Automating Resources Discovery for Multiple Data Stores Cloud Applications

Rami Sellami, Michel Vedrine, Sami Bhiri and Bruno Defude
Computer Science Departement, Institut Mines-Telecom,
Telecom SudParis, CNRS UMR Samovar, Evry, France

Keywords: NoSQL Data Stores, Relational Data Stores, Polyglot Persistence, Manifest based Discovery, ODBAPI.

Abstract: The production of huge amount of data and the emergence of cloud computing have introduced new requirements for data management. Many applications need to interact with several heterogeneous data stores depending on the type of data they have to manage: traditional data types, documents, graph data from social networks, simple key-value data, etc. Interacting with heterogeneous data models via different APIs, multi-data store applications imposes challenging tasks to their developers. Indeed, programmers have to be familiar with different APIs. In addition, developers need to master and deal with the complex processes of cloud discovery, and application deployment and execution. Moreover, the execution of join queries over heterogeneous data models cannot, currently, be achieved in a declarative way as it is used to be with mono-data store application, and therefore requires extra implementation effort. In this paper we propose a declarative approach enabling to lighten the burden of the tedious and non-standard tasks of discovering relevant cloud environment and deploying applications on them while letting developers to simply focus on specifying their storage and computing requirements. A prototype of the proposed solution has been developed and is currently used to implement use cases from the OpenPaaS project.

1 INTRODUCTION

Cloud computing has recently emerged as a new computing paradigm enabling on-demand and scalable provision of resources, platforms and software as services. Cloud computing is often presented at three levels (Baun and et al., 2011): the Infrastructure as a Service (IaaS) giving access to abstracted view on the hardware, the Platform-as-a-Service (PaaS) provides to the developers programming and execution environments, and the Software as a Service (SaaS) enables end users to run cloud software applications.

Due to its elasticity property cloud computing provides interesting execution environments for several emerging applications such as big data management. According to the National Institute of Standards and Technology[1] (NIST), big data is *data which exceed the capacity or capability of current or conventional methods and systems*. It is based on the 3-Vs model where the three Vs refer to volume, velocity and variety properties (McAfee and Brynjolfsson, 2012). Volume means the process of large amounts of information. Velocity signifies the increasing rate at which data flows. Finally, variety refers to the diversity of data sources. Against this background, the challenges of big data management result from the expansion of the 3Vs properties. In our work we focus mainly on the variety property and more precisely on multi-data store applications in the cloud.

In order to satisfy different storage requirements, cloud applications usually need to access and interact with different relational and NoSQL data stores having heterogeneous APIs. The heterogeneity of the data stores induces several problems when developing, deploying and migrating multi-data store applications. Below, we list the main 4 problems which we are tackling in this paper.

Pb_1. Heavy workload on the application developer: Nowadays data stores have different and heterogeneous APIs. Developers of multi-data store applications need to be familiar with all these APIs when coding their applications.

Pb_2. No declarative way for executing join queries: Due to the heterogeneity of the data models, there is currently no declarative way to define and execute complex queries over several data stores. That means developers have to cope

[1] http://www.nist.gov/

themselves with the implementation of such complex queries.

Pb_3. Code adaptation: When migrating applications from one cloud environment to another, application developers have to re-adapt the application source code in order to interact with new data stores. Developers have potentially to learn and master new APIs.

Pb_4. Tedious and non-standard processes of discovery and deployment: Once an application is developed or migrated, developers have to deploy it into a cloud provider. Discovering the most suitable cloud environment providing the required data stores and deploying the application on it is a tedious and meticulous provider-specific process.

In our work, we are are coping with these problems in order to support the developer during a multiple data stores based application life cycle. In a previous work, we proposed ODBAPI (OPEN-PaaS-DataBase API) a streamlined and a unified REST-based API (Sellami et al., 2014) for executing CRUD operations on relational and NoSQL data stores. The highlights of ODBAPI are twofold: (i) decoupling cloud applications from data stores in order to facilitate the migration process, and (ii) easing the developers task by lightening the burden of managing different APIs. In contrast, we present in this paper a declarative approach for discovering appropriate cloud environments and deploying applications on them while letting developers to simply focus on specifying their storage and computing requirements. A prototype of the proposed solution has been developed and is currently used to implement use cases from the OpenPaaS project.

The remainder of the paper is organized as follows. In section 2, we introduce the context and present a motivating example. In section 3, we give an overview of our approach and we detail in sections 4 the discovery step. In section 5, we present the implementation and validation of our solution. In Section 6, we discuss the related work. Section 7 concludes our paper and outlines directions of future work.

2 MOTIVATION

Our work is done in the context of the OpenPaaS project [2] aiming at developing a PaaS technology dedicated to enterprise collaborative applications deployed on hybrid clouds (private/public). It is a platform that allows to design and deploy applications based on proven technologies provided by partners such as collaborative messaging system, integration and workflow technologies that will be extended in order to address Cloud Computing requirements. Target applications of OpenPaaS are applications that use multiple data stores that corresponds to what is popularly referred to as the polyglot persistence. For instance, an application can interact with three heterogeneous data stores at the same time: a *relational data store*, a document data store that is *CouchDB*, and a key value data store which is *Riak*. But, this example exposes some limits. Linking an application with multiple data stores is very complex due to the different APIs, data models, query languages and consistency models. If the application needs to query data coming from different data sources (e.g joining data, aggregating data, etc.), it can not do it declaratively unless some kinds of mediation have been done before. Finally, the different data stores may use different transaction and consistency models (for example classical ACID and eventual consistency). It is not easy for developers to understand these models and to maintain its properties while coding their application. Moreover, it is more complicated when it comes to execute complex queries and especially join queries. But first, the developer has to discover all data stores capabilities of the available cloud providers in order to elect the most suitable cloud provider to his application requirements to deploy it. However this in itself is a tedious and meticulous work. In this paper, we will tackle these problems and we will propose an end-to-end solution and we will focus on the step of the cloud data stores discovery.

3 OVERVIEW OF OUR APPROACH

The lion's share of the contribution of our work is dedicated to application developers. Indeed, we attempt to simplify the developer task during the life cycle of an application (i.e. development, discovery and deployment, and execution) using multiple and heterogeneous data stores in its source code. The developer has (i) to write the source code of his application using various APIs, (ii) to discover data stores of each cloud provider in order (iii) to deploy his application, and (iv) to execute queries.

Hence, for the purpose of simplifying the developers task and getting them rid of all their onerous responsibilities, we propose our solution that is based on three points: (i) using a unique API enabling the

[2]This work has been partly funded by the French FSN OpenPaaS grant (https://research.linagora.com/display/openpaas/Open+PAAS+Overview)

coding and the execution of multiple data store based application, (ii) enabling join queries execution over heterogeneous data stores, and (iii) ensuring a neutral application deployment in a cloud provider that is already automatically discovered.

In Fig 1, we showcase an overview of our solution including the different steps of the application life cycle. Indeed, we depict the following four steps:

- *Development Step*: The first step is the application development. For this purpose, we propose to use a unique API enabling the interaction with relational and NoSQL data stores. In a previous work (Sellami et al., 2014), we defined a generic resources model defining the different concept used in each type of data store. These resources are managed by ODBAPI a streamlined and a unified REST API enabling to execute CRUD operations on different NoSQL and relational databases. ODBAPI decouples cloud applications from data stores alleviating therefore their migration. Moreover it relieves developers task by removing the burden of managing different APIs.
- *Discovery Step*: The second step consists in discovering data stores capabilities of each cloud provider. So, we can decide which cloud provider can support our application requirements. Doing so, the developer describe his requirements in the *Abstract application manifest* and sends it to the *discovery* module. The *discovery* module interacts with the cloud providers in order to discover the capabilities of each cloud provider in term of data store services. Then, it obtains as a result the *offer manifest*. Based on these two manifests, the *Matching module* elects the most appropriate cloud provider to the application in order to deploy it and edit its *deployment manifest*.
- *Deployment Step*: The third step represents the application deployment. An important part in this phase consists in generating the application executable and adding the *execution manifest* to it. This manifest contains the endpoint of the *ODBAPI server* or/and the data stores endpoints that the application requires. We propose to deploy an application by any deployment API (e.g. COAPS API(Sellami and et al., 2013), roboconf API[3], etc.). In our work, we are building on the COAPS API that is proposed in our team and allows human and/or software agents to provision and manage PaaS applications. This API provides a unique layer to interact with any cloud provider based on manifests. We model the *deployment manifest* based on the manifest of COAPS API and we enrich it with information about data stores to support ODBAPI-based application.
- *Execution Step*: The fourth step is the application execution. In this step, an application can execute two types of queries. On the one hand, it can execute CRUD queries; hence it interacts directly with the target data store based on its *execution manifest*. In this case, it uses a unique API like ODBAPI. On the other hand, it can execute join queries by interacting with virtual data stores. A virtual data store is created automatically for executing join queries. When a virtual data store receives a join query, it parses this query, rewrites it into sub-queries written in the ODBAPI syntax, and sends these sub-queries to the target data stores. Once a virtual data store receives the result of each sub-query, it forms the final result and returns it to the application. It is worth noting that when the user edits the *Abstract application manifest*, he has to precise that his application will manipulate complex queries. Hence, a virtual data store will be created and its endpoint will be added to the *execution manifest* also.

In the following section, we will focus only on the discovery of the data stores capabilities and the deployment of an ODBAPI-based application.

4 DISCOVERY OF DATA STORES OF CLOUD PROVIDERS

In this section, we present our logic to discover the capabilities of data stores of cloud providers meeting the application requirements (see Figure2). The developer coded an ODBAPI application and describes its requirements in the *abstract application manifest* in term of data stores and deployment. Then, he gives it as an input to the *matching algorithm*. This algorithms interacts with the *data stores directory* in order to obtain the data stores capabilities of each cloud provider stored in the *offer manifest*. This manifest represents the second input of this algorithm in order to obtain the *deployment manifest*. The *data stores directory* is automatically updated by interacting with the cloud providers using their APIs.

In the rest of this section, we introduce UML diagram classes illustrating the *Abstract application manifest* and *offer manifest* description. In addition, we introduce our *matching algorithm*.

[3]Roboconf home page: http://roboconf.net/fr/ index.html

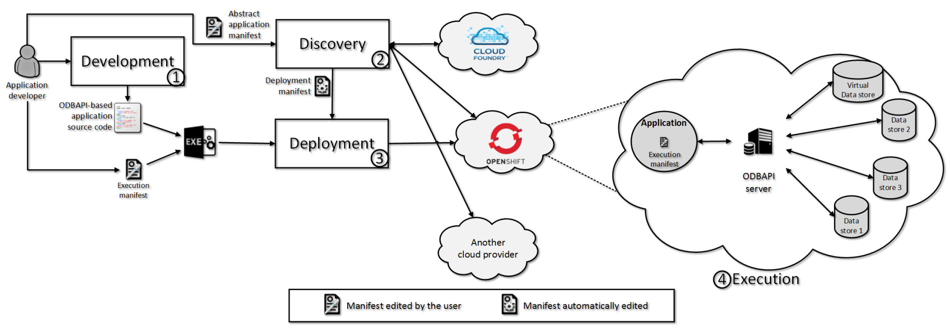


Figure 1: Overview of our solution.

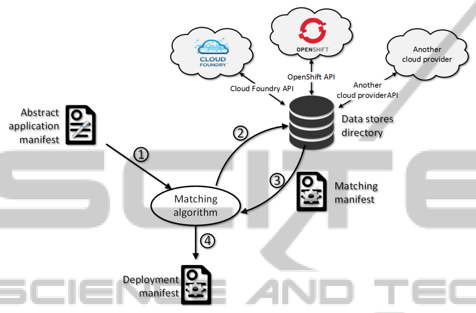


Figure 2: Zoom-in on the discovery module.

4.1 Abstract Application Manifest

This manifest contains two categories of requirements. First, we have requirements in term of data stores. The developer provides five information about the required data stores: its type, its name, its version, it size and the query type to execute. It is worth noting that when the developer fills this manifest, he has the freedom to specify one or multiple information. For each information, he gives a constraint expressed by a constant value, a joker (denoting any values) or some conditions (expressed as inequalities). Hence, we will ensure more flexibility in our model. For instance, one developer precises that he wants a data store having name MongoDB and type document and another developer precises that he wants a data store of type document without specifying its name (in this case any data store of type document fulfill the specification). Whereas the second category of the requirements is dedicated to the application deployment. Indeed, the developer precises the number of the virtual machines that he needs to run his application. In addition, he describes the executable of his application by giving its name, its type and its location.

We depict in Figure 3 an UML class diagram illustrating the *Abstract application manifest* model. The root class of our model is the *Abstract application manifest* class and it is identified by the attribute *name*. This class contains these following classes:

- The *User Information* Class: This class represents the required authentication information required to access the discovery module and consult the data stores repository. This class contains the *user login* class and the *user password* class that represents the developer identifiers.
- The *environment* class: This class represents the environment where an application will be deployed and it is instantiated from an environment *template* element that is characterized by a *name* attribute and a *memory* value. Each environment *template* contains at least one *node* class that represents resources in a cloud provider. A *node* class is identified by an *id* and a *content_type* attributes. This latter can be a *container* or a *database* to denote respectively engine resources to host and run services and storage resources. This class contains a *name* class and a *version* class. When the *content_type* attribute is equal to *database*, the *node* class contains also a *type* class, *query_type* class, and a *size* class to denote the type of data storage services, the query type (CRUD queries or join queries) and its size.
- The *application* class: This class represents the constraints that the user requires to deploy his application. It is characterized by a unique *name* attribute and the *environment* attribute where the application will be deployed. It has at least one *version* class that is identified by a *name* and a *label*. Each *version* class contains two types of classes: *deployable* class and *instance* class. The *deployable* class represents the application executable file. It is identified by a unique *id* attribute, a *content_type* attribute defining the executable file type, a *name* attribute denoting its name, a *location* attribute containing their URL, and a *multitenancy_level* attribute indicating the application tenancy degree. Whereas the *instance* class represents the running application instances required by the user. This class is identified by a unique *id* attribute, a *name* attribute, *initial_state* attribute

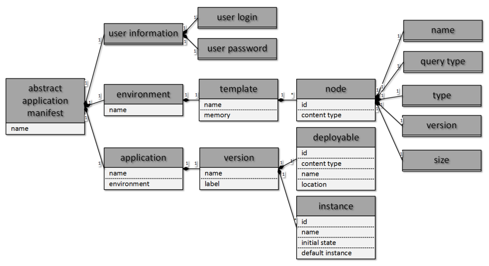


Figure 3: Abstract application manifest model.

defining the state of the application (e.g. running, stopped, etc) and *default_instance* attribute representing the running instances by default.

4.2 Offer Manifest

The offer manifest contains information about the capabilities of data stores of each discovered cloud provider. In Figure 4, we present a offer manifest modeling based on a class diagram. Indeed, the root class is *offer manifest* and it is identified by the *name* attribute. It contains one or multiple *cloud provider* class. This class represents a discovered cloud provider and it is identified by a unique *id* attribute. It contains the *name* class defining the cloud provider name and the *environment_response* class representing the capabilities that a cloud provider exposes according to an *abstract application manifest*. The *environment_response* class is composed by one or multiple *offer* class that contains one or multiple *node* class. This class is similar to the *node* element in the *abstract application manifest* modeling.

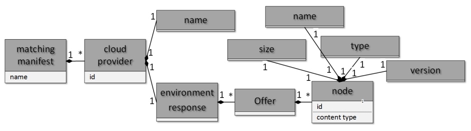


Figure 4: Offer manifest model.

4.3 Deployment Manifest

The *deployment manifest*'s structure is closest to the *abstract application manifest* and it is defined based on the COAPS API's manifest (Sellami and et al., 2013) (see Figure 5). Hence, in order to avoid the repetition, we do not describe this manifest structure in details. However, we will present we extend the COAPS API manifest structure. In fact, we add new attributes about data stores services in the *paas_node* class. These attributes are the *size* attribute, the *type* attribute and the *version* attribute.

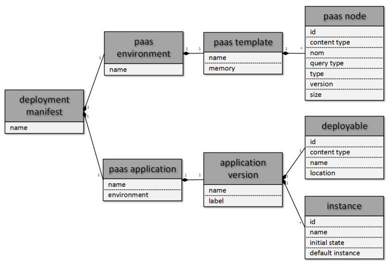


Figure 5: Deployment manifest model.

4.4 Matching Algorithm

In algorithm 1, we introduce the *matching algorithm* that elects the cloud provider that supports the best the application requirements in terms of data stores. We need a flexible matching allowing to elect a cloud provider which does not fulfill all the application's requirements but it is closed to them (that is the role of the computed distance). Furthermore, we impose that the result is a single cloud provider. First, the algorithm constructs the *offer manifest* using the operation *queryDataStoresRepository* by interacting with the data stores repository (line 4). We introduce this operation in Algorithm2. Second, it calculates for each cloud provider the number of differences between its capabilities and the user requirements described in the *abstract application manifest* (lines 6-18). Numbers of differences are stored in the data structure $distance$. These values are calculated as follows: for each property in the two manifest, if they are not corresponding then we update the distance by adding the appropriate penalty to the property. The two properties correspond if the actual value of the offer manifest property fulfill the requirement expressed by the abstract application manifest property (which is either a constant, a joker or a condition). These penalties are customized according to the property importance. By default, all penalties are fixed at 1; however the user can configure these penalties according to the importance that he gives to the properties. Once this step is achieved, we can now elect a cloud provider and construct the *deployment manifest* (lines 19-20). This is done through the operation *electCP* that takes as inputs the the data structure *distance* and the *threshold* and returns the identifier of the elected cloud provider if any. This identifier is the smallest value bounded between 0 and the $threshold$.

In algorithm 2, we present in more details the operation *queryDataStoresRepository* that returns a super-set of the result from the data stores repository. In fact, for each cloud provider it extracts all data stores corresponding to the data types of the *abstract application manifest*. If there are no corresponding data stores, the cloud provider is rejected (lines 10-

Algorithm 1: Matching algorithm.

```
1: input AAM: the abstract application manifest
2: input threshold: the threshold to limit the number of differences
3: output DM: the deployment manifest
4: OM ←queryDataStoresRepository(AAM) # see Algorithm 2#
5: $i \leftarrow 0$
6: while (exist(Cloud Provider CP in OM)) do
7:     while (exist(Offer O in OM)) do
8:         $distance[i] \leftarrow 0$
9:         for each node N in AAM do
10:            for each property prop in N do
11:                if (!valid(prop, OM.CP.O.node.prop)) then
12:                    $distance[i] \leftarrow distance[i]+$ updateDistance(prop, OM.CP.O.node.prop)
13:                end if
14:            end for
15:        end for
16:        $i \leftarrow i+1$
17:    end while
18: end while
19: electedCP ←electCP(distance, threshold)
20: return createDM(AAM, OM, electedCP)
```

Algorithm 2: The *queryDataStoresRepository* algorithm.

```
1: input AAM: the abstract application manifest
2: output OM: the offer manifest
3: $length \leftarrow 0$
4: for each node N in AAM do
5:     if (content-type == ”database”) then
6:         $T[length] \leftarrow$ getType(N)
7:         $length \leftarrow length+1$
8:     end if
9: end for
10: for each Cloud Provider CP in the data stores directory do
11:     for i=0 to length do
12:         if (CP contains T[i]) then
13:             Add all names of this type to the tree of the current CP
14:         else
15:             Reject the current CP
16:         end if
17:     end for
18: end for
19: return the resulted trees as the OM
```

18). For ease of presentation of this algorithm, we build ourselves on the Figure 6. Indeed, in the left side of the figure, we construct from the *abstract application manifest* a simple graph in which nodes represent the *type* elements in the *node* elements. This graph is a kind of a sample that will be used to construct the offer manifest. In the right side of Figure, we illustrate a data stores repository of a cloud provider having four types of data stores. Based on this repository and the graph based sample, we extract the list of the cloud provider's offers that we represent in the form of a graph. Once we check all cloud providers, we collect all the offers in the *offer manifest* (line 19).

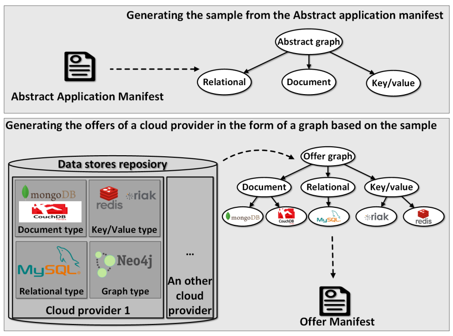


Figure 6: Generating the offer manifest.

5 IMPLEMENTATION AND VALIDATION

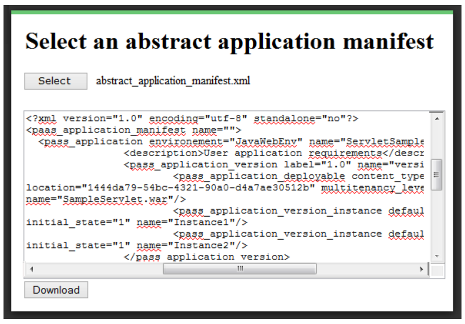
In this section, we present the tool implementing the *matching module*. Hence, in order to show the feasibility of this module, we propose to discover data stores of cloud foundry and open shift as cloud providers to deploy an ODBAPI-based application. This application is intended to handle the administration of relational and NoSQL data stores in a cloud provider. To do so, we propose to implement the manifests modeling with XML. We give in this section the example of the *abstract application manifest* (see Listing 1). Indeed, the developer provides *user1* as a login and *pswd* as a password. Then, he describes the environment that he requires in order to deploy his ODBAPI based application. Indeed, he chooses as name for the environment *ODBAPIEnv* and for the environment template *ODBAPIEnvTemp*. In this template, the user wants *tomcat* as an application container, a document data store without precising any name by filling the element *name* with the character $*$ and MySQL as a relational data stores. Regarding the application configuration, the user names his application *ODBAPIApplication*. He precises that the application is a runnable file and he requires to have two application instances: one is running by default and the other is stopped. In this section we present only the XML based representation of the *abstract application manifest* for lack of space. We wish to emphasize that we provide our tools and videos demonstration at http://www-inf.int-evry.fr/∼sellam_r/Tools/ODBAPI/index.html for more details.

```
<abstract_application_manifest name="AAM">
  <user_information>
    <user_login>user1</user_login>
        <user_password>pswd</user_password>
        <query_type>complex</query_type>
  </user_information>
  <environment name =”ODBAPIEnv”>
    <template name=”ODBAPIEnvTemp” memory=”128”>
```

```
            <node id ="1" content_type="container">
               <name> tomcat </name>
                  <version> </version>
            </node>
            <node id ="2" content_type = "database">
               <name>*</name>
               <version> 1.0 </version>
                  <type> document </type>
                  <size> large </size>
            </node>
            <node id ="3" content_type = "database">
               <name> mysql </name>
                  <version> * </version>
                  <type> relational </type>
                  <size> small </size>
            </node>
      </template>
   </environment>
   <application name="ODBAPIApplication" environement=
   "ODBAPIEnv">
      <version name="version1.0" label="1.0">
         <deployable id="1" content_type="artifact"
            name="ODBAPIApplication.war" location="1444d7"
            multitenancy_level="SharedInstance"/>
           <instance id="1" name="Instance1"
            initial_state="1" default_instance="true"/>
           <instance id="2" name="Instance2"
            initial_state="1" default_instance="false"/>
      </version>
   </application>
</abstract_application_manifest>
```

Listing 1: XML based representation of the abstract application manifest.

We programmed also a tool ensuring the discovery of cloud providers and the automatic deployment of an ODBAPI-based application. Indeed, the application programmer describes his requirements in the *abstract application manifest* and he uploads it through the interface that we illustrate in Fig. 7. Once this manifest is uploaded, this tool executes the *manifest based matching algorithm* to elect the appropriate cloud provider that supports the ODBAPI client requirements and returns the user the *deployment manifest*. Based on the *deployment manifest*, we deploy the ODBAPI client by the mean of COAPS API.

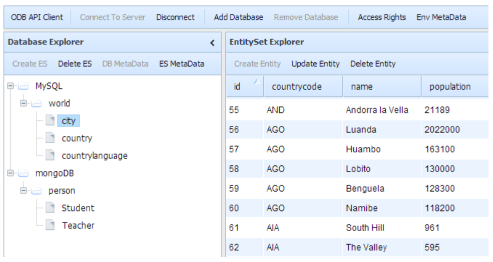


Select an abstract application manifest

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<paas_application_manifest name="">
  <pass_application environement="JavaWebEnv" name="ServletSample
                <description>User application requirements</descr
                <pass_application_version label="1.0" name="versi
                        <pass_application_deployable content_type
location="1444da79-54bc-4321-90a0-d4a7ae30512b" multitenancy_leve
name="SampleServlet.war"/>
                        <pass_application_version_instance defaul
initial_state="1" name="Instance1"/>
                        <pass_application_version_instance defaul
initial_state="1" name="Instance2"/>
                </pass application version>
```

Figure 7: Screenshot of the interface allowing to select the *user manifest* in order to get the *deployment manifest*.

In our work, we deploy an ODBAPI-based client intended to handle the administration of relational and NoSQL data stores in a cloud provider. In Fig. 8, we show an overview of the databases created in each data store in the Cloud Foundry. Indeed, there is a MySQL database called *world* and it contains



id	countrycode	name	population
55	AND	Andorra la Vella	21189
56	AGO	Luanda	2022000
57	AGO	Huambo	163100
58	AGO	Lobito	130000
59	AGO	Benguela	128300
60	AGO	Namibe	118200
61	AIA	South Hill	961
62	AIA	The Valley	595

Figure 8: Screenshot of all databases overview.

three entity sets: *city*, *country*, and *countrylanguage*. Added to that, we have the MongoDB database that is named *person* and it is composed by two entity sets: *Student* and *Teacher*. We show also an overview of the entities of the *city* entity set.

6 RELATED WORK

In our previous work (Sellami and Defude, 2013), we focused on existing solutions of the state-of-the-art supporting multiple data stores based applications in the cloud environment. More precisely, (i) we described different scenarios related to the way applications use data stores, (ii) we defined the data requirements of applications in cloud environment, and (iii) we analyzed and classified existing works on cloud data management, focusing on multiple data stores requirements. As a result, we pointed out six requirements of using multiple data stores in a cloud environment. One of these requirements consists in choosing a data store based on a data requirements. We present three sub-requirements: defining application needs and requirements towards data, defining data store capabilities, and defining application needs and data stores capabilities matching.

Against this background, we find several worksq (Truong and et al., 2012),(Truong and et al., 2011),(Vu and et al., 2012), (Ghosh and Ghosh, 2012), (Ruiz-Alvarez and Humphrey, 2011), (Ruiz-Alvarez and Humphrey, 2012) enabling an application to negotiate its Data Management Contract (DMC), often referred to as data agreement or data license, with various clouds and to bind to the specific DBMSs according to its DMC. Truong et al. (Truong and et al., 2012), (Truong and et al., 2011), (Vu and et al., 2012) propose to model and specify data concerns in data contracts to support concern-aware data selection and utilization. For this purpose, they define an abstract model to specify a data contract and the main data contract terms. Moreover, they propose some algorithms and techniques in order to enforce the data contract usage. In fact, they present a data contracts compatibility evaluation algorithm and they define how to construct, compose and exchange a data contract. In (Truong and et al., 2011), they introduce their model for exchanging data agreements in the Data-as-a-Service (DaaS) based on a new type of services which is called Data Agreement Exchange as a Service (DAES). This model is called DEscription MOdel for DaaS (DEMODS) (Vu and et al., 2012). However, Truong et al. propose this data contract for data and not to store data or to help the developer to choose the appropriate data stores for his application. In (Ghosh and Ghosh, 2012), Ghosh et al. identify non-trivial parameters of the Service Level Agreement (SLA) for Storage-as-a-Service cloud which are not offered by the present day cloud vendors. Moreover, they propose a novel SLA monitoring framework to facilitate compliance checking of Service Level Objectives by a trusted third part. Although Ghosh et al. try to enrich the SLA parameters to support the Storage-as-a-Service, this is still inadequate for our purpose in this paper. In (Ruiz-Alvarez and Humphrey, 2011), (Ruiz-Alvarez and Humphrey, 2012), Ruiz-Alvarez et al. propose an automated approach to selecting the PaaS storage service according an application requirements. For this purpose, they define a XML schema based on a machine readable description of the capabilities of each storage system. The goal of this XML schema is twofold: (i) expressing the storage needs of consumers using high-level concepts, and (ii) enabling the matching between consumers requirements and data storage systems offerings. Nevertheless, they consider in their work that an application may interact with only one data store and they did not invoke the polyglot persistence aspect.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a manifest-based solution for data stores discovery and automatic application deployment. Indeed, once the developer has completed the development of his application, we provided him the possibility to express his application requirements in terms of data stores in the *abstract application manifest*. Then, he sends it to the *discovery module*. This module interacts with the data stores directory to discover the capabilities of data stores of each cloud provider and constructs the *offer manifest*. Based on that, this module implements the *matching algorithm* in order to elect the adequate cloud provider to the application requirements. This algorithm takes as input the *abstract application manifest* and *offer manifest*, and returns the *deployment manifest* of the application. Once it is done, we deploy the application using the COAPS API that takes as input the *deployment manifest*.

Currently, we are working on applying our solution to other qualitatively and quantitatively various scenarios in the OpenPaaS project. This allows us to identify possible discrepancies and make our work more reliable for real use. Our second perspective is to implement virtual data stores in order to execute join queries across NoSQL and relational data stores and to introduce more elaborate query processing optimization techniques.

REFERENCES

Baun, C. and et al. (2011). *Cloud Computing - Web-Based Dynamic IT Services*. Springer.

Ghosh, N. and Ghosh, S. K. (2012). An approach to identify and monitor sla parameters for storage-as-a-service cloud delivery model. In *Workshops Proceedings of the Global Communications Conference, GLOBECOM 2012, 3-7 December, Anaheim, California, USA*, pages 724–729.

McAfee, A. and Brynjolfsson, E. (2012). Big data: The management revolution. (cover story). *Harvard Business Review*, 90(10):60–68.

Ruiz-Alvarez, A. and Humphrey, M. (2011). An automated approach to cloud storage service selection. In *Proceedings of the 2Nd International Workshop on Scientific Cloud Computing*, ScienceCloud '11, pages 39–48.

Ruiz-Alvarez, A. and Humphrey, M. (2012). A model and decision procedure for data storage in cloud computing. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16*, pages 572–579.

Sellami, M. and et al. (2013). Paas-independent provisioning and management of applications in the cloud. In *2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*, pages 693–700.

Sellami, R., Bhiri, S., and Defude, B. (2014). ODBAPI: a unified REST API for relational and NoSQL data stores. In *The IEEE 3rd International Congress on Big Data (BigData'14), Anchorage, Alaska, USA, June 27 - July 2, 2014.*

Sellami, R. and Defude, B. (2013). Using multiple data stores in the cloud: Challenges and solutions. In *Data Management in Cloud, Grid and P2P Systems - 6th International Conference, Globe 2013, Prague, Czech Republic, August 28-29, 2013. Proceedings*, pages 87–98.

Truong, H. L. and et al. (2011). Exchanging data agreements in the daas model. In *2011 IEEE Asia-Pacific Services Computing Conference, APSCC 2011, Jeju, Korea (South), December 12-15*, pages 153–160.

Truong, H. L. and et al. (2012). Data contracts for cloud-based data marketplaces. *IJCSE*, 7(4):280–295.

Vu, Q. H. and et al. (2012). Demods: A description model for data-as-a-service. In *IEEE 26th International Conference on Advanced Information Networking and Applications, AINA, 2012 , Fukuoka, Japan, March 26-29*, pages 605–612.