Accepted Manuscript

Trans-Cloud: CAMP/TOSCA-based Bidimensional Cross-Cloud

Jose Carrasco, Francisco Durán, Ernesto Pimentel

PII: S0920-5489(17)30318-5
DOI: 10.1016/j.csi.2018.01.005
Reference: CSI 3265

To appear in: *Computer Standards & Interfaces*

Received date: 22 August 2017
Revised date: 23 January 2018
Accepted date: 31 January 2018

Please cite this article as: Jose Carrasco, Francisco Durán, Ernesto Pimentel, Trans-Cloud: CAMP/TOSCA-based Bidimensional Cross-Cloud, *Computer Standards & Interfaces* (2018), doi: 10.1016/j.csi.2018.01.005

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

ACCEPTED MANUSCRIPT

Highlights

- A novel notion of trans-cloud management of applications
- An associated methodology for the description and operation of cloud applications
- A proposal for trans-cloud deployment (for IaaS and PaaS, by different vendors)
- A provider-agnostic TOSCA-based model to specify the topology of applications
- CAMP concepts (Apache Brooklyn) for the deployment and management of such models

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

Trans-Cloud: CAMP/TOSCA-based Bidimensional Cross-Cloud

Jose Carrasco, Francisco Durán, and Ernesto Pimentel

Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
Email: {josec,duran,ernesto}@lcc.uma.es

Abstract

The diversity in the way in which different cloud providers offer their services, give their SLAs, present their QoS, or support different technologies complicates the portability and interoperability of cloud applications, and favors vendor lock-in. Trying to solve these issues, we have recently witnessed the proposal of unified APIs for IaaS services, unified APIs for PaaS services, and a variety of cross-cloud application management tools. We go one step further in the unification of cloud services, building on the TOSCA and CAMP standards, with a proposal in which the management of IaaS and PaaS services, possibly offered by different providers, are integrated into a unified interface. The TOSCA standard is used for the definition of portable models describing the topology of cloud applications and the required resources in an agnostic, providers-and-resources-independent way. Based on the CAMP standard, we abstract from the particularities of specific providers. Indeed, to change the service on which any of the modules of an application is to be deployed, whether it be IaaS or PaaS, we just need to change its target location by picking from the catalog of supported locations. We provide insights into our implementation on Apache Brooklyn, present a non-trivial case study that illustrates our approach, and show some experimental results.

Keywords: Cloud applications, multi-deployment, cross-deployment, Apache Brooklyn, TOSCA, CAMP

ACCEPTED MANUSCRIPT

ACCEPTED MANUSCRIPT

1. Introduction

In recent years, Cloud Computing [1] has experienced a growth in the demand for its services, with a significant increase in its popularity. The Cloud promotes access on demand to a large number of resources throughout three main service models, namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [2], which allow cloud providers to offer a set of useful features for current IT requirements of the sector, among which we find scalability and elasticity.

Given the interest in this computing model, vendors such as Google, Amazon, and Microsoft have implemented solutions by developing their own cloud service layers, with custom APIs that expose their resources. Most of these providers offer a set of similar services as regards functionality, but developed according to their own specifications. For example, each supplier specifies its own Service Level Agreement (SLA), supports a concrete set of technologies, or provides ad-hoc services. The proliferation of these solutions has also increased the number of issues to be addressed in cloud computing, mainly related to the diversity of providers and their solutions, triggering the *vendor lock-in* problem (see, e.g., [3, 4]), and hampering portability and interoperability in the definition and usage of services.

Vendor-lock-in-related problems are aggravated by the differences between IaaS and PaaS. Both kinds of services allow developers to deploy applications and store and manage data, but they offer different mechanisms for the configuration and use of their functionalities. In IaaS, developers are offered an exhaustive control of the infrastructure, with control over virtual machine (VM) provisioning, management of networks and operating system, etc. With PaaS, users loose capability for low level configuration in exchange for gaining preconfigured environments where they can deploy their applications and easily take advantage of useful features as on-demand elasticity and scalability. Thus, each abstraction level provides services to reach similar goals, but they have to be managed by means of distinct mechanisms and interfaces which users must

ACCEPTED MANUSCRIPT

know and understand.

In order to mitigate this heterogeneity and find a vendor-agnostic solution, independent tools and frameworks have emerged with the goal of integrating, under a single interface, the services of multiple public and private providers (see, e.g., [5], [6] and [7]), or providing decentralised deployment environments (e.g., [8] and [9]). Over such mechanisms, most of these solutions support the building of models of application topologies, including dependencies and used resources, independently of the providers in which services will be executed. Thus, they offer a portable and interoperable environment where developers can describe their systems and select the resources that better fit their requirements, without worrying about technical details of the services' use, and focusing on the required features.

In a very short time, these platforms have evolved according to the way in which users can take advantage of integrated cloud services to expose and run their systems. Terms such as *multi-cloud* [10], *cross-cloud* [11], *federated clouds* [12], or *inter-clouds* [13] have been used for deployment platforms with the ability to distribute modules of an application using services from different providers. The main differences between these approaches lie in the different ways of handling the connections between modules deployed on different platforms. However, in all these attempts, platforms allow operating simultaneously with a single level of service to deploy applications, i.e., all the components of an application are deployed either at the IaaS level or all at the PaaS level (see, e.g., [14], [15] and [11]).

With the goal of unifying cloud services, we propose a second dimension in which deployment tools integrate all three levels under a CAMP-based single interface. Then, this will allow developers to deal with IaaS and PaaS differences to deploy their applications, combining services offered by providers at any of these levels. Following the evolution in terminology, *multi-/cross-/inter-cloud*, which shows the way in which cloud services offered by different vendors can be integrated and managed, we envision *trans-cloud* management tools without the limitations we currently face. The idea behind trans-cloud is to be able to build

ACCEPTED MANUSCRIPT

our applications by using available services and resources offered by different providers, at the IaaS, PaaS or SaaS levels, indistinctly and in combination, using virtual machines, containers and other services, according to our needs and preferences.

Which service to select from among the multitude of cloud services is still a challenge for users (see, e.g., [3], [9], [16] and [17]). Furthermore, once a service has been selected, we need mechanisms to ensure that the chosen cloud provider is delivering the promised computing resources (see [17] and [18]). The decision is indeed non-trivial, and the context and knowledge may change as time passes. We may decide today to use a PaaS provider for a particular module because it is more cost effective, or because it requires less management effort, but tomorrow our needs or business model may require more control over our virtual machines, e.g., for a better integration with our enterprise's infrastructure, or because we need to increase the security level of our services. This is problematic, since changes in these decisions require development effort (see Section 4.1 and, e.g., [19] and [20]). Changing from a PaaS provider to another may already require a significant effort, and moving from IaaS to PaaS or vice versa may be simply prohibitive. However, it may be unavoidable over time, because of change in the offered services, prices, security policies, or simply because a provider just stops providing its services. [1] We propose using a provider-agnostic TOSCA-based model to specify the topology of applications and their required resources, indistinctly using IaaS and PaaS services. Applications' models thus developed can then be deployed and managed using a CAMP-based approach.

In this paper, we present our CAMP-TOSCA-based proposal towards trans-cloud application management, which goes one step further in the management of cloud services by unifying the deployment of components using IaaS and PaaS of multiple vendors. We have developed a prototype by extending Apache

[1] Several services (indeed complete platforms), such as DotCloud or CloudBees, have been shut down by their providers.

ACCEPTED MANUSCRIPT

Brooklyn, [2] a CAMP-compliant system for the modeling, blueprinting, deployment and management of cloud applications. Brooklyn offers a complete catalog to describe application components and deploys them using compatible locations of among a wide range of IaaS supported providers, such as AWS or Softlayer, by means of a CAMP-based interface. For the present proposal, we have extended the aforementioned interface by adding PaaS services support, accomplishing the trans-cloud approach which integrates the unified management of different abstraction levels.

Having an agnostic model of our application may greatly simplify migration, or simply decision change. Indeed, with our approach, each application component may be deployed at one level or the other just by changing its location policy [21]. By making the change from IaaS to PaaS, and vice versa, so simple for each module in our applications, we are contributing to a real adaptation to our needs by minimizing the effort by the user. The underlying management tool, Brooklyn, will be in charge of the required provisioning and interoperation. Notice that other relevant aspects, for instance, security features of application components, such as servers and bound cloud resources like virtual machines, can be configured from Brooklyn. For example, the SSL protocol can be enabled and configured for a JBoss server and the necessary ports can be open in the location where it is running.

Our main contributions may be summarised as follows:

(i) We propose a novel notion of trans-cloud management of applications, and an associated methodology for the description and operation of cloud applications.

(ii) We make a detailed presentation of our proposal for trans-cloud deployment, providing support for the management of applications whose components may indistinctly be deployed on IaaS and PaaS services provided

[2] Apache Brooklyn was originally developed by CloudSoft Corp. (`http://www.cloudsoft.io/`), but has recently become an Apache project (`https://brooklyn.apache.org/`).

ACCEPTED MANUSCRIPT

by different vendors. [3]

(iii) We propose using a provider-agnostic TOSCA-based model to specify the topology of applications and their required resources, indistinctly using IaaS and PaaS services.

(iv) We propose using CAMP concepts to build a unified API to use cloud services, in particular its Apache Brooklyn implementation, to support the deployment and management of such models. We present the main architectonical decisions for our extension of Brooklyn and an overview of its design.

(v) We illustrate the details of the approach and its use in practice on a non-trivial case study based on a real application.

(vi) We evaluate our tool by analyzing both the effort required for changes in deployment decisions and the deployment times.

Our Brooklyn-based trans-cloud tool is publicly available at `https://github.com/scenic-uma/brooklyn-dist/tree/trans-cloud`. Some additional documentation, examples and evaluation are available at `https://trans-cloud.firebaseapp.com`.

The rest of the paper is structured as follows. Section 2 describes our approach for trans-cloud management, and provides insights into its design and implementation. We introduce in Section 2.2 our running example, on which all concepts and ideas presented in the paper will be illustrated. Then, we explain some preliminaries on the technology and standards we rely on in Sections 2.3-2.5, which helps us to introduce our extension of Brooklyn in Section 3. Section 4 illustrates the practical use of the tool and discusses some experimental results. In Section 5, we compare our proposal to related work. Finally, Section 6 concludes the paper and presents some future work.

[3] A preliminary statement of the proposal was presented in [22].

ACCEPTED MANUSCRIPT

2. Trans-Cloud Management

The vendor lock-in problem (see, e.g., [8] and [4]) affects all stages of cloud applications, including their design and operation. Application developers must know the features of the services to be used, and have a deep knowledge of providers' APIs. To minimize the knowledge required, we can find solutions based on the use of standards, such as CAMP [23] or TOSCA [24], modeling (see e.g., [8] and [9]), unified APIs, such as jclouds[4] or Nucleus [25], or container-based solutions like Docker[5] or Kubernetes.[6] These solutions are indeed very different, e.g., whereas jclouds provides a cloud agnostic API library to provision and configure secure communications with cloud virtual machines, a container-based solution like Docker allows describing and deploying applications and their dependencies through containers on machines with the corresponding engines.

2.1. Highlights of the proposal

Following CAMP ideas, our proposal goes a step further in the development of common APIs by unifying IaaS and PaaS services from different providers under the same interface, and using the TOSCA standard for the agnostic specification of applications' components and interdependencies. With such a proposal, we do not only reduce the need for vendor-specific knowledge to develop our applications for designing, deploying and operating them, but in fact, such a homogenized API greatly improves portability and interoperability as well.

Figure 1 presents an overview of our proposal. Given a TOSCA YAML description of the components and their relationships, an extended version of the Apache Brooklyn implementation of CAMP is used to manage applications. TOSCA YAML descriptions are processed by Brooklyn, which is then responsible for performing the requests and managing the services necessary for the deployment of each application's component and also for establishment of the specified connections.

[4]Apache jclouds: `https://jclouds.apache.org`.
[5]Docker: `https://www.docker.com`.
[6]Kubernetes: `http://kubernetes.io`.

ACCEPTED MANUSCRIPT

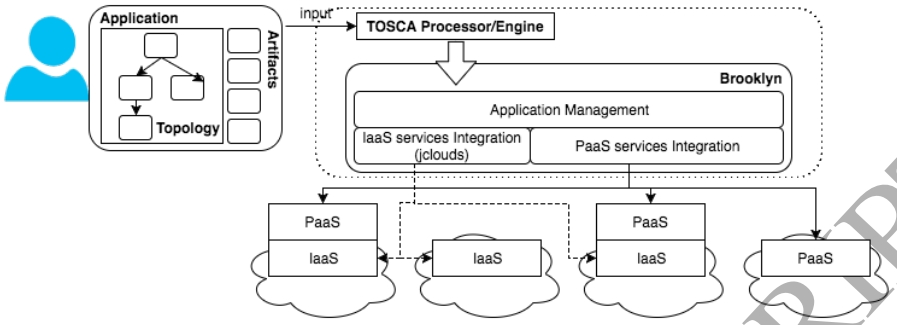


Figure 1: Trans-cloud deployment

Apache Brooklyn provides an API for the management of IaaS cloud services for a great number of providers and establishes a life cycle for the management of services and applications. However, trans-cloud proposes a unified management of IaaS and PaaS services. Thus, we have extended this API with features for the management of PaaS services of platforms based on Cloud Foundry interface, such as Pivotal or Bluemix,[7] providing a homogeneous access to IaaS and PaaS services. Our extension to provide support for PaaS services builds on the genericity and flexibility of Brooklyn's API, which has the independency between application descriptions and cloud services used in their operation as one of its goals.

Brooklyn only supports IaaS applications, and the target locations it handles are intended for specific datacenters on which the usual IaaS life cycle is to be operated. However, components deployed on PaaS follow a different life cycle, since there is no need, e.g., for the provisioning of virtual machines or low-level configurations. Our proposal includes a new PaaS life cycle to allow this kind of

[7] Cloud Foundry (`https://docs.cloudfoundry.org`) is an open-source PaaS which defines a baseline of common capabilities to promote cloud portability by means of an API. Several companies have built their platforms upon CloudFoundry, e.g., Pivotal (`https://run.pivotal.io`), Bluemix (`https://console.ng.bluemix.net`), or HPE Helion Stackato (`https://www.hpe.com/us/en/solutions/cloud.html`). Each of these platforms develops its own customization, but all of them use the CloudFoundry interface to offer the services.

ACCEPTED MANUSCRIPT

services to be managed by the unified API. Moreover, although TOSCA allows the specification of locations for the deployment of applications, this information is specified in the node templates associated with specific vendors, which prevents any form of provider independence in their specification. Following the Brooklyn-TOSCA initiative (see Section 2.4), we use agnostic TOSCA specifications, where the only information to be provided for deployment is the location of each component, which can be supplied in very late stages of the design, or even automatically if needed, still satisfying all the requirements specified in the TOSCA specifications.

Given the locations of each of the modules in our application included in the YAML files to be used by Brooklyn, the extended Brooklyn generates, for each component, an entity which is 'decorated' with IaaS or PaaS behavior depending on whether the location corresponds to a IaaS or PaaS service. Thus, given a TOSCA YAML description with the appropriate location specifications, the Brooklyn system will create suitable entities to manipulate each module.

When an entity is started—as part of the application deployment process—a cloud management strategy is found and added in accordance with the target provider. Then, depending on the entity's location, the aforementioned strategy will select the corresponding IaaS or PaaS life cycle (see Section 2.4), which injects IaaS or PaaS behavior, depending on the case, into the entity to allow it to work using the target location services. Life cycles describe the generic key steps required to manage an entity inside a location kind, but they do not know the final operation logic to carry out during the deployment of the entity on the expected cloud environment. Therefore, life cycles load and inject into an entity the appropriate IaaS/PaaS driver that contains specific knowledge for deploying and managing such an entity on such a provider. Thus, an entity has to define an appropriate driver, which contains the required behavior, for each kind of location that should be supported. In short, by using the appropriate IaaS/PaaS life cycle and drivers associated to each specific entity, the extended Brooklyn is able to handle trans-cloud applications.

ACCEPTED MANUSCRIPT

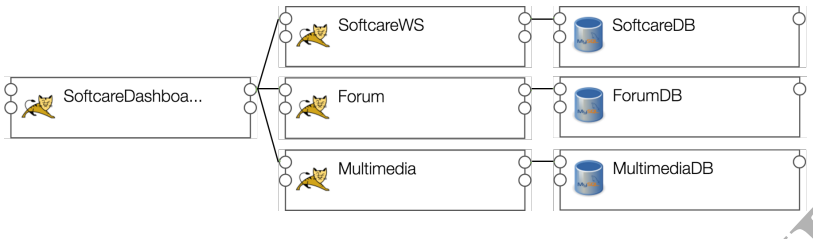


Figure 2: Brooklyn-TOSCA Softcare's topology

2.2. The Softcare case study

The use of our system, and the presentation of the technologies we rely on, will be illustrated in the Softcare case study. The case study corresponds to an application for the social inclusion of elderly people and for the management of their medical problems. The application is called Softcare, and was originally developed by Atos Spain [26]. Softcare is a cloud-based clinical, educational, and social application, based on state-of-the-art technology, that provides an innovative and integrated solution with the following main features: home as care-environment through the provision of user-friendly ICT tools for frequent, unobtrusive monitoring; risk assessment and early detection of deterioration symptoms; high-quality interaction between doctors, social workers and elderly people; monitoring and follow-up of the elderlies' progress; and self-care and self-management of chronic conditions, through the development of social networking, educational tools and dietary guidelines.

As depicted in Figure 2, the application is composed of seven modules: four web modules over respective Tomcat servers, namely SoftcareDashboard, SoftcareWS, Multimedia, and Forum (note the Tomcat icons), and three MySQL databases, namely SoftcareDB, MultimediaDB, and ForumDB (note the database icons). The SoftcareDashboard component provides the main graphical user interface, which depends on the Forum, Multimedia and SoftcareWS modules. Forum adds a forum service to the platform, Multimedia manages the offered multimedia content, and SoftcareWS contains the application's business logic. The databases ForumDB, MultimediaDB and SoftcareDB store, respectively, the forum's messages, the multimedia content, and the rest of the application's data.

ACCEPTED MANUSCRIPT

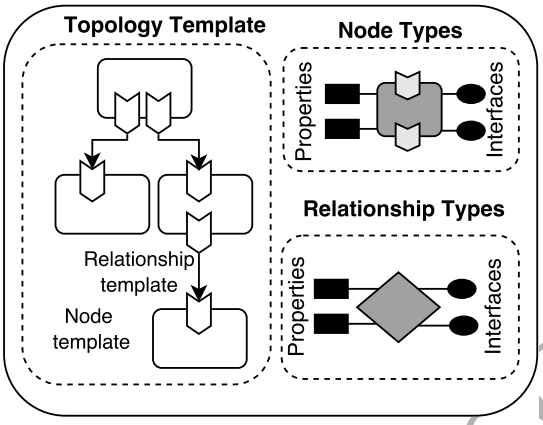


Figure 3: TOSCA Service Template elements

2.3. The TOSCA standard

Topology and Orchestration Specification for Cloud Applications (TOSCA) [24] is an OASIS standard language to describe cloud applications, the corresponding services, and their relationships using a service topology. In TOSCA, we can also provide the description of procedures to manage services using orchestration processes, by using plans which specify the calls to components' operations to instantiate the structure of the topology. In TOSCA, services are specified through *service templates*, as illustrated in Figure 3, by using node and relationship templates. These templates represent concrete components of the application at hand, of corresponding node and relationship types. A *node type* expresses the capabilities, requirements, properties and interfaces of the services. *Relationship types* are used to specify relationships between the elements of the system by means of a set of relations, which determine the connections and dependencies between components, taking into account their properties and interfaces. Type interface definitions include operations to describe their behavior. These operations are implemented by a set of *implementation artifacts*, which contain the necessary logic for executing the operations.

TOSCA provides two different syntaxes for the specification of the aforementioned concepts, one based on XML and another based on YAML [27]. Although

ACCEPTED MANUSCRIPT

both of them allow using the full power of TOSCA, we use the YAML-based language, as Brooklyn-TOSCA does, to model our applications, because we believe that the YAML profile offers a more accessible, concise and readable syntax than the XML one.

Figure 4 shows the topology of our Softcare case study as defined using TOSCA in Alien4Cloud. [8] As in the topology in Figure 2, we can observe that the application is composed of the seven modules previously described. However, we can see in this diagram how, in TOSCA, we specify the target platforms with appropriate *node templates*. In this case, each component is hosted on its own virtual machine, to be deployed on AWS or SoftLayer. Note that, although ready for multi-deployment, the deployment is originally specified using IaaS services. We will see in following sections how configuring the deployment by means of trans-cloud to uses indistinctly both, IaaS and PaaS services. Note also that here we just show the topology, although all details prescribed by TOSCA are specified using the Alien4Cloud template editors. Of course, we could use any other TOSCA editor that generates YAML TOSCA specifications, or even write them by hand and then use some validator such as Ubicity.[9]

2.4. CAMP, TOSCA, and Apache Brooklyn

CAMP is an OASIS standard [23] that focuses on the deployment, management and monitoring of cloud applications, regardless of the platform used. Basically, CAMP proposes an API for managing public and private cloud applications, introducing concepts such as *entities*, *locations*, *drivers* or *policies*.

[8] Alien4Cloud (`http://alien4cloud.github.io`) is an open project whose aim is to help in the design of applications and in the collaboration during the design phase. The Alien4Cloud platform offers a GUI for managing TOSCA topologies (blueprints) and monitoring the deployment phase. With Alien4Cloud, we can specify semantics and configurations by providing properties for each template element. Node operations can be defined and implemented by using implementation artifacts, which are completely integrated in the application's deployment life cycle. Alien4Cloud integrates Cloudify as its cloud service orchestrator, what allows Alien4Cloud to deploy applications using all the providers that are supported by Cloudify.

[9] Ubicity: `https://ubicity.com`.

ACCEPTED MANUSCRIPT

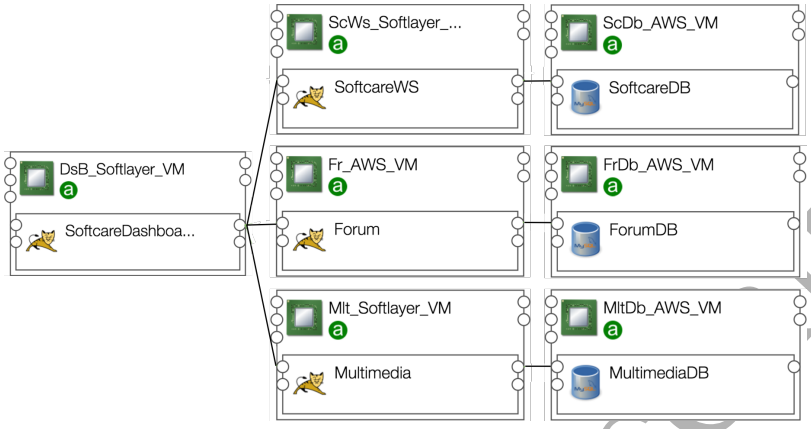


Figure 4: Softcare's topology with Alien4Cloud

These concepts play a relevant role both in the architecture and use of the Apache open-source-project Brooklyn, which quite closely follows the OASIS CAMP standard. [10]

Brooklyn is a multi-cloud application management platform that can manage the provisioning and deployment of cloud applications, can monitor applications' health and metrics, and resolve and handle the dependencies between application components during the application's life cycle. It enables cross-computing features through a unified API built over jclouds to manage IaaS services offered by various providers. The tool offers a REST API and a GUI that enables a single-click deployment of applications across multiple machines, locations and clouds, and can continuously optimize the running application to ensure ongoing compliance with specified policies. With Brooklyn's API, applications may be agnostically modeled, independently of providers' specificities. For instance, to specify a Tomcat server, we would use a TomcatServer entity, irrespective of the provider later chosen for its deployment.

Once the components of an application have been agnostically defined, its

[10] Detailed information on the use of standards in Brooklyn can be found at `https://brooklyn.apache.org/learnmore/theory.html`.

ACCEPTED MANUSCRIPT

deployment requires the specification of the provider or providers whose services are to be used. For this purpose, CAMP uses the concept of *location*, which specifies, in a generic, flexible and extensible way, where components are to be deployed, and those cloud services or resources used or managed. Brooklyn provides access to all locations supported by jclouds, at IaaS level.

The Brooklyn-TOSCA plug-in[11] adds to Brooklyn the capacity for deploying and managing applications that are modeled using Brooklyn's API components expressed as TOSCA concepts. Brooklyn-TOSCA uses Alien4Cloud technology to process TOSCA blueprints (topologies, CSAR files, etc.) and generate the necessary Brooklyn objects (entities and locations) to represent applications, which are then managed by Brooklyn to deploy them.

Brooklyn-TOSCA introduces new mechanisms to describe target locations. Following Brooklyn's API, locations are expressed as TOSCA policies, with the aim of creating agnostic topology templates, independent of the selected providers. Moreover, this approach allows a location to be added individually to each application module, ensuring the cross-cloud features that are offered by Brooklyn. With this mechanism, the Softcare's topology in Figure 4 may alternatively be given as in Figure 2, where the corresponding locations are specified as location policies.

2.5. Brooklyn's Architecture

In this section, we present some details on Brooklyn's architecture and operation. The complexity of Brooklyn makes it really challenging to present all the details of its architecture. Instead, for our explanation of Brooklyn, as for the presentation of our extension in Section 3, we will focus on a small set of classes and interfaces, as well as the relationships between them, that gives an insight into its design, enough to understand the extension we have developed. Figure 5

[11] Brooklyn-TOSCA (`https://github.com/cloudsoft/brooklyn-tosca`) is an open project principally being developed by CloudSoft and FastConnect, the main developers of Brooklyn and Alien4Cloud, respectively.

ACCEPTED MANUSCRIPT

depicts a few classes and interfaces as in the actual Java implementation of the extended Brooklyn. For a better understanding, specific classes for Tomcat and MySQL are considered, similar classes for other entities in the catalog are part of the architecture. Although modifications were performed on many different parts of the implementation of Brooklyn, we focus here on the elements added in our extension. These new elements are depicted in blue to make them more easily identifiable. Further details on Brooklyn and its implementation may be found in its official documentation [28].

2.5.1. Brooklyn entities

In Brooklyn, *entities* represent agnostic pieces of software, such as cloud resources, applications, and application modules (servers, databases, etc.), with the purpose of managing their deployment and operation.

An entity represents the core of any deployable artifact, so it can be extended to model new concrete software pieces, such as web servers, DBMSs, etc. For illustration purposes, Figure 5 shows interfaces TomcatServer and MySqlNode, which model, respectively, the software modules Tomcat server and MySQL server. Of course, Brooklyn also provides concrete software representations for JBoss servers, clusters, and many others. As shown in the diagram, the Entity interface is extended by the SoftwareProcess interface, which is described as the basis of any software process, that is, a piece of software to run somewhere. This interface is extended for each specific piece of software.

Following the interface-based architectural pattern, Brooklyn provides classes implementing each of these interfaces. For example, TomcatServerImpl implements TomcatServer.

The management of entities is performed through their provided endpoints, to know their status (*sensors*) and operate on them (*effectors*). Sensors allow entities to expose data. For example, a server can offer the number of requests per second or a cluster can offer the current number of server instances it handles. Effectors are used to model the actions that can be applied on an entity. For example, effectors are used for starting/stopping an entity in a location.

ACCEPTED MANUSCRIPT

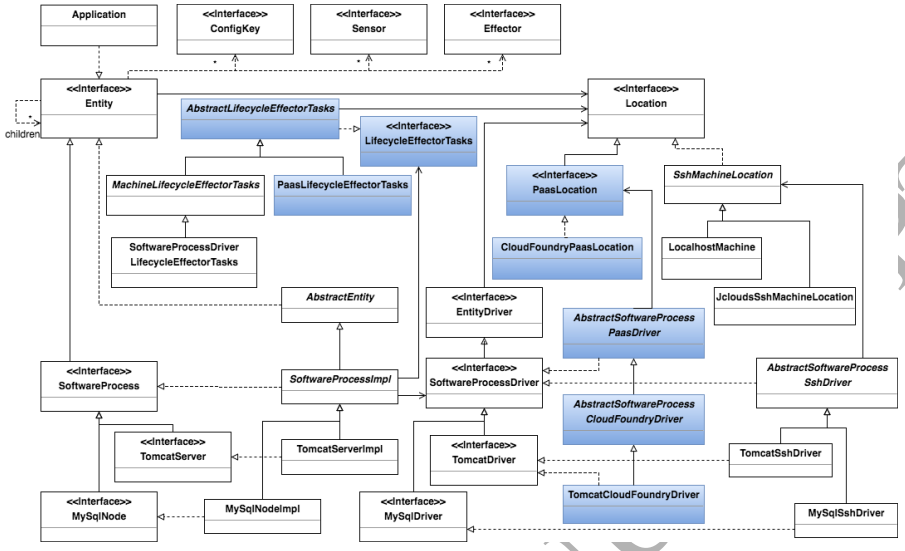


Figure 5: Extended Brooklyn's architecture (added elements in blue)

Effectors and sensors can be injected into an entity at runtime, following the classical Abstract Factory and Strategy patterns, and allowing a useful elasticity. These mechanisms are very helpful for improving the extensibility and flexibility of entities. In a similar vein, configuration keys are used to model the configuration of entities. Configuration keys and sensors allow the specification and establishment of topologies' relations. For example, a web application could require a config key to configure a database address, which could be provided by another entity (e.g., a DBMS) through a sensor.

2.5.2. Locations

Each entity has an associated *location*, which defines where the entity is or is to be deployed. A location represents a cloud service or a deployment resource, and contains the necessary information to access and to use the referenced elements. Thus, a local VM or an AWS datacenter will be modeled by different locations. Furthermore, these classes have to provide the necessary mechanisms to manage the target services and resources. For example, an AWS EC2 location should contain the necessary methods to connect to the specific AWS

ACCEPTED MANUSCRIPT

datacenter and to use the necessary services, such as creating and launching a VM, and configuring network resources.

On the right-hand side of Figure 5, we show the Location hierarchy. Locations represent the cloud services and providers where entities are to be deployed, such as remote or local virtual machines, remote hosts (MachineLocation), public and private datacenters, and so forth.

For instance, the SshMachineLocation class represents machines that can be managed using the ssh protocol, such as a VM. We find different specializations, such as LocalhostMachine which represent the machine where Brooklyn is running, and JcloudsSshMachineLocation which integrates jclouds-based locations.

2.5.3. Drivers

The location of a SoftwareProcess defines where the modeled piece of software will be deployed and run. For example, if a JBossServer has a LocalhostMachine as location then it will be deployed on the machine where the Brooklyn instance is running. Moreover, with this approach, entities may operate on their corresponding locations if they need to carry out some of their tasks.

To avoid requiring an intimate knowledge or management of any concrete location, these implementations have been developed generically by delegating the location management to another hierarchy of elements: the *drivers*. A driver contains the code to allow a specific entity to operate over a specific location. This scheme allows the location-independent definition of entities. Moreover, it is very useful if a new kind of location has to be supported, since in that case just a new driver will be required, which will be able to manage the entity in the new location. In other words, entities use a Strategy pattern to delegate the necessary behavior to carry out the location's management.

Class SoftwareProcessImpl implements the DriverDependentEntity interface, which determines that an entity must have a driver implementing SoftwareProcessDriver. To enable the management of different kinds of drivers, each entity declares a driver interface that the given driver will implement. For illustrative purposes, Figure 5 shows part of the sub-hierarchy of locations based on SshMachineLoca-

ACCEPTED MANUSCRIPT

tion.

2.5.4. Entities' life cycles

Although a driver contains the necessary knowledge to deploy and run an entity on a location, drivers operate in the context of an entity, which means that drivers expect configured locations to operate on. For example, a MySqlNode will require the port 3306, which is the one normally used by MySQL servers, open. Machine-based location management is extracted to the abstract class MachineLifecycleEffectorTasks, which defines the life cycle to carry out. This life cycle is responsible for different tasks, such as configuring the machines, adding the necessary information to the entities, and even machine provisioning. This class focuses on machine management, but SoftwareProcess will require it to form part of this life cycle in order to be deployed and run. Then, class MachineLifecycleEffectorTasks is extended by class SoftwareProcessDriverLifecycleEffector, which specializes the machine life cycle by adding driver management (IaaS life cycle). Thus, we can say that life cycles handle the different entities' processes, because they manage the locations, the drivers, and even the entities themselves, in a systematic and generic way, using specific effectors.

During the initialization of an entity, the operations of its effectors are modified by injecting appropriate body effectors, which are defined by the life cycle, fixing the entity's behavior to manage a certain kind of location. Thus, during the initialization phase of an instance of SoftwareProcessImpl, its effectors' default behavior is overridden with the appropriate effector bodies of an IaaS-based life cycle (SoftwareProcessDriverLifecycleEffector-based), setting up the entity to manage IaaS locations. IaaS life cycles assume that the entity implements the Startable interface, so the start, stop and re-start effectors executed by the life cycle will be those injected into the corresponding SoftwareProcessImpl entity. The (start routine of the) start effector is in charge of managing the entity's location and driver, getting the necessary flags to create and configure the location, opening ports, etc. The start effector is also in charge of launching the entity's children. Once its children have been started, the entity's driver is executed,

ACCEPTED MANUSCRIPT

which takes care of installing, configuring and launching the entity's software in its location. Along this process, the entity's sensors are connected. The stop effector is in charge of stopping the entity, disconnecting its sensors, stopping its children, and releasing machines if necessary. The re-start effector uses the previous effectors to stop and start again the entity but without releasing the machine.

Although effectors are injected during the initialization phase of an entity's life cycle, the Startable interface defines that the start effector takes a location as parameter. This location will be used to start the entity on it. As a result, an entity does not fix the specific location where it will run until the start effector is invoked. When SoftwareProcessImpl-based entities are initialized, their effectors are replaced by SoftwareProcessDriverEffectorTasks effector bodies, which are intended to manage machine locations. This means that, although SoftwareProcessImpl entities assume a MachineLocation-based management, the specific target location (JcloudsSshMachineLocation, LocalHostLocation, etc.) to use will not be specified until the start effector is called. This mechanism will be key for the agnostic definition of entities in our extension, since we will use this possibility, not only to change the specific location, but also to change the life cycle.

In summary, Brooklyn's IaaS life cycle is responsible for: (1) reserving and configuring the necessary resources, as for instance provisioning virtual machines, opening ports, etc., (2) installing, configuring and executing the environments on which to execute the servers and required dependencies, and (3) deploying applications and establishing connections between components.

3. An Extended Brooklyn for Trans-Cloud Management

As explained in the previous sections, Brooklyn provides an API for the management of IaaS cloud services for a significant number of providers, and establishes a life cycle for the management of IaaS services and applications. We have extended Brooklyn by adding some new elements to this API in order to

ACCEPTED MANUSCRIPT

facilitate the management of PaaS services, provided by platforms such as Cloud Foundry, thus providing an homogeneous access to IaaS and PaaS services.

Given Brooklyn's architecture, to be able to handle PaaS locations, it is not enough providing the corresponding locations and drivers to manage them. More importantly, PaaS services follow a completely different life cycle to orchestrate application deployment. The new locations allow PaaS platforms to be modeled inside Brooklyn, allowing the creation of a new family of PaaS-oriented drivers. These new drivers contain the logic to enable the deployment and management of entities on PaaS platforms. However, as for the management of IaaS locations, SoftwareProcess instances require a new PaaS-oriented life cycle. Briefly, the PaaS life cycle we have developed and added to Brooklyn's API for PaaS entities, is responsible for: (1) injecting effector logic to allow interaction with PaaS platforms, (2) creating drivers and adding them to entities, and (3) defining the key steps for managing application processing (deployment, stopping, removing, etc.) on the target PaaS platform. As explained, life cycles are agnostic pieces and do not know about final providers' management, so they require the specification of drivers to manage entities on target locations. For instance, an entity requires PaaS-based locations for signing in to the target platform and using the required services.

3.1. PaaS locations and drivers

As we can see in Figure 5, a new PaasLocation interface, extending Location, has been defined. Specific PaaS locations will then implement this interface. This is the case of class CloudFoundryPaasLocation, which models the Cloud Foundry platform's API. This class provides access to the services and resources of Cloud Foundry, allowing Brooklyn to deploy and manage components in Cloud Foundry-based environments.

Once PaaS platforms have been supported through an appropriate instantiation of the PaasLocation interface, entities need drivers and life cycles to enable the support for this kind of location. Following Brooklyn's approach, a new

ACCEPTED MANUSCRIPT

driver family for CloudFoundryPaasLocation allows SoftwareProcess entities to operate on Cloud Foundry platforms.

The abstract class AbstractSoftwareProcessCloudFoundryDriver provides the necessary general behavior to manage the platform (leveraging on a CloudFoundryPaasLocation), and offers the necessary endpoints to be extended according to the different elements that can be deployed and managed on a Cloud Foundry platform. We can see in Figure 5 how TomcatServerImpl uses a TomcatDriver, which is specialized by TomcatSshDriver. As part of our extension, TomcatDriver now has a new specialization, TomcatCloudFoundryDriver (see Figure 5), which allows TomcatServer entities to use a CloudFoundryPaasLocation. In other words, it allows the software that TomcatServer represents to be deployed using the services of a Cloud Foundry PaaS.

Whereas the driver AbstractSoftwareProcessSshDriver for ssh machines installs and configures the server that TomcatServer represents on a virtual machine and, then, it deploys the web applications on the server, TomcatCloudFoundryDriver does the corresponding work in a PaaS context, where the platform is responsible for configuring the deployment environment and the resources (containers, VMs, etc.), and installing and configuring the software. Once it is ready, the web applications are deployed and launched.

3.2. PaaS life cycles

Life cycles describe the generic key steps required to manage an entity inside a location. However, they require to load and inject into entities IaaS/PaaS drivers that contain specific entity knowledge for deploying and managing the entities on specific providers.

As explained, drivers can be loaded at runtime by entities according to the target location where the entity has to be started. However, drivers focus on the management of specific pieces of software, while life cycles are in charge of appropriately orchestrating the location configuration or the deployment process. The diagram in Figure 5 includes a life-cycle hierarchy that enables the management of machine locations and the integration of drivers for SoftwareProcess

ACCEPTED MANUSCRIPT

entities. To handle PaaS services, we can see in Figure 5 that this hierarchy has been modified. Specifically, interface LifecycleEffectorTasks has been added, which provides the life cycle for an startable entity following the pattern described in Section 2.5.

Figure 5 shows a new life-cycle implementation, PaasLifecycleEffectorTasks, which manages the PaaS locations and drivers and their orchestration. As for IaaS life cycles, PaaS life cycles need to provide start, stop and restart operations. The start routine of the start effector task begins by creating the corresponding driver according to the PaasLocation used and adding it to the entity. Then, the driver of the entity is executed, which takes care of the application processing, deployment, variable environments addition, application launching, etc. Finally, the sensors of the entity are connected to offer information about the software runtime.

Note that, since Brooklyn assumes IaaS locations, it does not consider the possibility of using different kinds of life cycles. As explained in Section 2.5, when an entity is initialized, it gets a default life cycle and its effector bodies are injected into the entity, setting up the entity to manage a concrete kind of location. We have extended the Brooklyn system to allow the indistinct use of PaaS and IaaS services just by suitably specifying the location of the services to be used. Instead of adding life cycle behavior during the entities's initialization phase, in our extension this is done when the entity is started—at runtime. Thus, depending on the entity's location, the aforementioned strategy will select the corresponding IaaS or PaaS life cycle, and will inject the appropriate IaaS or PaaS effector behavior into the entity which will work with the necessary drivers to allow it to work using the target location services. Since each entity comes with appropriate drivers for each kind of supported location, the extended Brooklyn is now able to handle trans-cloud applications, where a unified API allows different application components to be manage over different kind of cloud abstraction levels.

ACCEPTED MANUSCRIPT

4. The Tool in Practice

When dealing with the management of cloud applications, the most important performance measure is the running performance. However, note that this performance will depend on the decisions made in the topology design (components and connections) and the deployment plans (target locations), and our tool does not add any overhead at runtime. We therefore focus on the analysis of the deployment of applications, using deployment times as main metric, and on the effort required for changing deployment decisions.

The effort required for changing application-deployment decisions in IaaS and PaaS has hardly been examined. Throughout this section, we first analyze different alternatives used in the recent related literature. Then, we focus on the evaluation of our proposal on two aspects: (i) the effort required for a change in the deployment decisions, and (ii) the deployment times for two alternative deployment plans.

4.1. On the effort for deployment decision change

Whereas the migration from on-premise applications to the cloud has been studied by many researchers (see [29] for a systematic review), not much work has been published on changes in target providers, and just a few on multi-cloud deployment. Moreover, most studies on cloud migration focus on feasibility and experience reports (see [30] or [31]), and when measuring effort, the focus is usually on operational cost or person hours (see, e.g., [32, 33, 34] and [35]).

Although there is no consensus in the literature on how to measure alternative deployments, we find particularly interesting the one by Kolb et al. in [4]. They extend a framework for service orchestration and orchestration engines, to measure the effort required to deploy applications on the cloud. The deployability is captured and analyzed by using four metrics, namely deployment time, reliability, flexibility, and deployment effort. The deployment effort aggregates four different values, which gather information on the number of deployment steps, the deployment steps parameters, configuration and code changes, and

ACCEPTED MANUSCRIPT

the effort for package construction. These metrics capture the many diverse operations necessary for changing from one provider to another.

Kolb et al. in [4] use their proposal to compare and analyze the feasibility of the deployment of an application using seven different vendors in terms of portability and effort. Like us, they focus on functional portability, and assume service-based applications built to run in the cloud. For an application composed of modules similar to those used in our case study, in the analysis in [4], the deployment steps needed for a given set of PaaS providers are very different. Although these steps are semantically similar among vendors, they are carried out by proprietary tools, which do not permit them to be carried out in a standardized way. The experiments in [4] show that, on average, a deployment may require an effort of 17 actions with a maximum spread of 14 and a standard deviation of 5. A low number of steps is usually offset by a complex configuration of the initial code repository, and it makes the initial deployment of an application a non-elementary task.

Despite the differences in effort between the different alternatives shown in [4], this effort is reduced to 1 when their unified API [25] is used to interact with the different providers. This is true in our case also, not only for PaaS applications, and when all the modules of an application go to the same PaaS provider, but for any combination of IaaS and PaaS vendors used for each of our application's modules, since the knowledge to interact with a specific provider is encapsulated inside Brooklyn API. So, this allows the effort required for a change in our deployment to be only 1.

Flexibility may be an important issue, but since we are interested in the automation of the deployments to allow their reproducibility in DevOps environments, we do not find it to be a useful metric. Indeed, we can only target providers that offer public APIs or CLI tools. Finally, deployment time and reliability are important issues when it comes to deciding which specific providers to use, but they do not add much about how easy it is to change our deployment decisions, which is the main focus of our work.

In our Brooklyn-based solution for trans-cloud deployment, changing the

ACCEPTED MANUSCRIPT

target for our application's modules is as simple as going to its TOSCA model and choosing a different location from its catalog. If there is a location available for it, the effort for change is zero. Thus, the benefit is obvious. We only need an initial effort to specify the topology of our application in TOSCA, which is comparable with—even simpler than—the effort needed to do the deployment by hand.

4.2. The change of deployment plans

To illustrate our approach, in the following paragraphs we explain how to deploy our Softcare application with all its components deployed using IaaS services, and then we change the target host of some of the modules to PaaS. We show that the proposed approach supports deployment decision change with a minimum effort, and we confirm that this effort is drastically less than modifying from IaaS to PaaS by hand.

The initial deployment is shown in Section 2.1. Listing 1 shows the corresponding Softcare's TOSCA YAML topology schema.[12] As we can see in lines 29–37, Brooklyn-TOSCA uses policies (`brooklyn.location`) to specify the location on which the members of a group are to be deployed. The `groups` section defines groups by grouping one or more node templates for assigning special attributes, like policies, to each group element. In this case, we can see how two groups have been defined to be deployed on IaaS, specifically on AWS (Ireland's cluster) and SoftLayer (London's cluster).

Now, for each component, an entity which is decorated with IaaS or PaaS behavior depending on whether the location corresponds to an IaaS or PaaS service (see Section 3). For instance, we may decide to deploy some or all of the components in our Softcare case study on a different provider, and we can do so just by changing the corresponding locations. Listing 2 shows a modified version

[12]Note the ellipses in the TOSCA YAML, we have removed all the information specifying properties, capabilities, etc., with which we specify the war files to be deployed, ports to be used and other details needed for the correct operation of the application, but that are not relevant to our presentation.

ACCEPTED MANUSCRIPT

```
tosca_definitions_version: tosca_simple_yaml_1_0_0_wd03
...
topology_template:
 node_templates:
  SoftcareDashboard:
   type: org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
    ...
   requirements:
    - endpoint_configuration:
       node: SoftcareWS
       ...
    - endpoint_configuration:
       node: Forum
       ...
    - endpoint_configuration:
       node: Multimedia
       ...
  SoftcareWS:
   type: org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
    ...
   requirements:
    - endpoint_configuration:
       node: SoftcareDB
       ...
  SoftcareDB:
   type: org.apache.brooklyn.entity.database.mysql.MySqlNode
   ...
  ...
 groups:
  add_compute_locations:
   members: [SoftcareDB, ForumDB, MultimediaDB, Forum]
   policies:
   - brooklyn.location: aws-ec2:eu-west-1
  add_web_locations:
   members: [SoftcareDashboard, SoftcareWS, Multimedia]
   policies:
   - brooklyn.location: softlayer:lon02
```

Listing 1: Excerpt of Softcare's TOSCA YAML description

of the excerpt of the TOSCA YAML in Listing 1, focusing on the groups section. Now, the group with the components SoftcareDB, ForumDB, MultimediaDB, and Forum will again be deployed on Ireland's cluster of AWS, IaaS, as before, while the group with the components SoftcareDashboard, SoftcareWS and Multimedia will be deployed on Pivotal Web Services as PaaS. Of course, modules could be grouped differently, and deployed at will on any IaaS or PaaS platform supported by Apache Brooklyn. Our proposal presents similar portability features as other approaches either for IaaS or PaaS, but the main novelty and better results with respect to the current state of the art concerns to trans-cloud deployment, when the portability transits from IaaS to PaaS and vice versa.

ACCEPTED MANUSCRIPT

```
  ...
groups:
 add_compute_locations:
  members: [SoftcareDB, ForumDB, MultimediaDB, Forum]
  policies:
  - brooklyn.location: aws-ec2:eu-west-1
 add_web_locations:
  members: [SoftcareDashboard, SoftcareWS, Multimedia]
  policies:
  - brooklyn.location: pivotal-ws
```

Listing 2: Adding new locations to web modules

4.3. Deployment time analysis

A second important issue to be illustrated is as regards the quantitative level. In what follows we provide some insights into how the extended version of Brooklyn performs for different deployment plans. Comparing the performance or reliability of providers or abstraction levels is not the goal of the work presented here, and to draw conclusions on multi-cloud deployments, or even on IaaS and PaaS deployments, would require a more exhaustive analysis, with many more experiments, at different times, with a more varied set of modules, different providers, and so forth. However, our experiments show that we have not lost performance or reliability by using a trans-cloud deployment. The time is spent in provisioning, solving dependencies, and other tasks not due to management overhead.

We have carried out experiments with four different deployment plans: (AWS) all modules deployed on AWS; (AWS-Pivotal) all MySQL modules deployed on AWS, and all Tomcat modules on Pivotal; (AWS-SoftLayer (Forum)) all MySQL modules plus the Forum module deployed on AWS, the rest on SoftLayer; and (AWS-Pivotal (Forum)) all MySQL modules plus the Forum module deployed on AWS, the rest on Pivotal.

Each of these four deployment plans has been executed 100 times using the extended Brooklyn. With regards the discussion in Section 4.2, the only change between these alternative plans is the location in which some of the modules are deployed. The tool was instrumentalized to gather information at each subtask of the process for each module. Specifically, although entities targeting IaaS

ACCEPTED MANUSCRIPT

and PaaS locations have different life cycles, we identified in both cases tasks setup, pre.start, deploy, pre.launch, launch, post.launch, and post.start, and gather the times at which they were completed (all time amounts are in seconds). Charts in Figure 7 depict some of the data gathered.[13]

Charts in Figures 6a-6d show box plots for the deployment times of each of the modules for each of the four deployment plans (700 modules for each plan). In these charts, we have grouped the results by provider, since they differ quite significantly in their performance. If we focus, for example, on the boxes for modules deployed on AWS (in orange in all four charts), we observe a similar pattern in all four cases. With a different concentration of results, the times are quite similar for the modules deployed on SoftLayer (in red in Figure 6c). In the IaaS life cycle, the provision of VMs happens in the setup; artifacts, libraries and other dependencies are installed and solved in the deploy task; in the pre-launch, artifacts are uploaded, the application is configured, and the dependencies between the application modules are solved (TOSCA relations between components). Other operations, like preparing Brooklyn, setting up directories, etc. consume much less time. The plots show how the values are fairly concentrated around the median values for all providers, although with several outliers (around 30–40 out of 700), which show significant overtimes. Although all the deployments were executed sequentially and under similar conditions, some of the executions required more than 150% the average time. Although most of the deployments were completed in less than 450 seconds, some of then took over 700 seconds.

For modules deployed on IaaS modules, the divergences in the times for the deploy task are due to delays in the provisioning of virtual machines or in the creation of the environment to update components; divergences for the pre-launch task are due to delays in the resolution of dependencies between modules and

[13] Additional charts for these and alternative deployments and details on them are available at `https://trans-cloud.firebaseapp.com`, confirming the conclusions illustrated by figures presented in this section.

ACCEPTED MANUSCRIPT

in the pushing of artifacts to the environment. The charts in Figures 6b and 6d show an even bigger dispersion for Pivotal, mainly due to differences in the times taken in the pre.launch task. Times for setup and deployment are significantly bigger for IaaS than for PaaS, although the pre-launch and post.start is bigger for PaaS services. In the PaaS life cycle, the setup and pre-start tasks only initiate the user session and prepare Brooklyn for the deployment, the application is created in the deploy task, but it is in the pre-launch where the application is configured, all dependencies are resolved, and application artifacts are pushed. Note the amount of time and the variability of the pre-launch for the modules deployed on Pivotal (Figure 6b). This observation is key to understand the differences in the pre-launch boxes for Pivotal in Figures 6b and 6d. The fact that the Forum module is deployed on AWS or Pivotal produces a significant difference in the resolution of dependencies, and more importantly, a greater dispersion in the values obtained.

To get a better idea as to how the sequence of tasks performed, in Figures 7b-7d we have depicted the details for the execution of representative deployments, from which we took those with median deployment times. They show the times upon the completion of each of the tasks for each of the modules of our Softcare application. For modules deployed on IaaS services, most of the time is consumed in the setup and deploy tasks, since virtual machines must be provisioned, software loaded, etc. In the chart in Figure 7a we observe how, since all modules are deployed using AWS services, the times are similar for all of them. Although with some dispersion, when using AWS and SoftLayer, shown in Figure 7c, the times for the different tasks follow a similar pattern, with some extra delays for resolving dependencies. The charts in Figures 7b and 7d provide a clearer measure of the time required for dependency resolution. Whilst Figure 7b shows a clear differentiation between the modules deployed on AWS and Pivotal, the chart in Figure 7d, where the module Forum, on which SoftcareDashboard depends, clearly shows how this dependency has to be solved in the pre-launch, forcing the task to wait until the Forum module is available.

ACCEPTED MANUSCRIPT

4.4. Threads to analysis validity

After an analysis of our experiments, we can conclude that trans-cloud deployment does not affect performance or reliability. The time is spent, as expected, in provisioning, solving dependencies, and other tasks not due to management overheads.

In the results shown in the previous section there are certain outlier values. Indeed, there are also failing deployments that had to be repeated. These delays of failures were due to usual problems when deploying in the cloud using public providers. Unusual delays are due to times taken in the provisioning of virtual machines, connectivity problems, etc. However, note that they do not put in question the given results.

We also claim that the effort of changing location targets is almost zero if the proposed CAMP interface offers the location in its catalog. Of course, if we wanted to use a location not in the catalog, it should be added before using it. Its addition would require some effort, either by the developers of the interface or the user himself.

5. Related Work

In this section, we compare our proposal with some other research efforts, as well as independent tools and frameworks that have emerged with the purpose of mitigating the cloud heterogeneity in the usage of services, by proposing common interfaces or middleware platforms.

Federated multi-clouds [12] defines a PaaS federated platform to manage applications on IaaS and PaaS providers. Like us, federated multi-clouds uses an OASIS standard, the Service Component Architecture (SCA), to provide a unified management of an application's components, relationships, communications, etc., and to enable the distribution and handling of multi-cloud applications over IaaS and PaaS levels of different providers. They also offer a standard-based unified provider-management, and allow the description of application architectures. However, they do not offer a robust unified applica-

ACCEPTED MANUSCRIPT

tion modeling mechanism to represent the knowledge about applications, which makes the addressing of migration and elasticity more difficult.

COAPS [6] is a generic API to manage some PaaS providers, such as Google App Engine, Cloud Foundry, OpenShift, and some others. They define different models to represent application components and cloud services and how they are related. However, no standard is considered, and they focus in the simplification of PaaS management.

Kolb and Wirtz in [36] provide a conceptual analysis and classification of PaaS trends and contexts, and build a PaaS taxonomy-based profile to describe the core functional capabilities of offerings. In [25], Kolb and Röck present an interface to unify these core capabilities, providing mechanisms for the management of cloud applications and cloud environments. The proposal is validated by the implementation of Nucleus, which supports several PaaS cloud platforms. In Nucleus, each provider is represented by a specific adapter which implements the unified API, and the generic interface allows applications to be represented programatically, deployed and managed over the supported vendors. With our approach, we go one step further and provide mechanisms to define standard-based application topology descriptions and support for the interoperability of and portability between both IaaS and PaaS services.

Rabanahu et al. in [8] present SCALES, an abstraction-driven approach to address portability issues between different cloud providers, and to mitigate vendor lock-in related problems. They provide a DSL with which to provide full-detailed specifications of applications, including information on components, technological requirements, etc. From these descriptions, the necessary artifacts to deploy the application in a concrete provider are automatically generated. Like in our approach, users can model their applications in an abstract way and postpone the decision of selecting the target provider. Furthermore, they support IaaS and PaaS services, such as AWS EC2 or Google App Engine, facilitating the management of different abstractions levels. However, all modules must use the services of the same provider, since they do not provide a cross-deployment solution. Although they evaluate the advantages of using a unified

ACCEPTED MANUSCRIPT

DSL for system description, and the benefits of the proposed transformation by using different metrics, such as the relation between DSL scripts and generated lines of code, their work would have a greater impact if using some of the existing standardization efforts, such as those by OASIS or IEEE.

Rafique et al. in [37] present a middleware platform for hybrid PaaS cloud applications' distribution and management which enables, by means of a uniform API, portability over multiple services, such as data storage, asynchronous task execution or interoperability between PaaS platforms. In their proposal, changing target providers for applications requires the provision of specific drivers that resolve the specificities of the target providers.

Fang et al. in [38] propose a service management framework for the specification of cloud service operation. They provide an ontological modeling to describe applications, their components and relations, as well as vendors and service operations. Using these models, their tool can reason and assist in the management of applications. In comparison with our solution, the usability of their approach is limited by the complexity in the definition and use of ontology classes for cloud services and cloud service providers.

Different independent tools and frameworks are emerging in the cloud interoperability scope. Winery [39], developed as part of OpenTOSCA ecosystem, is a modeling tool for TOSCA-based cloud applications that offers support for the TOSCA standard as regards visual topology modeling. The OpenTOSCA environment offers a container [40], i.e., a runtime for processing and management, which can process TOSCA-based applications. OpenTOSCA does not provide node templates to represent and manage PaaS and IaaS providers. In this approach, to select new target providers, one has to modify the node templates of an application, which means that new compatible node templates should be used for the application components.

Different cloud orchestrators allow applications to be managed on a multi-cloud environment. A good example is Roboconf [5], an open-source distributed application orchestration framework for multi-cloud platforms. Although it principally focuses on IaaS integration, Roboconf also provides a generic and

ACCEPTED MANUSCRIPT

extensible infrastructure where new providers, including those of PaaS, can be added by means of a set of configuration and DSL-based description files. To deploy an application, developers have to provide a number of additional elements to specify the steps necessary to deploy and execute the application over the target providers. Application descriptions and target services are very interdependent, which makes it very difficult to modify the target providers without affecting the application models.

In the scope of commercial solutions, we can find some new platforms that provide support for the description and control of applications' life cycles which enables cross-cloud management. Just to mention a few, among these tools we find Flexiant, Terraform, Alien4Cloud, Cloud Foundry. For example, Flexiant supports cloud orchestration of multiple IaaS providers, bridging capabilities to scale, deploy and configure servers. Terraform provides a flexible abstraction of cloud resources, providers and applications, which allows representing physical hardware, virtual machines, containers, and cloud resources by means of an enriched application description that has its own DSL. Although most of its efforts focus on IaaS offering, it has put considerable effort into integrating some PaaS services of some providers. Like our proposal, it supports cross-deployment and takes advantage of the different cloud abstraction levels, IaaS, PaaS and SaaS, but it does not consider the unification of IaaS and PaaS modeling and does not offer standard-based applications descriptions.

Containers are a key technology when it comes to portability and interoperability of PaaS applications. Docker allows describing an application and its dependencies using a layer strategy over a container, which is then portable through any system supporting the Docker technology, thus ensuring that both the application and its dependencies will be installed and properly configured. Open projects, such as Docker Swarm or Kubernetes, have emerged to solve the problems related to the orchestration and clustering of applications. However, an important difference to our approach is that container-based technologies are not focused on cross-cloud management and orchestration. Moreover, as we stated at the beginning, we are trying to align our work with standards

ACCEPTED MANUSCRIPT

such as TOSCA and CAMP. In fact, application descriptions challenges in the containerization context are being researched to improve and standardize descriptions. The combination of Docker and Kubernetes is becoming popular for the packaging and managing solution for multi-PaaS (see, e.g., [41]). Pahl in [7] proposes a solution for the orchestration of complex application stacks based on TOSCA, to create topologies independent of service suppliers, cloud providers and hosting technologies.

There are several projects, initiatives and tools that target services deployed on the Cloud using different approaches, with the consequence that software developers need either to use special APIs or programming models to code their applications, or to model them using project-specific domain languages. The aim of the Cloud4SOA project was to solve the semantic interoperability issues that exist in current cloud platforms and infrastructures at PaaS level. Cloud4SOA provides facilities for the deployment and life cycle management and monitoring of applications on PaaS, offering the best matches to computational needs, and ultimately reducing the risks of a vendor lock-in. The PaaSage project also intends to match application requirements against platform characteristics and make deployment recommendations and dynamic mapping of components to the platform(s) selected. The mOSAIC project allows applications to be deployed on different IaaS providers using a sort of mOSAIC virtual machine. REMICS focuses on the development of model-driven tools for the reuse and migration of legacy applications to interoperable Cloud services.

6. Concluding remarks

We have presented *trans-cloud*, our proposal of a common API that unifies IaaS and PaaS cloud services, making their use completely agnostic, so adding a second dimension to previous attempts where the independency was only considered at the level of cloud providers. A TOSCA-based agnostic modeling of applications and cloud services allows us to specify the characteristics and requirements of any system to be deployed in the cloud. The standardized

ACCEPTED MANUSCRIPT

description of applications and cloud resources and the homogenous service API significantly reduce the portability and interoperability issues related to vendor lock-in, facilitating the reusability of cloud services. We have developed an operational tool built on the well-established Apache Brooklyn tool.

Through jclouds, Brooklyn provides support for a large number of IaaS providers. Thanks to our efforts in integrating Cloud Foundry-based platforms into Brooklyn, it now also provides access to PaaS providers such as Pivotal Web Services and Bluemix. Having an agnostic model of our system may greatly simplify migration, or simply decision change. Indeed, with our approach, each component may be deployed at one level or another just by changing its location. The underlying management tool is in charge of the required provisioning and interoperation.

Much work remains ahead. We plan to extend our support for PaaS by allowing other services and technologies in Cloud Foundry. In addition, new entities will be developed to allow Brooklyn to manage them, and new drivers will be added to the current Brooklyn entities, following the hierarchy that we have presented in Section 3. Support for other PaaS platforms may similarly be added to Brooklyn, thanks to the genericity of our proposal. We plan to add support for OpenShift and Heroku, but we are also considering using Nucleus as a unified API for PaaS providers management. We plan to study the possibility of using the flexibility and scalability mechanisms available for PaaS to develop management policies to react to applications' events, e.g., to carry out vertical and horizontal scaling. The independence of service suppliers, cloud providers and hosting technologies with which TOSCA topologies are developed, together with the current capacity for trans-cloud management of applications, opens up an interesting possibility to the dynamic reconfiguration of applications that we plan to explore.

ACCEPTED MANUSCRIPT

Acknowledgements

We are grateful to our partners in the SeaClouds project, and in particular to our colleagues Alex Heneveld, Andrea Turli, and the rest of Cloudsoft Inc., and Francesco D'Andria and Roi Sucasas from Atos Spain. This work has been partially supported by EU project FP7-610531 SeaClouds; Spanish MINECO/FEDER projects TIN2014-52034-R and TIN2015-67083-R; Andalusian Gov project P11-TIC-7659; and Univ. Málaga, Campus de Excelencia Internacional Andalucía Tech.

References

References

[1] M. Armbrust, et al., A view of cloud computing, Comm. of ACM 53 (4) (2010) 50–58.

[2] L. Youseff, M. Butrico, D. D. Silva, Toward a unified ontology of cloud computing, in: IEEE Grid Computing Environments Workshop (GCE), 2008, pp. 1–10.

[3] D. Androcec, N. Vrcek, P. Kungas, Service-level interoperability issues of platform as a service, in: World Congress on Services (SERVICES), 2015, pp. 349–356.

[4] S. Kolb, J. Lenhard, G. Wirtz, Application migration effort in the cloud, in: Intl. Conf. on Cloud Computing (CLOUD), 2015, pp. 41–48.

[5] L. M. Pham, A. Tchana, D. Donsez, N. De Palma, V. Zurczak, P.-Y. Gibello, Roboconf: a hybrid cloud orchestrator to deploy complex applications, in: Intl. Conf. on Cloud Computing (CLOUD), 2015, pp. 365–372.

[6] M. Sellami, S. Yangui, M. Mohamed, S. Tata, PaaS-independent provisioning and management of applications in the cloud, in: Intl. Conf. on Cloud Computing (CLOUD), 2013, pp. 693–700.

ACCEPTED MANUSCRIPT

[7] C. Pahl, Containerization and the PaaS cloud, IEEE Cloud Computing 2 (3) (2015) 24–31.

[8] A. Ranabahu, E. M. Maximilien, A. Sheth, K. Thirunarayan, Application portability in Cloud Computing: An abstraction-driven perspective, IEEE Trans. on Services Computing 8 (6) (2015) 945–957.

[9] A. Moustafa, M. Zhang, Q. Bai, Trustworthy stigmergic service composition and adaptation in decentralized environments, IEEE Trans. on Services Computing 9 (2) (2016) 317–329.

[10] K. Kritikos, D. Plexousakis, Multi-cloud application design through cloud service composition, in: Intl. Conf. on Cloud Computing (CLOUD), 2015, pp. 686–693.

[11] Y. Elkhatib, Defining cross-cloud systems, arXiv preprint arXiv:1602.02698 (2016) 1–4.

[12] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, L. Seinturier, A federated multi-cloud PaaS infrastructure, in: Intl. Conf. on Cloud Computing (CLOUD), 2012, pp. 392–399.

[13] N. Grozev, R. Buyya, Inter-cloud architectures and application brokering: taxonomy and survey, Softw., Pract. Exper. 44 (3) (2014) 369–390.

[14] E. Hossny, S. Khattab, F. Omara, H. Hassan, A case study for deploying applications on heterogeneous PaaS platforms, in: Intl. Conf. on Cloud Comput. and Big Data (CloudCom-Asia), 2013, pp. 246–253.

[15] D. Zeginis, F. D'andria, S. Bocconi, J. Gorronogoitia Cruz, O. Collell Martin, P. Gouvas, G. Ledakis, K. A. Tarabanis, A user-centric multi-PaaS application management solution for hybrid multi-Cloud scenarios, Scalable Computing: Practice and Experience 14 (1) (2013) 17–32.

[16] S. Kolb, G. Wirtz, Data governance and semantic recommendation algorithms for cloud platform selection, in: Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on, IEEE, 2017, pp. 664–671.

ACCEPTED MANUSCRIPT

[17] L. Qu, Y. Wang, M. A. Orgun, L. Liu, H. Liu, A. Bouguettaya, CCCloud: Context-aware and credible cloud service selection based on subjective assessment and objective assessment, IEEE Trans. on Services Computing 8 (3) (2015) 369–383.

[18] Z. Zheng, Y. Zhang, M. R. Lyu, Investigating QoS of real-world web services, IEEE Trans. on Services Computing 7 (1) (2014) 32–39.

[19] D. Petcu, Portability and interoperability between clouds: challenges and case study, in: Towards a Service-Based Internet, Springer, 2011, pp. 62–74.

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

[21] J. Carrasco, F. Durán, E. Pimentel, Component-wise application migration in bidimensional cross-cloud environments, in: D. Ferguson, V. M. Muñoz, J. S. Cardoso, M. Helfert, C. Pahl (Eds.), Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER), SciTePress, 2017, pp. 259–269.

[22] J. Carrasco, J. Cubo, F. Durán, E. Pimentel, Bidimensional cross-cloud management with TOSCA and Brooklyn, in: Intl. Conf. on Cloud Computing (CLOUD), 2016, pp. 1–5.

[23] OASIS, CAMP: Cloud Application Management for Platforms (V. 1.1), `http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html/` (2012).

[24] OASIS, TOSCA: Topology and Orchestration Specification for Cloud Applications (V. 1.0), `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf` (2012).

[25] S. Kolb, C. Röck, Unified cloud application management, in: World Congress on Services Computing, 2016, pp. 1–8.

ACCEPTED MANUSCRIPT

[26] A. Brogi, M. Fazzolari, A. Ibrahim, J. Soldani, J. Carrasco, J. Cubo, F. Durán, E. Pimentel, E. Di Nitto, F. D Andria, Adaptive management of applications across multiple clouds: The SeaClouds approach, CLEI Electronic Journal 18 (1) (2015) 1–14.

[27] OASIS, TOSCA YAML, http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csprd01/TOSCA-Simple-Profile-YAML-v1.0csprd01.html (2015).

[28] Brooklyn, Brooklyn 0.9.0 Documentation, `https://brooklyn.apache.org/v/0.9.0/` (2016).

[29] P. Jamshidi, A. Ahmad, C. Pahl, Cloud migration research: a systematic review, IEEE Transactions on Cloud Computing 1 (2) (2013) 142–157.

[30] M. A. Chauhan, M. A. Babar, Migrating service-oriented system to cloud computing: An experience report, in: Intl. Conf. on Cloud Computing (CLOUD), 2011, pp. 404–411.

[31] J.-F. Zhao, J.-T. Zhou, Strategies and methods for cloud migration, Intl. J. of Automation and Computing 11 (2) (2014) 143–152.

[32] A. Khajeh-Hosseini, I. Sommerville, J. Bogaerts, P. B. Teregowda, Decision support tools for cloud migration in the enterprise, in: Intl. Conf. on Cloud Computing (CLOUD), 2011, pp. 541–548.

[33] M. Menzel, R. Ranjan, L. Wang, S. U. Khan, J. Chen, Cloudgenius: A hybrid decision support method for automating the migration of web application clusters to public clouds, IEEE Trans. Computers 64 (5) (2015) 1336–1348.

[34] W. Qiu, Z. Zheng, X. Wang, X. Yang, M. R. Lyu, Reliability-based design optimization for cloud migration, IEEE Trans. on Services Computing 7 (2) (2014) 223–236.

ACCEPTED MANUSCRIPT

[35] K. Sun, Y. Li, Effort estimation in cloud migration process, in: Intl. Symp. on Service-Oriented System Engineering (SOSE), 2013, pp. 84–91.

[36] S. Kolb, G. Wirtz, Towards application portability in platform as a service, in: Intl. Symp. on Service Oriented System Engineering (SOSE), 2014, pp. 218–229.

[37] A. Rafique, S. Walraven, B. Lagaisse, T. Desair, W. Joosen, Towards portability and interoperability support in middleware for hybrid clouds, in: Conf. on Computer Communications Workshops (INFOCOM WKSHPS), 2014, pp. 7–12.

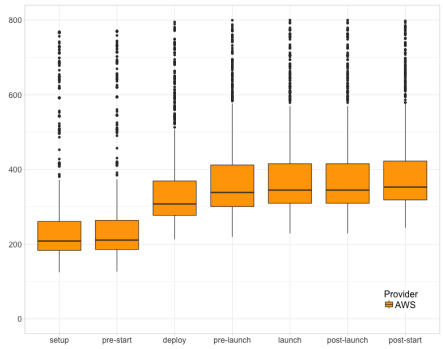
[38] D. Fang, X. Liu, I. Romdhani, C. Pahl, An approach to unified cloud service access, manipulation and dynamic orchestration via semantic cloud service operation specification framework, Journal of Cloud Computing 4 (1) (2015) 14.

[39] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann, Winery–a modeling tool for TOSCA-based cloud applications, in: International Conference on Service-Oriented Computing, Springer, 2013, pp. 700–704.

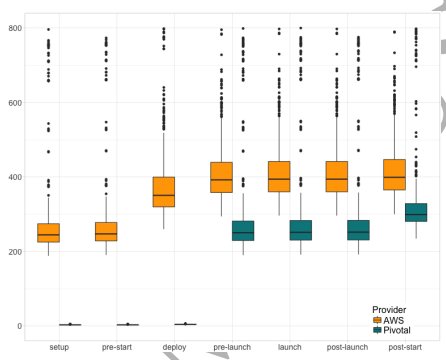
[40] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner, Opentosca–a runtime for tosca-based cloud applications, in: International Conference on Service-Oriented Computing, Springer, 2013, pp. 692–695.

[41] D. Bernstein, Containers and cloud: From LXC to Docker to Kubernetes, IEEE Cloud Computing 1 (3) (2014) 81–84.

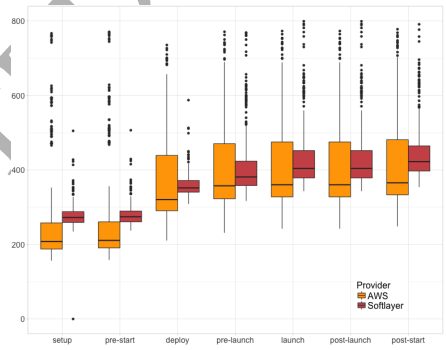
ACCEPTED MANUSCRIPT



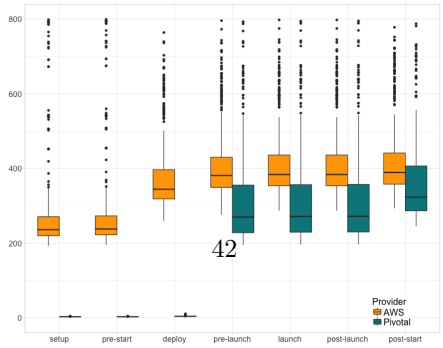
(a) AWS - Box plot



(b) AWS-Pivotal - Box plot



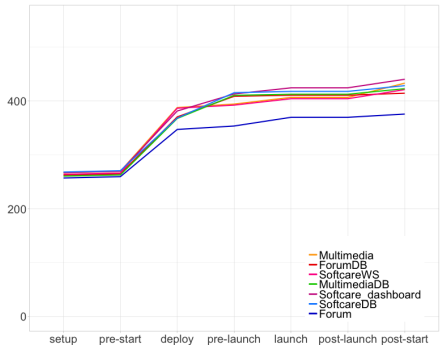
(c) AWS-SoftLayer (Forum) - Box plot



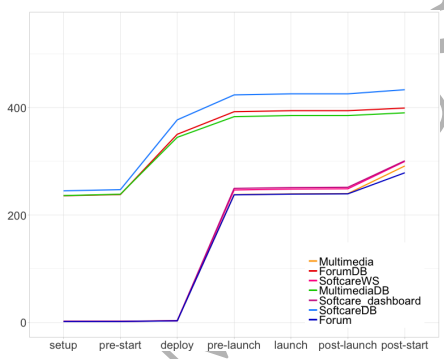
(d) AWS-Pivotal (Forum) - Box plot

Figure 6: Box plots of deployment times

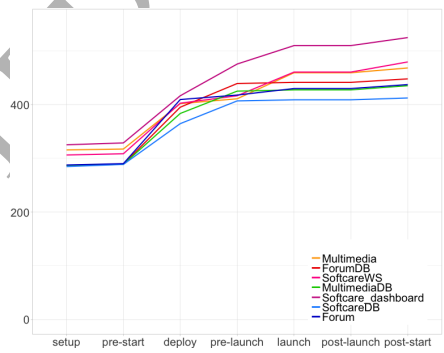
ACCEPTED MANUSCRIPT



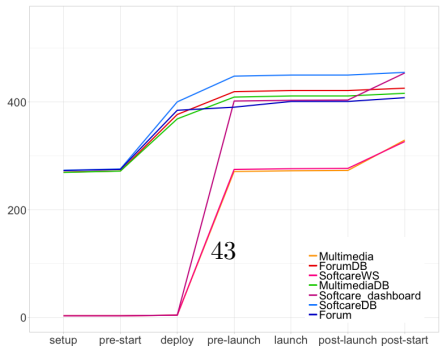
(a) AWS - Median (task 5)



(b) AWS-Pivotal - Median (task 54)



(c) AWS-SoftLayer (Forum) - Median (task 6)



(d) AWS-Pivotal (Forum) - Median (task 6)

Figure 7: Representations of the deployment times for the four plans