Supporting Multiple Persistence Models for PaaS Applications using MDE
Issues on Cloud Portability

Elias Adriano Nogueira da Silva[1], Daniel Lucrédio[2], Ana Moreira[3] and Renata Fortes[1]
[1]*ICMC-USP, São Carlos, Brazil*
[2]*DC-UFSCar, São Carlos, Brazil*
[3]*NOVA-LINCS, FCT - UNL, Lisboa, Portugal*

Keywords: Cloud Computing, Model-driven Engineering, Platform-as-a-Service, Portability, Persistence.

Abstract: In cloud computing, lock-in refers to the difficulty of porting an application from one platform to another. An example of such difficulty can be witnessed when porting an application from Platform-as-a-Service Google App Engine to Microsoft Azure. Differences in their implementations are substantial, yielding non-portable applications. Standardization could address this problem, but existing initiatives are still to be accepted. This paper addresses lock-in by proposing a model-driven engineering design approach that decouples platform specific code from the application logic. The resulting platform-independent models, as well as corresponding model transformations, can be reused to generate distinct platform-specific implementations, hence reducing the programming effort spent coding repetitive tasks. Such transformations can be made available for reuse on a repository for cloud providers. We have implemented transformations to handle persistence for Google App Engine and Azure, and discuss how model-driven engineering can reconcile the differences between features of the persistence models of GAE and Azure.

1 INTRODUCTION

In a recent report, the *IEEE Computer Society*[1] highlights 22 technologies with potential to change the scenario of computer science and its role in industry until 2022 (Alkhatib et al., 2014). Apart from foreseeing relevant research roadmaps, it also discusses a vision for each of those technologies. One such technology is cloud computing.

Cloud computing is not a new technological model, but the integration of technologies from the past (Chen et al., 2011). What is new though, is the different ways in which it is used to provide computing power as-a-service through the Internet. According to Armbrust et al., cloud computing permits acquiring computing resources on demand, enables payment according to the utilization volume, and allows a company to ignore the sources of the resources (Armbrust et al., 2009).

Several technological requirements are needed for the cloud model to operate properly. The most common are virtualization technologies, standards and interfaces that allow shared access, facilitated instantiation and management of virtual servers (IaaS – Infrastructure-as-a-Service). Moreover, there are different kinds of resources in the cloud. Load balancing, data persistence and analytics are just some of the many options available for application developers. Given this variety, and depending on its field of expertise, each cloud provider offers a different set of computational resources. Some even provide a complete development platform (PaaS – Platform-as-a-Service) that puts together many different resources under control of the cloud provider.

Both the heterogeneity and diversity of cloud services result in increased complexity as well as reduced reuse and portability of the applications (da Silva et al., 2013). In practice, some applications need to be highly specialized with respect to a particular type of resource (e.g. hardware, platform and/or set of services), yielding the *lock-in* problem (Armbrust et al., 2009). For example, when choosing a particular PaaS provider, the application developer usually has to follow a specific data management system and programming style. This typically reduces porta-

[1]http://www.computer.org/

Adriano Nogueira da Silva E., Lucrédio D., Moreira A. and Fortes R..
Supporting Multiple Persistence Models for PaaS Applications using MDE - Issues on Cloud Portability.
DOI: 10.5220/0005441203310342
In *Proceedings of the 5th International Conference on Cloud Computing and Services Science* (CLOSER-2015), pages 331-342
ISBN: 978-989-758-104-5

bility, resulting in applications "locked-in" to that particular environment.

Some strategies based on standardization have been proposed to address that portability issue (Armbrust et al., 2009; Petcu and Vasilakos, 2014). However, standards take time to define and approve, and require time to be accepted by a large part of development community. Currently, there are so many different standards being proposed (Petcu and Vasilakos, 2014) that even choosing one may be a difficult task. Moreover, cloud providers may wish to use specific technologies to create solutions that are aligned with their own business requirements, hence choosing not to follow the standards. Thus, until standardization becomes a fact, the portability problem, and in particular lock-in, remains.

We have been exploring how Model-Driven Engineering (MDE) (see Section 2) can be used to address portability (da Silva et al., 2013) Approaches based on MDE (France and Rumpe, 2007) have been investigated in several other contexts and may constitute an interesting alternative to address the problem. Our long-term goal is to build a repository of MDE transformations and use code generation to reduce the development effort for each platform, consequently reducing repetitive programing tasks, increasing portability and minimizing the lock-in effects.

The present paper takes Google App Engine (GAE) and Microsoft Azure, two well-known platforms available in the market, and shows how MDE conciliates the differences between their cloud persistence models. We show how these differences can be hidden behind a single conceptual model and discuss a set of MDE artifacts to support this idea. This can be seen as an abstraction layer that allows specifying entities and a set of code generators that use these entities to build similar persistence models even if the storage mechanisms are different.

The two central points of the idea are (i) using a DSL for modeling entities and (ii) building a set of transformations that can generate code for different targets from the same set of source models. Such code-generation approach allows developers focusing on platform-independent models, thus achieving portability by reducing the lock-in effects. Both the models created using DLS and their respective transformations for various different platforms can be made available in a repository for reuse.

In a previous paper (da Silva et al., 2013) we discuss the general approach, but we do not show how persistence is dealt with, which is one of the most interesting parts of our work. Here we extend that work, presenting the differences between the persistence models of GAE and Azure. We also show details of how these differences can be conciliated through an MDE process, resulting in applications that can be more easily ported between these two cloud providers.

As our approach is generic, transformations considering standards may also be added to the repository later. Although the typical claimed MDE-benefits are expected (e.g., facilitated maintenance and increased productivity), an analysis of the economic viability of creating and maintaining a repository of transformations is out of the scope of this paper[2].

The rest of this paper is organized as follows. Section 2 presents some conceptual background, including a more detailed definition of the lock-in problem, the different types of cloud portability, concepts of MDE and an overview of the previously proposed MDE approach. Section 3 discusses the two platforms that were the subject of this study (GAE and Azure), focusing on the differences in their persistence models. Section 4 presents our proposed solution using MDE and Section 5 discusses some points about the performed evaluation of the proposal. Section 6 presents related work and, finally, Section 7 concludes with some final remarks and future work.

2 BACKGROUND

This section starts with a discussion of lock-in and known types of portability. It then introduces model-driven engineering, and finishes with a summary of our vision on the use of MDE to support PaaS portability.

2.1 The Lock-in Problem

Lock-in is the difficulty faced to move data and programs from one cloud platform to run on another one (Armbrust et al., 2009). This is a major issue in the PaaS scenario: in order to take advantage of a very flexible cloud architecture, the applications are developed conforming to the specificity of the chosen platform. For example, to offer great elasticity, the GAE PaaS provider imposes a specific programming style and specific data management policy. Thus, an application developed for it may not be easily ported to a different PaaS provider, nor can its data. Even if the developer wants to host an application in his own private cloud later, considerable effort may be necessary to rebuild the code, redeploy it, and migrate all

[2]Mohaghegi and Dehlen presented a review of experiences from applying MDE in industry (Mohagheghi and Dehlen, 2008).

the data. This lack of portability causes the lock-in effect.

The possibility of becoming "locked in" on a particular platform, not being able to choose a different one later (customer lock-in), leaves developers in a difficult position. They mostly fear being charged abusive fees later, or having their applications unavailable due to lack of service quality (Armbrust et al., 2009).

2.2 Types of Portability

Prior to deciding on the adoption of a cloud model, an organization should take into account the viability of the one that better fits its business. It must carefully analyze the constraints related to cloud platforms, both technical and organizational (da Silva et al., 2013), as well as its business requirements (Khajeh-Hosseini et al., 2011).

Portability is a key attribute for the improvement and dissemination of the cloud model.The existing literature discusses four main types of portability in the cloud scenario: (i) portability of virtual machines between cloud providers, (ii) portability of applications in the context of IaaS, (iii) portability of PaaS applications and (iv) data portability between cloud providers(Bozman, 2010; Petcu et al., 2013; Ranabahu and Sheth, 2010; Shirazi et al., 2012).

This paper focuses on portability of PaaS applications. However, the code generators and repository proposed here can be used in other contexts. Petcu et al. present a list of initiatives to handle portability (Petcu and Vasilakos, 2014). They also discuss the reasons, scenarios, taxonomies, measurements, and requirements for portability. Several other authors are also looking at the problem and proposing alternatives to address it (see Section 6). One such alternative is the use of model-driven engineering (MDE).

2.3 Model-Driven Engineering(MDE)

Despite the advancements of the software development techniques, concerns about reuse, productivity, maintenance, documentation, validation, optimization, portability and interoperability are still under discussion.

Model-Driven Engineering (MDE) aims at solving some of those issues (Kleppe et al., 2003), shifting the focus of modern development methodologies from implementation to conceptual modeling. Thus, models are now first-class citizens, and transformation mechanisms are used to generate code from them, reducing developers' effort (Kleppe et al., 2003) and increasing portability and productivity (da Silva et al., 2013). The vision is that MDE will reduce the accidental complexity by increasing the level of abstraction used to develop software.

According to Schmidt, MDE technologies "offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively" (Schmidt, 2006). That is exactly our goal: use MDE to abstract away platform-specific details, building conceptual domain models that express the essence and logic of the domain. From these models, applications can then be generated through automatic transformations, thus reducing the development effort for implementations on different platforms.

Such models are abstract descriptions or specifications of the system and are usually represented as a combination of graphical (Domain-Specific Modeling Languages – DSML) and textual elements (Domain-Specific Languages – DSL) (Brambilla et al., 2012). DSLs are small languages focused on a particular problem/domain, and are normally declarative (Deursen et al., 2000). The language definition usually requires a metamodel which is capable of capturing the common and variable points of a specific domain (Brambilla et al., 2012; Deursen et al., 2000).

2.4 A Model-driven Approach for Cloud PaaS Portability

In a previous study (da Silva et al., 2013) we discussed a vision for using MDE to increase cloud PaaS portability, and discussed how to build a DSL and a set of code transformations, based on the Model-View-Controller (MVC) architecture, to reduce the effort of developing cloud applications. We also presented a DSL metamodel, samples of code transformations, the grammar of the DSL and a quasi-experiment (Wohlin et al., 2000; Juristo and Moreno, 2010) showing that MDE helps both reducing the development effort and achieving portability. Fig. 1 summarizes the methodology followed.

Adopting a typical MDE life-cycle, this methodology obeyed to the following strategy:

1. Case studies were developed to identify the main concepts of PaaS. These studies involved a careful analysis of the different providers' documentation, as well as the development of sample applications for different platforms.
2. Next, these concepts were used to prototype a specification language. This language serves to support the creation of platform-independent models that developers will use to specify the applications' structure and logic. This step involves

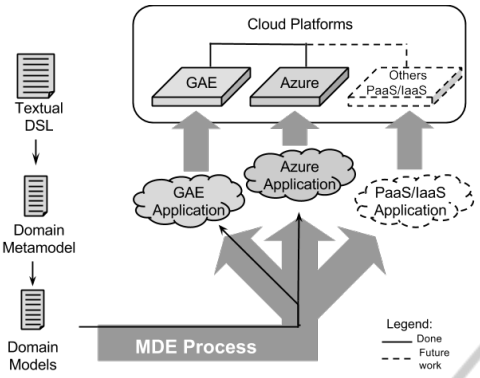

Figure 1: A MDE approach for cloud PaaS portability.

the development of a domain metamodel and a concrete syntax.

3. Based both on the case studies and on the specification language, transformations were defined to automatically generate code for cloud platforms.
4. Tests were performed to verify the conformance between the generated code and the platforms' requirements.

3 PERSISTENCE IN PaaS

The PaaS model leverages the flexibility of the cloud model, by providing a complete platform for software development. A cloud platform hides many of the complexities of developing cloud software, therefore increasing scalability and elasticity. In the PaaS model, the development platform is provided as-a-service. Applications that are developed for this particular platform can benefit from a specific programming model that can be fully, fine grained, managed by the platform provider.

Among the existing platforms, we selected two well-known ones for developing our prototype: the Google App Engine[3] (GAE) and the Windows Azure. However, as the developed DSL is platform independent (although domain dependent), it can be used to generate code for any other platform. One of the services managed by the provider is data persistence. By defining its own way to store data, a provider may incorporate services such as load balancing, automatic data distribution and optimized querying. This is so for the two selected platforms for our study. Both GAE and Azure offer PaaS solutions incorporating NoSQL storage. This service is provided to applications through simple configuration steps.

[3]https://cloud.google.com/appengine

Actually, Azure offers two types of cloud services: IaaS and PaaS. It is not hard porting an IaaS application because this offer is based on virtual machines (VM). Just migrating a VM from one provider to another causes little impact on the systems being virtualized, as all that is needed for them to run is a copy of the virtual disk. The Open Virtualization Format (OVF[4]) makes this task even easier, by providing a standard format so that there is little effort to port a virtual machine from one provider to another, as long as both support this format. The main issue in this case is to choose a different provider that accepts the same VM format[5].

However, in terms of PaaS, both Azure and GAE offer solutions based on Java servlets and JSP with NoSQL storage. Even if they allow the same set of technologies (Java based), applications implemented for them are not portable. This happens mainly because of the differences between their persistence models. (Section 3 discusses these models in detail.) Indeed, Gorton in a post[6] at Software Engineering Institute's blog and Armbrust et al. (Armbrust et al., 2010) highlight that the differences in data management technologies make applications less reusable by different providers.

The next subsections present specific details of each platform persistence model, and finish with a discussion of the main issues found.

3.1 Google App Engine

Google App Engine DataStore is typically one of the first choices for big data applications. The DataStore is GAE's native API and its scalability is managed by the platform itself, which means that the user does not need to worry about the actual storage details.

GAE's DataStore offers two mechanisms to specify persistent entities: Java Data Objects[7] (JDO) and Java Persistence API[8] (JPA) . The JDO and JPA interfaces are implemented using the Datanucleus[9] platform, which is an open-source implementation of JDO and JPA. With JDO/JPA, GAE allows the definition of simple entity relationships. As a result, even without direct relational support from the actual

[4]OVF: http://www.dmtf.org/standards/ovf

[5]Paasify may be an interesting solution to select compatible PaaS: http://www.paasify.it/vendors.

[6]http://blog.sei.cmu.edu/post.cfm/importance-software-architecture-big-data-systems-013

[7]http://www.oracle.com/technetwork/java/index-jsp-135919.html

[8]http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

[9]http://www.datanucleus.org/

database system, applications can use GAE's DataStore to manage related entities.

For simplicity reasons, we chose JDO for this study. To persist an entity in GAE's DataStore, all that is necessary is to annotate a class according to the JDO specification. Related entities (one-to-one and one-to-many) are also managed by GAE automatically through proper annotations.

Let us consider a simple example: a clinical laboratory system must maintain a record of customers, doctors and examinations; each customer has one doctor, and each doctor may choose among a set of examinations to be performed.

The first step is to annotate the classes that represent persistent entities according to the JDO specification. After this, calls to JDO's CRUD[10] methods can be used directly. In summary, all that is needed to make an entity persistent are some annotations. The actual storage of the entity and its related entities is performed by the platform.

It is important to stress that even with the possibility to define simple relationships through annotations, the GAE DataStore service is a NoSQL solution. If a relational solution is needed, a fully-fledged SQL solution, such as the MySQL-based service offered by GAE (Google Cloud SQL), is recommended. A trade-off between scalability and robustness is necessary in these cases.

3.2 Windows Azure

Windows Azure is the Microsoft's cloud platform, which offers different services, such as virtualization, storage and web hosting. Similarly to GAE, Azure's PaaS solution supports regular web-based applications (pages, controllers and other classes/libraries), but with a wider choice of languages (.Net, Node.js, PHP, Java, etc.).

Azure offers persistence through four main storage options:

- **Table Storage:** this is Azure's NoSQL persistence solution. It is a simple persistence model that allows applications to store basic data types (e.g., integer, string, boolean). It is a highly scalable solution, but with three major restrictions. First, Azure's Table Storage structures do not directly support relationships between entities. Second, there is a limit of 255 properties per entity, and every entity must define at least two properties for identification, which leaves 253 properties for general use. Third, data in a single entity cannot exceed one MByte.
- **SQL Database:** formerly known as SQL Azure, this is a fully managed relational database service. Being a relational database, it is not as scalable as the NoSQL service.
- **SQL Server in Windows Azure VM:** if the developer wants more control over the DBMS, he may opt to deploy his own instance of SQL Server in a virtual machine. This renders more control, but also requires more effort to setup, manage the database server, and the virtual machine.
- **Blob Storage:** this service supports the storage of large, non-structured data. It has great scalability, but it is focused on files like audio and video.

As Azure also allows NoSQL services, which offer a good combination between storage and scalability for big data applications, we also chose this model in Azure for our study. However, unlike GAE JDO/JPA-based implementation, Azure does not have an official support for JPA/JDO. As a result, the developer has to deal with relationships manually. Additionally, there are many restrictions in Azure, for example: persistence is defined through inheritance, and not annotations as in GAE; the identification field (primary key) has to be manually managed.

In Azure Table Storage, entities are stored in table structures called partitions. One partition can store multiple entities, which may be of different types. An entity has a unique identification field. Partition names and identification fields are both strings, and are inherited by the entity classes.

To perform CRUD operations, the Table Storage API has some predefined methods. Listing 1 shows an example of how an entity can be persisted. The method "saveOrUpdate" (line 1) is used to either create or update an entity. First, a table client object is obtained ("tableClient"), based on some predefined connection string (line 2). Next, a table operation is created, in this case, to insert or replace an entity (line 3). Then, the table (partition) is created, if it does not exist already (line 5). Finally, the operation is executed (line 6). In this example, for simplicity, the name of the partition and of the entity class will be the same.

Listing 1: Persisting an entity in Azure.

```
1 public void saveOrUpdate(
      TableServiceEntity tse) {
2     CloudTableClient tableClient =
          CloudStorageAccount.parse(
          storageConnectionString).
          createCloudTableClient();
3     TableOperation tableOperation =
          TableOperation.
          insertOrReplace(tse);
4     try {
```

[10]CRUD: Create, Retrieve, Update and Delete.

```
5          tableClient.getTableReference
               (tse.getClass().
               getSimpleName()).
               createIfNotExist();
6          tableClient.execute(tse.
               getClass().getSimpleName
               (), tableOperation);
7      } catch (StorageException e) {
           ... }
8  }
```

Dealing with relationships requires manual management of the id fields. For one-to-one relationships, it is possible to simply store the id field of the related (dependent) entity as a property in the container entity. For example, in the customer-has-a-doctor one-to-one relationship, to obtain the doctor for a given customer, first we obtain its id field, and then we perform a query in the Doctor table.

For one-to-many or many-to-many relationships, the strategy is to maintain a separate entity for relationships. Listing 2 illustrates the idea. In this example, "Relationship" (line 1) is a persistent entity that merely stores two string values: the "end1" and the "end2" (lines 2 and 3), each representing an end of the relationship. This entity will be stored in a partition of its own, called "Relationship" (line 5).

Listing 2: Persistent relationship in Azure.

```
1 public class Relationship extends
      TableServiceEntity {
2     private String end1;
3     private String end2;
4     public Relationship() {
5         this.partitionKey = "
              Relationship";
6     }
7     ... setters and getters ...
8 }
```

The "Relationship" entity from Listing 2 can be used to establish a relationship between any two entities. A Method "saveRelationship" realises a relationship between two entities, instantiating the "Relationship" entity and persisting it using calls to Azure's API.

Once the relationship is established, retrieving all related entities can be done by searching through the "Relationship" partition. The example of Listing 3 shows a way to implement this strategy. The method "getAllRelatedEntities" (line 1) gets all entities related to a given entity. The id of the containing entity and the class of the related entity are provided as arguments. First, all relationships are retrieved (line 2) through the "getAll" method, which is not shown here but should be trivial to imagine. Then, the resulting list is iterated in search for instances that have a matching "end1" property (lines 4-5). For those matching relationships, the instance corresponding to the "end2" is retrieved and added to the result (line 6). A method "retrieve", which is not shown here, looks into the partition of the corresponding entity class and returns the instance itself.

Listing 3: Retrieving related entities in Azure.

```
1 public List getAllRelatedEntities(
      String end1Id, Class end2Class) {
2     List<Relationship> temp = getAll(
          Relationship.class);
3     List result = new ArrayList();
4     for (Relationship r : temp) {
5         if (r.getEnd1().equals(end1Id
              )) {
6             result.add(retrieve(
                  end2Class, c.getEnd2
                  ()));
7         }
8     }
9     return result;
10 }
```

The implementation of Listing 3 is not very efficient, as it examines all relationships every time. However, it is not difficult to optimize this code with more refined structures such as trees or hash functions.

3.3 Difficulties in Conciliating Both Persistence Models

Although both GAE and Azure offer NoSQL services, GAE adds a layer that facilitates the management of relationships between persistent entities, while Azure demands some additional effort to be able to deliver similar functionality. The problem, however, is not the extra effort required by Azure. In fact, the jpa4azure[11] third-party API, adds an object-relational mapping layer to Azure, similar to what is natively available in GAE. (At the time we started our research, this API was not stable, at least according to our tests; so we decided to implement our own layer.) The problem, really, is that even allowing the use of the same set of technologies, the differences between the platforms impose specific programming styles on developing for each one. For this reason, the effort spent on specific programing tasks cannot be reused. Even considering the existence of a common API, the problem remains, due to the differences between the implementations and storage philosophies. Standardization could be an alternative, but as we discussed before, it is not the path followed in this work.

[11] https://jpa4azure.codeplex.com/

Hence, despite the apparent similarities of the platforms (which use the same set of technologies: Java back-end, web-based front-end, and NoSQL persistence), the resulting applications have considerable differences. If for a small application like the one presented here the differences are so substantial, in a real case, managing thousands of persistent entities, the effort of developing such a system can increase very fast. If we consider other platforms, supporting different technologies such as Redis[12] or memcacheDB[13], the problem becomes even worse.

We argue that MDE can solve the portability problem in a more fundamental way, reaching flexibility levels that no API or standard can provide. The next section describes our proposal, based on a single platform-independent development model that hides the details of the platforms. This proposal also helps to reduce the extra effort needed by Azure, or any other platform that uses different technology.

4 SUPPORTING MULTIPLE PAAS PERSISTENCE MODELS USING MDE

This section presents a model implemented using the previously developed DSL, discusses the specific details of the generated code for GAE and Azure, gives a synthesis of the whole generation process, and offers some highlights on the work done.

Listing 4 presents the model for the clinical laboratory system. This example uses the language presented in a previous work (da Silva et al., 2013), which is summarized next. First, the model defines some basic configuration properties, such as the application name (line 1), visual theme (line 2), version (line 3), title (line 4), and a set of tabs to be displayed in the main interface (lines 5-10). Next are the entities and their relationships. The syntax is straightforward. Some points to highlight are the definition of the primary keys (lines 14, 27 and 34), which are inspired by JDO's annotations, and the possibility to define custom labels to be displayed in the main interface (line 36).

Listing 4: Model of the clinical laboratory system.

```
1  application weblab {
2    theme = "default"
3    version 1
4    title "WebLab - Exam Requests"
5    tab tab1 {
6      title "Requests"
7      contains : Customer
8      contains : Examination
9      contains : Doctor
10   }
11 }
12
13 entity Customer {
14   pk { id:Key(strategy=IDENTITY)
         readOnly=true }
15   property name : String
16   property address : String
17   property email : String
18   property phone1 : String
19   property phone2 : String
20   property birth : Date
21   property doctor : Doctor
22   property gender : String
23   property examinations : Examination
         []
24 }
25
26 entity Examination {
27   pk { id:Key(strategy=IDENTITY)
         readOnly = true }
28   property name : String
29   property material : String
30   property price : Double
31 }
32
33 entity Doctor {
34   pk { id:Key(strategy=IDENTITY)
         readOnly= true }
35   property name : String
36   property nr : String title = "
         License Number"
37 }
```

[12]http://redis.io/

[13]http://memcachedb.org/

Listing 4 also shows the relationships established for this system. One customer has one doctor (line 21) and many examinations (line 23 - the `[]` suffix indicates that a property may have multiple instances). These appear in the model as properties mapped to other entities.

We developed two sets of transformations, one for GAE and another for Azure. A more generic view of this process can be seen in previous work (da Silva et al., 2013). Here we extend that description by detailing how persistence can be handled. The resulting transformations are to be collected to populate our repository.

4.1 Generating Persistence Code for GAE

Since GAE has JDO support, the transformations are not too difficult to define. One JDO-annotated Java class is generated for each persistent entity, including its properties and relationships. There is a single,

generic, non-generated data-access object (DAO) that performs basic CRUD operations. The invocations of the CRUD operations for each entity are generated in specific controller classes. One controller class is generated for each entity.

Listing 5 shows part of the generated controller class for the "Customer" entity in GAE. The method "saveCustomer" (line 3) persists a customer, given its properties and the doctor's id. Among other actions, such as obtaining parameters from the HTTP request and dealing with errors and page re-directions, this controller method retrieves the corresponding doctor (line 5), associates it with the customer being persisted (line 6), and saves the instance (line 7). Please, note the calls to the generic DAO in lines 5 and 7.

Listing 5 also shows how one-to-many relationships are persisted. The method "addExaminationToCustomer" (line 11) first obtains the related entities, in this case, customer (line 13) and examination (line 14), then it adds the examination to the customer's list of examinations (line 15), and finally it asks DAO to persist the customer and its examinations (line 16). Please note the calls to the generic DAO in lines 13, 14 and 16.

Listing 5: Generated Controller for GAE.

```
1  public class CustomerController {
2      ... // other controller actions
3      public void saveCustomer(Customer
           c, int doctorId) {
4          ... // other actions
5          Doctor d = (Doctor)
               GenericDAOJDO.INSTANCE.
               retrieve(Doctor.class,
               doctorId);
6          c.setDoctor(d);
7          GenericDAOJDO.INSTANCE.save(c
               );
8          ... // other actions
9      }
10
11     public void
           addExaminationToCustomer(int
           customerId, int examinationId
           ) {
12         ... // other actions
13         Customer c = (Customer)
               GenericDAOJDO.INSTANCE.
               retrieve(Customer.class,
               customerId);
14         Examination e = (Examination)
               GenericDAOJDO.INSTANCE.
               retrieve(Examination.
               class, examinationId);
15         c.addExamination(e);
16         GenericDAOJDO.INSTANCE.save(c
               );
17         ... // other actions
18     }
19 }
```

4.2 Generating Persistence Code for Azure

For Azure, one class per persistent entity is generated. For basic CRUD operations, as well as for dealing with relationships manually, there is a single, generic, non-generated data-access object (DAO). Listings 1, 2 and 3 illustrate the idea of how this generic DAO works. Finally, invocations to the CRUD operations are generated in controller classes, similarly to GAE. Listing 6 shows part of the generated controller class for the "Customer" entity in Azure. It is similar to the GAE controller, with the following three differences:

- the relationship between customer and doctor (one-to-one) is based exclusively on the doctor's id (line 5);
- ids need to be manually managed. In this case, and for simplicity, a random unique id is generated whenever a new entity is persisted (line 6);
- the relationship between customer and examination (one-to-many) is established by persisting a new relationship entity (line 13). This "saveRelationship" method is related to relationship shown in Listing 2.

Listing 6: Generated Controller for Azure.

```
1  public class CustomerController {
2      ... // other controller actions
3      public void saveCustomer(Customer
           c, String doctorId) {
4          ... // other actions
5          c.setDoctor(doctorId);
6          c.setId(UUID.randomUUID().
               toString());
7          TableStorage.INSTANCE.save(c)
               ;
8          ... // other actions
9      }
10
11     public void
           addExaminationToCustomer(
           String customerId, String
           examinationId) {
12         ... // other actions
13         TableStorage.INSTANCE.
               saveRelationship(
               customerId, examinationId
               )
14         ... // other actions
15     }
16 }
```

4.3 The Code Generation Process

The architecture of the generated applications is similar for GAE and Azure. Two JSP pages (for editing and listing entities) are generated for each entity, as well as one persistent class and one controller. Figure 2 illustrates this architecture.

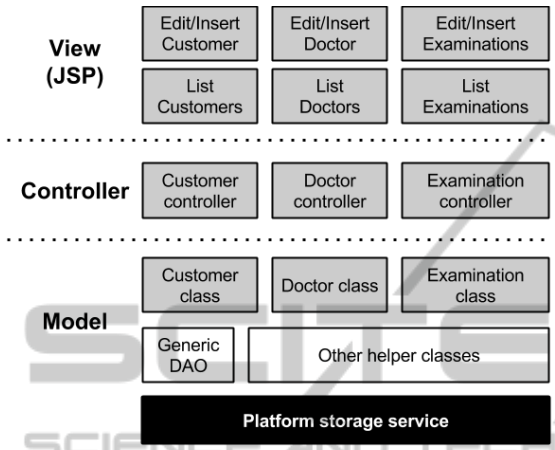

Figure 2: Architecture of the generated applications (GAE and Azure). Shaded elements are generated. White elements are non-generated. Black elements represent platform services.

However, there are significant differences between these two platforms. In GAE, there is no need to manually deal with relationships or identification fields. Hence, the generated application is simpler, with annotated entities and basic controller classes being generated straightforwardly. A simple, generic DAO was enough for basic CRUD operations.

For Azure, the generated applications are less simple. The extra layer to deal with relationships resulted in a more complex generic DAO. There is also the need to deal with identification fields.

5 FINAL DISCUSSION

The strategy presented here is similar to Object-relational mapping (ORM) frameworks like Hibernate. However, while Hibernate uses annotated java classes or a relational-entity model to generate SQL commands and other elements of the software such as classes, views, controllers and database structure, we use a DSL for modeling entities and generate MVC applications including annotated classes, based on specific details of each PaaS embedded in MDE transformations. Both strategies use code generation, but our approach can be considered as a level above and could even include ORM frameworks. Once the transformations are defined, the developer no longer needs to worry about platform-specific details. As long as the transformations are correct, s/he only needs to work on the platform-independent model. In the end, applications developed with our approach can be made as portable as necessary, by including new transformations to support other platforms or technologies.

Adjustments and adaptations in the code generation process, if necessary, become less frequent over time, and the investment made through this extra effort eventually pays off. In this research, the initial infrastructure described here was built by a single developer in a period of 3.2 months, including the time to study the related technologies (Xtext[14] and Xtend[15]).

From an evidence-based point of view (see for example (Tichy, 1998; Juristo and Moreno, 2010; Wohlin et al., 2000)), the case study discussed constitutes some evidence that it is possible to use our approach to deal with different persistence models at an higher level of abstraction and to port applications between different cloud providers. But to reinforce such evidence, we performed a more careful evaluation.

We defined a set of test cases, which 10 users executed on the same application generated for the two platforms (GAE and Azure). After executing the tests, the users perceived no difference in terms of functionality, what indicates an evidence that portability can be achieved by means of our approach. We also observed considerable gains in productivity, due to the automation power of MDE transformations.

From that evaluation, we concluded that it was possible to port an application between cloud platforms in such a way that the final users do not perceive the differences when using the two versions. This is particularly interesting if we consider that the underlying data management mechanisms are different, as discussed in Section 3. This promotes MDE as a possible alternative to port application between different cloud providers.

6 RELATED WORK

There are several different proposals for developing portable cloud applications, being standardization and open source software the more popular in the industry. In academia, many authors also attempt to use MDE to solve to lock-in problem.

Sharma and Sood (Sharma R., 2011) present a model-driven approach for interoperability in SaaS

[14]https://eclipse.org/Xtext/

[15]http://eclipse.org/xtend/

(Software-as-a-Service). They define the models at different abstraction levels, based on the separation of concerns between CIM (Computation Independent Model), PIM (Platform Independent Model) and PSM (Platform Specific Model), hence building on MDA[16]. Each level can be composed by one or more models to specify the structural, functional and behavioral aspects of a system. For PIMs a formal definition of the operations offered by the service is used, which can be accessed through an interface that must later be composed with other services to build a complete system. Business rules are specified through the declaration of restrictions, pre-conditions and post-conditions and invariants in OCL (Object Constraint Language). Transformations convert the PIM into a Web-Service Description Language (WSDL) PSM. The final step is the transformation of the WSDL PSM into WSDL specifications. The main difference from our work is that they use MDE to generate WSDL. Their approach is for SOA while ours is specific for cloud PaaS.

Miranda et al. present their vision on how MDE can support the development of adaptive multi-cloud applications, thus integrating MDE and Software Adaptation techniques (Miranda et al., 2013). Developers are requested to tag the components indicating in which cloud they will be deployed. MDE techniques are then applied to generate an XML-based cloud deployment plan. The source code and the XML deployment plan are processed to generate cloud compliant artifacts to access the underlying cloud services. This work aims at generating the deployment plan while our targets the design and development time.

MODAClouds[17] (MOdelDriven Approach for the design and execution of applications on multiple Clouds) aims at supporting system developers and operators in exploiting multiple clouds and in migrating their applications from cloud to cloud as needed (Ardagna et al., 2012). Its main objective is to provide methods, a decision support system, an open source IDE and runtime environment for the high-level design, early prototyping, semi-automatic code generation, and automatic deployment of applications on multiple clouds. It also helps administrators to monitor the services and measure their quality. While the project is developing a post-fact adoption standard (Petcu, 2011) with CloudML, a domain-specific modeling language and runtime environment that facilitates the specification of cloud application provisioning, deployment, and adaptation, we argue that each enterprise can build its own language or generation strategy more aligned with their business.

The REMICS project proposes an approach for migrating legacy systems to the cloud (Mohagheghi and Sæ andther, 2011; Mohagheghi and Dehlen, 2008). Formed by a consortium of several research institutions, consulting and cloud users, the REMICS has a robust design. Its main purpose is to specify, develop and evaluate a tool for migrating services using MDE. The proposed migration process consists of understanding the legacy system in terms of its architecture and functionality, and designing a new Service-Oriented Architecture (SOA) application that provides the same or better functionality. This project is more related to reenginering and migration strategies for legacy applications while ours is for new ones.

All these approaches use MDE to protect the developer from platform details, which is one of the intended uses of MDE. Our approach focuses on PaaS portability, with special emphasis on persistence. Our results are similar to what is seen in the literature, combining the portability of MDE with its inherent productivity benefits, we expect that our efforts support the leveraging of this new computation model.

A strategy to solve the portability without MDE is described in (Giove et al., 2013). Giove et al. propose a library called CPIM (Cloud Provider Independent Model), that encapsulates PaaS-level services such as message queues, noSQL, and caching. Instead of relying on the providers following a standard, they add a mediation layer that hides the details of the underlying PaaS provider and exposes a common API that allows platform-independent code to be developed on top of it. The result is that applications can be more easily ported between providers, as long as both sides of the implementation (application and supporting library) comply with the mediation layer. Their currently supported platforms are GAE and Azure, but new platforms can be added by providing a proper library to the layer.

Both our approach and the CPIM library attempt to deal with the differences between PaaS services. Both agree that standardization may not be the only solution. And both allow platform-independent applications to be specified. Our proposal has the advantage of allowing developers to work on a higher abstraction level. Therefore, we can collect additional benefits in terms of productivity and maintenance. On the other hand, CPIM requires no effort to setup a modeling and code generation environment, resulting in less upfront investment and being easier to adopt. In fact, an hybrid solution, combining MDE and a mediation layer, could bring benefits from both approaches.

[16]http://www.omg.org/mda/

[17]http://www.modaclouds.eu/

More research issues and approaches related to the development of systems to the cloud model can be found in Armbrust et al. (Armbrust et al., 2009) and our previous work (da Silva et al., 2013). Cloud computing is still evolving, and research opportunities are still being identified. The presented approaches are still being investigated and are far from being mature. More research and evaluations are still necessary.

7 CONCLUDING REMARKS AND FUTURE WORK

This paper shows how the differences in cloud persistence models can make an application difficult to reuse and/or be ported to a different provider. It extends our previous work (da Silva et al., 2013) on exploring the use of MDE to overcome portability in cloud computing, and shows how that previous approach can be used to solve the persistence related lock-in issue.

The main contribution of our work is to show that there is an alternative path to the standardization of cloud technologies. MDE can increase the portability of the applications, but it can also lead to additional benefits inherently associated with it, consequently, reducing the impacts of lock-in.

Our approach is focused on persistence, and therefore it has good support for CRUD operations.

A limitation of our approach, that is inherent to most MDE approaches, is that if the generated code needs to be adapted or modified, the MDE life-cycle can be broken. Changes in the generated code have to be replicated, either in the models or in the transformations, which is not a trivial task. This is why it is often recommended to leave generated code unmodified[18].

In the near future we plan to include more platforms to implement the repository of models and transformations, and to perform some more evaluations, which includes applying our approach to other case studies.

ACKNOWLEDGEMENTS

We would like to thank FAPESP (processes 2012/24487-3 and 2012/04549-4), Coordination of Superior Level Staff Improvement - CAPES and Brazil-Europe Erasmus Mundus project (process BM13DM0002) for partially funding this research.

[18]There are some efforts to solve the inconsistencies between changes made manually in generated code (Antkiewicz and Czarnecki, 2006; Hettel et al., 2008). Such research area is often called round-trip engineering.

REFERENCES

Alkhatib, H., Faraboschi, P., Frachtenberg, E., Kasahara, H., Lange, D., Laplante, P., Merchant, A., Milojicic, D., and Schwan, K. (2014). IEEE CS 2022 Report.

Antkiewicz, M. and Czarnecki, K. (2006). Framework-specific modeling languages with round-trip engineering. *Model Driven Engineering Languages and Systems*, pages 692–706.

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

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2009). Above the clouds: A Berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, 53(4):50–58.

Bozman, J. (2010). Cloud Computing: The Need for Portability and Interoperability. *IDC Analyze the Future, Sponsored by Red Hat, Inc*.

Brambilla, M., Cabot, J., and Wimmer, M. (2012). Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182.

Chen, Y., Li, X., and Chen, F. (2011). Overview and analysis of cloud computing research and application. In *E -Business and E -Government (ICEE), 2011 International Conference on*, pages 1–4.

da Silva, E. A. N., Fortes, R. P. M., and Lucredio, D. (2013). A Model-Driven Approach for Promoting Cloud PaaS Portability. In *Anual International Conference on Software Engineering-CASCON*.

Deursen, V., , Klint, A., and Paul and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36.

France, R. and Rumpe, B. (2007). Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA. IEEE Computer Society.

Giove, F., Longoni, D., Yancheshmeh, M. S., Ardagna, D., and Di Nitto, E. (2013). An approach for the development of portable applications on paas clouds. *Proceedings of CLOSER*, pages 591–601.

Hettel, T., Lawley, M., and Raymond, K. (2008). Model synchronisation: Definitions for round-trip engineering. *Theory and Practice of Model Transformations*, pages 31–45.

Juristo, N. and Moreno, A. M. (2010). *Basics of software engineering experimentation*. Springer Publishing Company, Incorporated.

Khajeh-Hosseini, A., Sommerville, I., Bogaerts, J., and Teregowda, P. (2011). Decision Support Tools for Cloud Migration in the Enterprise. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 541–548.

Kleppe, A., Jos, W., and Wim, B. (2003). MDA Explained, The Model-Driven Architecture: Practice and Promise. Addison-Wesley.

Miranda, J., Guillén, J., Murillo, J. M., and Canal, C. (2013). Development of Adaptive Multi-cloud Applications - A Model-Driven Approach. In *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, pages 321–330. SciTePress - Science and and Technology Publications.

Mohagheghi, P. and Dehlen, V. (2008). Where is the proof?-A review of experiences from applying MDE in industry. In *Model Driven Architecture–Foundations and Applications*, pages 432–443.

Mohagheghi, P. and Sæ andther, T. (2011). Software Engineering Challenges for Migration to the Service Cloud Paradigm: Ongoing Work in the REMICS Project. In *Services (SERVICES), 2011 IEEE World Congress on*, pages 507–514.

Petcu, D. (2011). Portability and interoperability between clouds: challenges and case study. In *Towards a Service-Based Internet*, pages 62–74. Springer.

Petcu, D., Macariu, G., Panica, S., and Crăciun, C. (2013). Portable cloud applications—from theory to practice. *Future Generation Computer Systems*, 29(6):1417 – 1430. Including Special sections: High Performance Computing in the Cloud & amp; Resource Discovery Mechanisms for {P2P} Systems.

Petcu, D. and Vasilakos, A. V. (2014). Portability in clouds: approaches and research opportunities. *Scalable Computing: Practice and Experience*, 15(3).

Ranabahu, A. and Sheth, A. (2010). Semantics Centric Solutions for Application and Data Portability in Cloud Computing. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 234–241.

Schmidt, D. C. (2006). Model-driven engineering. *Computer-IEEE computer society-*, 39(2):25.

Sharma R., S. M. S. D. (2011). Modeling cloud SaaS with SOA and MDA. *Communications in Computer and Information Science*, 190 CCIS(PART 1):511–518.

Shirazi, M. N., Kuan, H. C., and Dolatabadi, H. (2012). Design Patterns to Enable Data Portability between Clouds' Databases. In *Computational Science and Its Applications (ICCSA), 2012 12th International Conference on*, pages 117–120.

Tichy, W. F. (1998). Should computer scientists experiment more? *Computer*, 31(5):32–40.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA.