systems which support equal subsets of providers brings a complete picture of the current state of technology and is valuable for companies who need to take decisions and, once taken, need to avoid regressions. Such comparison would be valid for example to measure the time needed for creation and synchronisation of a specific virtual machine in AWS EC2.

Hence, this paper describes the challenges and proposes an approach and system implementation for comparative CMP evaluation. The primary objective of this work is to deliver

a centralised and standardised software solution for testing numerous platforms, for comparing results and for condensing output as tables or graphs. Furthermore, the results can be easily exploited and extended by other researchers and developers working with continuous integration as all experiments are reusable and recomputable. This design follows recent tooling support for recomputable science [6]. Concretely, our work presents a CMP testbed focused mostly on:

- Execution time for a specific cloud management request
- System consumption: CPU and memory

The research aim of this work is thus to systematise the approach of testing and comparing different CMP functionalities, to increase reproducibility beyond current approaches, and to experimentally evaluate the approach with representative scenarios.

The remainder of the paper is structured as follows: The second section, *Related Work*, analyses published academic works which focus on the evaluation of particular platforms. It defines needs and requirements reflected in solutions to relevant cloud management problems. *Architecture Design and Implementation* describes the classification of CMPs and abstraction APIs concerning the testbed design, an approach of creating extensible software, and the testbed implementation including high-level architecture and workflow. *Experiments and Exemplary Results* focuses on a description of a representative experimental setup of the CMP testbed and presents result from one exemplary analysis in the form of generated tables as well as visualised as graphs to demonstrate the testbed's capabilities. The final section concludes with links to the open source implementation and to the reference datasets.

II. Related Work

Most of the relevant works were systematically selected by keyword search from DBLP [7]. The main criteria for filtering results were relevance to the topic and contribution of practical results involving API abstraction libraries and CMPs.

The topic of advancing multi-cloud management has been raised repeatedly in the scientific community. One of the major early works which analyses existing abstraction libraries in detail is 'An Empirical Study for Evaluating the Performance of Jclouds' [8]. In this study, the primary focus is measuring the Apache Jclouds multi-cloud toolkit for Java [9]. Performance results are compared with results from platform-specific libraries and cloud vendor software development kits (SDKs). The expressed goal of the study is to provide a controlled experiment which assists developers in making decisions when the key concern is performance. To achieve such a goal, a 115 KB file is uploaded to Amazon AWS and Microsoft Azure endpoints using Jclouds and the platform-specific equivalents. For the null hypothesis, it assumed that the download time for a file through Jclouds is equivalent to the download time through the platform-specific library. As a result, the authors present relationship graphs of the number of requests and the resulting uploading time. In conclusion, the authors pointed out that the library's performance depends on the cloud services. In the experiments, Jclouds was faster compared to the AWS-specific SDK but worse compared to the Azure-specific SDK.

The same authors have recently expanded the work and published 'An empirical study for evaluating the performance of multi-cloud APIs' [10]. Again, the performance is measured, but in the expanded study two multi-cloud APIs are covered by adding the Apache Libcloud [11] library. Furthermore, file sizes are checked across a more representative range of 155 KB, 310 KB, 620 KB, 1240 KB and 2480 KB. In addition to the previous time evaluation criteria, both CPU time and memory allocation are considered, and in addition to download times also upload times are measured. This work is thus much more expressive and covers more relevant problems. In the results, it becomes evident that the performance of multi-instrument clusters is strongly off-limits from the platform-specific libraries. Jclouds is slightly worse in performance compared to the SDKs, while Libcloud is better in most experiments. In multi-cloud library selection, the main effort should therefore be on comparing particular attributes depending on the use case. However, the testbed remains focused on few operations and is neither extensible nor suitable for reproducing results over time.

A similar work is 'Critical evaluation on Jclouds and Cloudify abstract APIs against EC2, Azure and HP-Cloud' [12]. In this document the primary objective is stated as follows:

- Analyse 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 are highlighted in the current literature analysis.
- Create a prototype tool that will evaluate Jclouds and Cloudify.

As a result, the authors present cloud evaluation tables comparing multi-cloud API abstraction libraries and conclude that using abstract interfaces, most of the measured cloud criteria improve.

More recently, 'SeaClouds: An Open Reference Architecture for Multi-cloud Governance' [13] has emerged from a European research consortium as overlay architecture on top of Jclouds, Cloud4SOA, TOSCA and other multi-cloud approaches but has yet to be experimentally evaluated for practical usefulness.

According to the screened literature, it is evident that the existing works do not fully match the increasing practical relevance of the topic and that most solutions do not present an extensible test environment ready to be used by developers. In this work, we will raise the matching by contributing our solution and architecture with which it is possible to optimise and automate the performance tests for evaluation.

III. Testbed Design, Architecture and Implementation

The design of our testbed is inpired by an existing testbed for grid environments [14] which describes a highly configurable real-life experimental architecture that can be controlled

and monitored directly. A key assumption is the testing and benchmarking of large distributed systems with numerous parameters and complex interactions between resources which makes analytical modelling impractical. We conjecture that for complex cloud computing environments, a similar design philosophy applies. Hence, we first present design criteria for multi-cloud management evaluation, derive a suitable architecture, systematically select management platforms to evaluate prototypically, and summarise the implementation, its workflows and its extensibility.

A. Design criteria

Considering the emerging need to offer more insight into CMPs to developers and to improve the quality of research on CMPs, we design our testbed, called CoMParable CMPs (CMP^2), in a way which directly addresses two target groups. For software developers, as a ready-to-use open source toolkit which produces easy to understand reports; and for researchers, as a testbed for recomputable and comparative research enablement which produces statistically sound datasets ready for inclusion into articles, as we will expose in the next sections. For both target groups, low-effort extensibility for additional CMPs and the ability to publish the delta compared to the base version is another requirement which our design meets.

These requirements lead to the following distinct design considerations:

- Comfort: The testbed should have a high degree of automation and generate post-processed data on its own.
- Statistical correctness: Through repetitive invocations and outlier detections, users should be able to rely on the numbers produced by the testbed.
- Reproducibility: By presenting an open source prototype with light-weight configuration, experiments can be conducted in collaborative scenarios in which critical discussions about multi-cloud management can be guided by reproduction of previous experimental results. In particular, by offering a software tool, experiments can be repeated and automatically recomputed.
- Extensibility: As we only implement a prototype for some CMPs, it should be easy for software or lab engineers to extend the testbed both for new CMPs and for new methods per CMP.

B. Architecture approach

The designed testbed aims to improve the quality of research overall and particularly aims to create an environment which would create mostly-deterministically repeatable set of experiments for management platforms specifying authentication data (credentials, tokens or access keys) with simple YAML files. These files contain instructions about in which order, on which platforms, how many times should particular experiments be repeated. Based on the experiment, CMP^2 generates standardised output as raw and averaged metrics data, graphs and LaTeX tables.

During the experiment design process, every action which is supposed to be evaluated should be described with respect to the architecture using code-level decorators which are wrapping interpreted methods. These language-specific decorators, also named annotations, offer convenient pre-execution hooks for methods which receive the method instance as parameter (e.g. `@decorator def method ...`). There is a set of decorators for defining the methods and metrics which are extracted from the experiments, as follows:

1) Timing decorator: Created for simple time calculation of a particular method execution.

2) Docker consumption decorator: Has an input parameter expecting the list of containers, based on which it will collect the metrics of CPU time and used memory via the Docker API for each container.

3) Interpreter-level consumption decorator: Also collects the use of the CPU time and main memory, but unlike the Docker decorator, these values refer not to the container but rather to the embedding interpreter process which is determined automatically.

4) Tagging decorator: This decorator has a double benefit. The first one is for the more convenient output of information in the form of a JSON structure and further easier parsing. The second one is for registering and mapping all the methods which use it. As a result, a map of all methods and tags are recorded in a global variable which is necessary for the next steps in the experiment workflow.

C. Platforms classification and choice

For the prototype of the test environment and the determination of the workflow process and architecture, three manifestations of CMPs are distinguished: SaaS in the form of web platforms, containers as technical realisation of installable platforms, and libraries despite not strictly offering platform functionality.

1) Web Platforms: Web platforms are understood to offer a remote access point. By using platforms, there is regularly no opportunity to measure the CPU time and memory consumption; instead, clients can only measure the execution time of the query or process including any connection overhead due to network latency. In this paper, as a representative example, the CloudcheckR service [15] is used although other services could be integrated in a similar way. CloudCheckR provides an associated management platform for cost management, AWS inventory, continuous security and compliance auditing across all subscribed AWS services. It also provides comprehensive visibility into a user's cloud environment including billing details, resources, multi-accounts, services, configurations, logs, permissions and changes. Although CloudcheckR is a commercial offer, it belongs into the category of services offering limited (14-day) trial access which makes it suitable for basic comparison experiments. To gain access, it is necessary to create an admin access key which is then entered into the testbed's configuration file.

2) Libraries: Libraries are language-specific aggregators of several cloud platforms for standardised and straightforward

access and management. As evaluation example, we are using the previously mentioned Libcloud which is a library for interacting with many of the popular cloud service providers using a unified API. Libcloud was created to make it easy for developers to build products which work between any of the services that it supports. Compared to other platforms, libraries are the easiest to use due to the avoidance of any service setup and operation, and since the cloud management functionality runs in-process on the local machine, it is possible to test the performance in-process directly within the testbed.

3) Containers: There are two categories of containerised CMPs which are differentiated in the following.

Composed Containers. Platforms which are running with the help of orchestrators such as the '*docker compose up*' command are composed of multiple containers. An example is MistIO [16] which is managing a mix of public and private clouds, hypervisors, containers and bare metal, trying to optimise cost and policies across platforms but also providing visibility and control to govern more easily various infrastructures consistently. The main features of container compositions are:

- Control hybrid environments
- Enable self-service
- Keep track of usage and cost
- Workflows automation

Single Container. The platform consists of a single container image so that it is typically invoked through the *docker run* command. As an example, ManageIQ [17] is a platform which is encapsulated into one single image. Functionally, it is an open-source management platform which 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 processes, tasks and events, provisioning, workload management and orchestration.
- **Integrate**: Systems management, tools and processes, event consoles, web services.

In both cases, single and composed containers, system characteristics such as consumed CPU time and memory allocation can be extracted efficiently through the container orchestrator interfaces such as the open Docker API.

D. Testbed implementation

We have implemented the testbed in Python due to the ability to natively interact with the Libcloud API. 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 at least two decorators. The timing decorator should be at the lowest level so that time is calculated only for the method itself and not for other decorators. Conversely, the tagging decorator belongs at the highest level to systematise the output and results of all decorators. When passing a parameter to this decorator, the correct format is as follows: '*NameOfProvider:NameOfResource:Action*', where '*' stands for 'any'. For example, '**:system:start*' means that this method is responsible for the start of the system, and '*aws:provider:create*' defines a method for creating an AWS provider. Between these wrappers, there can be an arbitrary number of additional decorators. The evaluation of Libcloud uses a specially written decorator for the use of resources by the Python interpreter, while for the Docker-based platforms a decorator for the Docker resource consumption is used.

E. Workflow

While a number of detailed UML diagrams are available as documentation in the open source implementation, this section presents the general architecture and workflow in Figure 1. When the CMP^2 testbed starts, all the methods that are placed in source files within the directory *classes* are initialised (steps 1 and 2). All methods with their description are saved in a global map variable. Subsequently, the configuration file, as well as an associated configuration matrix, are loaded (steps 3 and 4). The configuration file contains the information necessary for authorisation, for example, a secret key and an access key for Amazon AWS, an access key for CloudcheckR, and similar credentials. The matrix is designed to simplify and systematise tests and experiments for multiple platforms. The matrix file format is given in Listing 1.

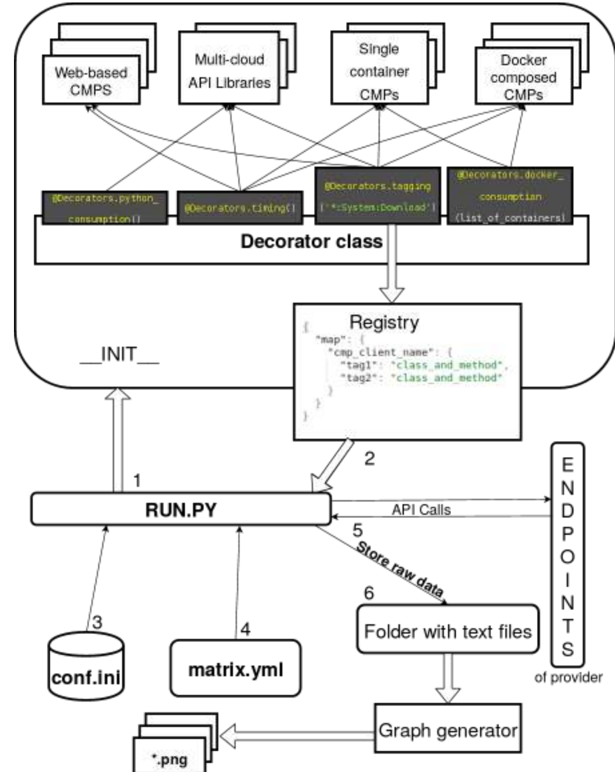

Fig. 1. Architecture and workflow of CMP evaluation testbed CMP^2

Listing 1. Configuration matrix format

```
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 structure is interpreted as follows: Create an excerpt under the name of "mistio", which will save all the results in the compound directory */home/ubuntu/experiments/mistio*, 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. Conduct the experiments on the provider (AWS) by creating the provider object within the CMP 50 times (by default), show the result, and delete the provider object again.

After reading all the files (configuration and matrix) and creating a common register with the decorator, these data are combined and tests are produced one by one. In parallel, whenever results are ready to be consumed, they are dynamically stored in the specified folder (steps 5 and 6). In the last step, from the raw data obtained during the experiment, graphs and tables are generated which compare the individual functions and lead to standardised and comparable result representations.

F. Extensibility

The architecture is very flexible and allows for adding new CMPs as well as characteristics for comparison with low effort. The current implementation demands the placement of additional handler files on the code level, as well as the registration of the handlers in a central file. The code is already prepared for extensibility at other levels as well. Given the ubiquity of portable encapsulated system containers in applied software research and software development [18], a layered container image could be constructed where publications would reference specific layers to maintain the reproducibility. With the matrix file, it is also possible to create any experiment with the most unusual conditions and arbitrary order to evaluate functions and actions without any code changes.

IV. Experiments and Exemplary Results

In this section, we depict in an exemplary setup how researchers can use CMP^2 to compare cloud management platforms and gain more insight about multi-cloud applications. All tables and graphs in the section are directly produced by CMP^2; a complete reference dataset is available as online addendum.

A. Experimental setup

All experiments were launched on a virtual machine running on a private OpenStack cluster with the following characteristics:

RAM	4 GB
VCPUs	2 vCPUs clocked at 2500 Mhz
Disk	40 GB
OS	Ubuntu 16.04.4 LTS

Tests were conducted with each CMP in 50 rounds. As evaluation criterion for demonstrating the test environment, system tests such as download, start, stop, delete were used. Since CloudcheckR is a web-based platform, it was not tested for these characteristics. Each platform worked with an out-of-order *provider* (service abstraction, e.g. EC2 or S3) on AWS servers, with the same region and access keys. All of the platforms were tested for creation, listing and deleting providers. Further, ManageIQ and CloudcheckR were tested for their synchronisation time because of their architecture design making it a significant metric. The following versions of the platforms were 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 this exemplary set of experiments, one single matrix file was used, although more complex experiments with multiple matrix files are possible to evaluate additional dimensions such as memory constraints across all CMPs.

B. Exemplary Results

The architecture of the testbed is designed so that runtime overhead is minimal and can be neglected for almost all use cases. This effect can be inferred from the minimum overhead results caused by abstraction libraries shown in the subsequent graphs. All the information which is given below, such as graphics and tables in the LaTeX format, are generated by the CMP^2 testbed itself. In this part, there will be no in-depth analysis of the data, since comparing different types of platforms along different dimensions is out of scope for this paper. The purpose of this work is rather to provide an approach to creating a reusable testbed for multi-cloud management platforms so that other researchers can achieve and convey their findings.

Table I shows the results of timing evaluation from which the conclusion is that Libcloud performs the fastest system operations since it is a lean library and with a much simpler structure than other middleware, especially long-running platforms. MistIO is a multi-image docker platform and because of this design falls behind ManageIQ in the boot time, but shows better results for start and termination of the platform. Hence, using CMP^2, an exemplary finding produced experimentally by data analytics based on raw numbers output by the testbed

could be that MistIO is suited better for occasional use in dynamic environments. Fostering such results with the general availability of experiment tools helps to better design and implement next-generation CMPs.

TABLE I
TIME CRITERIA EVALUATION

src	action	platform	metrics time (s) 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

In the results of provider management operations, shown in the same table, Libcloud is again the fastest system under test. Among the graphical interface platforms, good results are also shown by the MistIO, as unlike ManageIQ and CloudcheckR there is no need for synchronisation time. At the same time, for the creation of the provider the fastest results are delivered by the ManageIQ platform, the measurement of which can be studied in Figure 2.

Next, Tables II and III show the characteristics of the CPU time and memory allocation for the same experiments. From both results, assuming proper analysis which researchers using CMP^2 would perform, the conclusion would be that in Docker-based platforms the results are not very stable.

The instability of Docker-based CMPs can be seen in the deviations in Figure 3. The observation can be explained by the fact that the systems are complex and have to be loaded (or load parts by themselves) during the invocation, while the operations that are carried out afterwards are simple and not resource-intensive. This latter aspect can be seen in detail from Libcloud on the zoom-in Figure 4, where the platform itself is not resource-intensive. CMP^2 includes such statistical

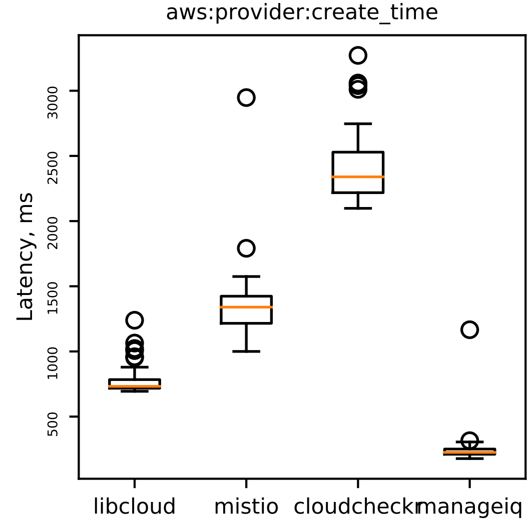

Fig. 2. Time to create provider object in CMP

TABLE II
PROVIDER OBJECT CREATION CPU TIME FOR LIBCLOUD

src	action	platform	metrics cpu time (s) 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

TABLE III
MEMORY CRITERIA EVALUATION

src	action	platform	metrics memory use (KB) 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

features in its results which reduces the risk that researchers publish results based on single or few rounds of experiments.

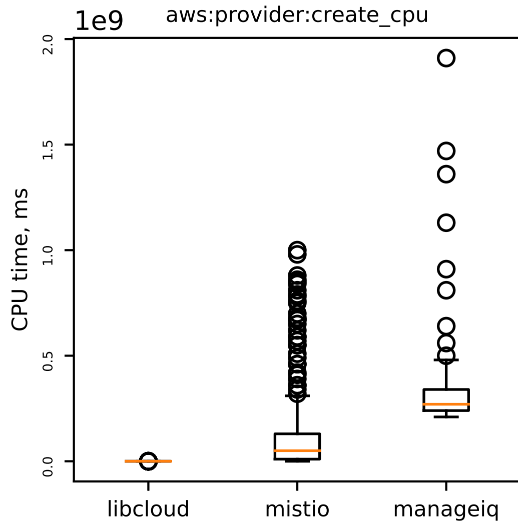

Fig. 3. Provider object creation CPU time

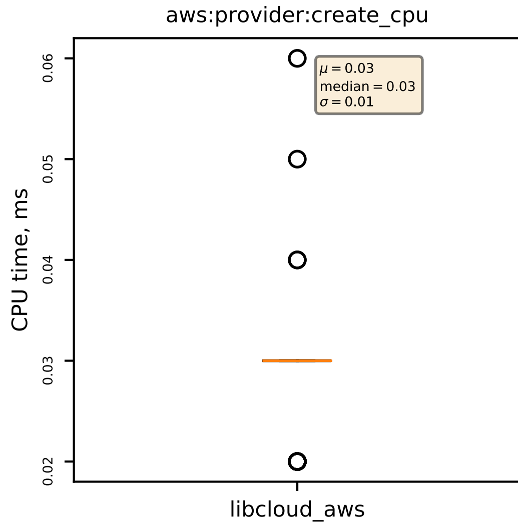

Fig. 4. Provider object creation CPU time in greater detail

C. Comparative Research Enablement

We argue that CMP^2 enables not only recomputation of results from single published works on multi-cloud management, but also eases comparisons between multiple such works assuming the authors publish sufficient details for independent reproducibility. To demonstrate the claim, we refer back to the related work on multi-cloud API performance evaluation [10]. Their work shows the performance of Jclouds, Libcloud and provider-specific libraries on storing files using AWS S3 and Azure, and in particular the Libcloud/Boto(both Python)/S3 combination in the 12th figure. We reproduce (with permission) their figure in our paper as Fig. 5 where 'multi-cloud' refers to Libcloud and 'platform-specific' refers to Boto. Their experiment measures the response time of download and upload invocations of differently-sized files and indicates a clear disadvantage of the multi-cloud CMP layer on the download side.

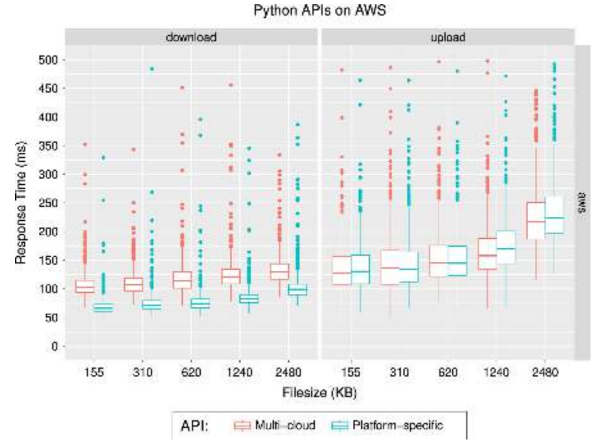

Fig. 5. Original 12th figure from [10] on cloud storage performance with Libcloud

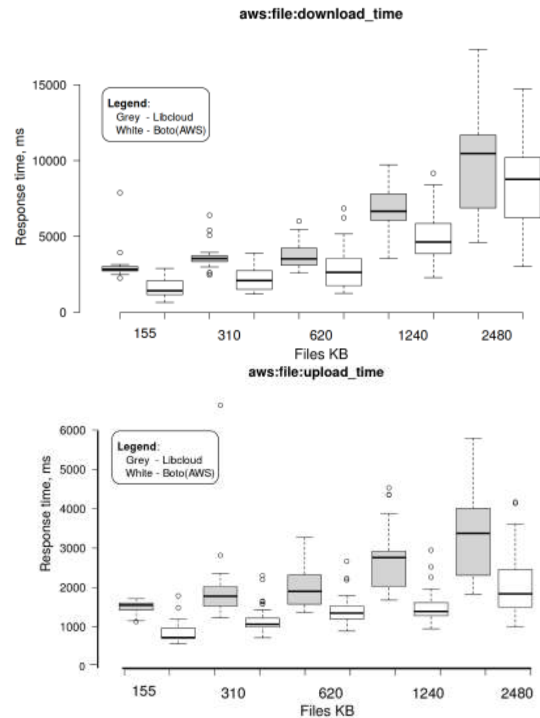

Fig. 6. Comparable reproduction of figures from [10] on cloud storage performance with Libcloud and Boto

Based on the raw data output by CMP^2, a custom plotting procedure to achieve comparable boxplots was implemented with little effort. The customisation of our testbed took 5 hours

of work including a comparable configuration of file sizes and credentials. Fig. 6 shows the graph output by our testbed.

It should be noted that CMP^2 always produces single graphs, hence our figure combines two automatically produced graphs in a matching order. While details differ, all important structural metrics are present in the graph. According to the statistics conveyed by the plot, it is evident that absolute response times cannot be compared, primarily due to different network configurations between research systems and cloud providers. Yet in relative terms, our measurement reveals a much higher spread in proportion to the file size.

V. Conclusion

With the popularity of hybrid cloud systems and multi-cloud applications, the number of platforms aiming to centralise their management and billing is continuously growing. The cloud management platforms (CMPs) differ depending on the software, and each of them is suitable for specific purposes. A lot of academic works that compare a separate platform functionality are written and published, which indicates the relevance of this topic. In this paper, we identified weaknesses in these publications and examined the possibility of centralised, standardised and recomputable testing of CMPs which are built on web platforms, local Docker platforms and libraries.

As part of this work, a test environment and an architecture for multi-platform testing are contributed. The architecture systematises 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 low-effort expansion. The raw results are stored in text form, from which in the future through the same testbed not only single and combined graphs as well as tables in LaTeXformat, but also other representations can be derived.

The contribution of the work is hence not to compare the platforms in specific dimensions but to create a testbed environment to facilitate such comparison and evaluation studies. Merely as a desirable side effect of our work, by choosing Libcloud and Boto (both libraries), MistIO (docker composed containers), ManageIQ (single image container) and CloudcheckR (website with open API), we show exemplary results which are valid for the specified software and service versions. The results are entirely consistent since libraries are the easiest way to manage platforms and they do not have an overhead, and they yield the best performance results. In the evaluation of provider management between two locally running platforms, MistIO shows itself better since ManageIQ has a wider range of functionality leading to more significant overhead.

All the data and findings are published as open source and open data to keep the study reusable and repeatable. The software can be found in an online code repository[1] while the reference data is available from a data repository[2].

[1]CMP software: https://github.com/serviceprototypinglab/cmp-testbed

[2]Evaluation data: https://zenodo.org/record/1311795

References

[1] Paul Alpar and Ariana Polyviou. Management of multi-cloud computing. In *Global Sourcing of Digital Services: Micro and Macro Perspectives - 11th Global Sourcing Workshop 2017, La Thuile, Italy, February 22-25, 2017, Revised Selected Papers*, pages 124–137, 2017.

[2] Kate Keahey, Pierre Riteau, and Nicholas P. Timkovich. Lambdalink: an operation management platform for multi-cloud environments. In *Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017*, pages 39–46, 2017.

[3] José Luis Lucas-Simarro, Rafael Moreno-Vozmediano, Rubén S. Montero, and Ignacio Martín Llorente. Cost optimization of virtual infrastructures in dynamic multi-cloud scenarios. *Concurrency and Computation: Practice and Experience*, 27(9):2260–2277, 2015.

[4] Rima Grati, Khouloud Boukadi, and Hanêne Ben-Abdallah. Saas cloud provider management framework. In *ICE-B 2015 - Proceedings of the 12th International Conference on e-Business, Colmar, Alsace, France, 20-22 July, 2015.*, pages 221–228, 2015.

[5] Nicolas Ferry, Franck Chauvel, Hui Song, Alessandro Rossini, Maksym Lushpenko, and Arnor Solberg. Cloudmf: Model-driven management of multi-cloud applications. *ACM Trans. Internet Techn.*, 18(2):16:1–16:24, 2018.

[6] Dennis Wehrle, Thomas Liebetraut, Isgandar Valizada, and Klaus Rechert. Emulation-as-a-service - workflows and infrastructure to support recomputable science. In *Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2014, London, United Kingdom, December 8-11, 2014*, pages 962–967, 2014.

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

[8] 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.

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

[10] 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.

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

[12] 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.

[13] Antonio Brogi, José Carrasco, Javier Cubo, Francesco D'Andria, Elisabetta Di Nitto, Michele Guerriero, Diego Pérez, Ernesto Pimentel, and Jacopo Soldani. Seaclouds: An open reference architecture for multi-cloud governance. In *Software Architecture - 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings*, pages 334–338, 2016.

[14] 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.

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

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

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

[18] Jürgen Cito and Harald C. Gall. Using docker containers to improve reproducibility in software engineering research. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 906–907, 2016.