Provisioning of Component-based Applications Across Multiple Clouds

Mehdi Ahmed-Nacer[1,2], Sami Yangui[3], Samir Tata[2,4] and Roch H. Glitho[3]
[1]*University of Sciences and Technology Houari Boumediene, Algiers, Algeria*
[2]*SAMOVAR, Telecom SudParis, CNRS, Universite Paris-Saclay, 9, rue Charles Fourier, 91011 Evry Cedex, France*
[3]*Concordia Institute for Information Systems Engineering, Concordia University, Montreal, QC, H3G 2W1, Canada*
[4]*IBM Research-Almaden, San Jose, CA, 95120, U.S.A.*

Keywords: Cloud Computing, COAPS, OCCI, PaaS, Resource Provisioning.

Abstract: The several existing Platform-as-a-Service (PaaS) solutions are providing application developers with different and various offers in terms of functional properties (e.g. storage), as well as, non-functional properties (e.g. cost, security). Consequently, developers may need to provision components of the same application across several PaaS depending on their related requirements and/or PaaS capabilities. This paper proposes generic mechanisms that allow seamless component-based applications provisioning across several PaaS. These mechanisms are based on the COAPS API; an already defined OCCI-compliant API that allows provisioning of monolithic applications in PaaS using generic descriptors and operations. To illustrate the proposed mechanisms, the paper showcases a realistic use case of provisioning of a JEE-based simulation application across Elastic Beanstalk and Cloud Foundry platforms.

1 INTRODUCTION

Cloud computing is an emerging model for enabling ubiquitous, convenient, and on-demand network access to a shared pool of configurable computing resources (Mell and Grance, 2009). Cloud computing providers offer their services according to three different models, namely Infrastructure as-a-Service (IaaS), Platform as-a-Service (PaaS), and Software as-a-Service (SaaS). According to these three service models, Cloud resources should be swiftly provisioned with minimal management effort.

The completion and the achievement of such model have induced the proliferation of not only cloud end-users but cloud service providers as well (Dash et al., 2009). The agreements between both sides are based on the pay-as-you-go model (Grossman, 2009). Generally, cloud providers (e.g. IaaS, PaaS) vary their offers in terms of functional properties (e.g. storage, networking), as well as, non-functional properties (e.g. cost, security) in order to attract as many clients as possible. Consequently, end-users are facing real challenges in order to fully optimize their cloud usage while meeting their functional and non-functional requirements. This inevitably involves the need of multi-provider provisioning of the handled resources, since probably none of the involved cloud providers will be able to provide the entire required services, with respect to all end-user requirements.

Provisioning cloud resources cover describing, deploying and managing them (Yangui and Tata, 2016). For instance, for cloud end-user applications case, the related provisioning process consists of: (1) Describing the application requirements, (2) allocating PaaS resources necessary to meet such requirements, (3) deploying the application over these resources and (4) managing (including executing) it. However, when considering such context, many limitations and issues still persist in order to be able to provision applications across several PaaS. These issues are mainly due to the diversification of the PaaS solutions and the strong heterogeneity of their provided resources, services, APIs, etc. In fact, existing PaaS solutions (e.g. Google Cloud Platform[1], Cloud Foundy[2], Heroku[3]) propose heterogeneous frameworks and runtimes for applications depending on their implementation technologies and capabilities. In addition, these frameworks are provisioned by PaaS in a specific way. Each PaaS has its own deployment

[1]cloud.google.com
[2]cloudfoundry.org
[3]heroku.com

Ahmed-Nacer, M., Yangui, S., Tata, S. and Glitho, R.
Provisioning of Component-based Applications Across Multiple Clouds.
DOI: 10.5220/0006296901320142
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 104-114
ISBN: 978-989-758-243-1
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

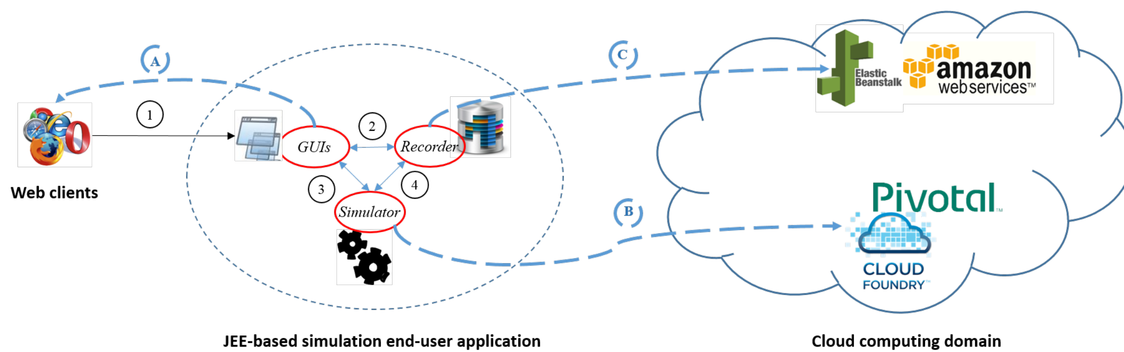

Figure 1: JEE-based simulation application deployment across several PaaS.

scenario and procedures. Consequently, these specificities and limitations make difficult provisioning of component-based applications, such as applications described according to Service-Oriented Architecture (SOA) specifications, in a multi-PaaS environment.

To address these issues, this paper proposes generic mechanisms that allow seamless component-based applications provisioning across several PaaS. These mechanisms enable the placement of applications' components into several PaaS in order to fully take advantage of their various offers when deploying and managing them. This provisioning is performed using the same operations and descriptors. The proposed solution uses and extends CompatibleOne Application Provisioning Service[4] (COAPS for short). COAPS provides unified description model and API that allows provisioning applications in a given PaaS (Yangui and Tata, 2016) (Sellami et al., 2013b). It is based on the Open Cloud Computing Interface (OCCI) recommendation for a standard (OCCI, 2016). A prototype was implemented in order to validate these findings.

The rest of the paper is organized as follows: Section 2 introduces a motivating use case. Section 3 presents the already existing COAPS API. Section 4 presents M-COAPS, the proposed extension for enabling generic multi-PaaS applications provisioning. Section 5 describes the implementation details and the developed prototype for its validation. Section 6 evaluates the existing related work. Section 7 concludes the paper and discusses the planned future works.

2 USE CASE AND MOTIVATIONS

Let us consider the provisioning of the JEE-based simulation application across several PaaS as use case. This application is shown in Figure 1. It allows simulating the operation of applications in cloud environments based on a given configuration provided by the user. The user provides a description of the application and the configuration of the cloud environment to simulate. Each configuration describes a prospective hosting cloud environment in terms of required datacenters, VMs, applications instances and so on. The simulation application is designed according to the MODEL-VIEW-CONTROLLER pattern. It consists of three components:

- The *Front-end* component provides a set of graphic Web interfaces that allow users to define and/or edit a simulation configuration template (Figure 1, action 1). A snapshot of such interface is shown in Figure 2. Specifically, a user has two options: (1) Define a new configuration and run its simulation or (2) display an existing configuration with its already calculated simulation data. This component refers to as the VIEW entity.
- The *Recorder* component allows the storing of a newly defined simulation template in a persistent database for reuse purposes (Figure 1, action 2). It also allows uploading and sending stored configurations, and eventually their related simulation results, to the *Front-end* component for display. The used database schema is No-SQL key-value. This component refers to as the MODEL entity.
- The *Simulator* component is based on *CloudSim*[5], an open-source cloud simulation tool. It simulates and evaluates cloud applications characteristics (e.g. performance) for a given set of configuration templates. The simulation result is sent to the *Front-end* component for display (Figure 1, action 3) and to the *Recorder* component for storage (Figure 1, action 4). This component refers to as the CONTROLLER entity.

On one side, developers may have to deploy and/or manage such application across several PaaS.

[4] compatibleone.com/cgi-sys/suspendedpage.cgi#coaps

[5] cloudbus.org/cloudsim/

Figure 2: The configuration templates management interface.

A multi-PaaS provisioning could be motivated by several reasons such as security, cost and/or performance. For example, for cost point of view, a basic compute node (i.e. Linux OS, 1 core CPU, 128 GB of RAM and 1Go of storage disk) costs $0.006/Hour in Cloud Foundry while it costs $0.013/Hour in Amazon EC2. In addition, the average cost of 1GB of basic key-value storage service is $71/Month in Cloud Foundry (*RedisDB* service) while it is $43/Month in Amazon (*DynamoDB* service). Therefore, based on these rates, developers might be interested by the alternative of provisioning the application's components that require intensive compute resources in Cloud Foundry rather than Amazon (Figure 1, action B). In the same way, they may consider provisioning components that handle data storage in Amazon rather than Cloud Foundry (Figure 1, action C). Finally, it should be noted that the GUIs could be provisioned locally in the end-users terminals for cost optimization and performance purposes (Figure 1, action A). This is very common practice for components that are part of the VIEW entity in applications designed according to the MODEL-VIEW-CONTROLLER pattern.

On the other side, runtimes, frameworks and hosting cloud resources are provisioned by existing PaaS in specific way. This makes the aggregation and the automation of multi-PaaS provisioning difficult to set. Each PaaS solution has proprietary model to describe and provision applications and their related hosting resources. Furthermore, the user APIs implementing these description models are strongly heterogeneous (e.g. proprietary operations, specific provisioning scenarios, etc.). These limitations are forcing developers to adapt their applications and procedures when they deploy and manage component-based applications across several PaaS in order to bypass the vendor lock-in (Satzger et al., 2013).

To address these issues, a couple of requirements need to be met. The first requirement is the need for mechanisms that enable and automate provisioning of component-based applications across several PaaS. This will allow taking benefit from the variety of the PaaS offerings. It also provides the developers with more alternatives to satisfy their needs and optimize their applications operating. The second requirement is the need for a unified model and generic provisioning mechanisms. PaaS-independent mechanisms will considerably harmonize and simplify the provisioning procedures of applications' components across several PaaS.

COAPS meets the latter requirement related to the need for unified model and generic operations. Indeed, as part of a previous work, COAPS was designed and implemented as PaaS-independent solution that provides a set of generic REST interfaces. These interfaces aggregate the APIs exposed by the PaaS offerings based on a unified resources description model (Sellami et al., 2013b) (Yangui et al., 2014). However, it should be noted that COAPS only supports the provisioning of monolithic applications. Furthermore, it does not support any multi-PaaS provisioning approach. Indeed, developers cannot use COAPS to deploy applications on top of more than one PaaS offering. Consequently, COAPS does not meet the first requirement.

In this paper, an extended COAPS solution that meets the first identified requirement related to the automation of applications provisioning across several PaaS is proposed. For understanding, COAPS is firstly introduced in Section 3. Then, Multi-PaaS COAPS solution (M-COAPS for short) is discussed in Section 4.

3 COMPATIBLEONE APPLICATION PROVISIONING SERVICE

COAPS is a PaaS-independent model and API for PaaS resources provisioning. The introduced model is based on the Open Cloud Computing Interface (OCCI) specifications. It enables the description of both platform and application resources independently of the target PaaS.

OCCI is a set of specifications that define a metamodel for abstract cloud resources and a RESTful protocol for their management (OCCI, 2016). It offers a flexible API with a strong focus on interoperability while still offering a high degree of extensibility. COAPS model consists of: (1) An OCCI platform specification which describes all PaaS resources that can be provisioned by a PaaS to set up an appropriate hosting-environment (Yangui et al., 2013) and (2) an OCCI application specification which describes the application resources to deploy in this environment (Sellami et al., 2013a). Indeed, the main resources handled by this model are *Environment* and *Application*. *Environment* resources are composed of platform resources (e.g. containers, databases). *Application* resources involve all necessary artifacts required to execute the application once it is deployed (e.g. source code, configuration files, scripts). Each one of the defined resources is characterized by a set of attributes and actions to handle and manage them according to OCCI specifications. The detailed list of these resources and their related attributes and actions are discussed in (Yangui and Tata, 2016).

COAPS API provides end-users with a set of generic operations for applications provisioning. These operations implement the actions that can be applied to platform and application resources according to the defined description model. COAPS API is based on OCCI HTTP Renderings (Metsch and Edmonds, 2011) and Representational State Transfer (REST) architecture. The listing of COAPS abstract generic interfaces and related specifications are detailed in (Sellami et al., 2013b). COAPS is designed according to a proxy system that allows implementing and adapting its abstract interfaces when integrating a new PaaS offering.

As shown in Figure 3, COAPS proxy architecture consists of three layers:

- *Front-end layer*, that exposes the COAPS generic interfaces. These interfaces represent abstract RESTful operations for application and related hosting-environment provisioning;

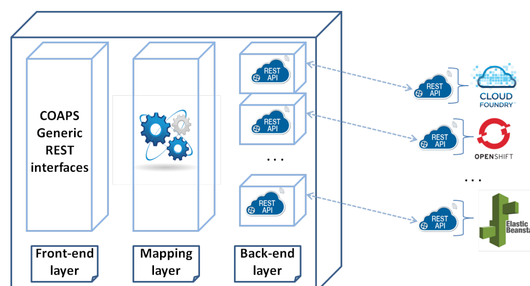

Figure 3: COAPS proxy system.

- *Back-end layer*, that represents the PaaS offerings interfaces. These interfaces expose the proprietary API operations of a given PaaS;
- *Mapping-end layer*, that constitutes the middle layer of the proxy. This layer ensures the mapping between the COAPS generic operations on one side and the proprietary PaaS API operations on the other side.

To create a new COAPS implementation for a given PaaS offering, one can simply instantiate its proxy after implementing its correspondent middle layer to map between the COAPS generic interfaces and the proprietary operations exposed by the API of the selected PaaS. Currently, proxies for Cloud Foundry, OpenShift, Elastic BeansTalk and Google App Engine platforms are implemented (available at (coa, 2016)). However, COAPS does not allow using more than one proxy at the same time to deploy one application across multiple PaaS.

4 M-COAPS FOR MULTI-PAAS PROVISIONING

The M-COAPS (M stands for Multi-PaaS) solution discussed in this section aims at meeting the requirement related to automatic applications provisioning across several PaaS. It addresses the identified limitations and drawbacks of COAPS described at the end of Section 3. M-COAPS supports component-based applications provisioning. This includes applications' components deployment and management in the several involved PaaS offerings. The decision to place the applications' components in the available PaaS offerings could be motivated by criteria such as hosting cost, applications' components requirements and/or hosting PaaS capabilities. While the issues related to the decision on components placement are important, the contributions discussed in this paper i.e. the effective deployment across multiple platforms are complex enough in themselves to deserve separate treatment. Placement decisions are out of the scope of this

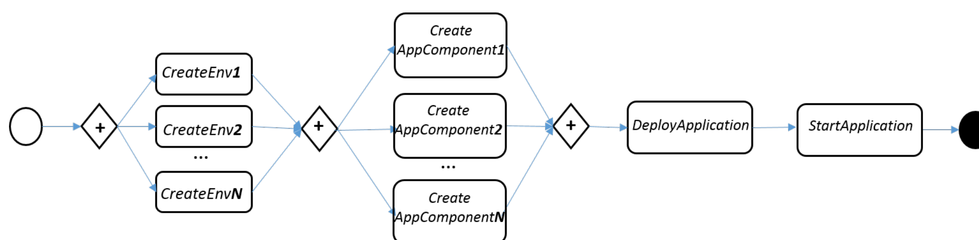

Figure 4: M-COAPS provisioning process.

paper.

As it is the case for COAPS, M-COAPS provides generic interfaces in order to allow deploying and managing components of the same application in different PaaS offerings. The provisioning steps are performed following the process shown in Figure 4. Broadly speaking, a developer first creates the required hosting environments resources for its application's components in the target platforms. Then, he/she creates the resources representing the application's components. After that, he/she proceeds to the concrete deployment and the activation of the whole application. Detailed description of each activity of the provisioning process is provided in the rest of this section.

4.1 *createEnvironment* Operation

`createEnvironment` operation instantiates an *Environment* resource and allows allocation of the required platform resources for hosting and executing an application (e.g. containers, DBMS). This operation is executed as many times as there are hosting-environments to provision. Each created *Environment* belongs to a given PaaS offering. The properties of each *Environment*, as well as, its target PaaS are described in a descriptor provided by the developer when calling this operation.

4.2 *createAppComponent* Operation

`createAppComponent` operation instantiates the required *ApplicationComponent* resources, that made up a component-based application. This operation is executed as many times as the number of components of the application to deploy. The components' properties (e.g. name, code version, and artifacts location) are listed in the application descriptor provided by the developer. The components listing order in the descriptor has meaning. It defines the flow and indicates the order of executing the several component in order to implement the final application's functionality (business logic). In addition, the developer have to mention in the descriptor to which already created *Environment* each *ApplicationComponent* belongs. It should be noted that the same *Environment* can be assigned to several applications' components if needed.

4.3 *deployApplication* Operation

`deployApplication` operation allows uploading the application's artifacts over the hosting *Environment* for the concrete deployment. It enables orchestrating the deployment of the several application's components. Specifically, it deploys the created *ApplicationComponent* resources over their related *Environment* in the target PaaS offerings. The orchestration consists on processing the wiring of the application's components once they are deployed. It follows the reverse of the application execution chain mentioned in its descriptor. Basically, `deployApplication` starts by processing *ApplicationComponent N*. It uploads its source code on the target PaaS, finalize the deployment and collects its associated public access URL returned by the hosting PaaS. Then, it provides this URL to *ApplicationComponent N-1* when processing its deployment until reaching the front-end component. The location of *ApplicationComponent N* is provided to *ApplicationComponent N-1* through a routing file attached to the application's component code at upload time.

4.4 *startApplication* Operation

`startApplication` operation allows activating the whole deployed application's components. An application is considered as available when all its related components are activated. The starting process order follows the same order of the application execution chain. It begins by starting the front-end component. Then, it starts the next component in the execution chain until reaching *ApplicationComponent N*. It should be noted that the same order is followed for `stopApplication` and `restartApplication` operations as well.

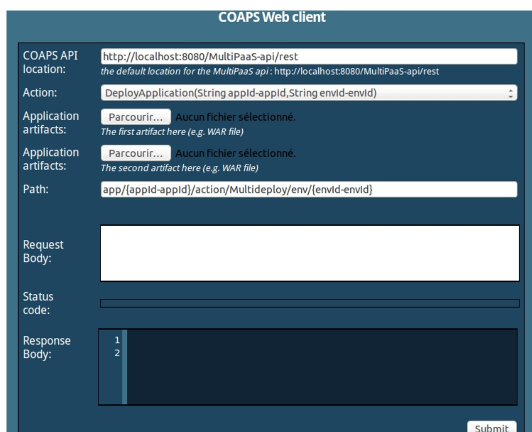

Figure 5: M-COAPS Web client.

5 IMPLEMENTATION AND VALIDATION

M-COAPS is implemented according to the specification and the provisioning process shown in Figure 4. It is developed as HTTP RESTful Web application with Java (JAX-RS) using Jersey. The descriptors are XML-based. Currently, the Cloud Foundry and Amazon Elastic Beanstalk proxies are integrated to M-COAPS. Adding new proxy support is done according to the RESTful architecture principles. In addition to that, the already performed COAPS Web client was adapted in order to be in-line with the new M-COAPS capabilities as it is shown in Figure 5. The client's interface allows henceforth several artifacts uploads from different locations. Each artifact corresponds to a specific application's component. Similarly, the resources path field allows henceforth specifying a list of *Environment* and *ApplicationComponent* resources *IDs*. In order to validate the implementation, the provisioning of the JEE-based simulation application introduced in Section 2 was performed using M-COAPS across Cloud Foundry and Amazon Elastic Beanstalk. The associated descriptors are shown in Listing 1. The source code of this implementation and a demonstration video showing the execution of the different steps are available at COAPS API Webpage (coa, 2016). According to the scenario discussed in Section 2, the *Simulator* component is provisioned in Cloud Foundry while the *Recorder* component is provisioned in Amazon Elastic Beanstalk. The details of the provisioning of this application are described in the rest of this section according to the provisioning process steps shown in Figure 4. The parallel operations are implemented as java threads for performance purpose.

5.1 *createEnvironment* Operation

This step allows the creation of the required *Environment* resources and the instantiation of these resources on the two target PaaS. `createEnvironment` is a POST REST request. It returns back a set of *EnvIDs* related to the newly created *Environment* resources. The XML elements describing *Environment* resources properties are shown in Listing 1 (lines 6-18).

SimEnv is the first hosting environment to create (lines 6-11). The target PaaS (i.e. Cloud Foundry) is indicated as value of *provider* attribute (line 6). The associated *description* is provided in line 8. This environment consists of a single PaaS resource node i.e. an Apache Tomcat Web container (line 9). *StorageEnv* is the second hosting environment to create (lines 12-18). The target *provider* is Amazon Elastic Beanstalk (line 12). The environment *description* is provided in line 14. This environment consists of two nodes: An Apache Tomcat Web container (line 15) and a DynamoDB database service (line 16).

It should be noted that for reuse purpose, it is possible to define environment templates and simply refer to the template name in the descriptor during future environment creations. For instance, *SimEnvTemp* is associated *SimEnv* template (lines 7-10) while *StorageEnvTemp* is associated to *StorageEnv* template (lines 13-17).

5.2 *createAppComponent* Operation

This step allows the creation of the required *ApplicationComponent* resources. `createApplicationComponent` is a POST REST request. It returns back a set of *AppIDs* related to the newly created *ApplicationComponent* resources. The XML elements describing *ApplicationComponent* resources properties are shown in Listing 1 (lines 19-33). *Simulator* is the name of the first component to create (lines 19-26). The hosting environment of this component is *SimEnv* (line 19). Consequently, it will be provisioned in Cloud Foundry. The *description* of this component is provided in line 20. Its associated artifacts *name* and *location* are provided in line 22. During the deployment of this component, the target PaaS has to instantiate and run two instances of it (lines 23-24). The main instance is *Instance1* (*default_instance="true"*). The two instances has to be started simultaneously (*initial_state="1"*). The second component to create is *Recorder* (lines 27-33). Its hosting environment is *StorageEnv* (line 27). This component will be provisioned in Amazon Elastic Beanstalk. Its *description* is provided in line

```
<?xml version="1.0" encoding="UTF-8"?>
<paas_application_manifest name="SimulationApplicationManifest">
  <description>This_manifest_describes_the_SimulationApplication_Environments_and_Components</description>
  <paas_application name="SimulationApplication">
    <description>This_App_simulates_deployment_costs_of_given_configs<description>
    <paas_environment name="SimEnv" provider="CF" template="SimEnvTemp">
      <paas_environment_template name="SimEnvTemp" memory="512">
      <description>SimulatorHostingEnvironmentTemplate</description>
      <paas_environment_node content_type="container" name="tomcat" version="7"/>
      </paas_environment_template>
    </paas_environment>
      <paas_environment name="StorageEnv" provider="AWS" template="StorageEnvTemp">
      <paas_environment_template name="StorageEnvTemp" memory="1024">
      <description>64bit_Amazon_Linux_Tomcat7Java 7_RecorderHostingEnvironment</description>
      <paas_environment_node content_type="container" name="tomcat" version="7"/>
      <paas_environment_node content_type="database" name="DynamoDB" version="2012-08-10"/>
      </paas_environment_template>
    </paas_environment>
    <paas_application_component name="simulator" environement="SimEnv">
      <description>simulatorServlet</description>
      <paas_application_version name="version1.0" label="1.0">
        <paas_application_deployable name="simulator.war" content_type="artifact" location="COAPS/tmp"/>
        <paas_application_version_instance name="Instance1" initial_state="1" default_instance="true"/>
        <paas_application_version_instance name="Instance2" initial_state="1" default_instance="false"/>
      </paas_application_version>
    </paas_application_component>
    <paas_application_component name="recorder" environement="StorageEnv">
      <description>recorderServlet</description>
      <paas_application_version name="version1.0" label="1.0">
        <paas_application_deployable name="storage.war" content_type="artifact" location="COAPS/tmp"/>
        <paas_application_version_instance name="Instance1" initial_state="1" default_instance="true"/>
      </paas_application_version>
    </paas_application_component>
  </paas_application>
</paas_application_manifest>
```

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Listing 1: JEE-based simulation application descriptor.

28 and the information related to its artifacts in line 30. A unique instance of this component will be instantiated once it is deployed (line 31).

5.3 *deployApplication* Operation

This step implements a POST REST request that enables the concrete deployment of the applications components in the target PaaSs. This is done by associating explicitly the *AppIDs* to the *EnvIDs* (see Figure 5). The upload of the components' artifacts to the two target PaaS is done during this step, as well as, the wiring between the two components. *DeployApplication* first processes the deployment of the *Recorder* component in Elastic Beanstalk. Then, it provides the returned public URL provided by Amazon to the *Simulator* component before its deployment in Cloud Foundry.

5.4 *startApplication* Operation

This step implements a POST REST request that enables activating the application. The application's components deployed in the two PaaSs are then started. `startApplication` first starts the *Simulator* component in Cloud Foundry before starting the *Recorder* component in Elastic Beanstalk.

6 RELATED WORK

Most of the existing solutions focus on enabling interoperability between cloud service providers, geo-diversity and/or portability (Ranjan, 2014) (Martin-Flatin, 2014) (Di Martino, 2014). Moreover, it should be noted that most of them are made for the IaaS domain such as *mOSAIC* (Petcu et al., 2013) and *PerfCloud* (Mancini et al., 2009). For example, *mOSAIC* is a European FP7 project that aims at enabling

data, services and applications portability and interoperability across multiple clouds (Petcu et al., 2013). The hosting multi-clouds system covers mainly IaaS providers. However, *mOSAIC* provides platform (i.e. *Software Platform Support layer*) that supports end-user applications provisioning. *mOSAIC* is based on brokering mechanisms that search for cloud services matching the applications' requirements. *mOSAIC* framework consists of several layers (SaaS, PaaS and IaaS) and offer a set of APIs in each one of these layers. The applications' execution framework is provided by *mOSAIC* on top of a set of integrated IaaS. The selection of the target IaaS is based on brokerage contracts. The interactions between *mOSAIC* and the IaaS providers is performed through appropriate adaptors. Currently, *mOSAIC* provides adaptors for Amazon EC2, Flexiscale, Eucalyptus and OpenNebula. *mOSAIC* does not meet the first requirement related to automatic provisioning of applications across several PaaS. It focuses mainly on IaaS providers integration. However, it meets the second requirement related to unified model and generic provisioning operations thanks to the semantic ontology for capabilities description and the generic API exposed by the *Software Platform Support layer*. On the other hand, several projects and academic works targeted end-user applications and PaaS domain. For example, in (Kamateri et al., 2013), the authors propose an approach to semantically interconnect heterogeneous PaaS that share the same technology. This solution was developed as part of the European FP7 *Cloud4SOA* research project. It aims to provide better accessibility and flexibility in the fragmented PaaS market. The aggregation of the integrated PaaS capabilities in *Cloud4SOA* framework is done thanks to a semantic common ontology and a harmonized API (D'Andria et al., 2012) . A proof-of-concept showing SOA applications management in hybrid multi-PaaS environment is also presented in (Zeginis et al., 2013). *Cloud4SOA* partially meets the first requirement related to the automatic provisioning of component-based applications across several PaaS. In fact, it supports only provisioning of applications that are designed according to SOA specifications. These applications are subset of the component-based applications set supported by M-COAPS. *Cloud4SOA* meets the second requirement related to unified model and generic provisioning operations thanks to the semantic ontology describing the aggregated resources and the harmonized API.

MODAClouds is a research project that aims at supporting developers and service providers when operating in a multi-cloud system (Ardagna et al., 2012). It defines a model-driven approach to design and deploy applications at large including IaaS and PaaS applications across several clouds. *PaaSage* project reuses *MODAClouds* basics and findings. It aims at designing and implementing a PaaS for end-user applications development and deployment in existing clouds (Baur et al., 2015). The defined methodology is inspired by *MODAClouds* model. Specifically, when deploying an application, appropriate interfaces and connectors are generated automatically for application integration with user systems in northbound and target cloud at the southbound. However, the generation process is not fully automated. It requires the involvement of the user to provide additional information such as the choice of cloud providers and services, and the reuse of external services (Ferry, 2015). Hence, *MODAClouds* and *PaaSage* did not meet the first requirement related to the automatic provisioning of applications across several PaaS. However, they meet the second requirement related to the unified model and provisioning operations.

PaaSHopper is a framework for operating SaaS applications on top of a multi-PaaS environment (Walraven et al., 2015). The application's owner specifies its properties and the required PaaS capabilities. These properties describe the constraints and rules in a declarative way, extracting them from the application code in a modular and reusable way. Depending on the provider-specific capabilities, *PaaSHopper* decides where in the multi-cloud a task will be executed or data will be stored. It uses deployment descriptor, specifying the different PaaS platforms and resources to be used in the multi-PaaS system. *PaaSHopper* meets the first requirement related to automatic provisioning of applications across several PaaS. However, it does not meet the second requirement related to unified model and generic provisioning operations. Indeed, even if the deployment descriptor is based on a model, it still poor and not rich enough to cover all the specificities and the strong heterogeneity of the resources handled by the existing PaaS offerings. Furthermore, it is not extensible. It is simply based on a command-line shell (i.e. Microsoft Windows PoweShell) and a limited set of pre-defined commands to describe the applications' requirement and the PaaS resources.

In (Paraiso et al., 2012), the authors present a federated multi-cloud system that enables the provisioning of applications described according to the Service Component Architecture (SCA) specifications. This approach provides a unified model and generic provisioning of applications across several PaaS and IaaS that should be provided with SCA containers. A prototype is also provided (Paraiso et al., 2016). However, it only supports SCA-based applcations. Sim-

Table 1: Related work evaluation synthesis.

Requirements / References	R1	R2
(Kirkham, 2013) (Petcu et al., 2013) (Ferry, 2015)	No	Yes
(Kamateri et al., 2013) (D'Andria et al., 2012) (Paraiso et al., 2012) (Paraiso et al., 2016)	Partially	Yes
(Walraven et al., 2015)	Yes	No
(Cunha et al., 2014)	Yes	Not mentioned
(Pahl, 2015) (Hadley et al., 2015) (Wei et al., 2011) (Ardagna et al., 2012)	No	No

ilarly, the targeted cloud environments should also support SCA containers. This work partially meets the first requirement related to the automatic provisioning of component-based applications across several PaaS. It supports only provisioning of SCA-based applications that are designed according to SOA specifications which are subset of SOA applications. Moreover, it meets the second requirement related to unified model and generic provisioning operations thanks to the introduced mapping model in between the cloud offerings.

In (Wei et al., 2011), the authors introduce Application Platform-as-a-Service for Cloud Computing called *Aneka*. It acts as a framework for building customized applications and deploying them on either public or private clouds. Basically, *Aneka* does not support multi-PaaS environment. But, a newly added extension supports distributed end-user applications deployment in multi-cloud environments (Buyya and Barreto, 2015). To illustrate the benefits of the new extension, the authors deploy an application composed of independent tasks by *Aneka* in multi-cloud environment using resources provisioned from Microsoft Azure and Amazon EC2. *Aneka* does not enable automatic provisioning and does not provide any unified model and/or generic provisioning operations. Consequently, it does not meet the two requirements.

In (Brogi et al., 2015), the authors provide an open source framework called *SeaClouds* to address the problem of deploying, managing and reconfiguring complex applications over multiple and heterogeneous clouds. This framework is based on Topology and Orchestration Specification for Cloud Applications (TOSCA) specifications to describe the topology of the applications independently of the target cloud providers and Cloud Application Management for Platforms (CAMP) API for its management. TOSCA and CAMP are standardization tentatives within OASIS consortium. *SeaClouds* automates the provisioning process using CAMP API operations; however, it imposes the support of TOSCA-enabled containers for the target PaaS. Consequently, it partially the first requirement related to the automatic provisioning of component-based applications across several PaaS. In addition, it meets the second requirement related to unified model and generic provisioning operations thanks to a mapping model in between the cloud offerings.

In (Cunha et al., 2014), the authors propose a framework denominated *PaaS Manager* that aims to struggle the existing lock-in in the PaaS market. *PaaS Manager* is intended to fit the many developer's needs in a multi-PaaS environment, such as developing and deploying applications, monitor operation date in real-time and migrate applications between PaaS offerings. Hence, it meets the first requirement related to automatic provisioning across several PaaS. To deal with the multi-PaaS environment, the user can fill a form with the technical profile of the application through a command-line or a Web interface. However, the authors did not detail on the properties that the user may enter.

Finally, it should be noted that several works use the containerization approach to enable distributed applications deployment across several clouds (e.g. see (Pahl, 2015), (Hadley et al., 2015)). These approaches consists on packaging the applications' components dynamically in a generated service containers. The generated service containers implement the applications' components requirements (e.g. required runtime, libraries). Then, the containers are pushed to the cloud as standalone applications. These approaches require manual deployment through command-line and do not provide any unified model and/or provisioning operations. Hence, they do no meet the two evaluation requirements.

Table 1 provides a summary of the critical evaluation provided in this section.

7 CONCLUSIONS AND FUTURE WORK

This paper presents M-COAPS API for component-based applications provisioning across several PaaS offerings. This approach uses and extends COAPS API that relies on a generic OCCI-based description model. M-COAPS provides a concrete solution to struggle the existing lock-in in the PaaS market and enables developers to deploy and manage their applications' components easily and seamlessly

in multi-PaaS environment. For example, the developers henceforth can use a unique and common descriptor when deploying the components in heterogeneous PaaS solutions. Moreover, they can manage (e.g. starting) the components in the same way whatever the target PaaS is. The implementation of the motivating use case was performed to validate the proposed approach and demonstrate its feasibility. Such approach can be considered as a step forward to achieve PaaS cooperation and federation. It also provides concrete perspective to enable cloud end-user applications portability. Furthermore, unlike the reviewed related work, M-COAPS and its associated OCCI model do not impose any integration contraints and/or modifications from the providers side which makes easy its adoption.

As next steps in the future, the integration of this solution to the open source CompatibleOne cloud broker is contemplated. In addition, the design and the implementation of a placement algorithm that can be integrated is also considered. Such algorithm will provide M-COAPS with the optimal components placement plan when deploying an application . Placement decisions will be based on well-defined requirements (e.g. cost, latency). Finally, the inclusion of migration capability is planned. It will enable moving components from one PaaS to another during runtime. The moving decisions can be triggered by events such as a rate change in the hosting PaaS.

REFERENCES

(2016). COAPS API Web Page. http://www-inf.it-sudparis.eu/SIMBAD/tools/COAPS/.

Ardagna, D., Di Nitto, E., Casale, G., Petcu, D., Mohagheghi, P., Mosser, S., Matthews, P., Gericke, A., Ballagny, C., D'Andria, F., et al. (2012). Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, pages 50–56. IEEE Press.

Baur, D., Wesner, S., and Domaschka, J. (2015). *Advances in Service-Oriented and Cloud Computing: Workshops of ESOCC 2014, Manchester, UK, September 2-4, 2014, Revised Selected Papers*, chapter Towards a Model-Based Execution-Ware for Deploying Multi-cloud Applications, pages 124–138. Springer International Publishing, Cham.

Brogi, A., Fazzolari, M., Ibrahim, A., Soldani, J., Carrasco, J., Cubo, J., Durán, F., Pimentel, E., Di Nitto, E., and D Andria, F. (2015). Adaptive management of applications across multiple clouds: The seaclouds approach. *CLEI Electronic Journal*, 18(1):2–2.

Buyya, R. and Barreto, D. (2015). Multi-Cloud Resource Provisioning with Aneka: A Unified and Integrated Utilisation of Microsoft Azure and Amazon EC2 Instances. *arXiv preprint arXiv:1511.08857*.

Cunha, D., Neves, P., and Sousa, P. (2014). PaaS manager: A platform-as-a-service aggregation framework.

D'Andria, F., Bocconi, S., Cruz, J. G., Ahtes, J., and Zeginis, D. (2012). Cloud4SOA: multi-cloud application management across PaaS offerings. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 407–414. IEEE.

Dash, D., Kantere, V., and Ailamaki, A. (2009). An Economic Model for Self-Tuned Cloud Caching. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 1687–1693.

Di Martino, B. (2014). Applications portability and services interoperability among multiple clouds. *IEEE Cloud Computing*, (1):74–77.

Ferry, N. (2015). MODAClouds evaluation report–Final version.

Grossman, R. (2009). The Case for Cloud Computing. *IT Professional*, 11(2):23–27.

Hadley, J., Elkhatib, Y., Blair, G., and Roedig, U. (2015). Multibox: lightweight containers for vendor-independent multi-cloud deployments. In *Embracing Global Computing in Emerging Economies*, pages 79–90. Springer.

Kamateri, E., Loutas, N., Zeginis, D., Ahtes, J., D'Andria, F., Bocconi, S., Gouvas, P., Ledakis, G., Ravagli, F., Lobunets, O., et al. (2013). Cloud4SOA: A semantic-interoperability paaS solution for multi-cloud platform management and portability. In *Service-Oriented and Cloud Computing*, pages 64–78. Springer.

Kirkham, T. (2013). Keith Je rey. PaaSage Project. Model Based Cloud Platform Upperware. Initial Architecture Design. Technical report, Technical report, November.

Mancini, E. P., Rak, M., and Villano, U. (2009). Perfcloud: Grid services for performance-oriented development of cloud computing applications. In *Enabling Technologies: Infrastructures for Collaborative Enterprises, 2009. WETICE'09. 18th IEEE International Workshops on*, pages 201–206. IEEE.

Martin-Flatin, J. (2014). Challenges in Cloud Management. *IEEE Cloud Computing*, (1):66–70.

Mell, P. and Grance, T. (2009). The NIST definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50.

Metsch, T. and Edmonds, A. (2011). Open Cloud Computing Interface - RESTful HTTP Rendering. Technical report.

OCCI (2016). OCCI. *Open Cloud Computing Interface*. http://occi-wg.org/.

Pahl, C. (2015). Containerization and the PaaS cloud. *IEEE Cloud Computing*, (3):24–31.

Paraiso, F., Haderer, N., Merle, P., Rouvoy, R., and Seinturier, L. (2012). A federated multi-cloud PaaS infrastructure. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 392–399. IEEE.

Paraiso, F., Merle, P., and Seinturier, L. (2016). socloud: a service-oriented component-based paas for managing portability, provisioning, elasticity, and high availability across multiple clouds. *Computing*, 98(5):539–565.

Petcu, D., Di Martino, B., Venticinque, S., Rak, M., Máhr, T., Lopez, G. E., Brito, F., Cossu, R., Stopar, M., Šperka, S., et al. (2013). Experiences in building a mOSAIC of clouds. *Journal of Cloud Computing*, 2(1):1–22.

Ranjan, R. (2014). The Cloud Interoperability Challenge. *Cloud Computing, IEEE*, 1(2):20–24.

Satzger, B., Hummer, W., Inzinger, C., Leitner, P., and Dustdar, S. (2013). Winds of change: From vendor lock-in to the meta cloud. *IEEE Internet Computing*, (1):69–73.

Sellami, M., Yangui, S., Mohamed, M., and Tata, S. (2013a). Open Cloud Computing Interface-Application. Technical report, Tech. Rep., 2013.[Online]. Available: http://www-inf. int-evry. fr/SIMBAD/tools/OCCI/occi-application. pdf 86, 94.

Sellami, M., Yangui, S., Mohamed, M., and Tata, S. (2013b). PaaS-independent Provisioning and Management of Applications in the Cloud. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 693–700. IEEE.

Walraven, S., Van Landuyt, D., Rafique, A., Lagaisse, B., and Joosen, W. (2015). PaaSHopper: Policy-driven middleware for multi-PaaS environments. *Journal of Internet Services and Applications*, 6(1):1–14.

Wei, Y., Sukumar, K., Vecchiola, C., Karunamoorthy, D., and Buyya, R. (2011). Aneka cloud application platform and its integration with windows azure. *arXiv preprint arXiv:1103.2590*.

Yangui, S., Marshall, I. J., Laisné, J., and Tata, S. (2014). CompatibleOne: The Open Source Cloud Broker. *J. Grid Comput.*, 12(1):93–109.

Yangui, S., Mohamed, M., Sellami, M., and Tata, S. (2013). Open Cloud Computing Interface-Platform. Technical report, Tech. Rep., 2013.[Online]. Available: http://www-inf. int-evry. fr/SIMBAD/tools/OCCI/occi-platform. pdf 86, 94.

Yangui, S. and Tata, S. (2016). An OCCI compliant model for paas resources description and provisioning. *Comput. J.*, 59(3):308–324.

Zeginis, D., D'Andria, F., Bocconi, S., Gorronogoitia Cruz, J., Collell Martin, O., Gouvas, P., Ledakis, G., and Tarabanis, K. A. (2013). A user-centric multi-PaaS application management solution for hybrid multi-Cloud scenarios. *Scalable Computing: Practice and Experience*, 14(1).

SCIENCE AND TECHNOLOGY PUBLICATIONS