LambdaLink: an Operation Management Platform for Multi-Cloud Environments

Kate Keahey
Argonne National Laboratory
Argonne, Illinois, USA
keahey@mcs.anl.gov

Pierre Riteau
University of Chicago
Chicago, Illinois, USA
priteau@uchicago.edu

Nicholas P. Timkovich
University of Chicago
Chicago, Illinois, USA
npt@uchicago.edu

ABSTRACT

The last several years have seen an unprecedented growth in data availability, with dynamic data streams from sources ranging from social networks to small, inexpensive sensing devices. This new data availability creates an opportunity, especially in geospatial data science where this new, dynamic, data allows novel insight into phenomena ranging from environmental to social sciences. Much work has focused on creating venues or portals for publishing and accessing such dynamic datasets. However access to data in itself is not sufficient—to turn data into information the data needs to be filtered, correlated, and otherwise analyzed using methods that are dynamically developed and constantly improved by a distributed community of experts. Further, these methods are increasingly required to deliver results with specific qualities of service, e.g., providing results by a certain deadline or ensuring a certain accuracy of the results. Delivering such qualities of service requires generic but often sophisticated tools managing the execution of operations and ensuring their correctness. This paper presents LambdaLink, an operation management platform for multi-cloud environments, and explains how it supports the structured contribution and repeatable, time-controlled execution of operations. We describe the architecture and implementation of LambdaLink, its approach to appliance management in a multi-cloud context, and compare it with related systems.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**;

KEYWORDS

Operation Management Platform; Cloud computing; Lambda; Repeatable Execution

1 INTRODUCTION

The last several years have seen an unprecedented growth in data availability with dynamic data streams from sources ranging from social networks to small, inexpensive sensing devices. The latter in particular is increasingly creating new sources of information: the proliferation of energy-efficient, cheap, and robust sensors, sometimes referred to as *second Moore's law*, is now creating new opportunities for measuring various physical, chemical, and biological characteristics of the environment. As small, specialized sensor devices, capable of both reporting on environmental factors and interacting with the environment, become more ubiquitous, reliable, and cheap, increasingly more domain sciences are creating instruments, composed of dynamic groups of sensors whose outputs are capable of being aggregated and correlated to answer new questions. This new data availability creates an opportunity, especially in geospatial data science where this new, dynamic, data allows unprecedented insight into phenomena ranging from environmental to social sciences.

Much work has focused on creating venues or portals for publishing and accessing such dynamic datasets. However access to data in itself is not sufficient—to turn data into information the data needs to be filtered, correlated, and otherwise analyzed using methods that are dynamically developed and constantly improved by a distributed community of experts. Further, the methods are often used to generate results with specific qualities of service, e.g., providing results by a certain deadline or ensuring a certain accuracy of the results. Delivering such qualities of service requires generic but often sophisticated tools managing the execution of such methods and ensuring their correctness.

Thus we propose to extend the concept of a dynamic data portal to a portal supporting the structured contribution and repeatable, time-controlled execution of operations (sometimes referred to as *lambdas*) as well as access to data. In particular, the proposed platform should support the following capabilities:

- *Operation Publishing.* It allows contributing users to easily publish new operations in such a way that they can be automatically executed by others without the need to understand any of their implementation dependencies or other details. Such operations should become referencable objects, i.e., they should be capable of being published via a Digital Object Identifier (DOI) and easily reenacted by others.
- *Versioning and Repeatability.* The platform should support users in repeating the execution of operations under the same condition as the original, i.e., should manage operation versions and record and provide sufficient information about the condition of the original execution for the user to recreate those conditions.
- *Time-Controlled Execution.* The architecture should support mechanisms for demand-based integration of resources to trade-off quality of service considerations such as response time, accuracy of results, and cost.
- *Variety of Platforms.* The solution should integrate support for multiple commercial and academic platforms including

UCC '17, December 5–8, 2017, Austin, TX, USA

ACM ISBN 978-1-4503-5149-2/17/12…$15.00
https://doi.org/10.1145/3147213.3147224

academic clouds such as Jetstream [42] and Chameleon [44], commercial platforms such as Amazon Web Services [3] or Azure [29], as well as Grid resources such as XSEDE [46] or OSG [33].

In this paper we present LambdaLink, an operation management platform and demonstrate how it fulfills the objectives stated above. Our focus is on developing the abstractions as well as design and interfaces for the system and demonstrating how existing technologies can be integrated to implement it. Since the questions of dynamic scaling by integrating on-demand resources [25, 38] as well as job execution management on remote resources [10] have been investigated in the context of other research, here we focus on mechanisms to bring those results together in a platform that supports repeatable execution.

Our paper is organized as follows. In Section 2, we demonstrate that the concept of appliance underlying our approach can be consistently and cost-effectively implemented across different types of platforms. In Section 3, we build on this concept to present the LambdaLink architecture, discuss the operation publication process it supports as well as its implementation. In Section 4, we discuss the architecture in the context of our goals stated above. We present related work in Section 5 and conclude in Section 6.

2 APPLIANCE AS ABSTRACTION

Our approach relies on the concept of an appliance [39]]: a complete and actionable representation of a user's environment. An appliance is capable of packaging all the software dependencies of a user's program—from operating system, through libraries and tools, to environment variables—in such a way that it is easy for the user to manipulate. In this section, we seek to explain how the concept of an appliance fulfilling our requirements above—whether representing an individual deployment or a cluster with complex relationships—can be supported by integrating existing tools.

Popular implementations of appliances include virtual machines such as KVM [23] and Xen [7], containers such as Docker [28], Shifter [8], or Singularity [24], or bare metal images such as those supported by Chameleon [44]. Most of those implementations consist of disk images that can be deployed to create an instance, i.e., an interactive environment based on a given appliance deployed onto a specific resource allocation. Once an appliance is deployed, a user can log into the instance interactively, modify it, and snapshot it (i.e., save the new disk image), thus creating a new appliance.

Most targeted cloud platforms, such as Amazon EC2 [3], Jetstream [42], or Chameleon [44] have specific requirements for the format of a disk image (e.g., raw or QCOW2 [35]), its disk layout (e.g., whole disk image or partition image), or the environment included in the image (e.g., cloud-init required to be installed and run on boot for injecting SSH keys, a DHCP client configured on specific interfaces, etc.) making images incompatible between various providers. Cloud users typically address this problem by deploying, manually customizing, and then snapshotting images for each platform. However, this approach is not sustainable: it is not only hard to automate (and thus costly) but also prone to errors that may result in an environment that is not consistent between platforms. This poses further questions in the light of our objectives to create an appliance compatible across a range of academic and commercial platforms: how can we generate appliance disk images for all those platforms in such a way that they are consistent, i.e., reflect the same properties for each platform? Will it be possible to maintain (i.e., update or upgrade) such appliances cost-effectively in practice without impacting consistency?

We investigated and compared two approaches in this space: (1) offline creation of images compatible with the requirements of the cloud platform, and (2) online customization and snapshotting. Each method starts with a base image (e.g., a bare-bones operating system installation, sometimes called JeOS for *Just Enough Operating System*), which can be created from scratch by populating its content using an operating system installation procedure, or are produced by some Linux distributions.

The first approach is exemplified by diskimage-builder [12], a tool for automatically building customized operating system images for use in OpenStack [32] clouds (both KVM and bare metal). It takes as input a set of elements, describing which disk image to use as base image, which image format to use, and which customizations to apply to this image (each customization element is a script, usually written in shell). It extracts the file system hierarchy from the base image, applies customizations to it using chroot, and from it creates a disk image in the intended format. Diskimage-builder can be run on any machine that has the required dependencies (libguestfs, QEMU image tools, etc.). The resulting disk image must then be uploaded to the target cloud platform(s).

The second approach relies on snapshotting capabilities of cloud platforms and is an automation of the *deploy, customize, and snapshot* approach described above. Its implementation is exemplified by the Packer [19] tool. Packer takes as input a configuration describing which cloud platform to use and how to access it (including credentials), which disk image to run, and how to customize it. Packer then performs all the steps required to generate the image which include: generating a dedicated SSH key, launching an instance on the cloud platform such that it is accessible with the generated SSH key, and applying customization steps defined by the user (e.g., via shell scripts or configuration management systems such as Puppet [34], Chef [9], or Ansible [5]). When all steps are successfully applied, Packer snapshots the instance on the cloud platform. The tool is supported for a wide range of platforms, in particular AWS [3], virtualized OpenStack clouds including Rackspace Cloud Servers [37], Google Compute Engine [17], and Azure [29].

To compare, the offline approach offers more control over the exact type of image being generated (e.g. whole disk image vs partition image, raw vs QCOW2), but requires knowledge of the format supported by the targeted cloud platforms. For example, while diskimage-builder may be able to create images for any OpenStack cloud based on KVM or bare metal, it does not provide out of the box support for other commercial clouds such as Amazon EC2. The online approach is directly compatible with each cloud platform, but it requires support for a call to the cloud platform's snapshotting API method which is not always available (e.g., OpenStack does not support snapshotting for bare metal deployments out of the box). In addition, the online approach requires credentials and credits for each target cloud platform so that an image can be uploaded, deployed, and later stored there—the offline method does not require uploading the image to the platform until it is actually

needed. Thus the two methods represent different trade-offs and provide coverage for different types of platforms.

To achieve the most complete coverage of platforms we have combined both approaches by first establishing a library of base images for a range of targeted platforms (i.e., a family of OS images, such as CentOS 7, or more specific, e.g. Ubuntu 16.04.3) that can be used as base for either method. For each image we then define what customizations should be applied (in our case usually Bash shell scripts, though other methods can also be used). The system works by selecting a base image for the right method and delegating to tools implementing either the online or offline approach as appropriate. Further, some appliances may use a modification of the *deploy, customize, and snapshot* method that omits the *snapshot* step. In this case, configuration is always redone when the image is deployed. This leads to long deployment times and is often unreliable as the installation process may access remote repositories that are not always available. For this reason, we decided agains using this process in our reference implementation.

An additional challenge is defined by the need to represent within the system platforms that are not appliance-based, i.e., do not let users deploy environments and instead rely primarily on fixed environments pre-configured on various sites, such as XSEDE [46]. While we cannot influence the configuration of those sites, we can represent their configurations as an appliance giving users the option to run at scale on one of the XSEDE sites—but also use appliance-based platforms as needed, e.g., when XSEDE resources are not available or—using an XSEDE appliance with slight modifications—to implement and debug their applications within an environment that allows for a hightened level of privilege such as e.g., superuser access.

Finally, the last challenge consists of deploying what we call *complex appliances*—appliances typically deployed as multiple instances implementing complex relationships—such as a virtual cluster (e.g., a Torque [1] cluster with potentially multiple specialized management nodes and a set of worker nodes) or a cloud deployment (such as e.g., OpenStack or Hadoop [6] deployments). In addition to the disk image, such deployments require integrating on deployment additional information e.g., exchanging security information (based on keys generated at deployment time) or configuration information (IP addresses generated at deployment time)—a process called contextualization [22]—and potentially recontextualizing [27] dynamically as new nodes are added to a virtual cluster or a cloud. These capabilities are supported by orchestration services that typically use an image and a template defining how the images are deployed and the information is exchanged among them. These services are implemented by Heat [20] for OpenStack, Cloud Formation [2] for AWS, Google Cloud Deployment Manager [16] for Google Compute Engine, etc., giving us a coverage of the platforms of interest.

3 LAMBDALINK ARCHITECTURE

In the previous section, we demonstrated the properties of appliances in LambdaLink. This section describes the architecture of the system.

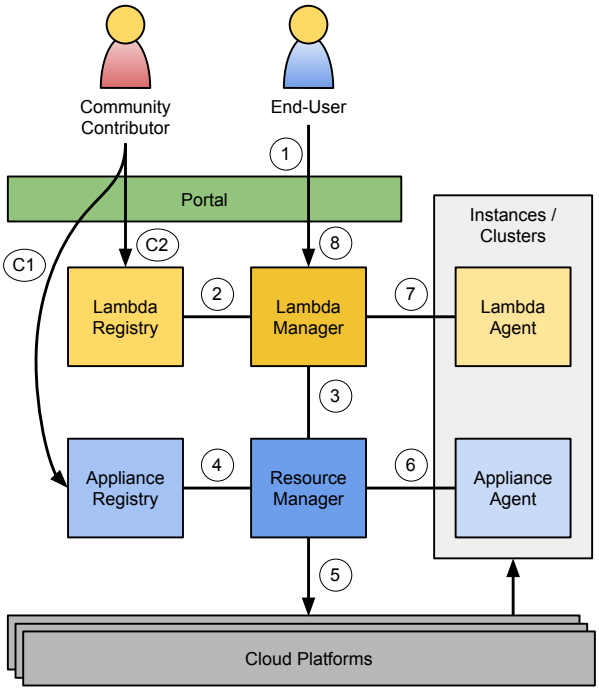

Figure 1: Diagram of the LambdaLink architecture

3.1 Critical Components

Figure 1 shows an outline of the LambdaLink architecture. The architecture is composed of the following components:

- A **Portal**, or another means for users to communicate with the system, which allows users to request the execution of specific operations on specific data. It also manages information relevant to users (e.g., credential information for multiple cloud services).
- The **Appliance Registry**, which stores appliances as well as the corresponding appliance implementations required to deploy them on different cloud platforms. To support repeatability at the level of environments, updates to appliances as well as appliance implementations are tracked using version numbers.
- The **Resource Manager** manages appliance deployments. It is in charge of choosing the best option between using an existing appliance instance (if available), expanding the resource allocation for one, or creating a new one on a new resource allocation, as needed to manage the overall response time. The Resource Manager uses the information about appliances in the Appliance Registry to deploy appliances on allocated resources. It then uses interfaces to multiple clouds to deploy the appliance, leveraging or integrating with orchestration mechanisms to create complex appliances as needed.
- The **Appliance Agent** carries out functions within the appliance, such as credential management or monitoring, on behalf of the Resource Manager.

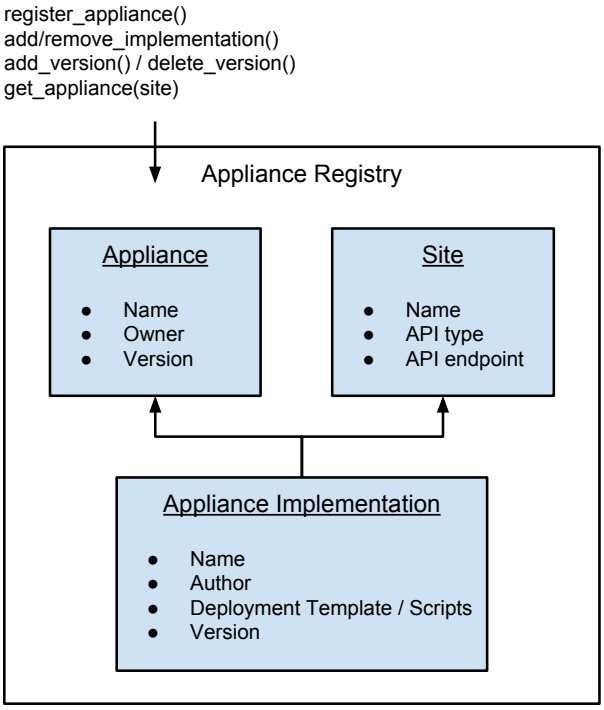

Figure 2: Appliance Registry interfaces

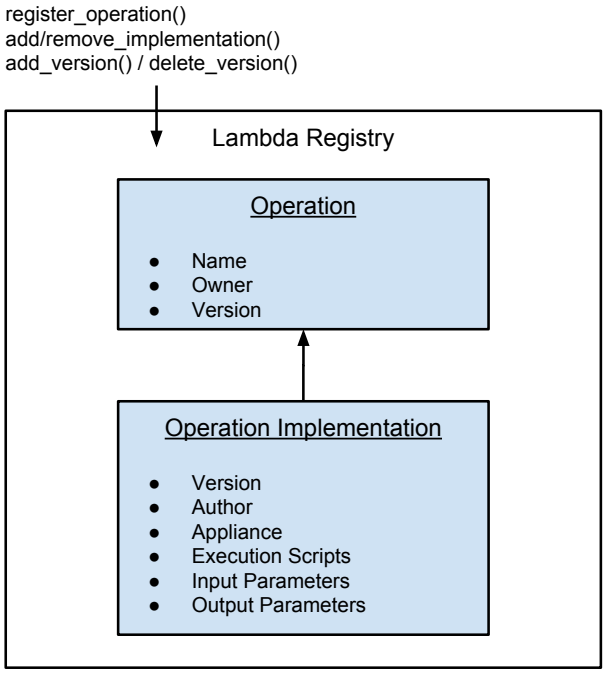

Figure 3: Lambda Registry interfaces

- The **Lambda Registry**, which stores information about operations in the form of scripts or execution recipes (e.g., how to execute them, what appliance they need, pre- and post-execution actions, etc.) that allow the Lambda Manager to execute them automatically. The operations are also tracked at the level of version numbers as multiple operation versions can map to the same appliance version.
- The **Lambda Manager**, which takes user requests for the execution of specific operations and executes them based on information provided by Lambda Registry. The Lambda Manager also works with the Resource Manager to ensure the availability of the required environment on the right amount of resources, possibly triggering the deployment of a new appliance.
- The **Lambda Agent** carries out functions within the appliance, such as starting a job locally and monitoring and reporting on its progress, on behalf of the Lambda Manager.

The system operates on virtual data, i.e., data that can be globally identified and efficiently managed based on global identifier, creating local copies as needed, such as implemented by Chimera [14].

3.2 User Workflows

The system assumes two types of users: (a) a community contributor, who contributes operations/lambdas to the system via the portal, and (b) an end-user, who uses those operations. Below we describe only the interfaces opened to community contributors and end users.

Community Contributor Workflow. To contribute an operation to the system, the user takes the following steps:

- The user conceptualizes a new appliance for the contributed operation and uses methods described in Section 2 to configure one or more implementation of this appliance for a set of resource providers. The user then tests these images to verify that they support the execution of the required operation across appliance implementations, and support other properties of the appliance.
- Once the appliance configuration step is complete, the user first adds an appliance entry/record to the Appliance Registry (Figure 1, step C1) as well as implementation records for each cloud provider using interfaces shown in Figure 2.
- After the appliance entry is created, the user proceeds to define a lambda/operation entry in the Lambda Registry (Figure 1, step C2) using interfaces shown in Figure 3.
- The user then defines a new operation implementation entry for a specific operation. The operation implementation record references the appliance required for its execution, and may consist of the command to execute the process, any pre- and post- processing commands, as well as links to virtual data representing input and output parameters.

End-User Workflow. The execution of operations/lambdas is triggered by the end-user and unfolds in the steps described below. We assume that the end-user has an account with the portal and that the account is associated with a set of credentials associated

with resources that the user wants to use; these can be either provided by the portal administrator or constitute (potentially partially delegated) user's credentials.

Invoking an operation triggers the following steps:

- The user logs into the portal and browses through a list of contributed operations, their versions and descriptions, and requests the application of a specific operation to specific data (Figure 1, step 1).
- After the user's request is conveyed to the Lambda Manager, the latter takes the following actions: a Lambda Instance record providing information about the executing operation is created (it will be updated with relevant information throughout operation execution); the Lambda Manager retrieves the information about how to run the operation from the Lambda Registry (Figure 1, step 2), and asks the Resource Manager to select a suitable appliance for its execution (Figure 1, step 3).
- The Resource Manager first checks if the appliance has been deployed within the system and takes the following actions:
  - If no appliance available to the user (via such mechanisms as e.g., membership in the same project) has been deployed in the system, the Resource Manager first selects a platform consistent with the user's credentials as present in the system. Users may provide hints on resource preferences, though the actual decision takes into account factors such as resource availability, the availability of an appliance implementation for the specific resource, and cost. Once a platform is selected, the Resource Manager uses information about the appliance implementation retrieved from the Appliance Registry (Figure 1, step 4) and deploys it on the platform (Figure 1, step 5), also creating an Appliance Instance record with relevant information. The Resource Manager then authenticates and interacts with the Appliance Agent to create an account associated with the requesting user (Figure 1, step 6).
  - If an appliance has been deployed in the system, but needs to be modified in some way, e.g., the user's account does not exist, or the resource allocation is insufficient to accommodate the user's request, the Resource Manager takes the appropriate action. In the former case, it asks the Appliance Agent to create an account for the user; the agent then securely returns the authentication token for it. If the appliance lacks resources to provide the desired response time, has been deployed but the resource allocation associated with the instance is insufficient to accommodate the user's request, the Resource Manager scales the instance up or out, following scaling strategies such as described in [38]. The allocations are monitored and scaled down as needed.
  - If the appliance has been deployed and fulfills the requirements, it is a no-op and the Resource Manager simply returns the resource record.

  In all cases, the Resource Manager returns a resource record containing the IP address of the appliance instance and the user's authentication token for this appliance deployment.
- Once the appliance is available, the Lambda Manager prepares for deployment, based on the requirements described in the operation record retrieved from the Lambda Registry, and launches the execution of the operation by interacting with the Lambda Agent running on the node (or the master node of a cluster) (Figure 1, step 7).
- Once the execution of the operation is finished, the Lambda Agent notifies the Lambda Manager. The Lambda Manager then orchestrates post-processing as per the operation record retrieved from the Lambda Registry and ultimately notifies the user (Figure 1, step 8).

3.3 Implementation

We have prototyped the LambdaLink architecture in the following reference implementation. Components of LambdaLink are implemented independently as microservices with HTTP REST APIs. Each service is written in Python, using Django for HTTP, object-relational mapping, and authentication, then Django REST Framework to provide the API. Persistent data is stored in a MySQL database.

Requests made to services are authenticated by custom Django middleware that validates a token. For external requests by a user, the token is provided by an authentication service, and for inter-service requests, a generated token is signed with a shared key. After authentication, authorization is validated, then the request is handled.

Endpoints that do not represent instantiated resources, for example, the appliance definition and implementation, have a simple CRUD API to modify properties of the objects. For endpoints reliant on external resources accessed in an asynchronous manner, the two Managers use additional processes running a Celery task queue [40] to make requests and poll for expected changes. For example, when the Resource Manager determines that it needs to launch an appliance on a cloud provider, it creates an object and immediately returns it to the user marked in a *pending* state. A task is created to command the provider to start creation, then to check for provisioning to finish, and finally to check if the Appliance Agent is contactable.

Our refernce implementation supports OpenStack clouds, such the ROGER OpenStack cloud at NCSA [31] or the Chameleon [44] KVM and bare-metal OpenStack deployments, with Heat [20] for deploying complex appliances. It has been used to deploy operations of UrbanFlow [41], a geospatial data analysis platform from the CyberGIS center [47] for synthesizing social media fine-resolution data with authoritative urban dataset.

Our deployment model assumes that LambdaLink will be operated as a service, where the service provider provides cycles and storage to manage and store the operations information, appliances, and their implementations (though some implementations may be cached at suitable cloud providers).

4 DISCUSSION AND ANALYSIS

We have defined the LambdaLink architecture, developed a reference implementation, and applied it in the context of CyberGIS computations [47]. The main innovation of our approach is a design separation of the function of resource provisioning/configuring and

job/operation management united in traditional schedulers—and then demonstrating how they can be used together to provide an architecture fulfilling our requirements, in particular generating information for repeatable execution and response time management.

We note that the operation/lambda management in our architecture is similar to mechanisms used in grid computing [10, 48]. The main differences consist in the ability of the Lambda Manager to (1) negotiate with the Resource Manager to provide specific types or amounts of resources and (2) the ability to provide a structured, persistent, and versioned definition of operations in the Lambda Registry. The existing mechanisms could thus be adapted to fit into this architecture with relatively little effort. The former could be implemented by extending them to provide the appliance selection operation (see Figure 1, step 3); the latter by providing an implementation of the Lambda Registry (a relatively lightweight elaboration on already existing mechanisms). For this reason, our reference implementation focuses on the appliance and resource management parts of the architecture, as well as articulating interfaces for better integration of existing methods.

We now turn to analysis of our system in the context of its stated objectives. We have provided a platform that supports contributing/publishing operations (lambdas) as well as their automated execution allowing for implementation of qualities of service. Section 3.2 outlines a contributor's workflow and showed how, based on specific and well-defined artifacts provided by the contributor, we can automate both the automatic execution of the operations and manage quality of service by automatically integrating resources. The ability to package, version, and publish operations in this way makes them not only shareable but also referencable entities that can be easily re-applied by others to different data sets or different problems. Execution based on appliances ensures a smoother experience which assures that the dependencies of a specific applications are met.

The ability to faithfully repeat an execution of a specific program operating on a specific dataset usually depends on two factors: the ability to execute on the exact same hardware, and the ability to recreate on this hardware the exact same environment used in the original execution. Though the first repeatability factor is sometimes downplayed—as not all changes to hardware will affect all executions—changes in hardware configuration and firmware upgrades can have a noticeable effect on the results injecting non-trivial inconsistencies into the results that only repeating an experiment in the exact same conditions can resolve. Two factors need to be present to resolve it: a record of the exact resources used, and versioning of those resources to describe changes that happened to them in the intervening time between executions. Having this information means that even if it is not economically possible to roll back the changes or restore decommissioned hardware based on those records, differences can still be reasoned about. The availabililty of this information is currently dependent on the provider; while commercial platforms provide little information in this space, both record of used resources and resource versioning are currently supported by the Chameleon platform [45] and the developed methods are published and shareable by other platforms. Versioning of appliances in the LambdaLink architecture allows us to point to the exact environment used for a specific execution. Since associating operations with environments is performed by the system we can manage and export exact records of how specific data was produced. Combining these two factors allows for exact reenactment of a specific run (currently not automated in the architecture though the relevant information is available).

Many applications and science portals have successfully managed to leverage on-demand cloud resources to adapt to a varying number of users/requests with varying workloads in order to provide a predictable response time for all requests [26, 36, 38]. Although integrating resources dynamically into an ongoing computation has proven effective in the case of e.g., high throughput computing (HTC) workloads [30], it is significantly more challenging for applications that have to manage a dynamic configuration, such as data distribution targeting a fixed number of nodes. Specifically, in the case of dynamically scaling Hadoop applications—used in many geospatial computations—the overhead of making the application aware of additional resources can incur more cost than it brings benefit if not done carefully. Thus, while we have proved that our approach will work for certain types of applications [41], we are currently investigating the boundaries of dynamic scaling of Hadoop and strategies for management of Hadoop workloads. Recognizing which applications are capable of consuming the additional resources and thus will benefit by their inclusion will ultimately form a part of the negotiation process with the user as described in [4].

The ability to support a range of platforms, commercial and academic alike, is dependent on two factors. The first one is implementing the appliance abstraction, i.e., developing models for generating and cost-effectively maintaining a set of appliance implementations (i.e., images) that is consistent across a set of those platforms; their advantages and limitations were described in Section 2. While we of course cannot deploy appliances on platforms that do not support this functionality (such as e.g., platforms providing an interface to batch-scheduled workloads), we can provide a one-way bridge allowing the users of those platforms to move to LambdaLink by configuring appliances with corresponding configuration. The increased interest in adopting container solutions such as Singularity [24] or Docker [28] in scientific platforms is likely to improve the situation on this front in the future. The second factor is the ability to adapt the Resource Manager to interface with and leverage a set of platform-specific tools to implement basic functions such as monitoring or deployment of complex appliances; this is currently well supported by tools [11] such as Apache Libcloud and Apache jclouds, and likely to develop in the future as more systems are interested in reaching out to multiple platforms.

5 RELATED WORK

Science gateways [21, 49] are a popular way to share catalogs of applications and services among a large scientific community. However, these gateways are generally linked to specific execution platforms, with their use in multi-cloud environments only explored recently. Farkas et al. [13] and Gugnani et al. [18] extend the WS-PGRADE/gUSE workflow-oriented science gateway with the CloudBroker Platform to support execution on multiple cloud platforms. However, neither of these systems deal with repeatable

execution of operations, which LambdaLink handles by integrating versioning into its registries.

Scientific workflow management tools [50] is another type of systems that can include comparable capabilities to LambdaLink. The main difference is that they focus on executing workflows of inter-dependent tasks, while LambdaLink operations are not tied to this concept: for example, an operation could be the deployment of a virtual cluster based on a complex appliance, providing a long-lived service to a community. Among these workflow management tools, the AWE/Shock ecosystem for bioinformatic workflow applications [43] is extended by Skyport [15] to use Linux container virtualization technologies (namely, Docker [28]) to handle software deployment across various cloud platforms. In comparison, the implementation of LambdaLink natively supports image-based deployments (either virtual machines or bare metal), but could also support Docker containers.

6 CONCLUSIONS AND SUMMARY

The unprecedented growth in data availability—with dynamic data streams from sources ranging from social networks to small, inexpensive sensing devices—creates an opportunity, especially in geospatial data science where this new, dynamic, data allows new insight into phenomena ranging from environmental to social sciences. While much work has focused on creating venues or portals for publishing and accessing such dynamic datasets, access to data in itself is not sufficient: data needs to be filtered, correlated, and otherwise analyzed using methods that are dynamically developed and constantly improved by a distributed community of experts.

In this paper, we have presented LambdaLink, an operation management platform for multi-cloud environments. Its architecture separates the management of appliances and operations/lambdas and fulfills the needs of two categories of users: community contributors, who create and share appliances and operations, and end-users, who run these operations on a variety of cloud platforms. We discussed the two main approaches available for appliance management in cloud systems and how they can both be leveraged by LambdaLink.

In future works, we plan to explore more deeply the integration of data management systems and protocols with LambdaLink, as well as advanced policies for dynamic scaling. In particular, we are currently investigating the boundaries of dynamic scaling of Hadoop and strategies for management of Hadoop workloads, with the aim of integrating the resulting algorithms and policies in LambdaLink.

7 ACKNOWLEDGEMENTS

This material was supported by the National Science Foudation grant 1443080, and, in part, by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. Work presented in this paper was obtained using the Chameleon testbed supported by the National Science Foundation.

REFERENCES

[1] Adaptive Computing. 2017. TORQUE Resource Manager. http://www.adaptivecomputing.com/products/open-source/torque/. (2017). [Online; accessed 15-Aug-2017].
[2] Amazon Web Services. 2017. AWS CloudFormation. https://aws.amazon.com/cloudformation/. (2017). [Online; accessed 15-Aug-2017].
[3] Amazon Web Services. 2017. Elastic Compute Cloud (EC2). https://aws.amazon.com/ec2/. (2017). [Online; accessed 15-Aug-2017].
[4] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. 2007. Web services agreement specification (WS-Agreement). In *Open Grid Forum*, Vol. 128. 216.
[5] Ansible HQ. 2017. Ansible. https://www.ansible.com. (2017). [Online; accessed 15-Aug-2017].
[6] Apache Hadoop contributors. 2017. Apache Hadoop. http://hadoop.apache.org. (2017). [Online; accessed 15-Aug-2017].
[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, Vol. 37. 164–177.
[8] Richard Shane Canon and Doug Jacobsen. 2016. Shifter: Containers for HPC. In *Cray Users Group Conference (CUG'16)*.
[9] Chef. 2017. Chef. https://www.chef.io/chef/. (2017). [Online; accessed 15-Aug-2017].
[10] Karl Czajkowski, Ian Foster, Nicholas Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. 1998. A Resource Management Architecture for Metacomputing Systems. In *Workshop on Job Scheduling Strategies for Parallel Processing*. 62–82.
[11] Beniamino Di Martino, Giuseppina Cretella, and Antonio Esposito. 2015. Cross-platform cloud APIs. In *Cloud Portability and Interoperability*. 45–57.
[12] Diskimage-builder contributors. 2017. Diskimage-builder Documentation. https://docs.openstack.org/developer/diskimage-builder/. (2017). [Online; accessed 15-Aug-2017].
[13] Zoltán Farkas, Péter Kacsuk, and Ákos Hajnal. 2016. Enabling Workflow-Oriented Science Gateways to Access Multi-Cloud Systems. *Journal of Grid Computing* 14, 4 (2016), 619–640.
[14] Ian Foster, Jens Vöckler, Michael Wilde, and Yong Zhao. 2002. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*. 37–46.
[15] Wolfgang Gerlach, Wei Tang, Kevin Keegan, Travis Harrison, Andreas Wilke, Jared Bischof, Mark D'Souza, Scott Devoid, Daniel Murphy-Olson, Narayan Desai, et al. 2014. Skyport: container-based execution environment management for multi-cloud scientific workflows. In *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds*. 25–32.
[16] Google Cloud Platform. 2017. Google Cloud Deployment Manager. https://cloud.google.com/deployment-manager/. (2017). [Online; accessed 15-Aug-2017].
[17] Google Cloud Platform. 2017. Google Compute Engine - IaaS. https://cloud.google.com/compute/. (2017). [Online; accessed 15-Aug-2017].
[18] Shashank Gugnani, Carlos Blanco, Tamas Kiss, and Gabor Terstyanszky. 2016. Extending Science Gateway Frameworks to Support Big Data Applications in the Cloud. *Journal of Grid Computing* 14, 4 (2016), 589–601.
[19] HashiCorp. 2017. Packer. https://www.packer.io. (2017). [Online; accessed 15-Aug-2017].
[20] Heat contributors. 2017. Welcome to the Heat documentation! — heat documentation. https://docs.openstack.org/heat/. (2017). [Online; accessed 15-Aug-2017].
[21] Péter Kacsuk. 2014. *Science Gateways for Distributed Computing Infrastructures: Development framework and exploitation by scientific user communities*. Springer.
[22] Kate Keahey and Tim Freeman. 2008. Contextualization: Providing One-Click Virtual Clusters. In *2008 IEEE Fourth International Conference on eScience*. 301–308.
[23] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Linux Symposium*. 225–230.
[24] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 2017. Singularity: Scientific containers for mobility of compute. *PLOS ONE* 12, 5 (05 2017), 1–20.
[25] Ming Mao, Jie Li, and Marty Humphrey. 2010. Cloud auto-scaling with deadline and budget constraints. In *11th IEEE/ACM International Conference on Grid Computing (GRID 2010)*. 41–48.
[26] Paul Marshall, Kate Keahey, and Tim Freeman. 2010. Elastic site: Using clouds to elastically extend site resources. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid 2010)*. 43–52.
[27] Paul Marshall, Henry M Tufo, Kate Keahey, David LaBissoniere, and Matthew Woitaszek. 2012. Architecting a Large-scale Elastic Environment: Recontextualization and Adaptive Cloud Services for Scientific Computing.. In *ICSOFT*. 409–418.
[28] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* 2014, 239 (2014), 2.
[29] Microsoft Azure. 2017. Virtual machines – Linux and Azure virtual machines. https://azure.microsoft.com/services/virtual-machines/. (2017). [Online; accessed 15-Aug-2017].
[30] Ruben S Montero, Rafael Moreno-Vozmediano, and Ignacio M Llorente. 2011. An elasticity model for high throughput computing clusters. *J. Parallel and Distrib. Comput.* 71, 6 (2011), 750–757.

[31] NCSA. 2017. ROGER: The CyberGIS Supercomputer. https://wiki.ncsa.illinois.edu/display/ROGER. (2017). [Online; accessed 15-Aug-2017].

[32] OpenStack contributors. 2017. OpenStack Open Source Cloud Computing Software. https://www.openstack.org. (2017). [Online; accessed 15-Aug-2017].

[33] Ruth Pordes, Don Petravick, Bill Kramer, Doug Olson, Miron Livny, Alain Roy, Paul Avery, Kent Blackburn, Torre Wenaus, Frank Würthwein, et al. 2007. The Open Science Grid. In *Journal of Physics: Conference Series*, Vol. 78. IOP Publishing.

[34] Puppet. 2017. Puppet. https://puppet.com. (2017). [Online; accessed 15-Aug-2017].

[35] QEMU contributors. 2017. QCOW2. http://bit.ly/qcow2. (2017). [Online; accessed 15-Aug-2017].

[36] Andres Quiroz, Hyunjoo Kim, Manish Parashar, Nathan Gnanasambandam, and Naveen Sharma. 2009. Towards autonomic workload provisioning for enterprise grids and clouds. In *10th IEEE/ACM International Conference on Grid Computing (GRID 2009)*. 50–57.

[37] Rackspace. 2017. Virtual Cloud Servers Powered by OpenStack. https://www.rackspace.com/cloud/servers. (2017). [Online; accessed 15-Aug-2017].

[38] Pierre Riteau, Myunghwa Hwang, Anand Padmanabhan, Yizhao Gao, Yan Liu, Kate Keahey, and Shaowen Wang. 2014. A Cloud Computing Approach to On-demand and Scalable Cybergis Analytics. In *Proceedings of the 5th ACM Workshop on Scientific Cloud Computing (ScienceCloud '14)*. 17–24.

[39] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. 2003. Virtual Appliances for Deploying and Maintaining Software. In *Proceedings of the 17th USENIX Conference on System Administration (LISA '03)*. 181–194.

[40] Ask Solem et al. 2017. Celery: Distributed Task Queue. http://www.celeryproject.org. (2017). [Online; accessed 15-Aug-2017].

[41] Kiumars Soltani, Aiman Soliman, Anand Padmanabhan, and Shaowen Wang. 2016. UrbanFlow: Large-scale Framework to Integrate Social Media and Authoritative Landuse Maps. In *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*. 2.

[42] Craig A. Stewart, Timothy M. Cockerill, Ian Foster, David Hancock, Nirav Merchant, Edwin Skidmore, Daniel Stanzione, James Taylor, Steven Tuecke, George Turner, et al. 2015. Jetstream: A self-provisioned, scalable science and engineering cloud environment. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*. Article 29.

[43] Wei Tang, Jared Wilkening, Narayan Desai, Wolfgang Gerlach, Andreas Wilke, and Folker Meyer. 2013. A scalable data analysis platform for metagenomics. In *Big Data, 2013 IEEE International Conference on*. 21–26.

[44] The Chameleon project. 2017. Chameleon Cloud Homepage. https://www.chameleoncloud.org. (2017). [Online; accessed 15-Aug-2017].

[45] The Chameleon project. 2017. Chameleon Hardware Discovery page. https://www.chameleoncloud.org/hardware/. (2017). [Online; accessed 15-Aug-2017].

[46] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, et al. 2014. XSEDE: Accelerating Scientific Discovery. *Computing in Science & Engineering* 16, 5 (2014), 62–74.

[47] Shaowen Wang. 2010. A CyberGIS Framework for the Synthesis of Cyberinfrastructure, GIS, and Spatial Analysis. *Annals of the Association of American Geographers* 100, 3 (2010), 535–557.

[48] Shaowen Wang, Marc P. Armstrong, Jun Ni, and Yan Liu. 2005. GISolve: A grid-based problem solving environment for computationally intensive geographic information analysis. In *Challenges of Large Applications in Distributed Environments (CLADE 2005)*. 3–12.

[49] Nancy Wilkins-Diehr, Dennis Gannon, Gerhard Klimeck, Scott Oster, and Sudhakar Pamidighantam. 2008. TeraGrid science gateways and their impact on science. *Computer* 41, 11 (2008).

[50] Jia Yu and Rajkumar Buyya. 2005. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing* 3, 3-4 (2005), 171–200.