Automatic Discovery and Selection of Services in Multi-PaaS Environments

Rami Sellami and Stéphane Mouton

Software and Services Technologies department, CETIC, Charleroi, Belgium

Keywords: Cloud Computing, Multi-PaaS, PaaS Service Discovery, PaaS Services Selection, Semantic Web, Ontologies.

Abstract: Over the past couple of years, a new paradigm has emerged which is referred to as DevOps. It is a methodology to efficiently manage the relationship between development and operations in order to automate applications lifecycle. Spurred by its popularity, it is used today to manage applications in the PaaS level of the Cloud. However, it becomes very challenging when it comes to deploying an application in multi-PaaS environments. The first challenge is to discover and select services taking into account the application requirements and on the PaaS capabilities. Indeed, PaaS providers do not use the same mechanisms to describe and expose their services. Added to that, there is no standard way to describe application requirements. To tackle these anomalies, we propose an automatic and declarative approach to discover and select services offered by PaaS providers. It enables developers to express their requirements and PaaS providers to expose their offers in manifests. To do so, a matching algorithm selects the most appropriate offer in terms of PaaS capabilities to deploy the application. An offer may involve either a single or multi-PaaS provider(s). The key ingredients of our solution are threefold: (1) manifests to describe application requirements and the offers, (2) an ontology to remove semantic ambiguities in PaaS providers capabilities, and (3) a matching algorithm to elect the most appropriate offer to the application. The solution is proposed as a REST API and is delivered with a Web client.

1 INTRODUCTION

Cloud Computing has become nowadays a buzzword in the Web applications world. It is defined by the National Institute of Standards and Technology (Peter and Tim, 2009) as a model for enabling on-demand remote access to a shared pool of configurable computing resources (i.e. processing, storage and networks) that can be released as fast as they have been provisioned to users. These resources are available in self service and without human interaction from Cloud service providers. Cloud Computing is characterized by its economic model referred to as "pay-as-you-go". This latter allows users to consume computing resources as needed.

Cloud services are delivered under three well discussed layers. First, the Infrastructure as a Service (IaaS) which ensures computer resources with a low level of abstraction such as virtual machines, storage and networks. Second, the Platform as a Service (PaaS) that maintains and manages all software components and libraries on top of the IaaS so that customers only need to deploy their applications to run it. Third, the Software as a Service (SaaS) which represents a set of tools and software ready for the use.

For each layer, resources can be provisioned pragmatically via Application Programming Interfaces (API). Doing so, it is particularly important in the case of IaaS and PaaS since applications execution relies on the precision of the configuration of computing resources needed to run them. The price for such automated configuration management is that developers have to program their applications requirements in terms of infrastructure and platform. Programming resources reservation is sometimes referred for IaaS as Infrastructure as Code (Hüttermann, 2012). In addition, remote application deployment and configuration allows to reliably reproduce an environment needed to run applications. Those configuration and deployment actions are usually assigned to the operation staff. The DEVOPS term (Gene et al., 2016) has been coined to describe the merge of the two roles, software DEVELOpment and OPerationS, in order to fully exploit the potential of Cloud Computing.

In this context, we find today a plethora of commercial solutions and research projects (Keith et al., 2013) (Sellami et al., 2017) that have sought to support the whole application lifecycle (especially the
discovery and the deployment steps). These solutions target single IaaS and/or PaaS environments. In our work, we mainly focus on the PaaS level. Indeed, a PaaS provider is supposed to support the whole application requirements during its lifecycle. However, these requirements frequently change. Thus, it seems illusory to find a single PaaS provider that efficiently supports various applications with different requirements.

To circumvent this obstacle, an application may be deployed in multi-PaaS providers (Athanasopoulos et al., 2015) (Sellami et al., 2017) (Ahmed-Nacer et al., 2017) and benefits from various PaaS providers capabilities at the same time. In a previous work (Sellami et al., 2017), we focused on the current state-of-the-art about applications deployment in multi-PaaS providers and we highlighted the main requirements of such environment. In this paper, we start realizing these requirements and we propose a solution to automatically and declaratively discover and select services in multi-PaaS environments. More precisely, our contributions are (1) two models enabling to describe application requirements and PaaS providers capabilities, (2) an ontology (called 2PCR) to automate the discovery process and remove semantic ambiguities in expressing PaaS capabilities, and (3) a matching algorithm to select the most appropriate environment. This latter may be composed by either a single or multi-PaaS provider(s).

The rest of this paper is organized as follows. Section 2 presents the MoDePaaS project and motivates our proposal. In section 3, we introduce the principles and the key components of our solution to automatically discover services in multi-PaaS environments. In Section 4, we present our matching algorithm. Section 5 introduces a proof of concept and the evaluation of our solution. Section 6 presents the related work and Section 7 provides a conclusion.

2 USE CASES AND MOTIVATION

The MoDePaaS project aims at defining a set of tools and techniques to automatically deploy software applications in multi-PaaS environments. It will help developers and alleviate the burden of their tasks. Indeed, they are faced with several problems and obstacles. They must manually discover the capabilities of each PaaS provider, taking into account the application requirements. Then, they have to select the most appropriate PaaS providers in which they will configure and deploy the application. However, these tasks are cumbersome and costly in terms of manpower and time. This is mainly due to the lack of automation when executing a given task. Against this background, MoDePaaS proposes to ease developers’ life by tackling these problems. It aims to define an end-to-end solution in order to automatically discover PaaS services and deploy applications on multi-PaaS providers. This solution is declarative so that application requirements and PaaS providers’ capabilities are easily expressed. MoDePaaS targets to provide the following aims:

- **Aim 1**: Define a model to declaratively express application requirements
- **Aim 2**: Define metadata to describe PaaS providers capabilities and propose a mechanism to collect it
- **Aim 3**: Define an algorithm to select one or multi-PaaS provider(s) in order to deploy an application
- **Aim 4**: Define a unique tool to automatically deploy applications in multi-PaaS providers

Against this background, we emphasize that it is important to discover and select one or multi-PaaS services correctly. Hence, it will ease developers’ tasks and automate applications deployment. In the upcoming sections, we will introduce our approach to realize the Aims 1, 2, 3 of the MoDePaaS project.

3 KEY INGREDIENTS OF THE DISCOVERY AND SELECTION APPROACH

In this section, we introduce the main components of our solution. Indeed, we start by introducing the principles of our approach (see Section 3.1). Afterward, we present the structure of the Abstract Application Manifest (AAM) (see Section 3.2). Then, we define the three components of the broker layer that are the PaaS capabilities repository, the 2PCR ontology, and the Offer Manifest (OM) (see Section 3.3). It is noteworthy that we provide examples of the introduced components in Section 5.1.

3.1 Principles of the Discovery and Selection Approach

Once the developers finish their application coding, they should discover Clouds’ services in order to deploy it. In Figure 1, we showcase an overview of our approach to discover and select services in multi-PaaS environment. First, developers describe their applications requirements in terms of storage and computing and some information about the deployment process in the AAM. Second, they send it to the Discoverer...
component which constructs a sample with respect to their requirements and sends it to the Broker layer. This layer integrates three main elements. Indeed, we find the PaaS capabilities repository that contains information about services capabilities organized by PaaS providers. The capabilities are gathered either by using proprietary API of PaaS providers or manually. Then, we have the 2PRC ontology which is used to (1) remove semantic ambiguities in the repository and (2) automate the discovery process. Finally, we have the Matcher component that constructs the OM. This latter contains a set of offers that meet the requirements described in the AAM. It is noteworthy that an offer may involve either a single or multi-PaaS provider(s). The resulted OM is sent to the Discoverer component which implements the matching algorithm in order to select an offer and construct the Deployment Model (DM).

3.2 Abstract Application Manifest

Using the AAM allows developers to declaratively express the requirements of their applications. In Figure 2, we showcase the structure of the AAM using a class diagram. Indeed, the root class of our model is the Abstract Application Manifest and a unique id and a name identify it. It contains two categories of classes: the Application and the Environment.

First, the Environment class represents information about the environment where the application will be deployed and from which the applications will provision its services. Such an environment can be formed by one or multi-PaaS solution(s). This class is identified by a unique id and a name. It is composed by one or multi-classes of the following types. For ease of presentation, we propose to designate these four classes by the term node.

- The Service class: It denotes a service offered by a PaaS provider. It is identified by a unique id, a type (e.g. a database, monitoring, search, etc.), a name, a version, a pricing (e.g. free, price per hour, price per month, etc.).
- The Dockerfile class: It presents a dockerfile which is a file containing a set of instructions used to build and run a docker image. It is identified by a unique id, a name and a description to give more details about its features. It is noteworthy that we use these three attributes to define the two following classes.
- The Buildpack class: It allows the developer to deploy and launch the application using the buildpacks. This latter is a set of scripts that enables to detect the framework and runtime of a given application, to compile it and to run it.
- The Cartridge class: It is dedicated to a specific PaaS provider which is Openshift. It is similar to the Buildpack class.

Elements in this part of the AAM are filled with constant values when the developer is sure about the requirements. However, the developer may fill it in a flexible manner. Indeed, if there is a doubt, it is possible to use a joker in order to denote any value. To do so, the developer should use the "∗" character. In addition, it is possible to express two kinds of constraints by using either the comparison operators or the logical operators. In fact, comparison operators enable to approximately express requirements regarding a given property. The specified operators are the following: "−" to represent the relationship lower than, "+" to represent the relationship greater than, "− =" to represent the relationship lower than or equal to and "+ =" to represent the relationship greater than or equal to. Whereas the logical operators express multiple requirements regarding a given

---

1https://docs.docker.com/engine/reference/builder/
2https://devcenter.heroku.com/articles/buildpacks
3https://developers.openshift.com/overview/basic-terminology.html
property. It can be either a conjunction (i.e. AND) or a disjunction (i.e. OR) between two values.

The second category defines information about the application to deploy illustrated by the Application class. It is characterized by a unique id, a name and an environment where the application will be deployed. The developer may specify several versions of the same application. And for each version, he/she needs to precise information related to the deployable artifacts and to the to-be-run instances. A Version class is identified by a unique id, a name and a label. It also contains a set of Deployable and Instance classes. The Deployable class represents the application executable file. It is identified by a unique id, a name, a content type defining the executable file type, and a location containing the URL where such element can be retrieved. Whereas the Instance class represents the running application instances required by the developer. This class is identified by a unique id, a name, an initial state defining the state of the application (e.g. running, stopped, etc.) and a default instance representing the running instances by default.

3.3 Broker Layer

This layer integrates three key components in order to correctly construct the OM. Indeed, it includes the PaaS capabilities repository that exposes information about the nodes delivered by each PaaS provider. It also includes the 2PCR ontology in order to remove semantic ambiguities in nodes information and to automate the discovery process. Finally, it uses the Matcher component that allows to construct the OM based on the Sample.

3.3.1 PaaS Capabilities Repository

In this section, we introduce the PaaS capabilities repository and its different parameters. This component plays the role of the catalog that exposes a list of nodes and its details. These details are gathered in two ways: either exported using PaaS providers proprietary APIs or manually added by analyzing the marketplace of a given PaaS solution. This information concerns the four categories of a node. Each category contains almost the same parameters that are: the name of a given node, its PaaS provider, its credentials, and its pricing. Nevertheless, the Service element is further characterized by a version and a type. Whereas the three other elements are characterized by a description. It is noteworthy that we remain faithful to the original names of these elements as it is offered in the marketplace of the PaaS Solutions. In order to unify the access to this information and to cover the different semantic ambiguities that may exist between two equivalent nodes, we propose to semantically annotate the PaaS capabilities repository using the 2PCR ontology.

3.3.2 PaaS Providers Capabilities Repository (2PCR) Ontology

Based on the different nodes delivered by PaaS providers, we define the 2PCR ontology (see Figure 4). This ontology enables to remove semantic ambiguities between two nodes coming from two different PaaS providers. In addition, it is used to automate the discovery process by semantically annotating the PaaS capabilities repository. Then, the 2PCR ontology is populated with information coming from the repository in order to create a knowledge base with PaaS capabilities. Finally, these capabilities are discovered using SPARQL queries. It is noteworthy that for lack of space and ease of presentation, we do not present all the concepts and the relationships in the rest of the section. To introduce our ontology, we rely on Definition 1:

**Definition 1.** The 2PCR ontology is defined by the 4-tuples \(<C_{2PCR}, DT_{2PCR}, OP_{2PCR}, A_{2PCR}>\):

- \(C_{2PCR}\): Concepts defining the different nodes stored in the PaaS capabilities repository,
- \(DT_{2PCR}\): Information about a concept,
- \(OP_{2PCR}\): Relationships between two concepts,
- \(A_{2PCR}\): A set of evident truths used to enrich the \(C_{2PCR}\) and the \(OP_{2PCR}\).

In the following, we separately define each element of the 4-tuples. We start by the \(C_{2PCR}\). The root concept is the PaaS capabilities repository which is composed by four concepts: a Service, a Cartridge, a Buildpacks, and a Dockerfile. By analogy, these concepts are the same as the four elements composing the PaaS capabilities repository and they are identified with the same parameters.

The second tuple is the \(DT_{2PCR}\). It provides additional information about the four nodes. Indeed, the Description element provides information about
a given node and the BelongTo element identifies the PaaS offering a node.

The third tuple is called OP2PCR. It defines relationships between two concepts. We introduce these relationships and its meaning in Table 1. Finally, the fourth tuple is the axioms set A2PCR. It provides evident truths used to enrich OP2PCR and C2PCR sets.

Table 1: The relationships OP2PCR set meaning.

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNameOfService</td>
<td>Denotes the name of a given service</td>
</tr>
<tr>
<td>hasNameMongoDB</td>
<td>Denotes the name of a service of type MongoDB</td>
</tr>
<tr>
<td>hasNameTomcatApache</td>
<td>Denotes the name of a service of type Tomcat Apache</td>
</tr>
<tr>
<td>hasPlace</td>
<td>Introduces the pricing model of a given service</td>
</tr>
<tr>
<td>hasVersion</td>
<td>Defines the version of a given service</td>
</tr>
<tr>
<td>hasType</td>
<td>Defines the type of a given service</td>
</tr>
<tr>
<td>hasTypeDatabase</td>
<td>Defines the type of a database service</td>
</tr>
<tr>
<td>hasTypeSearch</td>
<td>Denotes the type of a search engine service</td>
</tr>
</tbody>
</table>

Besides, we can enrich our ontology by defining some rules. These rules are an alternative way of defining new relationships between concepts in order to complement the ontology. The definition of a rule is based on the OP2PCR and the C2PCR sets and it is defined as follow:

Definition 2. A rule is defined as: \[ A(x, y) \land B(x) \land C(x, z) \land D(z) \land ... \implies E(y, z) \] where:

- \( A(x, y), C(x, z), E(y, z) \): predicates defining relationships (OP2PCR);
- \( B(x), D(z) \): predicates defining Concepts (C2PCR);
- \( x, y, z \): variables, literals, individuals, etc.

### 3.3.3 Offer Manifest

This manifest contains information about nodes capabilities coming from one or multi-PaaS provider(s). In Figure 5, we present the OM structure based on a class diagram. Indeed, the root class is Offer Manifest and it is identified by an id and a name. It contains one or multiple Offer(s) and each offer may involve single or multi-PaaS provider(s). An Offer class is identified by an id and is formed by a set of Service, Dockerfile, Buildpacks, and Cartridges classes. These classes are well presented in Section 3.2. And, we would like to clarify that we have added a new attribute referred to as cloud_provider_name to denote the Cloud provider offering the given service.

![Figure 4: The 2PCR ontology structure.](image)

![Figure 5: The OM structure.](image)

### 4 MATCHING ALGORITHM

Based on the different components that we have introduced in the previous section, we present our matching algorithm (see Algorithm 1). It selects the most appropriate offer with respect to the application requirements defined in the AAM. Such an offer, may involve one or multi-PaaS provider(s). The algorithm takes as input (1) an AAM, (2) a threshold to limit the number of differences between the AAM and the OM, (3) the 2PCR ontology and (4) the PaaS capabilities repository. This latter is semantically annotated using the 2PCR ontology. The algorithm outputs a DM containing information about the selected environment where the application will be deployed.

Algorithm 1: Matching algorithm.

```plaintext
1. input AAM: the abstract application manifest
2. input threshold: the threshold to limit the number of differences
3. input ontology: the 2PCR ontology
4. input repository: the semantically annotated repository using the 2PCR ontology
5. output DM: the deployment model
6. sample ← constructSample(AAM)
7. populatedOntology ← populateOntology(ontology, repository)
8. OM ← constructOM(sample, populatedOntology)
9. while (exist(Offer O in OM)) do
10.   distance[i] ← 0
11.   for each node N in AAM do
12.     for each property prop in N do
13.       if (valid(prop, OM.O.node.prop)) then
14.         distance[i] ← distance[i] + updateDistance(prop, OM.O.node.prop)
15.       end if
16.     end for
17.   end for
18.   i ← i + 1
19. end while
20. selectedOffer ← selectOffer(distance, threshold)
21. return createDM(AAM, OM, selectedOffer)
```
First, the algorithm constructs the sample based on the AAM (line 6) and populates the 2PCR ontology with information stored in the semantically annotated repository in order to construct a knowledge base (line 7). This knowledge base is denoted by the populatedOntology variable. Based on this, it constructs the OM by computing all the possible offers that may cover the application requirements (line 8). It is important to highlight that the OM is constructed using parameterized SPARQL queries based on the sample and executed on the knowledge base populatedOntology. Afterward, it computes the number of differences between each offer in the OM and the AAM (lines 9-19). Numbers of differences are stored in the data structure distance. These values are calculated as follows: for each property in the both manifests, if they are not corresponding then we update the distance by adding the appropriate penalty to the property. The two properties correspond if the value of the OM property fulfills the requirement expressed by the AAM property (which is either a constant, a joker or a constraint). By default, all penalties are fixed at 1; however the user can configure these penalties according to the importance that he/she gives to the properties. Once this step is achieved, the algorithm selects an offer using the operation selectOffer (line 20). This operation takes as inputs the data structure distance and the threshold and returns the identifier of the selected offer. The selected offer has the smallest value of distance bounded between 0 and the threshold. Finally, it constructs the DM (line 21).

5 IMPLEMENTATION AND EXPERIMENTATION

All along this paper, we proposed an approach enabling the support of developers during the discovery and selection of services in multi-PaaS environments. In this context, we propose to validate and evaluate our approach through two main steps. First, we present the implementations that we realized as a proof of concept (see Section 5.1). Second, we expose the experiments that we have conducted. These experiments enable to empirically evaluate the efficiency of our solution and to prove that it eases the developers task (see Section 5.2).

5.1 Proofs of Concept

In this section, we present the proof of concept that we have implemented in order to show the feasibility and the utility of our approach. Indeed, we develop the matching algorithm in the form of a REST API. To do so, we specified the API using an open source design tool referred to as Swagger editor and we implemented it using JAVA. Today, it is provided as a runnable RESTful web application. In order to easily use our API, we provide a web client to declaratively express the application requirements and to select an appropriate offer. This Web application integrates the REST API and it is implemented using the Restlet framework of JAVA.

In Figure 6, we showcase a screenshot of the Web interface enabling the user to edit the AAM. As it is depicted, we propose to present the AAM model using the XML syntax. Indeed, it is a universal syntax and it is easily understandable. In the example, the user requires three services. The first one is of type Database and it is required to be either a MongoDB or a CouchDB. It should have a version equal to 1. The second one is also of type Database and has the name Redis. Its version is lower than or equal to 1. The third one is of type MessageQueue and it is named RabbitMQ. It is noteworthy that all required services must be free. To ease the presentation, we illustrate a sample containing efficient information for the discovery and selection step (e.g.). And, we decide to hide the information that are used in the deployment step (e.g. the name of services, the versions, etc.) the name of the executable of the application, the number of instances, etc.).

Figure 6: Screenshot of the interface to edit the AAM.

In order to prove the feasibility of our approach, it would be certainly worthwhile to discover and select services coming from real PaaS providers. Hence, we select three open source PaaS providers that are DEIS, Openshift Origin, and Dokku. We install

---

4https://swagger.io/
5http://restlet.org/
6https://deis.com/
7https://www.openshift.org/
8http://dokku.viewdocs.io/dokku/
and test these solutions on our infrastructure. For interested readers, we provide more details about the installation of these solutions in a blog post. Then, we fill the PaaS capabilities repository with information about the offerings collected from their marketplaces. Afterward, we semantically annotate the PaaS capabilities repository using the 2PCR ontology in order to populate it and obtain a knowledge base. Once the ontology is populated, we dynamically construct the SPARQL queries based on the AAM in order to collect the offers. Based on the input AAM, our algorithm generates four SPARQL queries to collect the offers. Based on the input AAM, our algorithm generates four SPARQL queries to collect the offers. Based on the input AAM, our algorithm generates four SPARQL queries to collect the offers. Based on the input AAM, our algorithm generates four SPARQL queries to collect the offers.

Listing 1: SPARQL query to discover MongoDB services from the PaaS capabilities repository.

```
PREFIX 2PCR: <http://www.semanticweb.org/.../2PCR>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?service ?type ?parentType ?name
WHERE {
  ?service rdf:type 2PCR:Service .
  name a ?parentName .
  ?service 2PCR:hasType ?type .
  ?type a ?parentType .
  ?service 2PCR:belongsTo ?AAM .
  FILTER (regex(str(?parentName), "MongoDB", "i"))
  FILTER (regex(str(?parentName), "Thing", "i"))
  FILTER (regex(str(?parentName), "Name", "i"))
  FILTER (regex(str(?parentName), "Resource", "i"))
  FILTER (regex(str(?parentName), "Cloud", "i"))
  FILTER (regex(str(?parentName), "NamedIndividual", "i"))
  FILTER (regex(str(?parentName), "Database", "i"))
  FILTER (regex(str(?parentName), "MongoDB", "i"))
}
```

Listing 2: Samples of offers from the OM.

5.2 Ease of Use of the Approach

Our approach is intended to ease the developers task while discovering and selecting services in multi-PaaS environments. However, it would be certainly worthwhile to concretely evaluate this and prove that we really alleviate the burden on them. To do so, we propose to evaluate the gain in terms of time for developers by using our approach (i.e. Scenario1) instead of manually discover and select services (i.e. Scenario2). For this purpose, we asked three developers (i.e. a trainee, a researcher, and an engineer) to discover and select services taking into account a set of predefined requirements. Indeed, developers should (1) become familiar with both scenarios, (2) express the requirements of their application, (3) discover PaaS services capabilities, and (4) select a set of services supporting the predefined requirements. To realize this experimentation, we propose to use as PaaS providers OpenShift Origin, Dokku, and Deis that we presented in the previous section and the AAM depicted in Figure 6.

In Table 2, we illustrate a comparison between the familiarization time and the discovery and selection time to execute both scenarios. It is worthy to say that the familiarization time depends on the executed scenario. In Scenario1, it includes the time of the familiarization with our approach by discovering the AAM structure and its syntax. Whereas, in Scenario2, it denotes the time of the familiarization with the marketplaces of each PaaS provider and how it presents its services. Regarding the discovery and selection time, we evaluate the execution time of the end-to-end process. In Scenario1, we evaluate the time of

---

1 https://www.cetic.be/Open-Source-PaaS-Solutions-Analysis
editing the AAM with the application requirements and the execution of our algorithm in order to elect the most appropriate offer. In Scenario1, we compute the time spent by the developers to manually discover and select the appropriate services to their application. To compare between the two scenarios, we propose to evaluate the gain in terms of the familiarization time and the discovery and selection process time. The gain is obtained by calculating the ratio between the difference between the time recorded in Scenario1 and that recorded in Scenario2, and the time recorded in Scenario1:

\[
\text{gain}_{\text{familiarization}} = \left( \frac{\text{time}_\text{Scenario2} - \text{time}_\text{Scenario1}}{\text{time}_\text{Scenario1}} \right) \times 100
\]

\[
\text{gain}_{\text{disc&selec}} = \left( \frac{\text{time}_\text{Scenario2} - \text{time}_\text{Scenario1}}{\text{time}_\text{Scenario2}} \right) \times 100
\]

Table 2: Evaluation of the gain in terms of the familiarization time and the discovery and selection process time.

<table>
<thead>
<tr>
<th>Developer</th>
<th>Scenario1</th>
<th>Scenario2</th>
<th>Scenario1</th>
<th>Scenario2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Familiarization time (min)</td>
<td>10</td>
<td>3</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>Discovery and selection time (min)</td>
<td>3</td>
<td>10</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>gain_{\text{familiarization}} (%)</td>
<td>23</td>
<td>11</td>
<td>29</td>
<td>14</td>
</tr>
<tr>
<td>gain_{\text{disc&amp;selec}} (%)</td>
<td>70</td>
<td>64</td>
<td>60</td>
<td>50</td>
</tr>
</tbody>
</table>

Using these two formulas, we obtain an average gain_{\text{familiarization}} equals to 63.33% and an average gain_{\text{disc&selec}} equals to 64.66%. These results are important since they prove that we really alleviate the burden on developers and we ease their tasks. Indeed, this gain regarding the two times encourages the use of our approach since (1) it improves the developers productivity, (2) it increase the adoption of our algorithm, (3) it decreases the errors and the omissions during the discovery and selection process. For instance, when developers manually discover services, they may forget a given service that is more relevant than other services to the application requirements.

6 RELATED WORK

Several works deal with the problem of services discovery and selection in Cloud environments (Keppeler et al., 2014) (Sellami et al., 2017). Generally, these solutions enable developers to describe their applications requirements using a very specific model (e.g. a manifest, a SLA-based model, a metric-based model, etc.). Then, they propose mechanisms enabling the selection of the most appropriate PaaS provider(s) to deploy the application.

In a previous work (Sellami et al., 2015) (Sellami et al., 2016), we proposed an approach to automatically discover data resources in a single PaaS environment. Indeed, developers express their requirements in terms of data stores. Then, our matching algorithm discovers PaaS providers capabilities and, for each PaaS provider, it compares the application requirements and its capabilities in order to select the most appropriate PaaS provider. In this solution, we mainly focused on the data services discovery in a single PaaS environment.

Redl et al. (Redl et al., 2012) present an automatic approach to check if the elements of the SLA are valid or not. Doing so, they map each PaaS SLA to an ontology and they define a matching algorithm to compare the SLAs. Results of this matching are analyzed in order to select the most appropriate services provider. Although the idea of selecting Cloud services based on the SLA is interesting, this approach does not enable to describe application requirements in terms of deployment in multi-PaaS providers.

Wittern et al. (Wittern et al., 2012) propose a model based solution to select services in a Cloud environment in order to express users requirements and services capabilities. For this purpose, they define an algorithm to select services using a set of predefined criteria. However, they do not support the multi-PaaS discovery.

In the SeaClouds project, Athanasopoulos et al. (Athanasopoulos et al., 2015) propose an open source platform to support applications in a multi-clouds environments. Indeed, one of its key component is the SeaClouds Discoverer which enables to discover available capabilities and add-ons offered by available cloud providers. It allows to declaratively select multi-Cloud services based on the QoS expressed by the user. A matching algorithm is implemented in order to select the Cloud provider corresponding to the QoS required by the user. However, this solution lack of automation.

Li et al. (Li et al., 2010a) (Li et al., 2010b) define an automatic solution to select the most appropriate PaaS provider to application requirements. It is referred to as CloudCmp. It compares the performance and the cost of cloud providers in terms of computing, storage, and networking resources using benchmark tasks. It aims to select the cloud provider that has the best performance and the less cost. It supports four cloud providers that namely are Amazon AWS, Microsoft Azure, Google AppEngine, and Rackspace CloudServers. Although the importance of this solution, it does not support services discovery in a multi-PaaS environments.

Kang et al. (Kang and Sim, 2011) (Kang and Sim, 2016) propose an agent-based solution to discover cloud resources using ontologies in order to semanti-
cally describe resources and the relationships between each others. The user requirements are defined in the form of classAds. Authors introduce the discovery process using four stages: the selection, the evaluation, the filtering, and the recommendation. Authors do not consider the multi-PaaS environments.

7 CONCLUSION

In this paper, we proposed an automatic and declarative solution to discover and select Cloud services in multi-PaaS environments. The key ingredients of our work are threefold. First, we presented the AAM and the OM that enable to express application requirements and PaaS providers capabilities respectively. Second, we defined the 2PCR ontology that allows to semantically annotate the PaaS capabilities repository in order to populate the ontology and create a knowledge base. Third, we proposed a matching algorithm that enables to compare the AAM and OM in order to select the most appropriate offer to the application. This offer involves either a single or a multi-PaaS provider(s) in which the application may be deployed.

Currently, we are working on applying our approach to other qualitatively and quantitatively real use cases in order to identify possible discrepancies and make our work more reliable. Second, we are aware that we do not take into account the pricing aspect and we would like to integrate it in our approach. Finally, we target to realize the Aim4 that we have introduced in Section 2. Indeed, we will define a declarative and an automatic tool to deploy applications in multi-PaaS environments.

ACKNOWLEDGMENTS

This work has been funded by the Belgian research project MoDePaaS identified by the number 1610385.

REFERENCES


