Pattern-based Deployment Service for Next Generation Clouds

Hongbin Lu, Mark Shtern, Bradley Simmons, Michael Smit, and Marin Litoiu
York University, Canada
{*hongbin,mark*}*@cse.yorku.ca and* {*bsimmons,msmit,mlitoiu*}*@yorku.ca*

***Abstract*—This paper presents a flexible deployment service for cloud computing. The service facilitates the specification and the execution of cloud deployment plans for applications. An application is described through a pattern, an abstract view that captures the logical view of the application and its mapping into cloud resources. The services instantiate the pattern in the cloud and allows for runtime updates of the deployment. The service is accessible through a RESTful interface. We identify the requirements for the service, describe its interfaces and show several case studies that capture the main features of the service.**

***Index Terms*—multi-cloud, system management, application deployment**

I. Introduction

The multi-cloud [1], [2], [3] is a challenging topic in cloud computing research, and foundational concepts like brokers [4], [5], meta-data services [6], elasticity [7], and security [8] are being explored to facilitate its realization. Underpinning the concept of a multi-cloud is the notion of multiple domains (i.e., multiple cloud providers) in which various clouds (e.g., Amazon EC2, Rackspace, private clouds, ...) are aggregated together to achieve better quality of service or other desirable benefits.

One such model is being actively contemplated by a research project in Canada. The Natural Sciences and Engineering Research Council of Canada (NSERC) Strategic Network for Smart Applications on Virtual Infrastructure (SAVI) [9] is a national research project involving academia and industry that envisions a novel, hierarchical structure for future clouds composed of two key cloud types: *smart edges* and *core*. An edge represent a small cloud (in terms of numbers of machines and lower power consumption) and exists in close proximity to the end-user. An edge is connected by a fast pipe to the core (a large cloud much like today's public providers) and acts to provide a high-bandwidth, low latency point of service to users. This arrangement allows computation tasks to move between the edge and the core depending on the availability of resources, the characteristics of the application, and the needs of the user. This two-tiered model is expected to encourage the development of applications not currently possible with existing infrastructure.

A key functional requirement of SAVI is the ability to deploy future applications to the core and edges in an intelligent manner. This deployment should be dynamic in nature and alterable at runtime (to add/remove resources in response to workload or to move components between the edges and the core). This challenging task requires consideration of how an application might be split over two cloud locations (see for example the automated partitioning of an application to span multiple clouds [10]) and how this split application – or any application – might be deployed. In this paper we introduce a pattern-based deployment service (PDS) to automate the deployment of applications to the multi-cloud.

This novel deployment service integrates the concepts of system patterns [11], [12], multi-clouds and various industrial best practices (e.g., configuration management tools, RESTful services, etc.). The key design requirements were (a) to facilitate application deployments across the SAVI test-bed (i.e., on the SAVI cloud composed of a core and multiple edges running OpenStack) and (b) to enable dynamic adaptation through different management strategies (e.g centralized, decentralized, etc.). Automated deployment represents a critical capability in the context of the SAVI project in order to explore adaptive management [13] of applications on a two-tiered cloud. Beyond the functionality required for SAVI, the PDS is also expected to automate the deployment of applications across standard multi-clouds (e.g., OpenStack, Amazon EC2).

The inherent complexity associated with describing both a complex application topology and its respective deployment plan led us to abstract these concepts by way of a *system pattern*. This pattern is expressed using an expressive, declarative, XML-based domain specific language (DSL) that facilitates easy representation of complex deployment concepts, entities, relationships and services. Based on this representation, the PDS is able to dynamically construct a cross-cloud deployment including the acquisition and configuration of resources.

The remainder of this paper is structured as follows: we start with identifying the requirements for the deployment service in §II; in §III we present the service architecture and interfaces; the implementation and usage scenarios are described in §IV and §V, respectively; a concrete case study is discussed in §VI and conclusions are presented in §VII.

II. Background & Requirements

Interest is growing in deploying applications to cloud resources at all stages of the development lifecycle. Organizations are increasingly adopting *continuous deployment* in which application developers deploy new versions of web applications to production environments on infrastructure-as-a-service. Application testers and quality assurance engineers launch applications on temporary resources acquired from public clouds. Customers are launching new applications to various cloud providers or even private clouds. While a developer can picture how they want the application deployed, how that vision is realized varies based on the target clouds, the expected workload and the properties of the application.

In short, the challenge is translating from a vision of *what* the desired result is into a *workflow* that achieves this vision. The following subsection will provide an overview of this domain. The next subsection will describe our requirements with regards to design of the PDS. The final subsection will assess the gap between existing approaches and the requirements, motivating the design of the PDS.

A. Overview of Deployment Approaches

Traditionally, a workflow is implemented as a script that is dedicated to achieving some defined set of tasks (e.g., deploying a Java EE application to the cloud). Often the script is hard-coded for a specific application and a specific platform. Porting the script to another application / platform / cloud provider requires a significant amount of work and often affects the stability of the system, making migrations impractical and therefore unlikely. The root cause of the problem is that knowledge of the workflow is hidden in a dedicated script that is not easy to understand or reuse.

There are several configuration management tools [14] that help with automating complex deployments to heterogeneous systems (e.g., Chef, Puppet, and CFEngine[1]. These tools employ their own domain specific languages (DSL) which are easy to understand and reuse as they systematize the process of workload creation by applying clear semantics and removing code duplication.

At a higher level of abstraction, service orchestration tools like Juju[2] package multiple workflows together and make them available to users. In fact, both Chef and Juju include community-provided workflows (called *recipes* for Chef, *charms* for Juju); the user must combine these recipes/charms with their own scripted instructions to achieve their desired deployment.

Proprietary multi-cloud (e.g., hybrid cloud, two-tiered cloud) deployment/management services like Rightscale [16] aim to simplify the deployment/management processes for application environments across the multi-cloud by providing both a RESTful API and Web UI to users, hiding the complexity of the automation scripts (i.e, RightScripts).

Model-driven deployment allows a deployer to specify the elements of the desired system [17], which is automatically translated into a workflow behind the scenes that moulds the deployment in the shape specified by the deployer. To address the complexity of deploying complex web applications to the private cloud, Eilam et al. describe a mechanisms for translating *system patterns* into workflows [12], an approach currently used in the IBM PureApplication System[3] and IBM Workload Deployer[4].

B. Requirements

Given the requirements of the SAVI project in particular and the multi-cloud in general, we have defined the following requirements for a deployment service. The PDS was designed to meet these requirements.

Accessible to students and professors: Provisioning applications to complex cloud infrastructure must be accessible several types of users ranging in levels of comfort and expertise with systems, including students and faculty. The goal is that a user with little or no expertise in multi-cloud deployments be given access to a service for which they enter some parameters (e.g., location of a WAR file, etc.) and disregard complex configuration management and the multiple frameworks required to realize this deployment. As many students are present for relatively short periods of time, building expertise and gaining comfort and knowledge with regards to using the platform must be an achievable process. For example, a typical Masters student has up to one year of research time, most of which should be spent on research and not on learning tools.

Simple RESTful API and Web UI: The deployment service must offer a RESTful API offering full programatic access, to enable more advanced user to offer functions such as automation and elastic scaling. At the same time, the Web UI should act as a client for the RESTful API, providing access to documentation and facilitating the exploration and use of service for simple tasks.

Language for expressing patterns: A declarative, XML-based Domain Specific Language (DSL) is specified to simplify the expression of complex deployment structures and functions. The language should allow settings at the node level if needed, but also provide groupings at higher levels of abstraction to remove redundancy, which limits potential editing errors and increases comprehensibility. The use of a pattern language allows for focus to be placed simply on the declaration of what should be achieved by the service rather than on how to achieve it.

Abstract away the complexity of the multi-cloud: The service should allow for seamless deployment of applications across multiple cloud infrastructures. This allows the developer to focus on the application they wish to deploy rather than on the intricacies and complexities associated with multi-cloud configuration management.

Extensible for future research needs: This service is designed to facilitate research in many areas of distributed computing. A key requirement is the ability to add/remove features on the fly, and extensibility. The service should therefore decouple the DSL from the execution of workflows, allowing both to evolve as needed.

Ready to be used for different communities, open source, extensive documentation and easy installation:

Its use by researchers spanning a variety of organizations requires open licensing. Turnover in research staff and students requires the transfer of knowledge through documentation, and easy installation.

Elasticity/ auto-scaling: The service should be able to deploy an application across the multi-cloud/two-tiered cloud and then to dynamically add / remove cloud resources (e.g., nodes) to the deployment as required in accordance with the

[1]Respectively, http://www.opscode.com/chef/, https://puppetlabs.com/, and http://cfengine.com/ [15].

[2]https://juju.ubuntu.com/

[3]http://www.ibm.com/ibm/puresystems/us/en/pf_pureapplication.html

[4]http://www-142.ibm.com/software/products/us/en/workload-deployer/

deployer's specified elasticity policy [7].

Enable centralized, decentralized and hierarchical runtime management: The service should be able to facilitate autonomic management [18] of a deployed application [13], and in particular should integrate with other components and be agnostic to its role in management, supporting centralized, decentralized and hierarchical management approaches.

C. *Challenges with existing approaches*

Given the requirements defined above, existing solutions and configuration management tools are not entirely sufficient; however, they may serve as a base for a more complete service. There are several proprietary offerings that go beyond configuration management and work on the multi-cloud; however, our requirements include free and open-source licensing. We have chosen an open-source configuration management tool, Chef, on which to build our service.

The requirement that the service be usable by researchers at all levels of expertise and with various areas of focus was particularly challenging, as we manage the trade-off of simplicity and flexibility. While we aim for simplicity in the experience of using the service, the actual workflows we seek to run range from non-trivial to very complex. For example, deploying an application to a two-tiered cloud and then dynamically scaling the application's footprint at runtime in response to changes in workload and guided by the user's elasticity policy is a complex process that requires various types of knowledge, libraries, etc. This ever expanding functional complexity prevented us from electing to use some of the more simple configuration management tool alternatives (e.g., Babushka[5]).

In contrast, the turnover of participants on research teams (e.g., students and post-docs) along with the ongoing need to modify and change how various aspects of a system work suggest that the direct use of complex configuration tools alone is not feasible nor prudent. It is easier to extend or modify a meta-layer that operates at a higher level using selected features of an underlying tool, rather than modifying a major configuration management tool that seeks to be as general as possible and offers complex behaviours that may be entirely out of scope for the research project.

In the next section we will introduce the architecture of the PDS, demonstrating how this balance is negotiated by using a meta-layer with which the user interacts and that applies a distributed workflow abstraction onto a lower layer that utilizes Chef. The PDS takes advantage of this unique structure in order to provide a simple solution to the complex problem of application deployment on multi-clouds.

III. Anatomy of the PDS

In this section we will provide an overview of PDS. At a high-level (Fig. 1), a user (who has already registered with the system) uploads an system pattern description (SPD) file to the service. The SPD explicitly defines/specifies the desired

[5]http://babushka.me/

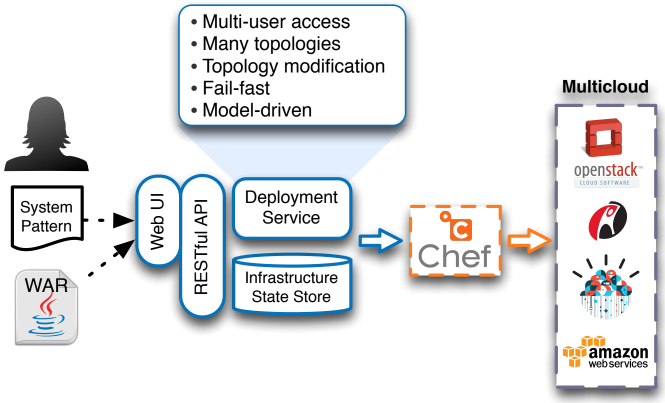

Fig. 1. Conceptual overview of the PDS.

application deployment to the PDS. The SPD is parsed by the PDS into a graph which is then dynamically converted into Chef commands that are issued, resulting in the instantiation of a set of nodes. The software stack described in the SPD for each node is then installed. Connections among various nodes in the topology are resolved from the graph structure. In the following sections we will explore this process and highlight key features of the PDS.

A. *XML-based DSL*

The PDS allows users to describe their system as a system pattern through use of its XML-based DSL. The language recognizes the following main elements; examples are shown in Figs. 2 and 3:

- `topology`: A topology represents the complete application to be deployed to the multi-cloud/two-tiered cloud.
- `container`: A container represents a collection typically of nodes (e.g., a cluster). A container has a special attribute, `num_of_copies`, that identifies how many copies of the contained node exist.
- `node`: A node represents a virtual machine instance (VMI). A node has several children elements including:
  - `key_pair_id`: a SSH key previously registered with a particular provider.
  - `ssh_user`: the username for logging onto an instance.
  - `cloud`: the cloud on which this node should be launched (e.g., EC2, Openstack, etc.).
  - `image_id`: denotes the particular flavour of the node (e.g., OS version/type, supporting software, etc.).
  - `security_groups`: specifies the named security group(s) to which the particular node belongs.
  - `instance_type`: specifies the configuration of the VMI in terms of CPU/RAM/Storage/Network, depending on the cloud.
  - `service`: There can be many services defined for each node; see below.
- `service`: Services represent any software process running on a given node that the user wishes to have

```
<topology id=”petstore”>
    <node id=”web_host”>
        <service name=”web_server”>
            <database_connection node=”web_host”/>
            <war_file>
                <file_name>petstore.war</file_name>
                <datasource>jdbc/pet</datasource>
            </war_file>
        </service>
        <service name=”database_server”>
            <script>petstore.sql</script>
        </service>
        <cloud>OpenStack</cloud>
        <instance_type>2</instance_type>
        <key_pair_id>demo</key_pair_id>
        <image_id>8</image_id>
        <ssh_user>ubuntu</ssh_user>
    </node>
</topology>
```

Fig. 2. Portion of XML-based DSL.

deployed and configured. Each `service` has an attribute `name` which identifies the type of the service (e.g., web_server).

- `database_connection`: This is the name of the node upon which a database server will be running, and to which this service may connect.
- `war_file`: This element has two child elements: `file_name` and `datasource`, referencing an uploaded file and the data source to use.

While functionally correct, as more nodes are added this system pattern becomes verbose. Two extension elements allow for (i) specifying templates (e.g., `instance_templates`) and (ii) referencing templates (e.g., `use_template`). The benefit of using these two additional element types lies in increased maintainability and readability of the document. In Fig. 3 the use of these tags is illustrated.

B. Graph Creation and Usage

The PDS automatically deploys a multi-node topology across a set of clouds (e.g., multi-cloud, two-tiered cloud). Each node in a given topology may run several services (e.g., a database, web application, VPN server, VPN client etc). Consequently, the deployment of one service requires the configuration information (such as IP address) from others. These are referred to as deployment dependencies; two algorithms are used to identify and resolve these when deploying a topology. Algorithm 1 extracts the dependencies from the system pattern and Algorithm 2 ensures the correct ordering of Chef invocations.

It should be noted that in Algorithm 2, line 5, a link is created between the Deployment Service (Fig. 1) and Chef. Specifically, a workflow is defined that ensures the correct recipes are run with the correct attribute settings in order to ensure dependencies are correctly deployed.

C. Stateful Service

The notion of stateful service is an important aspect of the PDS. It maintains a complete understanding of all deployed application topologies for a user. The user is then able to

```
<topology id=”scale”>
    <instance_templates>
        <template id=”openstack_small_instance”>
            <cloud>OpenStack</cloud>
            <instance_type>2</instance_type>
            <key_pair_id>demo</key_pair_id>
            <image_id>8</image_id>
            <ssh_user>ubuntu</ssh_user>
        </template>
    </instance_templates>
    <container num_of_copies=”2” id=”web_host_container”>
        <node id=”web_host”>
            <use_template name=”openstack_small_instance”/>
            <service name=”web_server”>
                <database_connection node=”data_host”/>
                <war_file>
                    <file_name>petstore.war</file_name>
                    <datasource>jdbc/pet</datasource>
                </war_file>
            </service>
        </node>
    </container>
    <node id=”data_host”>
        <use_template name=”openstack_small_instance”/>
        <service name=”database_server”>
            <script>petstore.sql</script>
        </service>
    </node>
    <node id=”web_balancer”>
        <use_template name=”openstack_small_instance”/>
        <service name=”web_balancer”>
            <member node=”web_host”/>
        </service>
    </node>
</topology>
```

Fig. 3. XML-based DSL showing the use of templates.

say *what* she wants the service to do (e.g., add 3 nodes to the application server tier) without requiring any low-level details be input. Due to the fact that the PDS deploys multiple topologies simultaneously a benefit of this stateful awareness is that the user is able to query it on the fly to get an up to date picture of her deployment as it proceeds.

D. Fail-fast and Recovery Model

As multi-clouds are complex systems on commodity hardware, failure of components is expected. PDS is able to detect failures of a deployment action and mark all affected nodes, and the overall topology, as `failed`. The user can then - through use of the RESTful API - invoke the `repair` action which corrects the failed deployment in an intelligent fashion. During the repair the topology is placed back in a `deploying` state and deployment continues to completion.

E. Maintainability

The code is written to be easily accessible to facilitate student involvement in continued development. Standard design patterns were employed (e.g., MVC). The code is documented (e.g., 6568 lines of comments / 9927 lines of code, see Table I). A particular focus of commenting is found in the /app/controllers directory as the Swagger[6] tool was used to generate both the Web UI and Java client automatically from comments embedded in the source code. There is also user guide available to assist with using and comprehending the system.

[6]https://developers.helloreverb.com/swagger/

Algorithm 1: Algorithm to construct a topology from the XML-based DSL. This algorithm assumes that a user has registered with the service and uploaded required files.

```
   Input: XML document xml
   Output: Topology topology
1  begin
2      Let nodes represent a list of nodes obtained from parsing
       the xml
3      foreach node in nodes do
4          Let $m$ be assigned the value of the node's multiplicity
5          foreach $i$ in $m$ do
6              Create a vertex $v$
7              Assign to vertex $v$ all services, attributes and files
               associated with node
8              Add vertex v to topology topology
9          end
10     end
11     Let dependencies be the set of dependencies compiled
       for the set of nodes
12     foreach dependency in dependencies do
13         Let vertex $i$ denote the source of the depenency
14         Let vertex $j$ denote the destination of the
           dependency
15         Create an edge $e_{ij}$ from vertex $i$ to vertex $j$
16         Add $e_{ij}$ to the topology topology
17     end
18     Ensure topology is a DAG
19     return topology
20 end
```

Algorithm 2: Algorithm to build the topology on the multi-cloud/two-tiered cloud.

```
   Input: Topology topology
   Output: Functioning Application Deployment on
           Multi-Cloud
1  Let undeployed denote the set of undeployed nodes in
   the topology
2  begin
3      while undeployed is not empty do
4          foreach node in set of undeployed nodes where
           the dependencies are resolved do
5              Define Chef configuration for this node
6              Instruct Chef to initiate the deployment
7          end
8      end
9  end
```

F. Dynamic Modification of Deployed Applications

From the user's perspective adding/removing resources to a cloud deployment is a straight-forward process (e.g., add five nodes to this cluster). However, in reality this is a non-trivial process that involves numerous interrelationships and dependencies among the involved services and nodes. Augmenting a topology's footprint is not the same process as initializing a deployment. Here we are dealing with running services which may be sensitive to service disruption. This must be carefully analyzed on a per-service basis and resolved and managed in a careful manner. The approach used by the PDS employs the notion of dirty nodes which represent a node that is impacted by the addition/removal of nodes to the system. Once the system discovers all the dirty nodes, the system attempts to "clean" these nodes by correcting their state and connections.

TABLE I
SOURCE LINES OF CODE AND COMMENTS IN THE PDS[7]

Language	Lines of Code	Lines of Comments
Ruby	9927	6568
JSON	4224	0
Ruby Templates	1638	25
YAML	286	10
XSD	265	4
XML	184	0

G. Chef

The PDS is responsible for translating system patterns to a workflow of actions; the actions generated are for Chef, for several reasons. Chef recipes are written in Ruby, as is the PDS. Chef, as a newer tool, has been designed from the ground up to be multi-cloud ready and has benefited from observing some of the limitations of older approaches.

H. PDS Deployment Options

There are four ways to deploy the PDS.

- A pre-deployed/hosted solution.
- An already running PDS can deploy a new independent PDS installation.
- An automated installer is available, which automatically deploys Chef (i.e., server, database, workstation), the Deployer Service, configures the system and generates an uninstall script.
- Manual installation from documentation.

IV. IMPLEMENTATION

The PDS was developed in the Ruby programming language and used many technologies including Phusion Passenger/Nginx. The core Deployment Service was developed using Ruby on Rails, a Mysql database and Chef was utilized as the configuration management tool. The complete system was significant in scale comprising almost 10,000 lines of Ruby code (see Table I) and 7000 lines in various supporting languages. We used the Swagger framework for both documentation purposes and the generation of client APIs.

V. USAGE OVERVIEW OF THE PDS

The PDS provides system pattern deployment as a service, and is intended for use on various complex cloud infrastructures (e.g., multi-cloud, two-tiered cloud, etc.). It is accessed over a network through a RESTful API; an organization can deploy their own installation (recommended) or use a publicly provided PDS.

The RESTful API exposes eight resources (Table II); some resources have sub-resources. Each resource supports up

[7]Note Ruby on Rails is responsible for some code generation that is included here. Code generated by the Swagger library is not included.

to five methods: `list`, `get`, `create`, `delete`, `modify` (which roughly correspond to GET, POST, DELETE, and PUT). The `topology` resource is used to create and modify system patterns; its supported methods are presented in more depth in Table III. Due to space limitations, the other resources are not presented in detail; an installed PDS includes live documentation, or one may consult the online documentation[8].

The remainder of this section will consider two illustrative scenarios for deploying an an application using the PDS. The first scenario will deploy the application to a single-node development environment. The second scenario will deploy the same application to a a multi-node environment on EC2 (e.g. for quality assurance).

A. Single Node Scenario

In this scenario, a standard Java EE application is deployed to an application server, with a database running on the same VMI. The first step is authoring a pattern describing the deployment; we use the example presented in Fig. 2. Second, the user uploads her credentials. This is done by invoking the `create` method on the /api/credentials endpoint. Next, the user uploads the application WAR file and a database generation script file using `create` on the /api/uploaded_files. Fourth, the user submits the pattern using the method `create` on the /api/topologies endpoint, receiving a unique id (*tid*). To actually deploy the topology the user invokes the method `modify` on the /api/topologies/{tid} endpoint, with `operation` set to `deploy`. At this point, the PDS begins deploying the application to the cloud identified in the pattern. As the user waits for deployment to complete, they can check the status of their application by invoking the method `get` on the /api/topologies/{tid} endpoint. The URL of the deployed application will be included in the response once the application is in the `deployed` state.

B. Multi-Node Scenario & Automated Repair

Typical production (and therefore quality assurance environments) involve multi-tiered, multi-node deployments. This scenario involves a more complex pattern including a front-end web balancer node, two application server nodes and a database server node. We will use the pattern presented in Fig. 3 for this example.

Despite the increased complexity of the deployment, using PDS proceeds as before: the new topology is uploaded using `create` on /api/topologies. The same credentials and uploaded files can be used, illustrating the high reusability of pattern-based deployments. The new topology can be deployed and undeployed as described previously. As this deployment involves several additional VMIs, which are notoriously unreliable, let us consider a failed virtual machine causes deployment to halt. The status reported by invoking `get` on the /api/topologies/{tid} endpoint will be "failed". In large distributed environments, there are best practices with regards to understanding what has gone wrong with any form of automation. A first check might involve determining whether the nodes are up. This might be followed by ensuring important services are running, etc. Assuming the problem is with the envorinment and not with the uploaded topology, PDS offers an automatic repair feature. By invoking the method `modify` on the /api/topologies/{tid} endpoint with the operation set to `repair`, PDS can be instructed to automatically correct issues (re-deploy nodes, restart services, etc.) and complete the application deployment.

In this multi-node scenario, we deployed two nodes. In anticipation of (or in response to) increased workload, a deployer may wish to scale up the application server tier. PDS allows the scaling of a running topology, by invoking the `modify` method on the `container` resource via the /api/topologies/{tid}/containers endpoint with an integer parameter expressing the desired size of the container.

After quality assurance is completed, the user can tear-down their application infrastructure by invoking the `modify` method on the /api/topologies/{tid} endpoint with the operation specified as `undeploy`.

VI. Case Studies

The PDS has been used to deploy applications within the SAVI project [19], but is also being used in more complex interactions as described in the following two case studies.

A. Supporting a Multicloud Application Management Platform

As the complexity of cloud systems increases, through multi-cloud deployments or two-tiered clouds like those proposed by SAVI, the ability to manage applications automatically becomes more important. With the growth of rapid/continuous deployment [20] and DevOps [21] approaches to application development and deployment, application developers are more interested in authoring code for self-managing their own applications. To address these needs, we introduced the X-Cloud[9] Application Management Platform (XCAMP) [13] which takes a developer-centric perspective allowing autonomic logic to be specified in the language of the developer's choice using their preferred environment and according to the methodology of their choice. This platform will be piloted in the SAVI project. The PDS is one of the cornerstones of this approach, along with a scalable multi-cloud ready monitoring system [22]. A key objective during the design of the platform was how to allow a developer to develop management logic with the same level of comfort and ease with which they develop business logic, which meant allowing a pattern-based model so they could specify the desired end result rather than the steps required to reach that end result.

XCAMP allows a developer to provide (i) a system pattern understood by PDS, (ii) a binary of their application, and (iii) an autonomic Management Logic Component (MLC), which is simply a web application that can be deployed by the PDS. Monitoring data, and information from the PDS about the state of the deployed application, is passed to the MLC, which

[8]https://github.com/ceraslabs/pattern-deployer

[9]The X is pronounced 'cross'.

TABLE II
LIST OF RESOURCES; MOST SUPPORT GET, POST, PUT, AND DELETE.

Resource	Description	Path
container	Used to group nodes in a collective unit (e.g. application server tier). Containers can be scaled up and down at runtime.	/api/topologies/{tid}/containers
credential	Used for uploading credentials that are used for authenticating users against their chosen cloud system.	/api/credentials
node	Represents individual instances / virtual machines / servers.	/api/topologies/{tid}/nodes /api/topologies/{tid}/containers/{cid}/nodes
service	Represents an application or service that runs on a node.	/api/topologies/{tid}/nodes/{nid}/services /api/topologies/{tid}/containers/{cid}/ nodes/{nid}/services /api/topologies/{tid}/templates/{temid}/services
supporting_service	A set of services shared by multiple topologies, e.g. DNS server, certificate authority, etc.	/api/supporting_services
template	Creates and modifies templates for nodes.	/api/topologies/{tid}/templates
topology	Represents the overall system pattern: how all of the containers, nodes, and services are related. Users can create, query, repair, deploy, or undeploy their system patterns.	/api/topologies
uploaded_file	Used to upload file(s): private keys, application archives (WAR, etc.), SQL scripts.	/api/uploaded_files

TABLE III
TOPOLOGY RESOURCE APIS

Methods	HTTP	Description	Path
list	GET	List all topologies.	/api/topologies
get	GET	Get topology with “tid”.	/api/topologies/{tid}
create	POST	Create a topology. This method requires users to submit their system pattern as a input parameter and create a topology based on the submitted pattern.	/api/topologies/
delete	DELETE	Delete the topology with “tid”.	/api/topologies/{tid}
modify	PUT	Modify topology with “tid” using an *operation*: “deploy”, “undeploy”, “repair”, “rename”, and “update_description”. “deploy” initiates translation from the pattern into a Chef workflow and deploys the application; “undeploy” terminates all resources involved in the topology; “repair” will recover a failed topology; “rename” and “update_description” change the metadata of the topology.	/api/topologies/{tid}

returns actions to modify the application topology. XCAMP components are responsible for translating the monitoring data and the PDS data to an abstract view understandable by the application developer (e.g. component names instead of IP addresses). The actions produced by the MLC are similarly translated into invocations of the PDS to enact the changes.

In a sample implementation on XCAMP, we demonstrated the ability and deploy to a hybrid cloud, with a local (edge) cloud running locally on Openstack and a remote (core) cloud in Amazon EC2, adaptively bursting to add nodes to the public cloud when we ran out of private resources. Using the Java client provided with PDS allowed for easy integration with the RESTful service.

Integration with PDS allowed XCAMP to focus on the platform for adaptive management, relying on PDS to deploy the application topology, maintain the state of a deployed topology, and modify a deployed topology.

B. Realizing the AERIE Architecture

AERIE [8] is a reference architecture for overlaying private clouds on public clouds. A set of supporting technologies are used to isolate the private cloud from shared physical resources, providing mitigation of security risks, a common fabric over heterogeneous providers, and more control. Deploying these supporting technologies and an application on top of the resulting private cloud is more complex than a standard cloud deployment. Here we describe an ongoing effort to automate the deployment of an AERIE-compliant system to the public cloud.

An overview of the reference architecture is shown in Figure 4. A key technology in AERIE is nested virtualization, where a virtual resource provided by the public cloud hosts a hypervisor (“container”), which hosts its own virtual machines (“inner instances”). These virtual machines are connected via virtual channels, implemented using VPNs, and run from encrypted disk images. Each outer instance includes a host intrusion detection and protection system. Public internet access to the inner instances, and the outer instances, is controlled by a gateway and a bulwark, which offer network intrusion detection and protection services and directs traffic to the correct inner / outer instance.

The services running on the actual instances are supported by a set of services running in a private datacenter: a DNS service, a private database, persistent storage for keys and disk images, and a controller with logic to manage the topology.

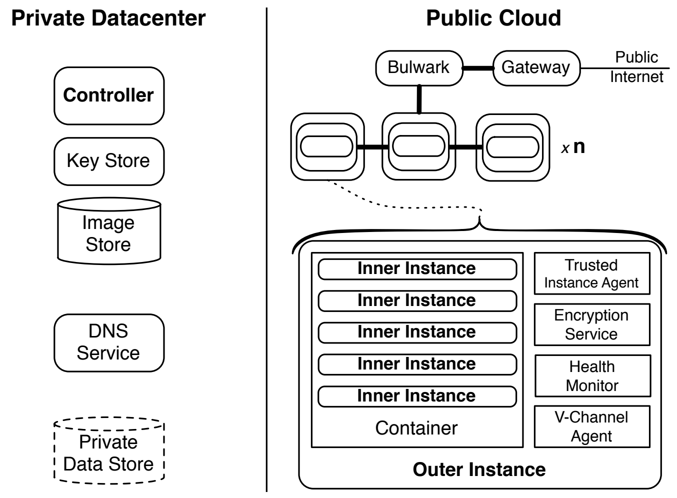

Fig. 4. The logical components of the AERIE reference architecture, from [8].

To deploy this complex topology with PDS, we include the tools and services running in the private cloud as supporting services, which can be created, deployed, and managed via the `supporting_service` resource. These components may only be deployed by privileged users, a rule enforced by the user management and permissions system of PDS. These services are deployed first, and information is available about these services as the remainder of the topology is deployed.

To build the nested virtual machines with the supporting services, we describe the desired nodes, their services, and the relationships among them using the XML-based DSL. For each inner instance, we insert an `node`, declare services inside that node, and request nested virtualization using the `nest_within` element which references the outer instance `node`. Each outer instance is its own node with declared services. Bulwarks and gateways are defined using their own `nodes`.

We are finding that using PDS to realize AERIE architectures reduces the effort required to achieve application deployments with desirable features that require complex and sophisticated deployment workflows.

VII. Conclusion and Future Work

We presented a deployment service for multi cloud environments. The service is installable in one or multiple clouds and allows the end user to specify a deployment, upload the deployment in the cloud, execute the deployment and monitor its execution. Also, the service facilitates changes in a deployed topology, hence enabling and autonomic runtime management. The service is accessible thorough both a RESTful interface and through a Web UI. We described the motivation for the service, identified its requirements, described the architecture and illustrated its usage through a case study.

Acknowledgment

This research was supported by IBM Centres for Advanced Studies (CAS), the Natural Sciences and Engineering Council of Canada (NSERC) under the Smart Applications on Virtual Infrastructure (SAVI) Research Network, and the Ontario Research Fund for Research Excellence under the Connected Vehicles and Smart Transportation(CVST) project.

References

[1] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow, "Blueprint for the intercloud - protocols and formats for cloud computing interoperability," in *Proceedings of the 2009 Fourth International Conference on Internet and Web Applications and Services*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 328–336.

[2] R. Buyya, R. Ranjan, and R. N. Calheiros, "Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services," in *ICA3PP (1)*, 2010, pp. 13–31.

[3] M. Shtern, B. Simmons, M. Smit, and M. Litoiu, "Navigating the cloud with a MAP," in *13th IFIP/IEEE International Symposium on Integrated Network Management (IM)*. To Appear, 2013.

[4] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski, "Introducing STRATOS: A cloud broker service," in *IEEE 5th International Conference on Cloud Computing*, 2012, pp. 891 –898.

[5] N. Grozev and R. Buyya, "Inter-cloud architectures and application brokering: taxonomy and survey," *Software: Practice and Experience*, 2012.

[6] M. Smit, P. Pawluk, B. Simmons, and M. Litoiu, "A web service for cloud metadata," in *IEEE Congress on Services*. Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 24–29.

[7] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring alternative approaches to implement an elasticity policy," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, 2011, pp. 716–723.

[8] M. Shtern, B. Simmons, M. Smit, and M. Litoiu, "An architecture for overlaying private clouds on public providers," in *8th International Conference on Network and Service Management, CNSM 2012, Las Vegas, USA*, 2012.

[9] SAVI, "Strategic network for smart applications on virtual infrastructure (savi)," http://www.savinetwork.ca/, 2012, last access 07/03/2013.

[10] M. Smit, M. Shtern, B. Simmons, and M. Litoiu, "Partitioning applications for hybrid and federated clouds," in *CASCON '12: Proceedings of the Conference of the Center for Advanced Studies*, 2012, pp. 27–41.

[11] Z. X. Chen, S. Imazeki, M. Kelm, S. Kofkin-Hansen, Z. Q. Kou, B. McChesney, and C. Sadtler, *IBM Workload Deployer: Pattern-based Application and Middleware Deployments in a Private Cloud*. International Business Machines Corporation, 2012.

[12] T. Eilam, M. Elder, A. Konstantinou, and E. Snible, "Pattern-based composite application deployment," in *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, 2011, pp. 217 –224.

[13] M. Shtern, B. Simmons, M. Smit, H. Lu, and M. Litoiu, "Toward an autonomic systems platform for multi-clouds," Submitted, 2013.

[14] T. Delaet, W. Joosen, and B. Vanbrabant, "A survey of system configuration tools," in *Proceedings of the 24th international conference on Large installation system administration*, ser. LISA'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.

[15] M. Burgess, "A site configuration engine," *Computing systems*, vol. 8, no. 2, pp. 309–337, 1995.

[16] T. Clark, "Quantifying the benefits of the rightscale cloud management platform," Fact Point Group Whitepaper, funded by Rightscale, 2010.

[17] T. Eilam, M. Kalantar, A. Konstantinou, and G. Pacifici, "Reducing the complexity of application deployment in large data centers," in *Integrated Network Management, 9th IFIP/IEEE International Symposium on*, 2005, pp. 221 – 234.

[18] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[19] E. Stroulia, I. Nikolaidis, L. Liu, S. King, and L. Lessard, "Home care and technology: a case study." *Studies in health technology and informatics*, pp. 142–152, 2012.

[20] J. Jenkins, 2011, presentation from O'Reilly Velocity Conference last accessed March 2013: http://assets.en.oreilly.com/1/event/60/Velocity%20Culture%20Presentation.pdf.

[21] D. Ohara, "Continuous delivery and the world of devops," GigaOM Pro Whitepaper, September 2012.

[22] M. Smit, B. Simmons, and M. Litoiu, "Distributed, application-level monitoring of heterogeneous clouds using stream processing," *Future Generation Computer Systems*, To Appear, 2013.