CloudBridge: a Simple Cross-Cloud Python Library

Nuwan Goonasekera[1], Andrew Lonie[1], James Taylor[2], Enis Afgan[2,*]

Victorian Life Sciences Computation Initiative
University of Melbourne
700 Swanston Street
Melbourne, Australia
+61 3 9035 5822
{ngoonasekera, alonie}@unimelb.edu.au

Department of Biology
Johns Hopkins University
3400 University Blvd
Baltimore, MD, USA
+1 410 516 7330
{jxtx, enis.afgan}@jhu.edu

ABSTRACT

With clouds becoming a standard target for deploying applications, it is more important than ever to be able to seamlessly utilize resources and services from multiple providers. Proprietary vendor APIs make this challenging and lead to conditional code being written to accommodate various API differences, requiring application authors to deal with these complexities and to test their applications against each supported cloud. In this paper, we describe an open source Python library called CloudBridge that provides a simple, uniform, and extensible API for multiple clouds. The library defines a standard 'contract' that all supported providers must implement, and an extensive suite of conformance tests to ensure that any exposed behavior is uniform across cloud providers, thus allowing applications to confidently utilize any of the supported clouds without any cloud-specific code or testing.

Categories and Subject Descriptors

Computer systems organization - Architectures - Distributed architectures - Cloud computing

Keywords

Cloud Computing, Cross-cloud, Interoperability, Portability

1. INTRODUCTION

Rapid adoption and deployment of cloud infrastructure, whether it be for new cloud applications [1], microservices [2], [3], or migration of legacy applications into the cloud [4], calls for treatment of infrastructure as code [5]. Furthermore, with an increased reliance on cloud providers as application deployment platforms, it is becoming even more important to be able to deploy increasingly complex application stacks in a cross-cloud compatible manner [6], [7]. With the core conceptual underpinnings of major cloud providers rapidly converging in terms of functionality and scope (e.g., virtual machines, firewall rules, block storage and object storage among others), cross-cloud compatibility would appear to be simple at first glance. However, writing such applications is often quite difficult in practice, as each vendor offers proprietary APIs with various and often subtle differences.

To support cross-cloud compatibility at an API level, there are two broad approaches: wrappers and adapters [8]. Wrappers hide the actual vendor API calls and expose a custom wrapper API, and internally translate the wrapper's API to the vendor APIs (e.g., Libcloud and jClouds). Adapters are implemented, or enabled, at the middleware layer and expose several popular APIs, allowing the end-user to utilize a familiar API of choice, which the adapter will relay to the native API. For example, Amazon's APIs are available in systems such as OpenNebula [9] and OpenStack [10].

While adapted APIs provide upfront relief to the application developer by being able to utilize a familiar SDK for multiple cloud providers (e.g., `boto` library for Python), in the long run, the subtle differences across the APIs call for special-casing the application code or limit the features of the application on select clouds. Some specific examples of this include the inability to name instances on OpenStack using the EC2 APIs and difficulties in correlating EC2 IDs with corresponding OpenStack IDs.

Many of the wrappers that are currently available (see related works section) have similar issues, resulting in abstractions which often require complex special cased code to deal with various differences between clouds. This is usually because the wrappers adopt a lowest-common-denominator approach for wider provider compatibility. In practice however, this poses a significant hurdle for application developers, as it is non-trivial to test an application for compatibility across a wide range of clouds or indeed, even obtain access to a wide range of clouds. This leads to reduced application portability and incurs temporal, financial and complexity costs.

In this paper, we present a Python library called CloudBridge. CloudBridge follows a fixed set of principles to provide a minimal, uniform, tested, and extensible programmatic interface to multiple cloud providers. It is minimal because the library forwards all calls to existing provider SDKs, forming a thin, simple layer with a consistent interface. It is uniform because it offers a consistent set of methods across all services and all providers, and all implementations. It is tested because the same test suite is used for all supported providers, obviating the need for applications to be individually tested against each implementation. It is extensible because the interface is well defined, and new provider implementations can be dynamically plugged in. The library focuses on the core Infrastructure-as-a-Service (IaaS) cloud features that are

** Corresponding Author*

likely to exist, or can be simulated, for multiple providers. At the moment, support for compute and object storage services on Amazon Web Services and OpenStack clouds is available while support for the Google Cloud is under development.

The library is very well documented, paving the path for community contributions to add support for additional clouds. There is dedicated usage documentation available at https://cloudbridge.readthedocs.org/. The library is released under the MIT open source license and the code is available at https://github.com/gvlproject/cloudbridge.

2. RELATED WORK

Providing solutions that target multiple clouds via a uniform layer has been a goal for several projects in the past. In this section we examine the current solutions and highlight how this new library is unique.

2.1 Language-agnostic solutions

A version of the adapter model is an HTTP service that provides its own APIs that target multiple clouds. Projects such as Apache Deltacloud (deltacloud.apache.org) and Dasein Cloud (dasein.org) allow HTTP requests to be sent to a single API provided by a single service, and the service internally translates the request to the native cloud provider. The benefit of using HTTP as a common interface is exciting due to its wide applicability, but it does require the application developer to operate at a lower level of abstraction than using a native, language-specific library, losing idiomatic expression and convenience.

The adapter approach can also be realized via a standard that multiple service providers implement and allow applications to uniformly interact with different clouds using that same API. Cloud Infrastructure Management Interface (CIMI) [11] and the Open Cloud Computing Interface (OCCI) [12] are the two standards that exist today for IaaS management. Although conceptually desirable, in practice, adoption of these standards has been low [13] as vendors (particularly the major ones) have little incentive to conform to the standard, plus it limits the amount of innovation and differentiation a vendor can supply. In the long run, as vendor services converge in terms of functionality and scope, standards-based implementations may become more prevalent.

2.2 Language-specific solutions

Thus far, the most widely adopted solution for achieving multi-cloud compatibility has been via language-specific wrapper libraries: jClouds for Java (jclouds.org), Fog for Ruby (fog.io), pkgcloud for JavaScript via Node.js (github.com/pkgcloud/pkgcloud), and Libcloud for Python (libcloud.apache.org). Graham and Liu [14] provide a detailed evaluation of jClouds against a number of cloud APIs and its suitability to the task. The approach undertaken by them can be used as a template for performing similar evaluations for other language libraries and clouds. Since this represents a deviation from the goals of the current paper, we instead focus on conceptual differences between CloudBridge and other similar solutions. We specifically focus on Libcloud as that is the only other Python library available in this space.

While Libcloud and CloudBridge may appear to be solving the same problem at first glance, we believe that these two libraries target two different levels of abstraction and are therefore conceptually and functionally very different, and are animated by fundamentally different philosophies. A rough analogy would be that Libcloud is to CloudBridge what `urllib` is to `requests`: a detail-oriented vs. a higher-level abstraction. In principle, Libcloud could be used as an underlying provider library for CloudBridge, although in practice, we have chosen not to do so at present for reasons explained below. In more detail, the differences are as follows:

- *Libcloud offers a lowest-common-denominator approach* by targeting a large number of clouds with a minimum level of abstraction, and offering extra functionality as cloud specific extensions. In practice, this requires that special-case code be written using cloud-specific "extended methods", unless the application's requirements are very modest. In contrast, CloudBridge focuses on a write-once, run-anywhere approach, rejecting a widest-possible-compatibility approach;
- *Libcloud wraps libraries at a ReST/HTTP level, instead of using native SDKs*. The positive aspects of this approach are that it minimizes the number of Cloudlib dependencies and gives complete control over the client-side implementation (e.g. switching to an asynchronous implementation). The negatives aspects are that the library becomes slow to incorporate new features because they have to be implemented directly in the library vs. reusing native libraries, meaning that Libcloud will always be slightly behind the curve (native libraries are more up-to-date since the cloud provider adds native library support with the release of a new service feature). Further, there is a significant duplication of work, often of a non-trivial nature (e.g. `boto` library has over 77,000 lines of code that largely has to be replicated in the 250,000 lines of Libcloud code – compared to roughly 1,000 lines of code for a single provider in CloudBridge). This design decision leads to a great deal of complexity within Libcloud, and does not leverage the large body of work that is made available through native provider libraries. In contrast, CloudBridge attempts to leverage existing bodies of work, thus simplifying implementation and improving confidence in correctness;
- *Libcloud's testing is provider-specific*, since the exposed methods tend to be provider specific. We see this as a very significant library design problem because it offers few guarantees to the developer of having a "write once, run anywhere" experience. Since cloud infrastructures are non-trivial to get access to, deploy and test on, this makes it extremely difficult for an application to support a large

variety of clouds in practice. In contrast, CloudBridge takes this burden upon itself by testing the implementation against all supported clouds, leaving the application developer free to test application code only;

- *Libcloud supports a wider range of provider implementations*. At present, Libcloud supports more than 30 cloud providers, exceeding CloudBridge by an order of magnitude. Over time, we will be adding support for additional providers to CloudBridge, with Libcloud itself being potentially used as an SDK. However, as described in the philosophy section, CloudBridge focuses on mature, widely used clouds and is unlikely to support as wide a gamut of providers. Instead of wide provider coverage, the focus of CloudBridge is instead on uniformity and reliability for the application developer;
- *Libcloud does not offer a way to determine provider capabilities at runtime*. Unlike libraries like jClouds, Libcloud does not offer a feature to determine provider capabilities at runtime. CloudBridge, at present, has limited supported for this capability (e.g., via the *has_service* function; see Section 5).

While we did consider using Libcloud as an underlying provider for CloudBridge, we found that in the case of AWS and OpenStack, the vendor provided SDKs were naturally at a higher-level of maturity and functionality as mentioned above, and were therefore more suitable choices. Nevertheless, Libcloud may be a more suitable choice for future provider additions to CloudBridge, and would need to be evaluated on a case by case basis.

We believe the outlined differences from existing approaches and projects position CloudBridge into its own space where it delivers value for cloud application developers as well as cloud systems developers. Specifically, application developers gain a uniform API for all providers that guarantees operation on all supported clouds without the need for cloud-specific testing. System developers wanting to add support for additional providers can do so quickly by wrapping native SDKs, leveraging existing bodies of work. They also do not need to write additional tests and can instead rely on the existing test infrastructure.

3. CLOUDBRIDGE ARCHITECTURE

We detail our description of CloudBridge by discussing the design philosophy. We then describe the conceptual layout of the library, followed by implementation details.

3.1 Design Philosophy

CloudBridge aims to provide a uniform and extensible core for manipulating infrastructure on multiple cloud providers. Since cloud vendors differ in their service model and technical realization [15] (partially due to their business models), a library such as this is forced to choose a set of principles that drive its development and accommodate for the technical differences of the underlying cloud environments. Although the cloud computing space is highly diverse and rapidly evolving, we believe that the design philosophy as presented transcends the surface differences among providers and is able to cater to a range of vendors and applications.

First, the core library offers *a uniform API irrespective of the underlying provider*. This means that no special-casing of the application code is required, implying that the application can be deployed to any supported provider without modification. Second, the library provides a set of *conformance tests for all supported clouds*. With high test coverage the application developer is in a position to reliably deploy their application without having to individually test against each supported cloud. From the library development standpoint, the extensive test coverage allows sweeping code changes and new features to be added with increased confidence that no existing functionality is affected [16]. Third, it focusses on *mature clouds with a required minimal set of features*, as opposed to following a lowest-common-denominator approach. The availability and implementation details of these services may differ across the providers but basic services such as the compute service, block storage service and a firewall service, among others, should exist for a cloud provider to be added. The fourth principle is that CloudBridge, as an additional layer of indirection, needs to *be as thin as possible*. This is realized by wrapping the cloud providers' native SDKs, which leads to reduced development time, improved reliability and quicker iterations when adding new features.

Design principle three ("*wrap mature clouds*") implies that certain tradeoffs may be necessary in cases where service availability across clouds is substantially different. There are two general paths that can be undertaken in this case: (1) a higher-level service is simulated within the library on behalf of a cloud provider to adhere to the first design principle or (2) the provider service is decomposed into its most basic parts and the higher-level service functionality is left up to the library user. For example, a notion of security group as a set of firewall rules that can be jointly applied to an instance is common for AWS and OpenStack. However, Google Compute Engine (GCE) operates with individual rules only. CloudBridge can, at the interface level, either simulate the notion of a security group for GCE or decompose each security group for the two other clouds into individual rules only. Since AWS at least limits the number of rules per security group, which could cause usability issues in the future, it is better to simulate the notion of a security group for GCE. Due to such potentially conflicting provider constraints and requirements, it is impossible to set a library-wide policy, and the decision must be made on a case-by-case basis.

3.2 API Overview and Interfaces

The CloudBridge library API revolves around three concepts: (1) providers; (2) services; and (3) resources. The providers encapsulate connection properties for a given cloud provider and manages the required connection. Services expose the IaaS provider functionality, offering the ability to create, query and manipulate resources (e.g., images, instance types, key pairs, etc.). Resources represent a remote cloud resource, such as an individual machine instance (`Instance`) or a security group (`SecurityGroup`). Figure 1 captures the relationships between these three concepts and supplies a list of resource method/types available for each service. For complete interface implementation details, see the project source code repository (github.com/gvlproject/cloudbridge).

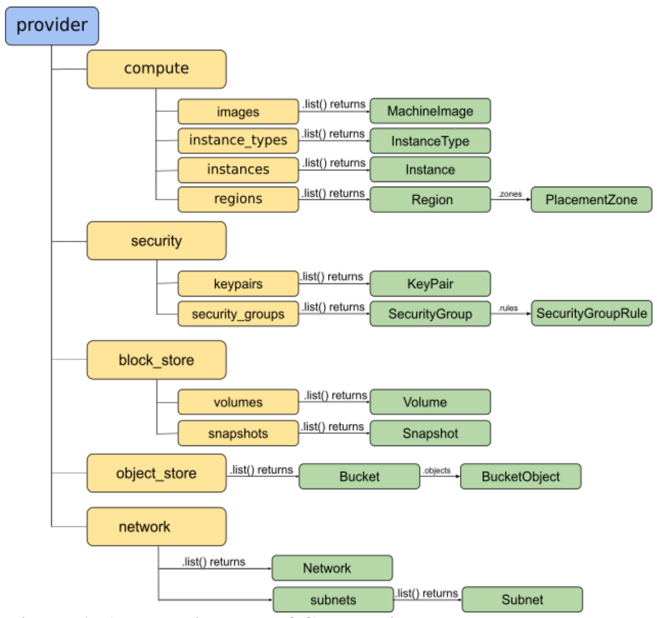


Figure 1. A class diagram of CloudBridge concepts.

Driven by the first design principle (‘*be uniform*’), CloudBridge offers a consistent set of methods across all the available services (see next section; also Table 1). The available service methods enable basic activities (`List`, `Get`, `Find`, and `Create`) to be performed for all resource types. This level of uniformity across the available services enhances developer experience as common service functionality is always present (e.g., `provider.compute.images.list()` or `provider.security.keypairs.list()`). Similarly, individual resources offer a common set of basic properties (e.g., `id`, `name`). In addition, each resource is equipped with a set of fields and methods appropriate for that resource type (e.g., `attach/detach` methods for volume objects).

3.3 Implementation

We have implemented CloudBridge in about 5,000 lines of Python, with the core code and conformance tests consisting of 3,000 lines, and provider implementations taking up 2,000. The library can be used with Python 2.7, 3.4 and 3.5 on Linux, OS X, and Windows. There is comprehensive documentation covering both usage and development, including an introductory tutorial and a contributor’s guide.

Structurally, the library implements the Bridge pattern [17] for relaying the providers’ native SDKs to the defined library interface. The adopted pattern allows the library interface to evolve independently of the underlying libraries and is supportive of adding new providers in a low-impact fashion (see Figure 2). Specifically, one needs to provide an implementation for the interface to translate the native SDK API calls into the ones defined by the library interface; no library-wide modifications are necessary. Further, the Factory pattern [17] is implemented for instantiating the requested provider implementation.

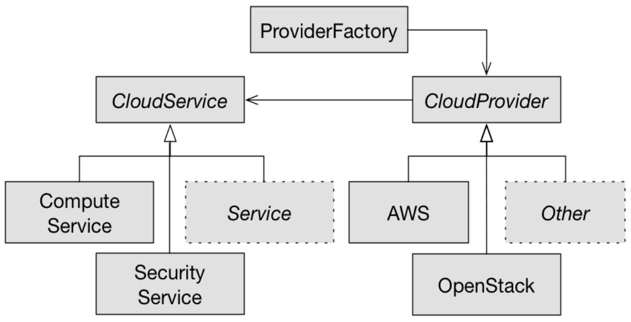


Figure 2. Bridge pattern as implemented in CloudBridge.

Driven by the fourth design principle (‘*be a thin layer*’), the library heavily relies on third party SDKs and libraries. This greatly simplifies the implementation aspects but also introduces a significant risk with dependency management. As the cloud computing space evolves, so will the SDKs’ functionality. To combat this challenge, we have adopted a model where the development branch of the library tracks the latest dependent SDK versions and any changes introduced by a dependency are dealt with during the development cycle. Just prior to a CloudBridge version release, the current dependent library versions are fixed in the library installation setup. Hence, at any point in the future, the specific versions whose operation has been verified will get installed and the library, as well as any higher-level applications, will continue to be operational.

Before services and resources for a provider can be implemented, it is necessary to define the provider connection properties. This is achieved by supplying an implementation of the `CloudProvider` abstract class, which initializes the underlying SDK connection with the required authentication tokens. Typically, the implementation will lazily establish several connection objects for various provider services (e.g., compute, network, object store). Next, each provider implementation implements all the interface services and resources. As mentioned in the previous section, resources implement a core set of methods and properties as well as additional properties applicable to the given resource type.

Table 1 captures the core methods for the available services as defined in the interface. Figure 3 (top) provides an example of the `Instance` class with its attributes and methods. The implementation for each attribute or method

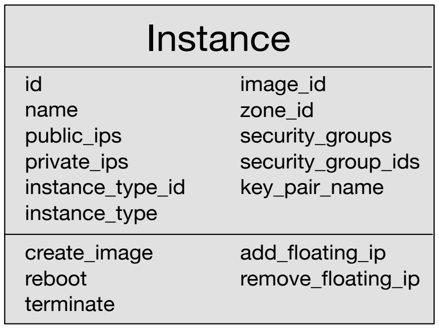
Table 1. A set of services currently supported in CloudBridge and the methods available in each service.

Service name	`list`	`get`	`find`	`create`
`InstanceService`	yes	yes	yes	yes
`VolumeService`	yes	yes	yes	yes
`SnapshotService`	yes	yes	yes	yes
`ImageService`	yes	yes	yes	no
`NetworkService`	yes	yes	yes	yes
`SubnetService`	yes	yes	no	yes
`ObjectStoreService`	yes	yes	yes	yes
`KeyPairService`	yes	yes	yes	yes
`SecurityGroupService`	yes	yes	yes	yes
`InstanceTypeService`	yes	yes	yes	N/A
`RegionService`	yes	yes	no	N/A

uses the native provider SDK to obtain the required information or perform the action defined by the library interface. The implementation manipulates the obtained information to return the results in a format that matches the CloudBridge definition. As an example, implementation for the `instance_type` method is captured in Figure 3 (bottom), for OpenStack and AWS providers.

For application developers, CloudBridge provides support for some additional features. As managing lists of resources is a common operation, CloudBridge resources can be iterated on (e.g. *`for instance in provider.compute.instances`*). This provides for more compact application code that is idiomatic Python. Similarly, consuming large lists of resource objects can quickly become a performance and usability problem. To alleviate this issue, CloudBridge provides support for paging object lists. Note that certain provider and resource combinations provide support for this natively via their SDK, which is then simply relayed via CloudBridge. In cases where this is not supported, CloudBridge provides a client-side implementation by caching a full list of resources.

The CloudBridge implementation also provides access to native provider SDK objects. Although this breaks the uniformity of the application's implementation and requires special-casing application source code, it offers an avenue for using provider functionality that is not available in CloudBridge. This feature is not advocated by the authors but may be an appropriate solution in certain circumstances.



AWS

```
@property
def instance_type(self):
    """Get the AWS instance type object."""
    return self._provider.compute.instance_types.find(
        name=self._ec2_instance.instance_type)[0]
```

OpenStack

```
@property
def instance_type(self):
    """Get the OpenStack instance type object."""
    flavor = self._provider.nova.flavors.get(
        self._os_instance.flavor.get('id'))
    return OpenStackInstanceType(self._provider, flavor)
```

Figure 3. Interface definition for the *Instance* class (top) and an example of method implementation for AWS and OpenStack providers (bottom).

3.4 Testing

Testing is one the four core design principles driving the development of this library and it is one of the main differences from existing solutions. Namely, CloudBridge implements a single test suite that is automatically applied to all supported providers. This approach makes it easier to write the tests since only a single test case needs to be designed and implemented (in contrast, Libcloud implements dozens of test cases, one for each provider). In turn, the tests validate functionality of the implementation and its consistency for all the providers at once, so no functional drift can occur. We have taken it upon ourselves to obtain access to instances of all supported cloud providers, alleviating users from needing to do the same. This allows an application developer to rest assured that the application using the library APIs will consistently operate across all supported clouds without it needing to be explicitly tested.

The complete test suite automatically runs for each code commit. Currently, the test coverage is 95% and it is automatically recalculated with each code commit; we strive for 100% coverage but this is challenging due to the library framework code that is difficult to capture as a test case.

For the purposes of speeding up the tests, mock provider libraries are used for daily commits. This has led to the complete test suite completing in under 10 seconds, vs. ~20-30 minutes using the real infrastructure. Periodically, however, the test suite is run against the actual infrastructure.

Table 2. A snippet of code used to launch an instance using CloudBridge, shown for two different clouds.

	AWS	OpenStack (NeCTAR)
1.	`from cloudbridge.cloud.factory import CloudProviderFactory`	
2.	`from cloudbridge.cloud.factory import ProviderList`	
3.	`config = {‘aws_access_key’: ‘A_KEY’,`	`config = {'os_username': 'username',`
4.	`          ‘aws_secret_key’: ‘S_KEY’}`	`          'os_password': 'password',`
5.		`          'os_tenant_name': 'tenant name',`
6.		`          'os_auth_url': 'authentication URL',`
7.		`          'os_region_name': 'region name'}`
8.		
9.	`provider = CloudProviderFactory()` `.create_provider(ProviderList.AWS, config)`	`provider = CloudProviderFactory().create_provider(` `ProviderList.OPENSTACK, config)`
10.	`image_id = ‘ami-d85e75b0’`	`image_id = ‘c1f4b7bc-a563-4feb-b439-a2e071d861aa’`

```
11.  kp = provider.security.key_pairs.create('cloudbridge_intro')
12.  sg = provider.security.security_groups.create('cloudbridge_intro', 'A CloudBridge security group')
13.  sg.add_rule('tcp', 22, 22, '0.0.0.0/0')
14.  img = provider.compute.images.get(image_id)
     # Find the smallest instance with at least 2 vcpus and 4 GB of ram
15.  inst_type = sorted([t for t in provider.compute.instance_types.list() if t.vcpus >= 2 and t.ram >= 4],
                        key=lambda x: x.vcpus*x.ram)[0]
16.  inst = provider.compute.instances.create(name='CloudBridge-intro', image=img, instance_type=inst_type,
                                              key_pair=kp, security_groups=[sg])
17.  # Wait until ready
18.  inst.wait_till_ready()
19.  # Show instance state
20.  inst.state
21.  # 'running'
22.  inst.public_ips
23.  # [u’54.166.125.219’]
24.  if provider.has_service(CloudServiceType.VOLUME):
25.    vol = provider.block_store.volumes.create(name=‘CB vol’, size=10, zone=inst.zone_id)
26.    vol.attach(inst, ‘/dev/sdb’)
```

4. USAGE

We demonstrate usage of CloudBridge via a code snippet to launch a virtual machine on a select cloud (Table 2); after the initial imports (lines 1-2), the first step is to set up a provider-specific object by supplying credentials and cloud-specific parameters (lines 3-10). Before an instance can be launched, we need to create a key pair and a security group (lines 11-13). Each time a resource is created or a reference to an existing one fetched, CloudBridge wraps the resource in an appropriate CloudBridge resource object that subsequently provides uniform fields. Once the preliminary requirements for launching an instance have been satisfied, we launch an instance by supplying desired parameters (lines 14-16). After the instance has launched, we can inspect its properties (lines 17-23). CloudBridge also offers an ability to inspect if the current cloud offers a certain service (see more about this in the next section); lines 24-26 will create and attach a new volume to the launched instance in case the provider offers a volume service. This allows the application developer to write a single version of the application code and have it apply to multiple clouds (making appropriate decisions based on the availability of a service). As visible from this example, all the code after this initial setup is equivalent irrespective of the provider. This is true for any service or resource available within CloudBridge and is the enabler of cloud-agnostic applications.

5. DISCUSSION

As captured by the design philosophy item on uniformity, any of the available API methods are uniform across all the implemented providers allowing the application to, conceptually, be developed without special-cased code. However, cloud vendors may not provide all the underlying services. For example, the Australian national cloud NeCTAR (nectar.org.au/research-cloud) does not provide the volume storage service for all users. Similarly, the US-based Chameleon cloud (chameleoncloud.org) does not provide an object storage service. In such scenarios, the application code must still account for the availability of a service before consuming it.

CloudBridge offers two methods for dealing with this scenario: `CloudProvider` class implements a `has_service` method that allows the application to check for existence of a service (e.g., `provider.has_service(CloudServiceType`

.OBJECT_STORE)). Another option is to create a hybrid provider object. For services that can be used independently, CloudBridge offers the ability to specify a per-service configuration object. Table 3 shows an example where all the services except the object store service (Swift) will target Chameleon while the object store service targets NeCTAR cloud. Unfortunately, this cannot be achieved for any combination of services because, for example, a volume service must exist on the same cloud as the instance service.

Table 3. Hybrid provider configuration sample.

```
config = {
  'os_username': 'Chameleon username',
  'os_password': 'Chameleon password',
  'os_tenant_name': 'Chameleon tenant',
  'os_auth_url': 'Chameleon auth url',
  'os_region_name': 'Chameleon region'
  'os_swift_username': 'NeCTAR username',
  'os_swift_password': 'NeCTAR password',
  'os_swift_tenant_name': 'NeCTAR tenant name',
  'os_swift_auth_url': 'NeCTAR auth url',
  'os_swift_region_name': 'NeCTAR region name'}
```

In future, we plan to extend these capabilities to broadly determine a provider's feature set so that application developers can adjust according to available features in a service (e.g., availability of block device mapping support in the ComputeService). However, this creates a tension between too fine a granularity of feature determination (which poses a greater burden on the developer to respond appropriately to a larger number of possible conditions), and too coarse a granularity, which would mandate that all clouds have all features at all times. In general, CloudBridge will err on the side of coarser granularity with clear guidelines on usage, to reduce the burden on application developers. The side-effect of this is that, by design, CloudBridge will be unable to support a broad gamut of providers with disparate capabilities. The upside of this design is that CloudBridge's abstraction is simpler.

6. CONCLUSIONS AND FUTURE WORK

Although there are existing libraries that facilitate cross-cloud application deployment, they tend to lack uniformity across their APIs. This reduces application portability and slows down development because developers need to learn what methods are available for each individual provider, conditionally handle the differences, and test their applications against each cloud. In this paper, we described a new Python library that follows a fixed set of principles to produce a minimal, uniform and extensible interface for all supported clouds and thereby deliver an interoperable interface for them. By design, the library is focused on a set of cloud features that are common across mature cloud providers such as Amazon, OpenStack and Google and, in our experience, those features are sufficient to address the requirements of a wide range of applications. Therefore, we believe the simplicity and portability of CloudBridge will go a long way in promoting its adoption.

Looking forward, we are currently adding support for the Google Cloud provider and and will be looking into adding support for additional cloud services, such as the Metadata service, support for Projects/Tenancies and the increasingly available Container Service.

We are also considering extending CloudBridge with support for orchestrating and managing container technologies, such as Docker [18] and LXC. We hope to integrate this with the cloud specific container services mentioned above, and provide a uniform abstraction for managing the lifecycle of containers, their storage connections and their deployment.

Another possible future enhancement for CloudBridge is to add support for aggregate clouds, through application of the Composite pattern [17]. By allowing a composite provider instance to be created out of two simple provider instances, it would be possible to have features such as transparent failover or cloud-bursting for example, using a user supplied policy. While there are several challenges in providing such support, it is something that we hope to explore in future.

ACKNOWLEDGMENTS

This project was supported in part through grant VLS402 from National eCollaboration Tools and Resources, grant eRIC07 from Australian National Data Service, grant number HG006620 from the National Human Genome Research Institute, and grant number CA184826 from the National Cancer Institute, National Institutes of Health.

REFERENCES

[1] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns*. Vienna: Springer Vienna, 2014.

[2] J. Thones, "Microservices," *IEEE Softw.*, vol. 32, no. 1, pp. 116–116, Jan. 2015.

[3] S. Newman, *Building Microservices*. "O'Reilly Media, Inc.," 2015.

[4] A. Botta, W. de Donato, V. Persico, and A. Pescapé, "Integration of cloud computing and Internet of Things: A survey," *Futur. Gener. Comput. Syst.*, vol. 56, pp. 684–700, Oct. 2015.

[5] B. Fitzgerald, N. Forsgren, K.-J. Stol, J. Humble, and B. Doody, "Infrastructure Is Software Too!," *SSRN Electron. J.*, Oct. 2015.

[6] D. Petcu, "Multi-Cloud: expectations and current approaches," *Proc. 2013 Int. Work. Multi-cloud Appl. Fed. clouds - MultiCloud '13*, pp. 1–6, 2013.

[7] K. Kritikos and D. Plexousakis, "Multi-cloud Application Design through Cloud Service Composition," in *2015 IEEE 8th International Conference on Cloud Computing*, 2015, pp. 686–693.

[8] B. Di Martino, G. Cretella, and A. Esposito, *Cloud Portability and Interoperability*. Cham: Springer International Publishing, 2015.

[9] R. Moreno-Vozmediano and I. M. Llorente, "IaaS Cloud Architecture: From Virtualized Datacenters to Federated

Cloud Infrastructures," *Computer (Long. Beach. Calif).*, vol. 45, no. 12, pp. 65–72, Dec. 2012.

[10] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "OpenStack: Toward an Open-source Solution for Cloud Computing," *Int. J. Comput. Appl.*, vol. 55, no. 3, pp. 38–42.

[11] D. Davis and G. Pilz., "Cloud Infrastructure Management Interface (CIMI) Model and REST Interface over HTTP," vol. DSP-0263, no. May, 2012.

[12] A. Edmonds, T. Metsch, A. Papaspyrou, and A. Richardson, "Toward an Open Cloud Standard," *IEEE Internet Comput.*, vol. 16, no. 4, pp. 15–25, Jul. 2012.

[13] S. J. Ortiz, "The Problem with Cloud-Computing Standardization," *Computer (Long. Beach. Calif).*, vol. 44, no. 7, pp. 13–16, 2011.

[14] S. T. Graham and X. Liu, "Critical Evaluation on jClouds and Cloudify Abstract APIs against EC2, Azure and HP-Cloud," in *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, 2014, pp. 510–515.

[15] R. Prodan and S. Ostermann, "A survey and taxonomy of infrastructure as a service and web hosting cloud providers," in *2009 10th IEEE/ACM International Conference on Grid Computing*, 2009, pp. 17–25.

[16] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," in *Future of Software Engineering (FOSE '07)*, 2007, pp. 85–103.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: elements of reusable object-oriented software," Jan. 1995.

[18] D. Merkel, "Docker: lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, Mar. 2014.