Shade: Addressing Interoperability Gaps Among OpenStack Clouds

Samuel de Medeiros Queiroz[1], Monty Taylor[2], Thais Batista[1]
[1]DIMAp, Federal University of Rio Grande do Norte, Natal, Brazil
[2]Infrastructure Team, OpenStack Community
e-mail: samueldmq@gmail.com, mordred@inagust.com, thais@dimap.ufrn.br

***Abstract*— As much as OpenStack promised a utopian future where an application could be written once and target multiple clouds that run OpenStack, the reality was that vendor choice leaked through the abstractions to the point where the end user must know about deployment and configuration details, compromising interoperability and favoring vendor lock-in. Shade is a middleware written in Python by the OpenStack community which stands between users and clouds, abstracting vendor differences in order to allow a seamless experience in multi-cloud environments. It is widely used by OpenStack Continuous Integration systems nowadays, booting thousands of servers every day in numerous deployments distributed around the globe. This paper enumerates, categorizes and exemplifies the interoperability issues found in OpenStack deployments and then describes how Shade addresses most of them.**

Keywords-Interoperability; IaaS; OpenStack.

I. INTRODUCTION

Cloud computing is a model for enabling ubiquitous, convenient, on-demand self-service access to a shared pool of configurable computing resources over the network [1].

Infrastructure as a Service (IaaS) is the cloud computing model that allows users to consume processing, storage, and networking resources from a data center, providing users the ability to deploy and run arbitrary software. Such resources may be served in a private, public or hybrid deployment model. OpenStack [2] is the largest open source IaaS solution nowadays, empowering hundreds of companies around the globe to run production environments with no license cost.

With many options available, users may benefit from the ability of moving between providers when convenient, avoiding vendor lock-in. In order to make that possible, different OpenStack clouds must be interoperable. As an open source project, OpenStack is designed to support various use cases and configurations via highly flexible and configurable services. By allowing such a flexibility in its use cases, the responses returned by different clouds may vary significantly, compromising both syntactic and semantic interoperability.

After identifying syntactic and semantic interoperability issues, this paper presents Shade [3], a middleware standing between the clouds and end users that was proposed in the OpenStack ecosystem to abstract such issues. Shade is a library that exposes the most common cloud operations, making deployment and configuration choices transparent to end users. This paper classifies the identified issues, and then shows how Shade addresses them. Despite the fact that Shade is able to perform many use cases in OpenStack clouds, the examples in this paper focus on creation and management aspects of servers.

The next sections of this paper are organized as follows: Section II is a background section highlighting interoperability definitions, what OpenStack is and what syntactic and semantic gaps exist in it; Section III presents Shade, the technical solution abstracting those gaps in OpenStack clouds; Section IV presents related work, describing how this study is unique; and Section V presents the final remarks.

II. BACKGROUND

This section describes what interoperability and OpenStack are, and then enumerates the syntactic and semantic interoperability issues that exist in OpenStack.

A. Interoperability

Interoperability is the capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units [4]. In an interoperable environment, great user experience is achieved because users are able to communicate with all functional units seamlessly, disfavoring vendor lock-in.

There are two levels of interoperability: (i) syntactic: all functional units use the same data formats and communication protocols; and (ii) semantic: the results returned by all functional units have the same accurate interpretation, i.e., after performing requests to the units, users understand that they have executed the same functions and thus have been put into the same state.

Interoperability is a characteristic that may be achieved in different phases of the system lifecycle, in two manners: by design and post-facto. The former is when the functional units are all designed to be interoperable, and then built to comply with the well-defined interoperability syntactic and semantic specifications; while the latter is when the functional units exist and, without being prior designed to, are redesigned to become interoperable. The latter is expected to be much more complex, since there will be very well defined use cases using protocols, data formats and semantics particularities that will need to be given away for the sake of interoperability, affecting end users.

 ISBN: 978-1-61208-607-1

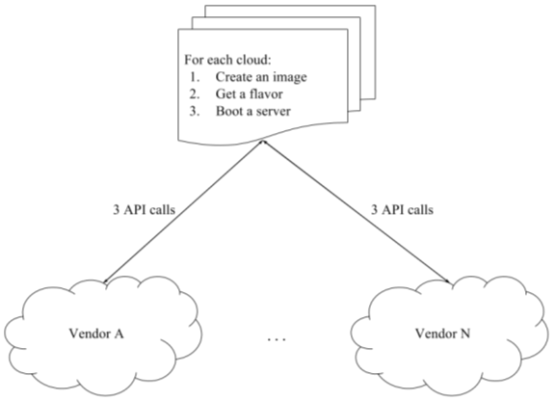

Figure 1. Users want to transparently create servers across multiple clouds via a single application.

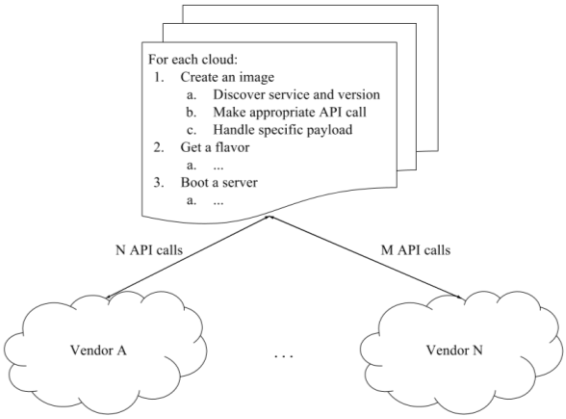

Figure 2. Users need to figure out vendor specific deployment and configuration choices to communicate to multiple clouds.

B. OpenStack

OpenStack is an open source IaaS platform, consisting of interrelated services exposing REST APIs to control diverse, multi-vendor hardware pools of processing, storage, and networking resources throughout a data center. In this context, interoperable functional units may represent clouds run by multiple vendors, where the user can communicate to all of them, equally; or a single cloud, where users are able to communicate seamlessly upon upgrade or downgrade. In both cases, deployment and configuration choices must be transparent.

Achieving and keeping interoperability within a single cloud is much simpler and more natural, as vendors do not want to break their customers. On the other side, however, not all vendors struggle to be interoperable with its competitors, favoring vendor lock-in.

From a cloud user point of view, interoperability translates into the utopian use case represented in Figure 1, where users create servers with 3 steps: (i) create an image, (ii) get a server configuration (flavor) and then (iii) boot the server; without any specific logic depending on what deployment and configuration choices have been made by vendors. In this model, the code would be written once and target multiple clouds that run OpenStack.

In reality, however, choosing a vendor leaks through the abstractions to the point where the end user must know about what deployment and configuration choices were made. This causes logic to require a-priori knowledge about clouds, as well as conditional complex logic even on discoverable differences, which would result in many extra API calls and conditional statements in the user application, as illustrated in Figure 2.

By analyzing multiple OpenStack clouds from different vendors, we were able to identify several syntactic and semantic interoperability issues.

1) Syntactic: when different clouds expose a functionality that is semantically equivalent, but it is exposed in a noninteroperable manner because there are differences in the communication protocols or data formats, i.e., the REST parameters or payloads, respectively.

The two patterns for the occurrence of strictly syntactic issues are listed below. Let A and B be two OpenStack clouds.

- The functionality is exposed through different APIs. Cloud A deploys the Nova Network service for networking operations. Cloud B deploys the Neutron service. As a user, how may you write an application that shows floating IPs in both clouds?
- The underlying functionality mechanism is pluggable, such as when a vendor requires password authentication and another requires a proprietary authentication mechanism. Both would return a token upon successful authentication, but each require specific REST payloads. How do you get a token in both clouds?

2) Semantic: when different clouds expose behaviors through syntactically equivalent protocols and data formats, but the results returned do not have the same accurate interpretation.

The five patterns for the occurrence of strictly semantic issues are listed below. Let A and B be two OpenStack clouds.

- Different authorization requirements for the functionality: cloud A requires a user to have member role in order to upload an object, whereas cloud B requires admin role. As a user with member role in both clouds, how may you upload an object in both A and B?
- Cloud-wide restrictions on resources: cloud A sets the maximum size of an image to 512 megabytes, while cloud B sets it to 1 gigabyte. How do you upload your 700-megabyte Linux image to both clouds?
- User account-wide restrictions on resources: you need to boot 20 servers, of which 10 go in cloud A, and 10 in cloud B. Your quota in cloud A allows you to boot up to 6 servers, and your

 ISBN: 978-1-61208-607-1

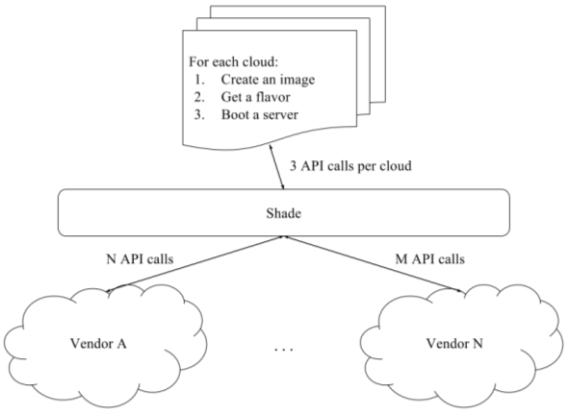

Figure 3. User executing the same program seamlessly across multiple clouds, with Shade as a middleware.

account in cloud B allows you to boot up to 12 servers. How do you boot 10 servers in each cloud?

- Inconsistent resource discovery: how do you discover the latest version of your preferred Linux image in both A and B?
- Pluggable underlying mechanism: all users in cloud A are backed by an LDAP server which is read-only by OpenStack. Cloud B uses a read-write SQL backend. How do you write an application that needs to create users in both clouds?

3) Syntactic and Semantic: there are two patterns for issues affecting both syntactic and semantic interoperability simultaneously. Let A and B be two OpenStack clouds.

- Multiple workflows for complex operations: booting a server with a floating IP attached to it is a functionality that involves many API calls and may happen in many manners, depending on how the cloud is configured, and what services are available. How do you boot a server in both cloud A and B without needing to know what deployment and configuration choices were made?
- Functionality is not provided: you write an application that uses Database as a Service (DBaaS) to create and configure a database at execution time. How do you deploy that application in both clouds A and B, given only cloud A deploys the OpenStack DBaaS solution?

III. SHADE

OpenStack has a large Continuous Integration (CI) system that launches thousands of servers every day to run tests on. It spins up servers in several clouds distributed around the globe. As a result, the CI team has learned a lot about what needs to be done to communicate with multiple clouds. Shade emerges as a promise of sharing that knowledge as a reusable library, as opposed to keeping it all inside CI scripts.

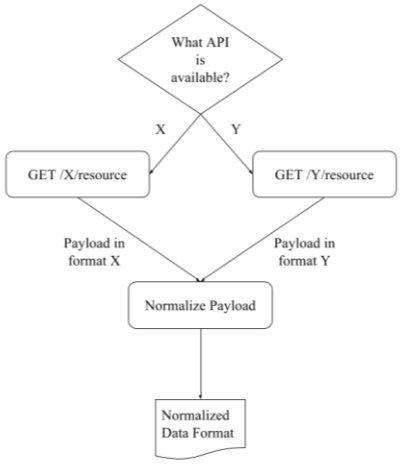

Figure 4. Discovery of the GET API for a resource and normalization of the retrieved resource.

Shade is a library written in Python standing between the user and the OpenStack clouds, abstracting most of the interoperability gaps. A consumer of Shade should never need to put in logic, such as "if my cloud supports X, then do Y, else Z". Shade will handle all the differences between clouds when possible, allowing users to seamlessly run applications across multiple OpenStack clouds, regardless what deployment choices were made. This is illustrated in Figure 3, which makes the use case in Figure 1 possible without adding complexity to the user application, as shown in Figure 2.

The next subsections will go through the issues described in Section II-B, detailing how Shade fix most of them. For the issues Shade cannot fix, we will give suggestion on how they can be addressed in the user side.

A. Syntactic

The mechanism Shade developed to abstract vendor choices on protocols and data formats to its users is by discovering what underlying APIs are available to serve the requested functionality and then standardizing resource representation through a normalization process. The normalization process consists of mapping attributes of different data representations to a common data format, which in this case is a JSON representation format that is exchanged in the REST calls, allowing users to safely rely on it. The overall process is illustrated in Figure 4.

1) Functionality is exposed through different APIs: when the functionality is exposed through different services or by the same service but in different versions, it is implemented by different OpenStack REST APIs, meaning multiple URLs and payload formats to be handled. As the URLs are not equal, the request protocol is not the same. Since the input or output payloads change, the data format is affected as well.

In order to solve this, Shade identifies what service and version are available in the service catalog, then proceed

 ISBN: 978-1-61208-607-1

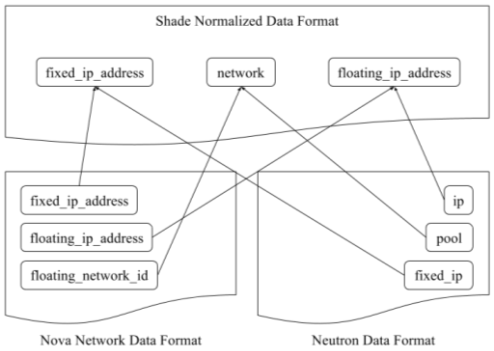

Figure 5. Normalizing a floating IP returned by Nova Network (bottom left) and another returned by Neutron (bottom right).

with the appropriate call. After getting the return from the service, it normalizes the result before returning to the user.

Networking capabilities were initially supported by Nova, the OpenStack Compute Service, via a subservice named Nova Network. Later on, Neutron, the OpenStack Networking Service, was created to centralize all those capabilities.

If an application requests Shade to show a floating IP, it identifies if Nova Network or Neutron is available, then proceed with the appropriate API call. After getting the return from the service, it normalizes the floating IP resource by mapping attributes from the heterogeneous data format to attributes in the normalized format, as shown in Figure 5 for both Nova Network and Neutron.

2) Pluggable underlying mechanism: OpenStack is designed to be flexible, supporting multiple vendors and technical solutions in most of its functionalities via plugins. While the semantic is preserved, different plugins may take different payloads to perform the requested operation. Since vendors are in charge of defining what plugins are available in a cloud, if the sets of plugins in different clouds are mutually exclusive, the user would need to use multiple payload formats to communicate to multiple clouds.

In order to communicate with OpenStack services, users must use tokens. As the authentication mechanisms are provided by plugins, there are multiple ways to authenticate and get a token.

Consider A and B are two OpenStack clouds, both will return a token upon successful authentication. Cloud A provides a password authentication plugin, that expects a request payload as in Figure 6, while cloud B provides a proprietary plugin that takes specific arguments, expecting a request payload as in Figure 7.

If the sets of plugins are mutually exclusive, there is absolutely nothing that can be done to get around and authenticate the user seamlessly across clouds with the same arguments. However, most OpenStack cloud

```
{
  "auth": {
    "identity": {
      "methods": ["password"],
      "password": {
        "user": {
          "name": "admin",
          "domain": {"name": "Default"},
          "password": "devstacker"
        }
      }
    }
  }
}
```

Figure 6. Payload for a POST /v3/auth/tokens API call to get a token using the password authentication plugin.

```
{
  "auth": {
    "identity": {
      "methods": ["xpto"],
      "xpto": {
        "id": "0a33--"
        "sequence": "2",
        "secret": "230"
      }
    }
  }
}
```

Figure 7. Payload for a create token POST /v3/auth/tokens API call to get a token using a proprietary plugin.

providers support at least the password authentication plugin, which is what Shade uses.

B. Semantic

The purely semantic issues found in OpenStack are related to how the cloud and the user account are configured, thus they cannot be solved by a technical workaround. Despite the fact Shade not being able to workaround them, there are some approaches users may take to avoid such issues when negotiating their contracts with cloud vendors and when developing their applications.

1) Different authorization requirements for the functionality: OpenStack uses Role Based Access Control (RBAC) to protect its functionalities. For each API exposed, the roles required to access it are configured by the cloud provider. When different providers configure

 ISBN: 978-1-61208-607-1

access control differently, a user with the same set of roles in multiple clouds will get unauthorized errors when performing the same operations in the cloud with least privilege.

Let A and B be two OpenStack clouds, both support the upload and storage of objects. However, cloud A requires the admin role, while the cloud B is more permissive and requires the admin role or the member role. A user with member role trying to upload an object in both clouds will get an unauthorized error when trying to upload an image to cloud A, because they do not own the required permissions.

In this case it is up to the user to negotiate with the cloud vendor what functionalities will be available to their account.

2) Cloud-wide restrictions on resources: due to nonfunctional requirements such as reducing complexity and optimizing available storage, vendors have to configure some options that establish upper limits for resources upon creation. When clouds have different limits for a given resource, a user may get an error when trying to create resources in the cloud with the lowest limit.

When requested to create a resource, Shade makes the appropriate call to the underlying services. If different clouds have different limits for resource creation, such as the maximum disk size an uploaded image may use or how deep a project hierarchy may go, there is nothing Shade can do to work around that.

It is up to the user to understand what limits the cloud vendor sets and decide if they are acceptable. If not, try a different vendor.

3) User account-wide restrictions on resources: restrictions on resource creation in a user account basis are called quotas. They define how much resources a user may use up to, such as number of virtual machines that can be instantiated. They are assigned by the vendors to users upon request. If the limits are not consistent across different clouds, the user may get errors in a cloud with lower limits when trying to perform the same create operation across clouds.

When Shade tries to perform a create call and the user quota is not enough, Shade will simply raise to the user the error it got from the underlying service.

An example is when a user needs to boot more servers than what they are allowed. In that case, the user would need to negotiate a consistent quota for booting servers across cloud vendors.

4) Inconsistent resource discovery: some resources are created by the vendors and are cloud-wide, such as the public network, user roles and default images. The lack of standardization on what is available by default and how those cloudwide resources are labeled disfavors users to programmatically discover them in a multi-cloud environment.

How can Shade find the latest Ubuntu image available in all clouds? There is no standardization in resources names across clouds, neither helpful metadata to make it possible. Image metadata is entirely vendor-defined, thus there is no way Shade can understand it precisely in a multi-cloud environment.

Despite the fact the user cannot fully understand the default cloud-provided resources, they can create and name their own resources. Thus, a possible solution for this issue is that the users create their own resources. In the example above, the user could upload the same image to all clouds in use, ensuring both the image contents and name are the same.

5) Pluggable underlying mechanism: as stated in Section III-A2 Pluggable underlying mechanism, OpenStack supports multiple vendors and technical solutions via plugins. Plugins act as backends for the REST APIs, whose are always available, regardless the plugin implementing the operation for that API or not. An error stating the functionality is not implemented may be raised, or the API call may be silently ignored. In that case, multiple clouds using different plugins might have inconsistent behaviors when requested to execute the same API call.

It is very common to organizations to maintain a central source of truth for authentication, such as an LDAP server, when they need to have a consistent user management across the whole organization, including its applications. OpenStack provides a mechanism to integrate LDAP servers for authentication purposes. Companies do not want, however, that a deletion of an OpenStack user propagate and delete that user for all their applications. In this scenario, OpenStack would have read-only access to the LDAP server.

Consider A and B two OpenStack clouds, both will return an access token upon successful authentication. Cloud A uses a read-only LDAP backend, while cloud B communicates with a read-write SQL backend.

When performing a create user call, Shade would be successful when calling cloud B. However, it would get an exception in the call to cloud A. There is nothing Shade can do about it, since it is a functionality that is not supported by some vendors depending on how they configure their clouds.

The users need to understand what plugins the vendors support and if those meet their needs. If not, they would need to try a different vendor.

C. Syntactic and Semantic

1) Multiple workflows for complex operations: providing IaaS involves non-trivial operations, such as instantiating a virtual machine on a hypervisor and assigning a public IP address to it. By supporting many vendors and technical solutions, there are multiple manners to solve such complex tasks, each one taking a

 ISBN: 978-1-61208-607-1

```
 import shade

for cloud_name, region_name in [
    ('cloud-a', 'region-a'),
    ('cloud-b', 'region-b')]:

    # Initialize the cloud
    cloud = shade.openstack_cloud(
        cloud=cloud_name,
        region_name=region_name)

    # Upload an image to the cloud
    image = cloud.create_image(
        'devuan-jessie', wait=True,
        filename='devuan-jessie.qcow2')

    # Find a flavor with at least 512MB
    # of RAM
    flavor = cloud.get_flavor_by_ram(512)

    # Boot a server, wait for it to boot,
    # and then do whatever is needed to
    # attach a public IP to it.
    cloud.create_server(
        'my-server', image=image,
        flavor=flavor, wait=True,
        auto_ip=True)
```

Figure 8. Using Shade to boot a server with a public IP attached to it in multiple clouds.

different workflow involving multiple API calls. Even if the final semantic result is the same, executing such operations does not consistently use the same data formats neither have the same accurate interpretation throughout the process.

In the example of booting a server and assigning a public IP to it, the first step is to figure out what is the networking service in the cloud: Nova Network or Neutron. In this example, let's assume Neutron is available. The second step is to query Neutron in order to figure out if there is a public network to boot the server on. If there is, then a single API call to Nova, the Compute Service, may be performed requesting the virtual machine to be instantiated and be put directly in that public network. If there is not, the solution will be first to create a virtual machine with a private IP and then to assign a floating IP later on via NAT mechanism.

In order to assign a floating IP via NAT mechanism, first try to pass the port ID of the private IP of the server to the floating IP create call. If that is not possible, create a floating IP and then attach it to the server. Executing all this complex functionality with Shade is as simple as shown in Figure 8.

Another complex example is the upload image functionality, managed by Glance, the Image Service. There are two mechanisms for that: (i) upload data directly to Glance via HTTP PUT, or (ii) upload the data to the Object Storage service, Swift, and then import it to Glance with an import task. Both alternatives are available in every Glance version 2 service. In some clouds, upload via PUT is disabled, and in other clouds the task import mechanism does not do anything, just ignores the requested action.

More specifically in the task import path, the accepted payloads are all vendor or plugin specific, presenting the issues described in Section III-A2 Pluggable underlying mechanism.

2) Functionality is not provided: cloud vendors may opt to not deploy or to remove some of the OpenStack services for whatever reason, such as it is not part of their market strategy. In that case, users would not be able to use the same functionality across multiple clouds.

As an example, Trove, the Database as a Service (DBaaS) service may not be available in all clouds. That would make it unfeasible to deploy an application that needs DBaaS in the clouds that do not deploy it.

Before choosing what cloud vendors to go with, the users need to understand well their service catalog to make sure all the expected functionalities are provided.

D. Validation

Shade is currently the library handling all the clouds differences for the whole Continuous Integration system of OpenStack, which spins up thousands of servers every single day across many non-interoperable clouds. It is a project developed by the OpenStack community and the authors work on this project, which is also used in a master's thesis.

In addition to the above mentioned use, Shade is also used in Ansible modules, which enable several cloud providers to orchestrate their clouds via scripts. Such modules were used to make the program The Interoperability Challenge possible, where multiple cloud vendors were challenged to run the same workloads against their clouds, live, in front of thousands of attendees at two editions of the OpenStack Summit.

In the Barcelona edition, there were 16 participating companies: Canonical, Cisco, DreamHost, Deutsche Telekom, Fujitsu, HPE, Huawei, IBM, Intel, Linaro, Mirantis, OVH, Rackspace, Red Hat, SUSE and VMware. In the Boston edition, the 15 participants were IBM, VMware, Huawei, ZTE, SUSE, EasyStack, T2Cloud, Red Hat, Rackspace, Canonical, VEXXHOST, Deutsche Telekom, Platform9, Wind River and NetApp.

All companies, in both editions, were successful on running the workloads defined by the community and

 ISBN: 978-1-61208-607-1

implemented via orchestration scripts using the Ansible modules. Without Shade, there would be no way to communicate to all those clouds transparently without implementing the Shade logic in the Ansible modules themselves.

The patterns for the interoperability gaps detailed in this paper are enumerated in Table I, which summarizes what is solved by Shade and what requires user intervention, be it negotiate with the service provider or to use a work around when writing applications. Despite the fact that Shade solves fewer issues in terms of quantity, the ones it solves are the most impeditive for interoperability in OpenStack, because they bring a lot of complexity to the user side, while the issues solved by the users are related to understanding what functionalities are available in the clouds and how their accounts are configured.

TABLE I. PATTERNS ADDRESSED BY SHADE

Pattern	Action	
	Shade	***User***
3.A.1 Functionality is exposed through different APIs	X	
3.A.2 Pluggable underlying mechanism		X
3.B.1 Different authorization requirements for the functionality		X
3.B.2 Cloud-wide restrictions on resources		X
3.B.3 User account-wide restrictions on resources		X
3.B.4 Inconsistent resource discovery		X
3.B.5 Pluggable underlying mechanism		X
3.C.1 Multiple workflows for complex operations	X	
3.C.2 Functionality is not provided		X

IV. RELATED WORK

Even in 2010, when cloud computing was growing as a concept, Dillon et al. already signaled that interoperability deserved substantial further research and development [5].

In a literature review, we were able to identify studies focusing on interoperability among different IaaS cloud platforms. Zhang et al. [6] conducted a comprehensive survey on the state-of-the-art efforts for understanding and mitigating interoperability issues. Parák et al. [7] discussed challenges in achieving IaaS interoperability across multiple cloud management frameworks.

No study reporting that interoperability issues occur within a single platform was found, and that is the case being reported in this paper with OpenStack.

As opposed to defining open protocols and making the existing vendor adapt their deployments to it, the solution as presented in this paper is a post-facto high-level end-user broker for facilitating effective interoperability in the cloud, as clarified in by Parák et al. [7].

Loutas et al. [8] highlighted that creating different interoperability standards and frameworks can possibly lead to different interoperability solutions which are not interoperable between each other. However, we found that creating platform-specific interoperability frameworks such as Shade is a good strategy because another middleware could be built on the top of it and consider all OpenStack deployments interoperable, without caring about particularities of the OpenStack world, and then solve interoperability limitations between different cloud platforms. Creating multiple solutions of that higher level middleware would certainly be a problem.

V. CONCLUSION AND FUTURE WORK

As an open source platform, OpenStack is deployed by numerous vendors. By allowing great flexibility in its functionalities, it compromised interoperability, with the issues reported in this paper.

Shade is a Python library that was implemented to solve the issues when there is a programmatic way to discover how to perform the operations and how to interpret the results accurately. In the other cases where the issues are inherent to the platform, such as the lack of standardization on what is available by default, how cloud-wide resources are labeled and what is the available quota for a given resource, this paper recommended that the users should workaround themselves when possible, otherwise analyze the cloud offering and negotiate with the vendor directly.

Since interoperability was developed post-facto, being fully interoperable in OpenStack will never become a reality because that would mean giving up flexibility and, for that, backward incompatible changes would need to be introduced. One of the key attributes of OpenStack is that it strives to always be backwards compatible. Furthermore, it would require vendors to standardize their deployments, changing their market strategy and breaking their customers for the sake of being interoperable with their competitors.

This study was important because it showed that interoperability issues may emerge even within a single cloud platform. The issues were categorized, exemplified and a solution was proposed, allowing further improvements and studies to be placed on the top of it.

Future work may include creating another middleware on the top of Shade that is not language-specific, such as a Remote Procedure Calls (RPCs) or a REST API, allowing users to consume Shade in other languages than just Python.

Another important study would be to investigate other open source IaaS platforms to report what interoperability issues they present, then compare with OpenStack and analyze if a solution similar to Shade would apply. Example of platforms are Apache CloudStack, Eucalyptus and OpenNebula.

REFERENCES

[1] P. Mell and T. Grance, “The NIST definition of Cloud Computing”, NIST Special Publication, USA, 2011.

[2] OpenStack, https://www.openstack.org, [Online; accessed 14 January 2018].

[3] Shade Git, https://github.com/openstack-infra/shade, [Online; accessed 14 January 2018].

[4] ISO/IEC 2382:2015, "Information technology -- Vocabulary". Switzerland: ISO/IEC JTC 1, 2015.

[5] T. Dillon, C. Wu, and E. Chang, “Cloud computing: Issues and challenges”, Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications. USA: IEEE, 2010, pp. 27-33.

[6] Z. Zhang, C. Wu, and D. W. Cheung, “A survey on cloud interoperability: Taxonomies, standards, and practice”, ACM SIGMETRICS Performance Evaluation Review, vol. 40, no. 4, pp. 13-22, Mar. 2013.

[7] B. Parák and Z. Šustr, "Challenges in achieving IaaS cloud interoperability across multiple cloud management frameworks” Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing. USA: IEEE, 2014, pp. 404-411.

[8] N. Loutas, E. Kamateri, F. Bosi, and K. Tarabanis, “Cloud computing interoperability: The state of play”, Proceedings of the Third IEEE International Conference on Cloud Computing Technology and Science. USA: IEEE, 2011, pp. 752-757.

 ISBN: 978-1-61208-607-1