

Towards a solution avoiding Vendor Lock-in to enable Migration Between Cloud Platforms

Alexandre Beslic ‡, Reda Bendraou ‡, Julien Sopena ‡, Jean-Yves Rigolet †

Department of Computer Science and Engineering, Pierre and Marie Curie
University, 4 place Jussieu 75017 Paris, France ‡
{alexandre.beslic,reda.bendraou.julien.sopena}@lip6.fr

IBM, 9 rue de Verdun 94250 Gentilly, France †
rigolet.j@fr.ibm.com

Abstract. The *Cloud Computing* paradigm is used by many actors, whether companies or individuals in order to harness the power and agility of remote computing resources. Because they target developers and offer a smooth and easy way to deploy modern enterprise software without dealing with the underlying infrastructure, there is a steadily increasing interest for Platforms as a Service (PaaS). However, the lock-in makes the migration to another platform difficult. If the vendor decides to raise its prices or change its security policies, the customer may have to consider to move to the competition or suffer from these changes. Assistance and tooling to move to the competition at the PaaS layer still does not exist thus requiring tremendous re-engineering effort. In this regard, we propose an approach to the challenge of software migration between PaaS using Model-Driven Engineering coupled to Program Transformation

1 Introduction

Cloud Computing is now a popular paradigm, offering computing resources on a "pay-as-you-go" basis. It allows a remote and on-demand access to a wide range of services alleviating the need to own and maintain an internal infrastructure. The service model is standardized by the NIST [16] and is divided into three major layers. These layers vary in the amount of abstraction they provide to the consumer. The more you climb this service model, the more you will face restrictions.

Infrastructure as a Service (IaaS) provides the ability for consumers to provision fundamental computing resources such as processing power, storage capacity or networks. They have control over the operating system and software stack giving them the freedom to deploy any kind of software. *Platform as a Service* (PaaS) came as an abstraction to the infrastructure layer. Because maintaining and updating a whole infrastructure requires knowledge and time, platform provides with a fully prepared runtime environment to deploy applications. It targets developers to further fasten the development process and to focus on the

product features rather than configuring the underlying infrastructure. *Software as a Service* (SaaS) is the highest level of the Cloud service model. The software itself is provided as a service to the end-user.

While Infrastructure as a Service (IaaS) and Software as a Service (SaaS) are still prevalent in the Cloud computing service model, Cloud platforms (PaaS) are becoming increasingly used. According to the Gartner study on Cloud Computing, the use of Cloud Platforms will increase at 43% in 2015 compared to 3% in 2012. With a major struggle between cloud providers to dominate the PaaS market, the use case of software migration between providers is to be considered. But this task is far from being easy. Indeed, the platform layer suffers from a well known issue: the vendor *lock-in*. Early platforms are providing tools and libraries to use during the development process to access their own features thus locking the application to this platform. The advent of NoSQL solutions with data denormalization makes it even more difficult because of choices made on the program's design to ensure best performance. As a consequence, migrating onto another platform requires tremendous re-engineering effort that a few are able to provide.

The will to migrate is explained by several factors. The price is the first one considering that computers are now a commodity that we need at the lowest price, thus explaining the popularity of the Cloud Computing paradigm. Some other factors are the *Lock-in avoidance*, an *Increased Security*, a *Better availability* (99,95% versus 100%), a *Better Quality of Service* (QoS guarantee), a *Major shift in technology trends* or *Legal issues* (forced to move) among others. As of today, no such tool exists to achieve this migration. Existing work like mOSAIC [19] is taking the approach of the middleware abstracting cloud software stacks or APIs. mOSAIC offers a thin layer of abstraction over software stacks and data storage on PaaS in order to develop applications from scratch. Thus it only supports newer applications and the user is still entangled by the compatibility list of the middleware. Even if it tries to support a wide variety of Cloud providers hence being a first step for Cloud platform interoperability, the use of a middleware just moves around the lock-in and businesses are reluctant to this.

In this regards, we present our approach to deal with this major issue of software migrations between Cloud Platforms. The idea is to provide assistance and tooling to re-engineer a software using the Model-Driven Architecture and Refactoring approach. It is divided in several stages. The first one is the discovery of a Cloud software deployed on a platform using MoDisco [12]. Follows the Transformation on the program structure/layout using Model transformation on the discovered model. Then fine grained transformations are defined between an API and its counterpart on the targeted platform using a general purpose transformation language such as TXL. Assistance on software transformation is provided by applying these rules. The final step is the migration on the targeted platform with offline and online tests suites validation to be sure that the software runs as usual. In order to achieve this, as the number of scenarios is huge (especially for the data storage part) and the cloud environment always evol-

ing, we aim to provide the ability to add new knowledge for processing source to target transformations using a dedicated DSL.

The paper is organized as follows. In section 2, we give examples of challenges to be tackled when migrating a software tied to the provider's data storage solution or APIs. In section 3, we introduce our detailed approach to deal with software migration between Cloud platforms. Section 4 discusses related work and the limits of using middlewares to deal with this issue. Section 5 concludes the paper and discusses future work.

2 Vendor lock-in issue

The Platform as a Service (PaaS) appeared shortly after the Infrastructure (IaaS) and Software (SaaS) layers of Cloud Computing. Heroku, which has been in development since 2007 is a owned subsidiary of Salesforce.com and is one of the very first Platform as a Service provider. It provides a fully prepared stack to deploy automatically Ruby, Node.js, Clojure, Java, Python and Scala applications and runs on top of Amazon Web Services (AWS). Since then many new providers have entered the market as Google with its App Engine (GAE) platform or Microsoft with Windows Azure both in 2008. These are three of the very early platform providers but there are many others as of today creating a large ecosystem of PaaS solutions. The particularity of Cloud platforms is that every provider has its own set of features, frameworks or language supported.

With those early Cloud platforms, the customer is using a well defined set of tools and libraries originating from the provider. Achieving best performance is a result of using the providers data storage solution, not supported on other platforms. Google App Engine is using BigTable while Windows Azure is using Azure Table to store data. Both are categorized as NoSQL solutions meaning that they differ drastically from classical RDBMS as MySQL or PostgreSQL which are relational.

Both have a strikingly similar data structure, even if they have also subtleties in their design like the way they handle fragmentation on servers, thus impacting the design of the program. Both are schema-less and both are using two keys with a timestamp to quickly query data using a distributed binary index. Both are also categorized as Key-Value stores in their documentation (even if the right definition for BigTable as defined in the white paper is a sparse, distributed, persistent multi-dimensional sorted map [13]). Achieving best-performance on both systems requires effort and knowledge. Thus and because they are exotic in design compared to other platforms supported databases, you end-up being locked in. These types of NoSQL databases are not represented elsewhere as a true Storage as a Service offering. Your choices are to use similar solutions like DynamoDB from Amazon, HBase (the open source implementation of BigTable) on EMC (an IaaS provider), or try a migration from Azure Table to BigTable or vice versa. The fact is that DynamoDB and HBase are not included as built-in PaaS features but only at the Infrastructure layer. Moreover, they have different properties in the way they handle the data compared to BigTable or Azure

Table. The data denormalization broadens the possibilities but trying to move from a solution to another is a difficulty introduced recently with the advent of NoSQL databases. Every NoSQL is used for a purpose and differs slightly or completely from another solution. But as they offer the scaling properties that relational databases couldnt offer, they are mandatory on PaaS to leverage best performance and scalability.

Considering those design decisions, what if one of these two platforms on which you deployed your application decides to raise its prices or do whatever you disagree with as a customer? Either you accept these changes or you consider moving onto another Cloud. But this will require tremendous efforts to adapt your software that is locked-in by specific APIs and data storage proprietary implementations.

As the lock-in became a sore point for customers willing to move on a platform, companies are now taking the bet to offer *Open Platforms as a Service*. Initiatives like TOSCA [7] to enhance the portability of Cloud applications have been supported by several partners like IBM, Red Hat, Cisco, Citrix and EMC. Red Hat with *OpenShift*, VMWare with *CloudFoundry* and IMBs *SmartCloud Application Services* are three of the most known projects for portable PaaS solutions. As of writing, CloudFoundry is still in beta while OpenShift has been released. The idea behind Open PaaS is that being restricted to a framework or library to develop an application is not offering the flexibility desired by the developers. Instead they offer the widest range of languages, frameworks and data storage to vanish the lock-in still present on older Platforms. These platforms are extensible with new technologies or patterns (in the case of IMBs PaaS solution).

Still older Platforms have their benefit. Because the architecture is mature and that they improve by offering the latest technologies to attract new customers. There are a lot of examples of successful websites with a huge amount of requests per day while newer platforms lacks such examples that could appease new customers in their choice. Moreover, with the example of OpenShift, the support is large but still limited to older versions of languages such as Python used in its 2.6 version (while there is a 2.7 and 3 version of the language supported on many other platforms). It also misses built-in services present in other PaaS like Redis, Memcached or RabbitMQ (some of them are available on carts, which are pluggable components on OpenShift but are not straightforward to use). Also proprietary implementations of database systems are still more scalable and BigTable is known to scale to petabytes of data with highly varied data size and latency requirements which makes it particularly powerful for high traffic websites. Given the diversity of configurations for developed applications, Open platforms are not the general response to the vendor lock-in problem. Because the lock-in is traded off with the support of newer technologies and possible increased availability and security on other platforms. A dominant Platform crushing the competition is unlikely to happen because of the wide range of customers needs. As a consequence: *How to offer the possibility to migrate from a platform to another considering this ecosystem of PaaS providers?*

While on the infrastructure layer, the migration was restricted in the study of moving Virtual Machines from an Hypervisor to another [11], here the issue is much larger. Because the diversity of modern applications is huge and that finding a "one-fits-all" answer is rather impossible. At least a way could be found to assist the shift from a platform to another as for the migration of legacy software to the Cloud.

3 Our Approach

In this section, we describe the details behind our approach to migrate applications between Cloud platforms. Given the similarities that could be found on technologies of the same kind, we could define source to target transformations by leveraging knowledge amongst the Cloud developer community as well as Cloud vendors themselves. We strongly believe that assisting the migration of applications between Cloud platforms using the re-engineering approach is more flexible than the middleware approach because of the independence to any intermediate technology that could be harmful in the future, in terms of performance and support.

3.1 Overall Design

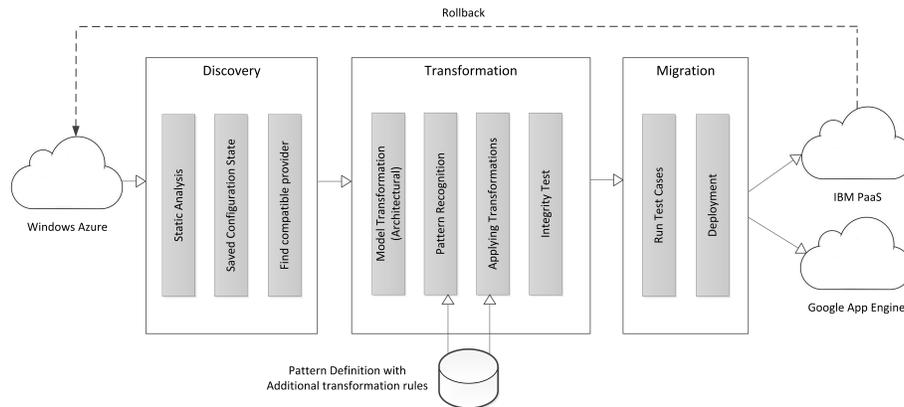


Fig. 1. PaaS migration workflow

Figure 2 shows our approach which contains three steps in the lifecycle of a software about to be migrated on a new platform: *Discovery*, *Transformation* and *Migration*. It also contains pluggable components: *Pattern Definition with Additional transformation rules*.

The *Discovery* step includes the discovery on sources taken back from the source platform. Because platforms generally have version control deployment options using git, getting the sources is straightforward at this step. By having a high level representation of sources, we have all the informations to transform and adapt the software accordingly at an architectural level.

Transformation step is going to adapt the software regarding the targeted platform. It includes a Model Transformation step to address architectural (coarse grained) violations that could prevent a successful migration. Because fine grained transformations on sources from a given pattern to its counterpart on the targeted platform are necessary as well, we include a Pattern recognition step. Patterns are defined portions of code with a given semantic. Every pattern comes with its counterpart for the targeted platform and mappings between method calls and attributes are made to bridge the two representations. These mappings are part of the *Additional transformation rules* that are provided by Cloud users with a dedicated *Domain Specific Language*.

Migration is the last phase of the process, it tests the transformed application prior to the deployment on the new platform and gets feedbacks at runtime. If nothing goes wrong in the deployment and at runtime, we take back resources on the ancient provider. But if something goes wrong after the resources are taken back, rollback is applied on the ancient platform with the help of saved sources and configurations. Specific insights on each phase are covered on subsections.

Discovery phase As we are treating of the use case of migration between platforms, we assume that an existing project is already deployed on a PaaS. Whatever the provider, as long as the sources are accessible by any mean, the process can go further. With these sources, we get back a higher level representation of the software. For this, we use MoDisco [12] which is a reverse engineering tool (the term of discoverer is more appropriate) that traduces sources to UML models or even an Abstract Syntax Tree (AST). MoDisco is extensible by new definitions of legacy software to discover. However in our case, the OMG's Knowledge Discovery Metamodel (KDM) [18] already provides the required metadata for the source discovery. KDM is a standard providing ontology (a set of definitions) for system knowledge extraction and analysis. This *KDM discoverer* is a built-in feature of MoDisco which takes away the complexity of the process. The target KDM model gives insights of the technology used by the application. It represents the project structure and overall design. At this state, the target model as well as configuration options are saved for future purposes (Rollback which is further explained in the migration section).

For a given configuration we could guide the user to a new platform that has the same kind of technologies to reduce the work on the transformation phase. This could be done by defining a configuration pattern for each provider and match our configuration among these definitions. This choice could also be guided by the price and other parameters taking inspirations from customer needs. Given the diversity of platform providers, a right behavior is to forbid the choice to a target provider with a completely different language with an opposed data representation. Such a case will make the transformation step weaker and likely to break the integrity of the sources. Otherwise, we could also find a perfect matching configuration for our software. Not every software are locked in by a specific API or data solution (especially on an Open Platform which promote

the use of open source software) thus enabling an easy migration to another provider.

```

1 // Create a new student entity.
2 StudentEntity student1 = new StudentEntity("Foo", "Bar");
3 student1.setEmail("foo.bar@upmc.com");
4 student1.setPhoneNumber("0102030405");
5
6 // Create Insert operation
7 TableOperation insertStudent1 = TableOperation.insert(student1);
8
9 // Submit the operation
10 tableClient.execute("people", insertStudent1);

```

Listing 1.1. New entity on Azure Table

```

1 // Create a new student entity
2 Entity student1 = new Entity("Student");
3 student1.setProperty("name", "Foo");
4 student1.setProperty("lastName", "Bar");
5 student1.setProperty("email", "foo.bar@upmc.com");
6 student1.setProperty("phoneNumber", "0102030405");
7
8 // Add the student entity
9 datastore.put(student1);

```

Listing 1.2. New entity on BigTable

Transformation phase Transformation process takes place after the Discovery step and benefits from the KDM model discovery. KDM provides a set of abstraction layer to understand existing softwares. There are three main layers:

- *Program Elements Layer*: Code primitives and actions
- *Runtime Resource Layer*: Data, Events, UI and Platform
- *Abstractions Layer*: Conceptual, Build and Structure

All these layers are helpful to define the context of the migration, giving us insights of the specific code portions, frameworks and APIs used, to higher level details about runtime environment. The KDM model is used for two goals: finding the best target platform and apply architectural changes to the project. By architectural changes, the idea is to address project varying layouts forced by some frameworks (mostly for web applications). Examples are project layouts of Struts, JSF or the webapp lightweight framework of Google App Engine. Changes to the project configuration and deployment scheme are guided to comply to the new provider as each offer different options of deploying on their platform.

However, these architectural changes are not sufficient to adapt the software for the new platform. As such, our approach focuses on the use of pattern-based techniques so that fine grained transformations are realized on the sources. Patterns are defined as a provider’s code portion with a meaningful semantic. As every pattern comes with an associated transformation definition bridging the

methods and attributes to their counterparts in the targeted platform, transformation are made in a fine grain fashion. Thus the architectural changes mixed to the program transformation depicts an hybrid way to adapt an application to be migrated on a new platform, covering most of the re-engineering effort.

Listing 1.1 and 1.2 are showing Java code portions to store an entity respectively on Azure Table and BigTable. The similarity between both source code is striking, and a move from the one to the other is straightforward with manual intervention. But doing this on large sources is tedious and error prone. Both code are storing the same entity using the same scheme. Defining a strategy of transformation between those two elementary operations is possible, and could help to this re-engineering effort to enable a migration. Although differences still remains between the two datastores by design (mostly on the way to handle fragmentation over multiple servers), this variance could also be dealt within the program transformation process. Developers are the first source of knowledge for the huge amount of technology used across all this platform ecosystem. Thereby, providing a DSL helping to define those sets of transformation could leverage the potential of this approach. The DSL would come as an abstraction of tools such as TXL and other general purpose transformation languages.

Fine grained transformations on specific method calls and class wise modifications could be made using TXL [14], a Language processing transformation on sources. It provides with rewrite rules, strategies to apply those rules with support for context-sensitive transformations.

Migration phase Migration is the last step of the workflow. It moves the transformed sources on the new provider. Prior to the deployment, some tests are going to be applied on sources to avoid the case of applications that are not providing a correct behavior. At this time, we are uncertain on the manner tests are going to be handled. They will need to fit the targeted Cloud provider, then adding complexity to the process for the developer that is going to write those tests. After the application being validated, deployment on the target platform could be launched. Runtime informations are caught after the deployment process to validate the success of the migration. Then, we take back resources on the ancient provider. Still, if something goes wrong we could *rollback* to the ancient state, that was saved during the discovery phase.

4 Related Work

Existing work addresses the problem by using middlewares. mOSAIC is one of those projects. However there are several reasons for us not to use the middleware approach. First the *Overall complexity*: In order to support the widest amount of technologies, middlewares are strongly tied to the software and often provides with configuration files and eventual cluttered logic. Also, it could cause a *Performance overhead* during program execution. Cloud software are especially developed to be accessed by a huge amount of users, generating an enormous quantity of data to write and read. This makes considering a middleware to

support the portability for a potential migration a huge tradeoff. Finally, the *Compatibility list*: The user is tied to the technology supported by the middleware. In the case of a major shift in technology trends, the customer will always face this transformation process if the middleware still don't offer support for the targeted technology. Existing work are already relying on models to adapt legacy software to be migrated on the Cloud through modernization (from the company's infrastructure to a cloud infrastructure or platform) such as REMICS [17], MODAClouds [9] and artist [1] amongst notable cloud migration projects. These solutions are focusing on the provisioning or on the migration of legacy software to the Cloud. None of these are actually dealing with the migration between PaaS. Another existing work is CloudMIG Xpress [15] which is also using KDM to reverse engineer cloud software to check violations for the deployment but does not try to correct those.

5 Conclusions and Future work

In this paper, we presented our approach to deal with software migration between Cloud platforms. We introduced the overall design depicted in three phases: Discovery, Transformation and Migration. We rely on program transformation to enable the migration between PaaS solutions. The discovery on sources is performed with MoDisco, which has a built-in KDM (Knowledge Discovery Meta-model) discovery feature. This discovery provides insights on the software configuration to help choosing the best target platform regarding numerous parameters (price, availability, etc.). Instead of providing yet another middleware much likely to cause a performance hit, we will build a system to provide the ability to define transformation rules on cloud software. Those rules will be defined by developers (and we may imagine, Cloud providers to attract new customers), with the help of a dedicated Domain Specific Language. Transformations are made at an architectural level using models and on source code using tools like TXL, which is a language to define transformations.

Future work includes the realization of a dedicated Eclipse plugin. This plugin will directly points to the changes to be made on a Java project to adapt for the migration on a targeted provider. This work is also going to include insights from the discovery phase to choose wisely platform that fits the best in terms of technology and customer needs. Finally, we will go onward with the program transformation support, validating the approach on real use cases. This part being the heart of the contribution.

6 Acknowledgement

The author's work is funded by the MERgE project (ITEA 2 Call 6 11011).

References

1. artist project. <http://www.artist-project.eu/>.
2. Google app engine. <https://appengine.google.com/>.
3. Heroku. <https://www.heroku.com/>.
4. Ibm smartcloud application services. <http://www.ibm.com/cloud-computing/us/en/paas.html>.
5. Microsoft windows azure. <http://www.windowsazure.com/>.
6. Red hat openshift. <https://www.openshift.com/>.
7. Tosca. http://cloud-standards.org/wiki/index.php?title=Main_Page.
8. Vmware cloud foundry. <http://www.cloudfoundry.com/>.
9. Danilo Ardagna, Elisabetta Di Nitto, P Mohagheghi, S Mosser, C Ballagny, F D'Andria, G Casale, P Matthews, C-S Nechifor, D Petcu, et al. 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, 2012.
10. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.
11. David Bernstein, Erik Ludvigson, Krishna Sankar, Steve Diamond, and Monique Morrow. Blueprint for the intercloud-protocols and formats for cloud computing interoperability. In *Internet and Web Applications and Services, 2009. ICIW'09. Fourth International Conference on*, pages 328–336. IEEE, 2009.
12. Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 173–174. ACM, 2010.
13. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.
14. James R Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.
15. Sören Frey and Wilhelm Hasselbring. Model-based migration of legacy software systems to scalable and resource-efficient cloud-based applications: The cloudmig approach. In *CLOUD COMPUTING 2010, The First International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 155–158, 2010.
16. Michael Hogan, Fang Liu, Annie Sokol, and Jin Tong. Nist cloud computing standards roadmap. *NIST Special Publication*, page 35, 2011.
17. Parastoo Mohagheghi and Thor Sæther. 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. IEEE, 2011.
18. Ricardo Pérez-Castillo, Ignacio Garcia-Rodriguez De Guzman, and Mario Piattini. Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33(6):519–532, 2011.
19. Dana Petcu. Portability and interoperability between clouds: challenges and case study. In *Towards a Service-Based Internet*, pages 62–74. Springer, 2011.
20. Dana Petcu, Georgiana Macariu, Silviu Panica, and Ciprian Crăciun. Portable cloud applications-from theory to practice. *Future Generation Computer Systems*, 2012.