The Journal of Systems and Software 86 (2013) 2294–2308

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A service-oriented framework for developing cross cloud migratable software

Joaquín Guillén[a,*], Javier Miranda[b], Juan Manuel Murillo[b], Carlos Canal[c]

[a] GloIn, Calle Azorín 2, Cáceres, Spain
[b] Department of Information Technology and Telematic Systems Engineering, University of Extremadura, Cáceres, Spain
[c] Department of Computer Science, University of Málaga, Spain

ARTICLE INFO

Article history:
Received 1 March 2012
Received in revised form
17 September 2012
Accepted 6 December 2012
Available online 3 January 2013

Keywords:
Cloud framework
Adaptation
Cross-cloud applications

ABSTRACT

Whilst cloud computing has burst into the current scene as a technology that allows companies to access high computing rates at limited costs, cloud vendors have rushed to provide tools that allow developers to build software for their cloud platforms. The software developed with these tools is often tightly coupled to their services and restrictions. Consequently vendor lock in becomes a common problem which multiple cloud users have to tackle in order to exploit the full potential of cloud computing. A scenario where component-based applications are developed for being deployed across several clouds, and each component can independently be deployed in one cloud or another, remains fictitious due to the complexity and the cost of their development. This paper presents a cloud development framework for developing cloud agnostic applications that may be deployed indifferently across multiple cloud platforms. Information about cloud deployment and cloud integration is separated from the source code and managed by the framework. Interoperability between interdependent components deployed in different clouds is achieved by automatically generating services and service clients. This allows software developers to segment their applications into different modules that can easily be deployed and redistributed across heterogeneous cloud platforms.

1. Introduction

Cloud computing has experienced an admirable growth throughout the past years due to the high acceptance rate it has had among companies with technological backgrounds (Leavitt, 2009). The combination of a powerful technology with a competitive business model, which in this case results from combining cloud computing with utility computing, has been presented as extremely beneficial for companies with limited amounts of resources. Such benefits have been widely documented by many researchers (Armbrust et al., 2009; Zhang et al., 2010), emphasizing the lack of an initial investment, the lower operation costs and the apparently infinite computing resources offered by the technology. These benefits, in conjunction with the great amount of cloud providers that exist, each with their own strengths and weaknesses, allow companies to choose which cloud platform they want to develop their software for (Li et al., 2010).

However, besides the outstanding benefits provided by the technology, applications are often subject to numerous limitations that keep them from exploiting the potential of cloud. Most cloud vendors establish a series of restrictions for their cloud platforms, and certify that they are satisfied by providing tools that must be used throughout an application's development process. The software developed with these tools will be coupled to the libraries and services provided by each cloud provider, hence preventing these applications from being deployed in other clouds. This problem, which is commonly known as vendor lock-in (Chow et al., 2009), presents one of the main drawbacks of cloud computing: applications developed for specific cloud platforms cannot easily be migrated to other cloud platforms and users become vulnerable to any changes made by their providers (Armbrust et al., 2010). Additionally the lack of standardization among cloud platforms prevents users from creating software that can be deployed across multiple clouds (Petcu et al., 2011), where different components of a single system could reside in different platforms.

Thus, the current scenario does not allow cloud users to take full advantage of the utility computing model since it only considers choosing a cloud platform at early stages of projects, where development has not begun. Ideally, users should be able to migrate their applications towards different clouds if they are not satisfied with the services and costs of their providers. Moreover, considering that different modules or components of an application may be subject to different requirements or technological restraints, users may find it difficult to choose a cloud platform that complies with all the application's requirements and may thereby have to consider deploying their applications across multiple clouds (Tsai et al., 2010). Cross cloud development is an arduous task which

* Corresponding author. Tel.: +34 685556021.
E-mail addresses: jguillen@gloin.es (J. Guillén), jmircar@unex.es (J. Miranda), juanmamu@unex.es (J.M. Murillo), canal@lcc.uma.es (C. Canal).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.12.033

involves using the tools provided by each different cloud provider, as well as developing the services and clients for integrating the components deployed in separate clouds. Consequently, migrating a component from one cloud to another would require redeveloping its dependent components, either if they are hosted or not in other clouds.

Considering that standardization is currently not an option for most significant cloud computing providers (Dillon et al., 2010), the aforementioned limitations suggest that new tools are required for developing applications that can be deployed across multiple clouds as well as migrated from one cloud to another in order to take advantage of the business model associated to cloud computing. These tools should contribute to lower the barriers that users continue to encounter in their leap towards cloud computing, and thereby impulse cloud as the IT infrastructure of the future. Software should be developed in the same manner independently of which platform it will be hosted on, thereby not having to consider matters related to integration between the software and the cloud architecture as well as the integration between software hosted by different clouds.

This paper presents a framework for developing applications that are decoupled from the architecture, services, and libraries provided by cloud vendors, such that they can be deployed across one or several clouds. The framework allows developers to choose which components are deployed in each of the supported cloud platforms. Deployment metadata is separated from the source code and managed by the framework. The metadata represents a deployment plan that describes how dependent modules hosted in different clouds communicate with each other and the services provided by their cloud, this way relieving developers from implementing cloud integration modules and interoperability services. The application source code, in conjunction with the deployment plan, is interpreted by the framework, which generates cloud compliant software artefacts that are deployed in each cloud platform.

The development of cross-cloud migratable applications is currently a topic of great interest which is being tackled from many perspectives by different authors. The main contributions of the framework, which differentiate it from alternate solutions, are summarized in the following points:

- *Feature models*, commonly used in Product Line Engineering, are used for modelling the variability of the cloud platforms supported by the framework.
- *Software adaptation* techniques are applied; adapters are automatically generated to allow the software artefacts to be integrated with their corresponding clouds.
- Applications can be developed as if they were going to be deployed in an in-house environment, being completely agnostic to any cloud related information.

Such contributions have be integrated in the framework in order to improves the current state of the art in cloud application development. Their adoption allows applications to be designed without having to consider the intricacies of each cloud platform and without being coupled to any cloud or intermediate platform.

The framework described in this work has been conceived for developing software for cloud platforms that allow hosting and deploying applications. The prevailing categorization of cloud computing models considers the following variants (TechTarget, 2010):

- *Infrastructure as a Service* (IaaS): physical or virtual machines are provided as a service, upon which users can install an operating system and any required software.
- Platform as a Service (PaaS): in this model a computing platform which will usually include an operating system, a programming language an execution environment, a database and a web server, is provided as a service.
- Software as a Service (SaaS): in this model services an applications are provided as a service. The underlying infrastructure is not accessible to the users.

Hence, the framework is intended to be applied upon applications targeted towards IaaS and PaaS clouds, as only these two models can host externally developed software.

Both the framework and its associated tools have been developed in conjunction with Gloin S.L., a Spanish start-up company that works on providing technology for enhancing the software development processes of its customers. The framework has been applied to a real industrial case in one of Gloin's projects; the results of this experience are reviewed in the final sections of this paper.

The remainder of this paper is structured as follows. Section 2 contains a description of the background and the motivations that have led us to our work. Section 3 describes our proposal; we discuss the principles on which our work is based and the architecture of the framework. Section 4 contains the technical design of the framework; in this section technical details are provided about the technology in which the framework has been constructed. Section 5 explains the development processes that must be followed by developers in order to use the framework. Section 6 contains a case study where the framework is used to implement and deploy an application across more than one cloud. Section 7 analyses the results obtained from the case study; it presents its strengths and weaknesses in order to plan future milestones. Section 8 summarizes the work carried out by other authors that is related to ours. Finally, Section 9 contains the conclusions extracted from our work.

2. Background and motivation

Cloud computing is a technology that provides numerous benefits to both its providers and its users, however the interests of each of these parties are of a very different nature. Whilst providers have seen in this technology a means of expanding their business models whilst recouping the cost of their underutilized server farms, users have seen the opportunity to access powerful computing resources flexibly and at an accessible price.

Becoming a cloud provider requires companies to have extremely large-scale data centres and highly qualified staff for their maintenance. The initial investment required for setting up such infrastructure is inaccessible for most companies; however companies such as Microsoft, Google and Amazon, among many others, have already invested in large server farms over the years. For these companies cloud computing has emerged as a great opportunity for leveraging the investment that they have made (Armbrust et al., 2009). Furthermore, cloud computing has been commercialized with a competitive business model which allows providers to target both small and large companies, thereby making the technology accessible to a wider range of users. The underlying interest of companies to return the investment they have made by gaining as many customers as possible and guaranteeing that these customers consume as many of their services as possible, has frustrated most efforts for standardization and cloud platforms integration (Dillon et al., 2010).

On the other hand, cloud users in the form of companies with technological interests, have seen that cloud can provide them with access to an unlimited number of computing resources that are accessible from numerous devices, such as smartphones, tablets and PCs (Anderson and Rainie, 2010). This allows cloud users to manage their services and applications in a much more flexible manner, being able to scale up and down the number of computing resources that they manage, as well as the cost of such resources.

These different interests in cloud computing clash with one another due to what is known as the vendor lock-in effect. The flexibility that cloud users demand stands in counter position to the growing need that vendors have for maintaining their customers (Armbrust et al., 2010). Such need has led them into imposing limitations upon the applications hosted by their platforms, in such way that software is commonly tightly coupled to its underlying cloud platform. Cloud applications are developed using different cloud-specific APIs, libraries, and even different project structures that vary depending on which cloud the software will be hosted. Furthermore, the services provided by each cloud vendor are explicitly invoked in the source code using the libraries provided for this effect, and service provision and remote service consumption is hardcoded into the source code based on the protocols and technological restraints imposed by the cloud vendor.

The vendor lock-in problem sows distrust among potential future cloud users since migrating applications to different clouds will often require redeveloping that application. Additionally, considering that a single cloud platform often does not fully comply with the technological and business requirements of a software project, architects may choose to deploy an application across more than one cloud, with different components deployed in different clouds (Tsai et al., 2010). Under the current context this implies developing several applications, each of them tightly coupled to their corresponding cloud platform, where the integration between each component and its dependent components has to be hardcoded as services and service clients. The elevated cost that has to be assumed on behalf of companies in these cases is a risk that hinders many companies from relying on cloud technology.

In contrast, we believe in a future scenario of cloud computing where both cloud providers and cloud users are free to evolve their technology without any restrictions. On one hand, cloud users should be free to develop and deploy cross-cloud migratable applications not tightly coupled to any cloud platform, and on the other hand cloud vendors should be free to provide their technology as they wish without being pressured into the standardization of their APIs and services. However, this can only be achieved if software development tools, such as the one presented in our work, are available and adopted by developers.

3. A framework for cloud application migratability and interoperability

In this section a service-oriented framework for developing cloud applications is presented. The framework allows applications to be constructed as a composition of software components, further referenced as cloud artefacts, where each cloud artefact can be freely migrated between cloud platforms without having to redevelop the application. The main principles that have been considered in the framework's design are presented. This is followed by a description of the high level design of the framework in order to provide a conceptual view of how it has been constructed.

3.1. Principles of the framework

The main goal behind the design of the framework is to allow developers to create their software as if it were finally going to be deployed in an in-house environment, without having to consider any cloud related information or having to use different development tools. This implies that the application's source code cannot contain cloud-specific information; i.e. all information about the cloud libraries and services that are used by the application once it is deployed in a cloud environment must be separated from the source code.

Different software engineering principles have been integrated in the framework in order to support these features. Each of these principles and their role in the framework are described in the following paragraphs.

3.1.1. Obliviousness

The obliviousness principle was initially introduced in the scope of Aspect Oriented Programming (Filman and Friedman, 2000) as a means of allowing developers to build applications without having to deal with a series of concerns that *Aspects* would take care of. In our case the obliviousness principle allows developers to code the applications without having to take into account which cloud platforms the application are deployed in, and which restrictions and rules must be applied to the software for each platform. The framework interprets the cloud deployment plan that is generated separately from the application's source code and transforms the original source code into artefacts that can be deployed in the targeted clouds.

3.1.2. Source code transformation

The source code generated by developers is not immediately deployed in each cloud platform, instead source code transformations must be applied in order to generate the software artefacts that are compliant with each cloud.

3.1.3. Service adapters

Software adaptation (Becker et al., 2006; Bracciali et al., 2005; Canal et al., 2006) is a branch of Software Engineering whose objective is to define techniques for arranging existing third-party software elements in order to combine and reuse them in new systems, accommodating any potential mismatch arising from their interaction (Canal et al., 2008). Adapters have an inherently black box nature and are therefore non-intrusive mediator elements specifically built for guaranteeing that a set of mismatching software elements will interoperate correctly.

Service adapters are automatically generated by the framework in order to favour the development of applications that are decoupled from cloud platforms. Developers simply have to identify the adaptation needs of their applications and match these needs with the services provided by a cloud platform. Adapters are automatically generated by the framework and injected into the cloud artefacts.

The work presented in Miranda et al. (2012a,b) presents the adaptation needs that software components have in the scope of this framework. Different adaptation scenarios are presented and mapped with a specific adaptation interoperability level (Canal et al., 2005).

3.1.4. Inversion of control and dependency injection

Inversion of control and dependency injection (Fowler, 2004) are used in the transformations carried out by the framework. The service adapters generated by the framework are initialized and injected into the generated cloud artefacts. This allows developers to decouple the use of cloud-specific services from the application, allowing the generated artefacts to easily be migrated to another cloud by simply changing their configuration and injecting a different cloud service adapter.

3.1.5. Stateless and stateful services

Considering that an application may finally be deployed across more than one cloud, services and service clients are generated in order to convert class dependencies into service dependencies. As a result of this, stateless or stateful services may be generated by the framework. For example, classes containing static methods, where objects do not maintain a state, are converted to stateless services

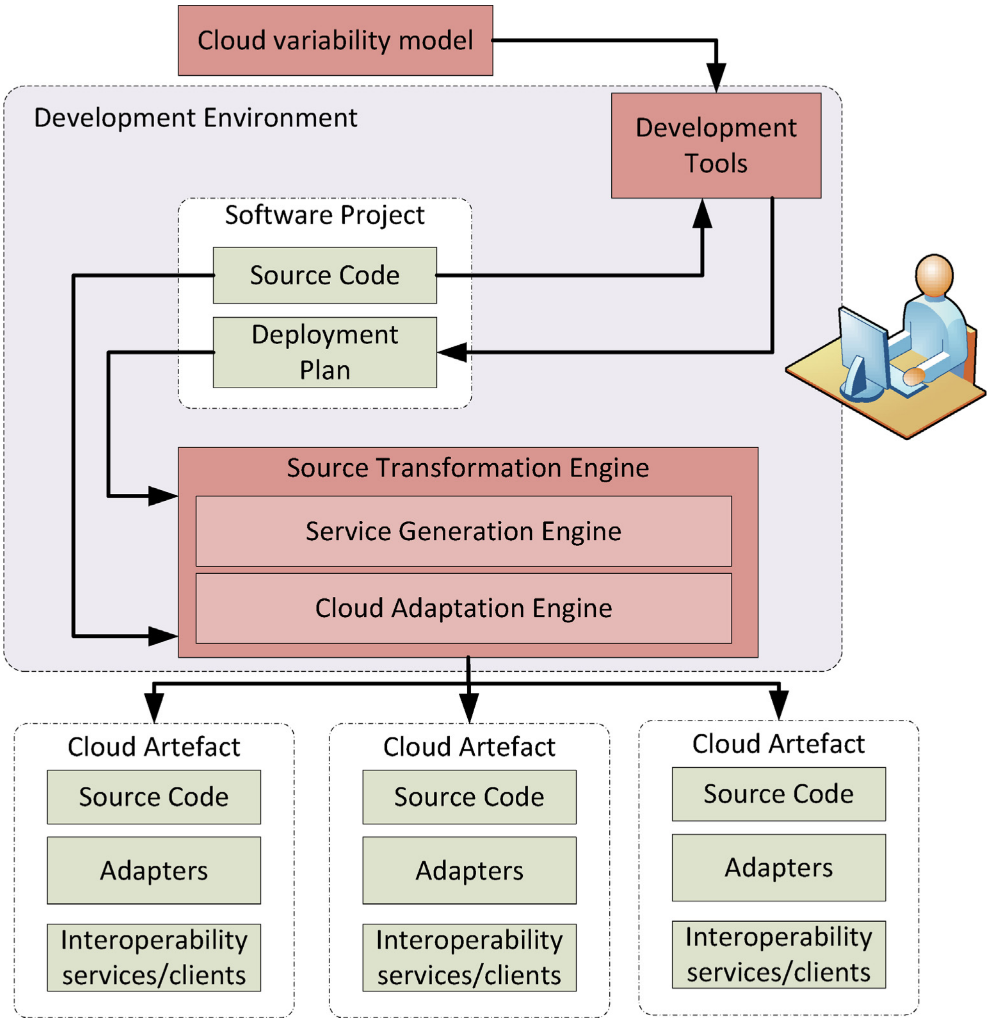


Fig. 1. Components involved in the use of the framework.

and classes that maintain an internal state are converted to stateful services (Foster et al., 2004) provided by cloud artefacts.

The combination of these principles allows the framework to achieve its goal of allowing developers to design cloud agnostic applications that can be segmented and deployed indifferently across multiple cloud platforms.

3.2. High level design

Fig. 1 illustrates all of the components that are involved in the use of the framework and the relationships between each other. The components that make up the framework as such have been coloured in red. The figure shows how the framework has been designed as a set of components that are integrated into the common development environment in order to avoid developers from having to use additional environments. The final output of the framework consists on a set of *cloud artefacts*. These are described along with the main components included in the design in the following sections.

3.2.1. Cloud artefacts

Cloud artefacts are the final output of the framework; their high level design is described before the rest of components in order to clarify how each of these components contribute to generate the cloud artefacts.

For an application to be deployed across more than one cloud, with different components of the application deployed in different clouds and interoperating with one another, it must be converted into sets of *cloud artefacts*. Cloud artefacts are defined as atomic groups of software components that can be deployed in a specific cloud platform and interoperate both with their underlying cloud as well as with dependent cloud artefacts deployed in other cloud platforms. This scenario is illustrated in Fig. 2.

Communication between the software components found in a cloud artefact is carried out locally without the need for mediation,

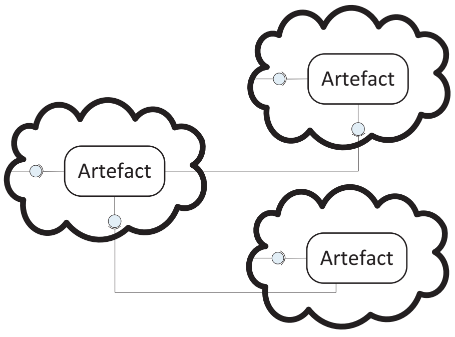


Fig. 2. Cloud artefact deployment.

whereas communication between software components found in different cloud artefacts is carried out remotely, thereby requiring mediation.

Therefore, as illustrated in Fig. 1 each cloud artefact is composed of the following elements:

- *Predefined structure*: some cloud providers require applications to have a predefined structure before they can be deployed in their cloud. Each artefact's structure must comply with the requirements of its targeted cloud provider.
- *Source code*: the source code of the software components encapsulated by the artefact.
- *Adapters*: cloud service adapters are included, if necessary, in order to allow each artefact to consume the services provided by its targeted cloud platform. Examples of this may be the authentication, persistence or file storage services provided by some platforms. Adapters are automatically generated by the *Cloud Adaptation Engine*.
- *Service/client interoperability*: communication between dependent components found in different cloud artefacts is transformed from the local model used in the source application, to a remote model once the artefacts are deployed in a cloud environment. Therefore each cloud artefact contains a set of services and service clients for providing services and accessing those found in other artefacts, respectively. Each service or service client complies with the technological requirements of its corresponding cloud platform. Services and service clients are automatically generated by the *Service Generation Engine*.

3.2.2. Software project

A software project contains the source application, libraries and configuration files. As we commented in Section 3.1, the source application is agnostic to its final deployment platform, which may or may not be a cloud platform. Therefore, if the software is going to be deployed in one or more clouds, all cloud related data must be included in a separate deployment plan. A deployment plan contains the following information:

- *Cloud artefacts*: the cloud artefacts that the application is decomposed into are identified in the deployment plan. Each artefact, composed of a set of software components, is configured to be deployed in a specific cloud platform.
- *Artefact configuration parameters*: depending on which cloud an artefact is targeting, different configuration parameters have to be set in the deployment plan. The deployment plan contains all of the configuration parameters that must be configured in each artefact before it is deployed in its corresponding cloud.
- *Interoperability services/clients*: the deployment plan includes information about the services and service clients that have to be generated in order to allow dependent cloud artefacts to communicate with one another. Each cloud artefact found in the deployment plan contains information about how its services and service clients have to be generated; among others, information about the technology that must be used (SOAP, REST, etc.), the types of services that are required (stateful, stateless, etc.) and the communication model (synchronous or asynchronous) is included.
- *Cloud integration adapters*: each cloud artefact may consume services provided by its targeted cloud platform. In such case the deployment plan contains information about the adaptation needs of each artefact. It identifies the adaptation points of the artefact with the services provided by each cloud. This information is used by the Cloud Adaptation Engine in order to automatically generate the required adapters.

3.2.3. Cloud variability model

The keystone component of the framework is the *Cloud Variability Model*. It contains information about all of the cloud platforms that are supported. It models all of the features of the cloud platforms required by the framework to generate cloud-specific adapters and interoperability services for the cloud artefacts. More specifically it contains the following information for each platform:

- *Service catalogue*: the cloud-specific services provided by each cloud platform are documented in the variability model. This information is used by the *Cloud Adaptation Engine* to generate the adapters required by each artefact.
- *Technological restrictions*: the technological restrictions imposed by each cloud provider determine how the cloud artefacts are generated. An example of the information modelled in this point is the service provision technologies supported by each platform (SOAP, REST, publish/subscribe technologies, etc.).
- *Templates*: some cloud providers establish a predefined project structure for the applications hosted in their platforms. In these cases the variability model contains a project template that is used to generate cloud compliant cloud artefacts. Each template contains information on how to configure its content correctly based on the cloud deployment plan.
- *Configuration parameters*: depending on each cloud platform different configuration parameters are required and have to be set in each cloud artefact in order for it to be deployed. The variability model documents all of the configuration parameters required by the supported cloud platforms.

3.2.4. Development tools

Depending on the complexity of the applications that are migrated to the cloud as well as on the cloud platforms which they are targeted towards, creating a full deployment plan can turn out to be a tedious task. Therefore tools and configuration assistants have been integrated into the development environment in order to allow developers to visually build their desired deployment plans.

Based on the application's source code and the cloud variability model, the tools provide developers with an intuitive interface that allows them to group their software components into cloud artefacts and to assign each artefact to a cloud platform. Additionally the tools supplied by the framework also allow developers to map any of the developed service consumers with a specific cloud service. The tools interpret the configuration requested by the developer and automatically generate a valid deployment plan for the application.

The development processes supported by the tools that have been integrated in the development environment are further detailed in Section 5.

3.2.5. Source transformation engine

Once the application has been fully designed and a valid deployment plan has been generated, the source transformation engine generates the cloud artefacts. As illustrated in Fig. 1 the transformation engine is composed of a *Service Generation Engine* and a *Cloud Adaptation Engine*. The former generates the services required for interoperability between cloud artefacts; i.e. it detects the class dependencies that have been broken in the process of segmenting the source application, and converts these dependencies into services and service clients generated for the correspondent cloud artefacts. The latter interprets the adaptation needs of each artefact contained in the deployment plan and automatically generates the adapters required by each artefact for its integration with its targeted cloud's services.

4. Technical design

Once the framework's high level design has been presented, in this section an insight of its main components is provided detailing the technologies in which they have been implemented and the main design decisions that have been taken.

The first high-impact decision that was taken whilst designing the framework was to construct it for applications developed in J2SE and J2EE. Java was chosen since it is currently one of the most extended programming languages (TIOBE, 2012) and also because many relevant cloud providers such as Amazon, Microsoft, Salesforce and VmWare support Java applications for their platforms. Based on this decision different tools and frameworks commonly used in Java projects have been integrated into the framework; nevertheless similar tools exist for other platforms in case the framework should be extended. Furthermore, the ideas and principles on which the framework has been based are easily applicable to different programming languages, frameworks and tools.

Another influential decision taken during the design of the framework was to integrate the framework's development tools into the Eclipse IDE. Other alternatives such as NetBeans were considered for this; however Eclipse provides many facilities for extending the common IDE via the use of plugins. The latest versions of Eclipse have been developed under the OSGi framework and this has greatly reduced the complexity of developing plugins for the IDE.

Once these decisions, that have an impact on the design of all of the remaining components, have been clarified, an insight of how the main components of the framework have been constructed is provided in the following sections.

4.1.1. Software project structure

Maven, a tool that is used for building and managing Java-based projects, was chosen as the preferred tool for managing the framework's software projects. Maven is based on the concept of a project object model (POM), a central piece of information which allows it to manage a project's build, reporting and documentation. It is easily extensible, with the ability to write plugins in Java or scripting languages. Additionally it provides a series of Project templates, known as *archetypes.*

Java projects that wish to use the framework's capabilities must be created as Maven projects based on the *webapp* archetype. These projects enclose an XML document that contains the application's deployment plan. When the project is compiled, the deployment plan is interpreted by the *Source Transformation Engine*, and the cloud artefacts are generated by the framework.

Maven and other equivalent project management tools are commonly used since they provide developers with multiple benefits. In this case the decision of using a project management tool like Maven is mainly motivated by the need to integrate a set of source transformation tools that are required by the framework. Maven allows us to include such transformation tools as project dependencies and to invoke them homogeneously. More specifically, the following technologies and components are included in the POM created for all software development projects that use the framework:

- *Source transformation engine*: the source transformation engine consists on a Maven plugin that is included in each project's POM. The plugin is configured in such way that it generates the application's cloud artefacts during Maven's *generate-sources* phase.
- *Maven WAR plugin*: the WAR plugin is used to configure how the source compilation phase is carried out. More specifically this plugin is used to establish the output directories for the cloud artefacts generated by the framework.

4.1.2. Cloud variability model

The framework models the variability between different cloud platforms using a feature model (Lee et al., 2002), a representation that is commonly used in Software Product Lines. The model is processed by the *Source Transformation Engine* in order to generate cloud artefacts that comply with the restrictions imposed by their corresponding cloud. Feature models may be used as centralized models accessible to all developers, or on the contrary, local models located in each developer's machine. Fig. 3 illustrates an excerpt of the feature model for Microsoft's Azure platform

As we can appreciate, the model has been constructed hierarchically. The top level is used to differentiate each of the cloud platforms supported by the framework, and the underlying levels describe specific features of each cloud platform. As mentioned in Section 3.2, the feature model contains information about the technologies supported by each platform, the cloud-specific services that it provides, the configuration parameters that it requires, etc.

Some features may however exist for some platforms and not for others. For example, Azure requires developers to include the application server that will be used as part of the project that is deployed in the cloud platform; therefore Azure's features include an *artefacts* feature where references to supported application servers are provided so that they can be included into the generated artefacts. This feature is not included in other platforms such as Cloud Foundry, which always uses an integrated VMWare vFabric tc Server for application deployment.

The feature model and the information that it contains is an extremely important part of the framework. The higher the number of documented cloud platforms, the higher the potential of the framework, as it will allow developers to create applications that can be deployed across more platforms. Therefore, keeping this model up to date with the changes made by each cloud vendor and increasing the number of documented cloud platforms enriches the quality of our proposal.

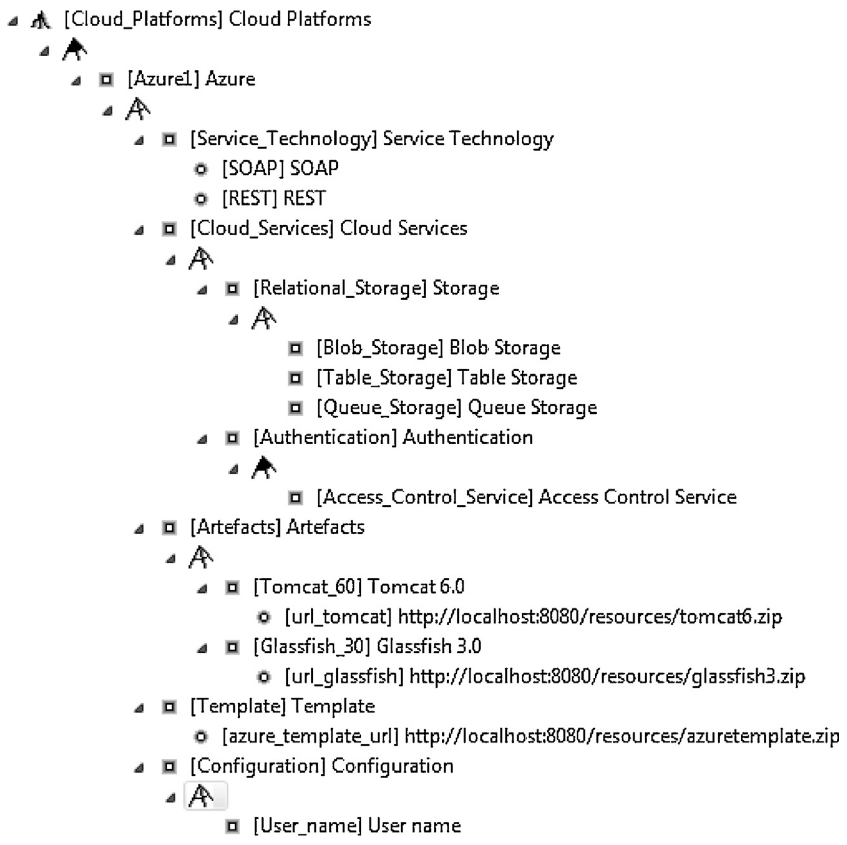
4.1.3. Cloud deployment plan

An XML representation was chosen for the deployment plan in order to make it easier to validate its structure as well as to allow experienced users to manually edit any deployment configuration. Additionally, for validation purposes, an XSD schema diagram was created to define the structure of the deployment plans. A graphical representation of the XSD diagram is illustrated in Fig. 4.

In this figure we appreciate that the file's root element is composed of a configuration section and a cloud artefacts declaration section:

- The configuration tag references the feature model used for generating each of the cloud artefacts declared in the *cloud-artefacts* tag. The cloud-artefacts tag is composed of a list of *artefact* elements, where each artefact is a segment of the application that will have to be deployed in a cloud platform. Each artefact contains relevant information about how it will be integrated with other artefacts and with its underlying cloud architecture.

In order for the cloud artefact projects to be generated correctly by the *Source Transformation Engine*, each *artefact* tag must reference an existent cloud platform in the features model using its *platform* attribute. Additionally the set of configuration parameters required by each cloud platform have to be included in each artefact. The parameters required by each platform are specified in the feature model, and therefore vary depending on the targeted platforms. This is why a generic property description structure has been chosen where the name of each property and its value is specified.



```
[Cloud_Platforms] Cloud Platforms
  [Azure1] Azure
    [Service_Technology] Service Technology
      [SOAP] SOAP
      [REST] REST
    [Cloud_Services] Cloud Services
      [Relational_Storage] Storage
        [Blob_Storage] Blob Storage
        [Table_Storage] Table Storage
        [Queue_Storage] Queue Storage
      [Authentication] Authentication
        [Access_Control_Service] Access Control Service
    [Artefacts] Artefacts
      [Tomcat_60] Tomcat 6.0
        [url_tomcat] http://localhost:8080/resources/tomcat6.zip
      [Glassfish_30] Glassfish 3.0
        [url_glassfish] http://localhost:8080/resources/glassfish3.zip
    [Template] Template
      [azure_template_url] http://localhost:8080/resources/azuretemplate.zip
    [Configuration] Configuration
      [User_name] User name
```

Fig. 3. Excerpt of the feature model used for describing each cloud platform, showing Azure's feature model.

Other than this, each artefact also contains a list of packages and classes. These lists determine which parts of the application are included in the artefact once it is generated; complete packages and separate classes may be selected in order to create partitions with as much flexibility as possible. These partitions may dissect dependencies with other classes and packages assigned to different artefacts. In order to allow these class dependencies to be maintained in the generated artefacts, services and service clients are declared using the *service* and *client* tags. Each service or service client is associated to a class declared in the *classes* element via the *class-ref* attribute. Furthermore, each has to be generated using a platform compliant technology specified by the *type* element. The information found in the *services* and *clients* tags is interpreted by the Service Generation Engine in order to create the services and clients that allow cloud artefacts to interoperate.

Finally, each artefact included in the deployment plan may have a list of service adapters associated in order to integrate it with the architecture and services provided by its targeted cloud platform. Developers must identify the adaptation needs of the application and match its interfaces with the services provided by the cloud platform. The application interfaces that require adaptation are identified by the *class-ref* element found in the *adapter* tag and matched with a service included in the platforms service catalogue, as it is modelled in the feature model.

4.1.4. Source transformation engine

This component has been designed as a Maven plugin that is included into the POM file of projects that use the framework. Having implemented the transformation engine as a plugin allows it to be invoked from a wider range of IDEs that support Maven, as well as from the operating system's command line.

The *Source Transformation Engine* takes the application's source code and its deployment plan as an input, and generates the cloud deployable artefacts as an output. The pseudocode of the transformation algorithm is presented in Pseudocode 1.

Pseudocode 1
Cloud agnostic application to cloud artefact transformation.

Algorithm parameters	Source code **S**, Deployment Plan **D**

Pseudocode

```
Validate deployment plan D
If D is valid then
    For each Cloud Artefact C in D
        Obtain the artefact template T for C
        Configure T based on C
        Generate cloud adapters CA for C
        Generate interoperability services IS for C
        Extract artefact specific source code AS from S
        Copy CA, IS and AS into T
    End for
Else
    Return Error
End If
```

Here we can appreciate how the transformation engine creates the cloud artefacts based on their templates, and later configures them and inserts the corresponding source code, services, service clients and cloud adapters.

4.1.5. Cloud artefacts

The artefacts generated by the Source Transformation Engine differ depending on the platform for which they are generated.

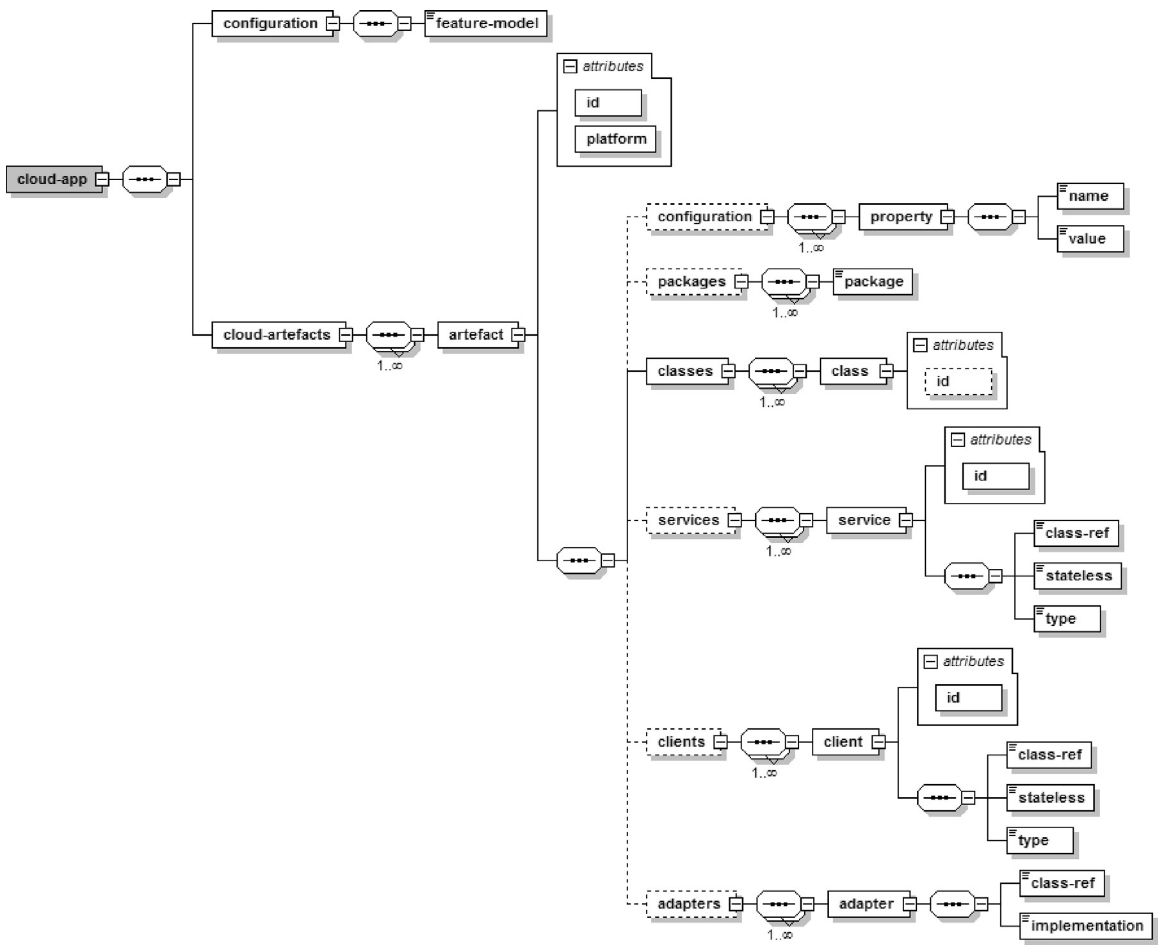


Fig. 4. XSD schema diagram used for cloud deployment plans.

However the following architectural components are common to all projects:

- *Spring*: the services, service clients and adapters generated by the *Source Transformation Engine* are configured in each artefact using Spring. The dependency injection mechanisms provided by Spring allow these components to easily be substituted without having to regenerate the complete artefact. I.e. if a cloud artefact is migrated to a new cloud using the new services, clients and adapters will only require updating the Spring configuration.
- *JAX-WS*: services are generated in order to integrate dependent artefacts deployed in different clouds. JAX-WS allows artefacts to provide and consume SOAP and REST services using the standard API provided with Java.

Both technologies are widely used for developing Java enterprise applications and are supported by a wide range of cloud platforms such as Microsoft Azure, Amazon Beanstalk and Cloud Foundry. Each of these platforms support applications developed using J2EE, where both are common technologies.

5. Framework-specific development processes

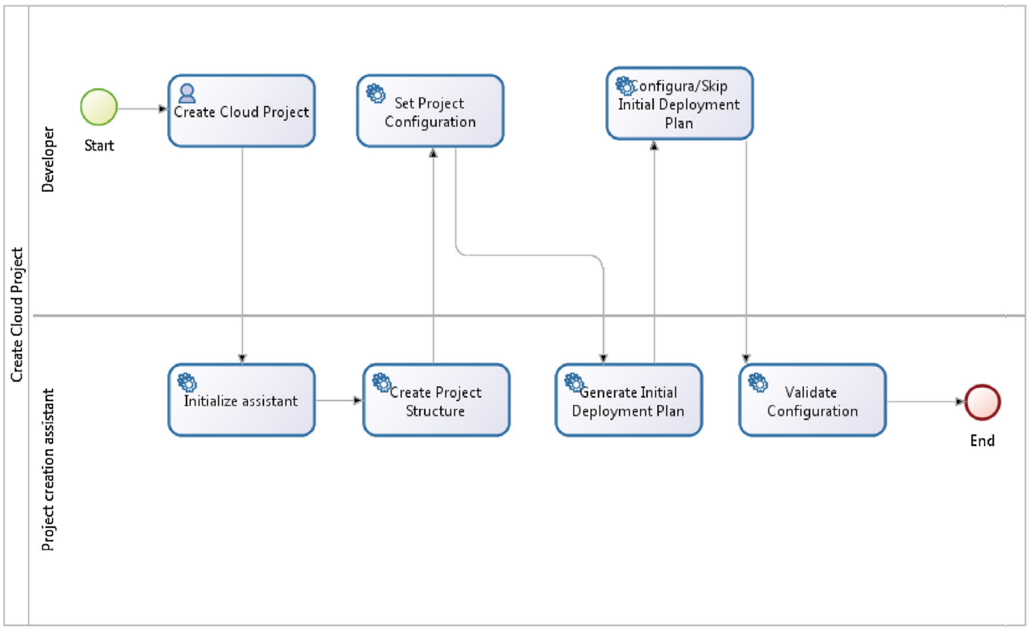
In Fig. 1 we illustrated how tools have been integrated into the development environment for allowing developers to generate a valid deployment plan for their applications, as well for granting them an easy access to the source transformation engine. Such tools have been designed to lower the burden of integrating the framework into the IDE used during the development stage, and for helping to carry out the development process associated with the framework. The following steps provide an insight of the development processes.

5.1.1. *Creating a project and beginning the development process*

In order to start coding the application, a framework compliant project as described in Section 3.2.2 must be created. Creating this project structure is not very complex; however it is a mechanical task that is always done in exactly the same way. Therefore an assistant for creating these projects has been integrated into the IDE. Process 1 illustrates the tasks that must be fulfilled for creating a project.

The developer requests to create a new cloud project; this request initializes the cloud project creation assistant and generates the project structure. Additionally a new deployment plan is created, and the developer is given the choice of configuring a deployment plan. At this stage the developer may already be aware of which cloud platforms the application will be deployed across, and he may be interested in generating the empty cloud artefacts in order to assure that they can be correctly deployed in their targeted cloud platforms. In any case, no classes, packages, services, clients nor adapters are included in the artefacts considering that no source code exists yet in the project. Once these steps have been completed the assistant validates the configuration and assures that it can correctly access the feature model referenced by the developer in the project configuration.

The framework also makes it easy to migrate legacy software to cloud platforms, as well as to deploy this software across multiple



Process 1. Cloud project creation.

platforms. Once a project has been created following Process 1, legacy code can be imported into the project. After this the most complex task consists on identifying the adaptation needs of the software and mapping such needs to specific services provided by the targeted cloud platforms. The better the legacy code has been structured, the easier it is to identify the software's adaptation needs. For example, if a cloud persistence adapter is required, it will be easier to identify where the adapter must be used if the legacy software uses the DAO (data access object) design pattern, than if all database accesses are distributed among the source code.

5.1.2. Cloud deployment configuration

At any stage during the development process developers may choose to specify how the application must be segmented and which cloud platform each segment will be deployed in. This can be done by manually editing the *deployment plan*; however due to the complexity of some cloud platforms this may be a tedious error prone task. For this reason a configuration assistant has been created for Eclipse, where developers can configure how the software is segmented, which adaptation needs each artefact has, and which cloud platform is targeted by each artefact.

The following diagram illustrates the process that must be followed for configuring the application's segmentation and cloud deployment information:

With the use of the configuration assistant created for this effect, developers are presented with a list of the classes that exist in the project, grouped by the packages which they belong to. On the other hand a list of the cloud platforms described by the feature model are presented and developers can choose which cloud platform should be matched with each class or package. The assistant detects the class dependencies that are dissected in the process of segmenting the source code and declares the services and service clients in the xml document that will allow communication between components to be maintained once they are deployed in different clouds. The services are declared as stateless or stateful services depending on whether the original classes contained fields that maintained a state. Additionally, if different types of services are supported by the cloud platforms for communicating remote artefacts, developers are asked to select the service type that they wish to use in each case: SOAP, REST, etc.

The next step consists on configuring the adaptation points of each artefact with its corresponding cloud. Classes and interfaces from each artefact can be mapped to either of the services provided by the artefact's targeted cloud. The catalogue of candidate services on which adaptation can be applied for each cloud is documented in the cloud platform feature model.

Finally a cloud deployment plan is generated with the configuration chosen by the developer. At this stage the deployment plan contains all the information required by the *Source Transformation Engine* to generate each of the cloud artefacts.

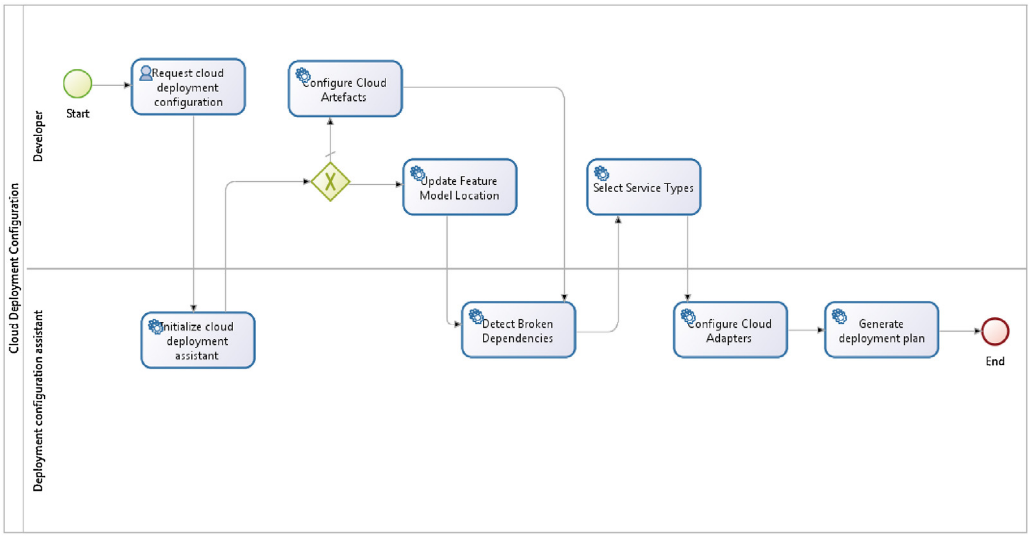
5.1.3. Project building and cloud artefact generation

Once a valid configuration has been set for a cloud application, and all of the classes have been correctly assigned to a cloud platform, developers can build the project and generate the cloud artefacts that are deployed across several cloud platforms. Process 3 illustrates the tasks that are executed for generating the cloud artefacts.

The artefact generation process begins by parsing the deployment plan in order to validate that the file is syntactically and semantically correct. Thereafter the cloud compiler analyses which cloud platforms have been targeted in order to generate the project structure of the cloud artefacts. This step is done using the project templates contained in the feature model for each cloud platform. The project templates may have to be configured with parameters that are specific to each cloud platform. Therefore each template contains an XML coded file that specifies which files have to be configured in the template, and maps each configurable parameter with a parameter contained in the deployment plan.

Once each cloud artefact has been correctly created and configured, the cloud-specific configuration of each artefact found in the deployment plan is interpreted in order to generate its interoperability services and service clients, and its cloud-specific adapters. These automatically generated components are copied into the cloud artefact with its source code, and configured using the Spring framework.

Each artefact can then be deployed in its correspondent cloud platform. If an existent application has to be deployed in a new cloud or distributed differently across the clouds, Processes 2 and 3


Cloud Deployment Configuration
Developer
Deployment configuration assistant
Start
Request cloud deployment configuration
Configure Cloud Artefacts
Update Feature Model Location
Select Service Types
Initialize cloud deployment assistant
Detect Broken Dependencies
Configure Cloud Adapters
Generate deployment plan
End

Process 2. Cloud deployment plan configuration.

have to be executed once again in order to generate a new deployment plan and the application's cloud-artefacts, respectively.

6. Integrating the framework in an industrial research project

The framework proposed in this paper was applied in an industrial research project for developing a cloud-based application for a bank, which was one of Gloin's customers. In this section we will analyze the application that was developed and how the framework was used to generate artefacts that could be deployed across multiple clouds.

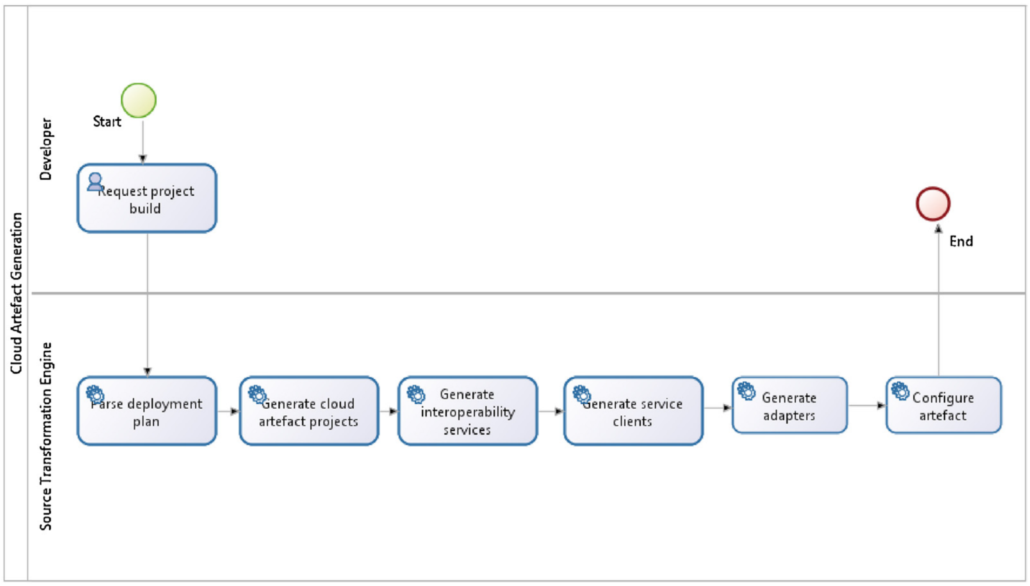
6.1. Application description

In order to evaluate the possibilities of migrating some of the client's secondary systems to the cloud a pilot project was proposed to explore the capabilities of cloud technologies for developing reports and statistics about the bank's customers, as well as their use of the bank's services, their economic activity and financial information.

To be precise, the pilot project aimed to generate statistics about the following points:

- Personal information: age, gender, professional activity and studies of its customers.
- Financial information: salary, expenses, investments, electronic commerce, loans and mortgage payments of its customers.
- Bank/customer related information: information about the bank's products consumed by each customer, interaction of each customer with the bank's online services, average number of times customers visit the bank's offices, etc.

This information had to be crossed in order to generate an exhaustive set of statistics that allowed the bank to have a profound knowledge on the behavioural patterns of its customers. Additionally first-hand information on the adoption of cloud technologies would be gathered.


Cloud Artefact Generation
Developer
Source Transformation Engine
Start
Request project build
End
Parse deployment plan
Generate cloud artefact projects
Generate interoperability services
Generate service clients
Generate adapters
Configure artefact

Process 3. Project building.

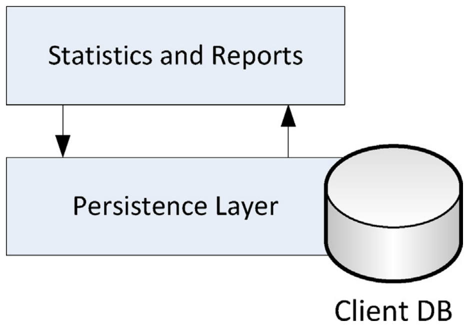


Fig. 5. Statistics and reporting application structure.

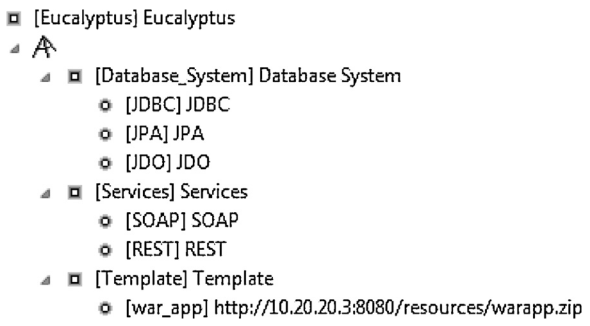


Fig. 6. Eucalyptus project specific feature diagram.

The bank had more than 100,000 customers which were generating an average of 14 financial transactions per client per month, which resulted in approximately 1.4 million transactions per month. This information had to be analyzed and crossed with each customer's information in order to generate the statistics and reports on a monthly basis. The process required a high computing power due to the complexity of the statistics processes and to the large amount of data that had to be managed.

One of the primary goals of the project consisted on reducing the costs of hardware as much as possible. At the time it was launched a limited amount of hardware was available, and undergoing a strong investment in buying hardware was something to avoid. For this reason a cloud solution was proposed, where hardware resources were accessed at an affordable price without undergoing an important initial investment. However the bank's executives were highly reluctant to export customer related information to a public cloud platform considering that the information managed by their system was extremely confidential. Hence, a hybrid cloud solution was proposed, where a private cloud would be deployed in their proprietary systems for data storage, and a public cloud with a higher computing potential would be used for analyzing the data stored in the private cloud and generating the statistics and reports. All information extracted from the private cloud needed to be depersonalized in order for data confidentiality to be kept in the process of sending data between cloud platforms.

The framework presented in this paper was used for developing the application; therefore developers did not have to consider which cloud platforms would be used for hosting the applications during the development process. An application with the structure illustrated in Fig. 5 was coded:

In this case the persistence layer, responsible for extracting client information from a relational database had been clearly separated from the statistics and reports logic of the application in order to emphasize the need of cleaning all personal and confidential information before returning it to the statistics and reports component. The nature of the project determined that the statistics and reports component would be located in a different cloud and this conditioned which information could be sent out of the bank's network. In any case, customer names, bank account numbers, ID card numbers, etc., were not at all relevant for generating statistics and reports.

6.2. Framework usage

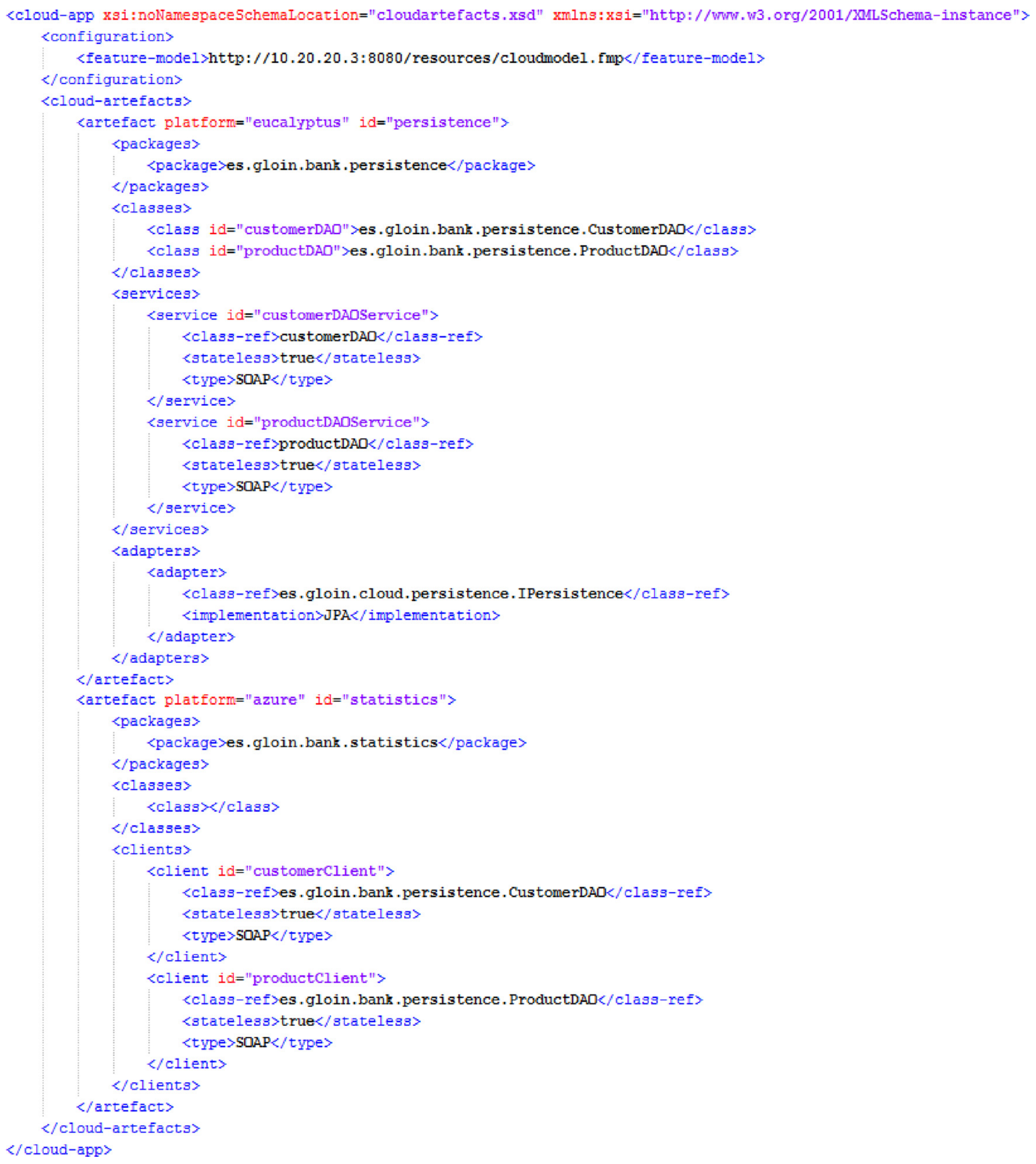
Once the application had been completely implemented a decision was made as to which cloud platforms each component would be deployed in. The client database and the persistence layer was deployed in a private cloud in order to guarantee that confidential data was kept in a controlled environment. For this purpose an open source Eucalyptus cloud platform was set up in one of the available systems of the bank in order to lower the costs of the project. On the other hand Azure's cloud was chosen by the bank for hosting the statistics and reports generation component, as they had previously developed other projects using this cloud and they were satisfied with the results.

In order for this deployment plan to be supported by the framework, Eucalyptus was included into the framework's feature model; this allowed us to generate Eucalyptus compliant cloud artefacts. However since Eucalyptus is a private IaaS cloud platform it imposes much less restrictions upon its applications than other public clouds. The feature model for Eucalyptus applications is a lot more flexible and it may have to be customized by each organization depending on how they set up their Eucalyptus images. In our case a Eucalyptus image was set up with a MySQL database and a Tomcat Fig. 6 application server. For this image the feature model shown in Fig. 6 applied.

As we can appreciate in the feature model no cloud-specific services were modelled for Eucalyptus. The only resource that was consumed from the Eucalyptus image was its MySQL database. However Eucalyptus does not provide any services for managing relational data. Instead direct connections to the database were established; this being the reason why the supported connection capabilities of the framework to the database were modelled (JDBC, JPA and JDO). Regarding interoperability, services could be provided using SOAP and REST; even though a wider range of service technologies were supported by the platform these were not included in the feature model as they had not yet been implemented for the framework. Finally, since no special restrictions were applied on the structure of the applications deployed for this cloud, a generic WAR application structure that could be deployed in the Tomcat 6 server was used for generating the cloud artefact.

Once the Eucalyptus image was modelled, the aforementioned deployment scenario was configured in the application's deployment plan; a process that required developers to identify the adaptation needs of each artefact with its targeted cloud platform. In this case the statistics and reports generation component did not require any integration with Azure's services, whereas the persistence layer hosted in Eucalyptus did have to be integrated with the relational database hosted by the cloud. The adaptation process consisted on identifying the CRUD operations carried out by the application upon the clients database and generating a CRUD SQL operation based adapter that accessed the database. Additionally the interoperability services and service clients required by each artefact were configured accordingly in the deployment plan. Fig. 7 illustrates an excerpt of the deployment plan generated for the application. Some of its content has been omitted for the sake of clarity.

In this configuration we observe that two cloud artefacts were deployed: a Eucalyptus artefact implemented in the *es.gloin.bank.persistence* package, and an Azure artefact implemented in the *es.gloin.bank.statistics* package. The Eucalyptus artefact provided services that managed the application's



```xml
<cloud-app xsi:noNamespaceSchemaLocation="cloudartefacts.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <configuration>
        <feature-model>http://10.20.20.3:8080/resources/cloudmodel.fmp</feature-model>
    </configuration>
    <cloud-artefacts>
        <artefact platform="eucalyptus" id="persistence">
            <packages>
                <package>es.gloin.bank.persistence</package>
            </packages>
            <classes>
                <class id="customerDAO">es.gloin.bank.persistence.CustomerDAO</class>
                <class id="productDAO">es.gloin.bank.persistence.ProductDAO</class>
            </classes>
            <services>
                <service id="customerDAOService">
                    <class-ref>customerDAO</class-ref>
                    <stateless>true</stateless>
                    <type>SOAP</type>
                </service>
                <service id="productDAOService">
                    <class-ref>productDAO</class-ref>
                    <stateless>true</stateless>
                    <type>SOAP</type>
                </service>
            </services>
            <adapters>
                <adapter>
                    <class-ref>es.gloin.cloud.persistence.IPersistence</class-ref>
                    <implementation>JPA</implementation>
                </adapter>
            </adapters>
        </artefact>
        <artefact platform="azure" id="statistics">
            <packages>
                <package>es.gloin.bank.statistics</package>
            </packages>
            <classes>
                <class></class>
            </classes>
            <clients>
                <client id="customerClient">
                    <class-ref>es.gloin.bank.persistence.CustomerDAO</class-ref>
                    <stateless>true</stateless>
                    <type>SOAP</type>
                </client>
                <client id="productClient">
                    <class-ref>es.gloin.bank.persistence.ProductDAO</class-ref>
                    <stateless>true</stateless>
                    <type>SOAP</type>
                </client>
            </clients>
        </artefact>
    </cloud-artefacts>
</cloud-app>
```

Fig. 7. Excerpt of the cloud artefact configuration for the statistics application.

persistence layer. In order to manage Eucalyptus' artefact connection to the database a JPA persistence adapter was used by the component. On the other hand the Azure cloud artefact contained service clients for invoking the aforementioned services.

The final step consisted on generating the cloud artefacts based on the configuration illustrated in Fig. 7. These artefacts were correctly generated and configured by the *Source Transformation Engine* Maven plugin thereby leading to the deployment scenario illustrated in Fig. 8:

7. Lessons learned

The project described in the previous section was developed in conjunction between professional developers from Gloin and the customer's internal development team. In order to gather information about the use of the framework on behalf of the development teams, a qualitative study was carried out between some of the developers that had used the framework. Developers were interviewed firstly during the mid-stage of the project and finally at the end of the project. The goal of these interviews was to gather their impressions on the use of the framework.

Considering that qualitative information about their experience with the framework was being sought for, the sample of interviewed developers was chosen by selecting only those that had more than two years of experience working with Java frameworks. This selection criterion was chosen in order to filter out inexperienced developers who had not yet grown a personal opinion on which features they would look after in a software development framework. As a result of this a sample of two developers from Gloin and three developers from the customer's development team were chosen, making a total of five experienced developers.

Considering that a case study methodology (Yin, 1994) was used to gather the personal experience of the samples, the interviews that were conducted at the mid-stage and the final stage of the

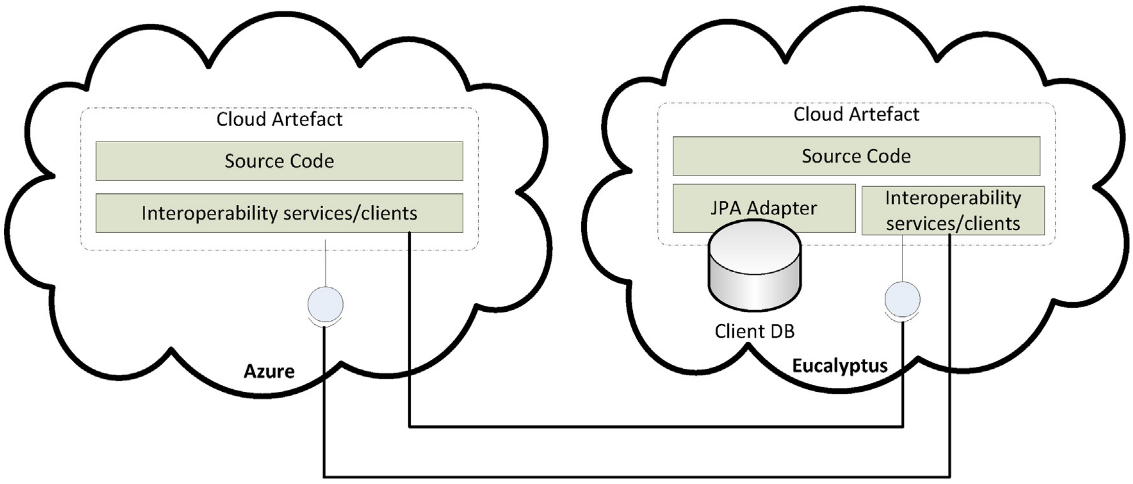


Fig. 8. Statistics and reporting cloud artefacts.

project consisted on open questions that were made to the developers with the following goals:

- To obtain information on whether the framework being analyzed was fully functional and that all of its components and tools were working correctly.
- To evaluate their experience with the framework in terms of its ease of use.
- To evaluate whether at any stage during the development process they were conditioned by specific features of the cloud in which the application was going to be deployed in.
- To gather a list of enhancements that could be applied on the framework.

The questionnaires provided a very useful feedback about the use of the framework. During the interviews carried out at the middle stage of the case study developers indicated that they had had a low interaction with the framework. Since their efforts were still being put into developing the application's functionality they had not yet configured a deployment plan. Their interaction with the framework and the tools that had been integrated into their development environment simply consisted on creating the cloud application. Nevertheless most developers did remark that at that stage of the development process they had to be aware of which cloud services the application would have to be integrated with as this conditioned the development process. This information was useful since the services that would be provided by the cloud did not have to be implemented by the application.

The interviews carried out at the final stage of the case study revealed many more details than their predecessors. At this stage the application's deployment plan had been fully configured and much more experience had been gathered in the use of the framework.

In general terms the interviewed subjects were satisfied with the functionality of the framework. They considered that the tools that had been integrated into the IDE were very useful for configuring the deployment plan and generating the cloud artefacts.

A lesson learnt from the experience of the development team was that for the framework to be useful a feature diagram containing all of the supported clouds had to exist. This was the main limitation that came up since Eucalyptus was selected as the private cloud platform used by the project. A significant amount of time had been used for creating a valid virtual image in which to host the applications and documenting this virtual image in the project's feature model. Furthermore, since Eucalyptus' virtual image had been created with a series of components that were specific for this project, the documentation included in the feature model for Eucalyptus would not necessarily be the same for other projects where virtual images would be configured differently. This would be applicable to any IaaS environment where the framework was used and it was a drawback to consider. In such cases it would be highly recommendable for companies to adopt and document a set of preconfigured virtual images to use for most of their applications, as this would allow them to reuse the documentation found in the feature model.

Another limitation that was brought up by developers had to do with the application's testing process. As a consequence of not having to code service or service clients and not having to integrate the application's source code with a database access technology, testing the source code had to be done differently. The original application was tested using JUnit and mock objects had to be created in order to simulate the database connections. On the other hand, once each cloud artefact was generated it had to be tested using the mechanisms provided for this effect by each provider. For Azure the local simulator provided by the platform had to be set up, whereas a test image was created in the Eucalyptus cloud for deploying the tested cloud artefacts. Setting up this heterogeneous test environment required an important effort to be done on behalf of the development and testing teams involved in the project. This is a cost that must be taken into account in projects where cross-cloud applications are being developed, independently if the framework presented in this work or an alternate framework is being used.

Using the framework in a real development project allowed us to clearly identify its strengths and weaknesses and to establish a roadmap for our future work based on the feedback that had been gathered from the development teams. The framework, combined with the plugins and tools which it was composed of, established a powerful tool for developing migratable cloud artefacts that could be deployed across several clouds, however work remained to be done on documenting a wider range of cloud platforms in order for the deployment to be done more freely. Documenting more cloud platforms would also imply generating their corresponding project templates and documenting their services in order to be capable of generating adapters for them, hence implying that an important amount of work remains to be done for this. Additionally, the service technologies supported by the framework (SOAP and REST) should be further expanded in order to support alternative technologies such as JMS messaging, EJB invocations, etc. This would provide much more flexibility to the framework in cloud environments, such as Eucalyptus, which do not impose as many restrictions on their applications.

Another point that is currently being worked on is related to the dynamic instantiation of the generated cloud-artefacts. The current approach, where each cloud artefact relies on generated source code for interoperability and cloud adaptation purposes, has a clear trade-off in dynamic environments. If an application's deployment plan changes, its newly generated interoperability services and cloud adapters have to be replaced in each of its cloud artefacts, which have to be redeployed in their corresponding cloud platform. Depending on the complexity of the application itself as well as the complexity of its deployment plan this may turn out to be disturbing. Nevertheless work is currently being done in this direction by analyzing dynamic adaptation techniques (Gomaa and Hashimoto, 2011; Camara et al., 2007) that will allow us to integrate a dynamic adaptation engine into the framework.

8. Related work

Applications developed for cloud platforms often have to confront the lock-in problem caused by the restrictions imposed by many cloud providers. This is a recognized problem (Chow et al., 2009) that has concentrated the efforts of many researchers. Commonly solutions are provided in the scope of model-driven development for modelling cloud applications that can be transformed to platform-specific applications, or architectures are proposed to allow managing cloud instances; however little effort has been made on providing tools for assisting developers in the software development process, and for generating software that can indifferently be deployed across different cloud platforms.

The mOSAIC project, a reference initiative carried out as a Europe funded project, is related to our work in its motivation, perspective and manifesto (Martino et al., 2011). mOSAIC aims to provide a platform for developing applications that can easily be deployed in different clouds. It also allows the development of cross-cloud applications, known in their work as multicloud applications that can interoperate with one another and with the services provided by each cloud. Their solution is based on an API and a middleware platform that creates an abstraction layer between the developed applications and the cloud in which it will be deployed. The technology is also complemented with cloud brokers that manage SLA negotiations with each cloud platform as well as the deployment of each application. The solution provided in this case differs significantly from our approach in the use of an intermediate abstraction layer.

Tsai et al. (2010) point out in their work many of the problems discussed in this paper, specially emphasizing on how cloud applications are tightly coupled to cloud platforms, and how migrating these applications between clouds can be extremely difficult. A Service Oriented Cloud Computing Architecture (SOCCA) is proposed where cloud computing resources are componentized, standardized and combined in order to build a "cross-platform virtual computer". An ontology mapping layer is configured over these services as a means of masking the differences between cloud providers. Cloud brokers interact with the ontology mapping layer for deploying applications in one cloud or another depending on a series of parameters, such as the budget, SLAs and QoS requirements that are negotiated with each provider. SOCCA applications can be developed using the standard interfaces provided by the architecture or the platform unique APIs of a cloud provider.

Maximilien et al. (2009) use a different approach for combatting the heterogeneity of cloud platforms by proposing a middleware that behaves as a broker for cloud clients. The middleware provides tools and REST based APIs for deploying applications and consuming the services exposed by each cloud platform. Homogeneity on the invocation of cloud-specific services is achieved by modelling clouds according to a metamodel defined in the work and providing REST management services based on these models. Applications are deployed in one cloud or another using these services. Additionally adapters are provided in order to decouple the application from each cloud provider.

Middleware-based solutions such as those proposed by the mOSAIC project, Tsai et al. (2010) and Maximilien et al. (2009) constitute valid solutions for application migratability across clouds. They provide an abstraction layer that sits between the applications and the clouds allowing these applications to consume all of the supported clouds' resources homogeneously. Their adoption shifts the vendor lock-in problem to a middleware lock-in problem since applications become coupled to the intermediate architecture and APIs provided by the middleware. This contrasts with our approach where coupling does not occur at a low level, i.e. system execution level, but at a higher level with the adoption of the development model that is proposed for generating cloud dependant code from the combination of cloud agnostic code and a cloud feature model. Additionally middlewares are usually large atomic and complex software entities; hence, certain applications that do not require an extensive use of all of their functionality may find them exceedingly heavy. In such cases, our approach based on the use of software adapters, is a lightweight solution that may result beneficial.

Another approach based on model driven development is proposed by Hamdaqa et al. (2011), in this case for modelling cloud applications. A metamodel for cloud applications is defined, centralized in the definition of a cloud task as a "composable unit, which consists of a set of actions that utilize services to provide a specific functionality". The metamodel aims to decouple the design process from specific cloud platforms, however decoupling the source code from cloud platforms is not dealt with by the authors.

Another proposal based on MDD techniques is presented by Frey and Hasselbring (2010) as a means of mapping models of existent cloud environments to legacy software models and transforming the result to cloud-specific code through a series of iterations and result evaluations. This approach is fully oriented towards legacy software and it requires subsequent development tasks to be carried out upon generated source code; this may be an arduous task depending on the complexity of the generated source code.

The common factor found in these works is the wish for creating an abstraction of the peculiarities of each cloud platform in order to favour their management, as well as the migration of applications from one cloud to another or from non-cloud environments to cloud environments. Most proposals have been oriented towards the final result, however little has been mentioned on how the development of migratable cloud applications should be done, considering that the work carried out by developers has not been taken into account. Neither of the proposals provide an environment in which the development process is not conditioned by architectural restraints or cloud-specific requirements. No tools are provided for use in the development process, in order to generate manageable and structured source code which will favour its maintenance.

9. Conclusions

Developing applications for cloud platforms is currently a process that is often conditioned by the requirements and restrictions imposed by each individual cloud provider. As an immediate consequence, software that is tightly coupled to specific cloud platforms is often developed by organizations, thereby making vendor lock-in an inherent problem of cloud application development. This paper proposes a framework which aims to favour cloud agnostic software development. By separating all cloud related information from the source code the development process is no longer conditioned by external requirements and constraints. Additionally the benefits of the utility computing business model can be exploited by allowing applications to be deployed across more than one cloud, where each part of an application may take advantage of a particular set of features provided by a cloud provider (lower CPU cost, higher storage rate, privacy, etc.).

Organizations that choose to base their technological infrastructure on cloud technologies are searching for a reliable solution that will allow them to lower their costs; however under the current context many risks have to be assumed. An important investment is made on behalf of companies for creating cloud compliant software that in most occasions cannot easily be migrated to different clouds. Hence, cloud users become vulnerable to the conditions and terms of services of their providers, and cloud environments become an unreliable solution for organizations.

Considering that the nature of cloud platforms is unlikely to change during the upcoming years, we believe that tools have to be provided in order to make cloud technology much more attractive and accessible to its potential users. The work proposed in this paper aims to boost the confidence of companies in cloud environments by providing a framework that will allow them to exploit the potential of the technology without introducing any foreign elements into the development process. Companies can continue to build their applications without having to modify their technology, and most importantly, without having to choose a single immovable cloud provider.

Acknowledgements

This work has been partially funded by the Spanish Ministry of Science and Innovation under Project TIN2011-24278, as well as by Junta de Extremadura and FEDER funds.

References

Anderson, J., Rainie, L., 2010. The Future of Cloud Computing, Pew Internet & American Life Project, Report. http://pewinternet.org/Reports/2010/The-future-of-Cloud-computing.aspx

Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Kowinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M., 2009. Above the Clouds: A Berkeley View of Cloud Computing. EECS Department University of California Berkeley Tech Rep UCBEECS200928 (UCB/EECS-2009-28), p. 25. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf

Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Kowinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M., 2010. A view of cloud computing. Communications of the ACM 53 (4), 50–58.

Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M., 2006. Towards an engineering approach to component adaptation. Architecting Systems with Trustworthy Components, vol. 3938. LNCS, Dagstuhl Castle, Germany, pp. 193–215.

Bracciali, A., Brogi, A., Canal, C., 2005. A formal approach to component adaptation. Journal of Systems and Software 74 (1), 45–54.

Camara, J., Salaun, G., Canal, C.,2007. Run-time composition and adaptation of mismatching behavioural transactions. In: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods. IEEE Computer Society, Washington, DC, USA, pp. 381–390.

Canal, C., Murillo, J., Poizat, P., 2005. Coordination and adaptation techniques for software entities. In: Malenfant, J., Østvold, B. (Eds.), Object-Oriented Technology. ECOOP 2004 Workshop Reader. Springer, Berlin/Heidelberg, pp. 133–147.

Canal, C., Murillo, J.M., Poizat, P., 2006. Software adaptation. L'Objet. Software Adaptation 12 (1), 9–31.

Canal, C., Poizat, P., Salaun, G., 2008. Model-based adaptation of behavioral mismatching components. IEEE Transactions on Software Engineering 34 (4), 546–563.

Chow, R., Golle, P., Jakobsson, M., Shi, E., Staddon, J., Masuoka, R., Molina, J., 2009. Controlling data in the cloud: outsourcing computation without outsourcing control. In: Proceedings of the 2009 ACM Workshop on Cloud Computing Security, CCSW, pp. 85–90.

Dillon, T., Wu, C., Chang, E., 2010. Cloud computing: issues and challenges. In: 24th IEEE International Conference on Advanced Information Networking and Applications, pp. 27–33.

Foster, I., Frey, J., Graham, S., Tuecke, S., Czajkowski, K., Ferguson, D., Leymann, F., Nally, M., Sedukhin, I., Snelling, D., Storey, T., Vambenepe, W., Weerawaranna, S., 2004. Modeling stateful resources with web services. Library Hi Tech 24 (4), 496–503, Available from: http://www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf

Filman, R.E., Friedman, D.P., 2000. Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns, OOPSLA'00.

Fowler, M., 2004. Inversion of control containers and the dependency injection pattern. Interface M (1), 1–19, Available from: http://www.martinfowler.com/articles/injection.html

Frey, S., Hasselbring, W., 2010. Model-based migration of legacy software systems to scalable and resource-efficient cloud-based applications: the cloud MIG approach. In: Cloud Computing 2010: Proceedings of the 1st International Conference on Cloud Computing, GRIDs, and Virtualization, pp. 155–158.

Gomaa, H., Hashimoto, K., 2011. Dynamic software adaptation for service-oriented product lines. Proceedings of the 15th International Software Product Line Conference, vol. 2. ACM, New York, NY, USA, pp. 35:1–35:8.

Hamdaqa, M., Livogiannis, T., Tahvildari, L.,2011. A reference model for developing cloud applications. In: Proceedings of the 1st International Conference on Cloud Computing and Services Science. SciTePress, pp. 98–103.

Leavitt, N., 2009. Is cloud computing really ready for prime time? Computer 42 (1), 15–20.

Lee, K., Kang, K., Lee, J., 2002. Concepts and guidelines of feature modeling for product line software engineering. In: Gacek, C. (Ed.), Software Reuse: Methods, Techniques, and Tools. Springer, Berlin/Heidelberg, pp. 62–77.

Li, A., Yang, X., Kandula, S., Zhang, M., 2010. CloudCmp: comparing public cloud providers. Compute 57 (2), 1–14.

Martino, B.D., Petcu, D., Cossu, R., Goncalves, P., Máhr, T., Loichate, M.,2011. Building a mosaic of clouds. In: Proceedings of the 2010 Conference on Parallel Processing. Springer-Verlag, pp. 571–578.

Maximilien, E.M., Ranabahu, A., Engehausen, R., Anderson, L.C., 2009. Toward cloud-agnostic middlewares. In: OOPSLA09: 14th Conference Companion on Object Oriented Programming Systems Languages and Applications, pp. 619–626.

Miranda, J., Guillén, J., Murillo, J.M., Canal, C.,2012a. Enough about standardization, let's build cloud applications. In: Proceedings of the WICSA/ECSA 2012 Companion Volume. ACM, New York, NY, USA, pp. 74–77.

Miranda, J., Guillén, J., Murillo, J.M., Canal, C., 2012b. Identifying adaptation needs to avoid the vendor lock-in effect in the deployment of cloud SBAs. In: Proceedings of ESOCC 2012.

Petcu, D., Craciun, C., Neagul, M., Lazcanotegui, I., Rak, M., 2011. Building an interoperability API for sky computing. In: 2011 International Conference on High Performance Computing and Simulation (HPCS), pp. 405–411.

TIOBE Software: The Coding Standards Company, 2012. TIOBE Software: The Coding Standards Company. Available from: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Tsai, W.-T., Sun, X., Balasooriya, J., 2010. Service-oriented cloud computing architecture. In: 2010 Seventh International Conference on Information Technology New Generations, pp. 684–689.

TechTarget, 2010. What is Cloud Computing? – Definition from WhatIs.com. Available from: http://searchcloudcomputing.techtarget.com/definition/cloud-computing

Yin, R.K., 1994. Case Study Research: Design and Methods (Applied Social Research Methods), 2nd ed. Sage Publications Inc., California, USA.

Zhang, Q., Cheng, L., Boutaba, R., 2010. Cloud computing: state-of-the-art and research challenges. Journal of Internet Services and Applications 1 (1), 7–18.

Joaquín Guillén Melo received the MS degree in Computer Science from the University of Extremadura in 2008 and is currently a PhD student at the same university. He works as a software engineer and project manager at Gloin. His research interests include model driven engineering and cloud computing.

Javier Miranda received the MSc degree in computer science and the Master of Advanced Studies (MAS) from the University of Extremadura, Spain, in 2006 and 2009, respectively. In 2011 he joined Quercus Engineering Group, where he is currently working towards the PhD degree in Computer Science. His current research interests include business process engineering and cloud computing.

Juan Manuel Murillo received his PhD in Computer Science from the University of Extremadura in 2001. He is a co-founder of Gloin and a Professor at the University of Extremadura. His research interests include model driven engineering and cloud computing.

Carlos Canal received his PhD degree in Computer Science from the University of Málaga, Spain, in 2001, where he is currently Associate Professor of Software Engineering. His current research interests include component and service-based software development and software adaptation, in particular with the application of model driven techniques and formal methods to the specification, safe composition, and adaptation of interacting software components.