Rapid Development of Service-based Cloud Applications:

The Case of the Cloud Application Platforms

Fotis Gonidis, Iraklis Paraskakis
South-East European Research Centre, Greece, The University of Sheffield, UK
Iraklis Paraskakis
South-East European Research Centre, Greece
Anthony J. H. Simons
The University of Sheffield, UK

ABSTRACT

Cloud application platforms gain popularity and have the potential to alter the way service-based cloud applications are developed involving utilisation of platform basic services. A platform basic service provides certain functionality and is usually offered via a web API. However, the diversification of the services and the available providers increase the challenge for the application developers to integrate them and deal with the heterogeneous providers' web APIs. Therefore, a new approach of developing applications should be adopted in which developers leverage multiple platform basic services independently from the target application platforms. To this end, the authors present a development framework assisting the design of service-based cloud applications. The objective of the framework is to enable the consistent integration of the services, and to allow the seamless use of the concrete providers. The optimal service provider each time can vary depending on criteria such as pricing, quality of service and can be determined based upon Big Data analysis approaches.

Keywords: Platform Basic Service, Code Generation, Cloud Computing, Big Data, Platform as a Service, Heroku, Fotis Gonidis, Iraklis Paraskakis, Anthony Simons, South East European Research Centre, The University of Sheffield.

INTRODUCTION

In recent years two major technological trends have emerged and are able to drive the evolution of service oriented computing, namely cloud computing and Big Data. The rise and proliferation of cloud computing (Armbrust et al., 2010) and cloud platforms in specific (Cusumano, 2010), has the potential to change the way cloud based service applications are developed, distributed and consumed. Cloud platforms popularity stems from their potential to speed up and simplify the development, deployment and maintenance of cloud based software applications. Nevertheless, there is a large heterogeneity in the platforms offerings (Gonidis, Paraskakis, Simons, & Kourtesis, 2013), which can be classified into three clusters. On one cluster application development time is drastically decreased with the use of bespoke

visual tools and graphical environments at the expense of a restricted application scope which is usually limited to customer relationship management (CRM) and office solutions. At the other end of the spectrum platforms offer basic development and deployment capabilities such as application servers and databases. The intermediate cluster consists of cloud platforms, which offer additional functionality via the provisioning of, what the authors call, platform basic services (e.g. mail service, billing service, messaging service etc.). A platform basic service can be considered as a piece of software, which provides certain functionality and can be reused by multiple users. It is typically provisioned via a web API. The platforms offering such services are also referred to as cloud application platforms (Kourtesis, Bratanis, Bibikas, & Paraskakis, 2012). The rise of the cloud application platforms has the potential to lead to a paradigm shift of software development where the platform basic services act as the building blocks for the creation of service-based cloud applications.

Almost in parallel with the emergence of cloud computing, data volumes started skyrocketing leading to what is commonly referred nowadays as Big Data (Jacobs, 2009). The term Big Data has been initially attributed with certain features (McAfee, & Brynjolfsson, 2012) such as: *a) Volume, b) Velocity and c) Variety*. Volume refers to the large amount of data which are generated each day. Velocity dictates the need for analysis of the rapidly collected data as those are becoming outdated quickly. Variety denotes the diverse forms and formats in which data are collected and stored. In addition to these features, Demchenko et al. (2013) propose a wider definition of Big Data by adding two extra characteristics: *d) Veracity* and *e) Value*. Veracity refers to the amount of uncertainty that the collected data contain and to the extend they can be considered trustworthy. Value denotes the added-value that data can bring to the predictive analysis. Therefore, what once used to be a technical problem due to limited storage and processing capacity now it is being transformed into a business opportunity (The Data Warehouse Institute, 2011). The collection and analysis of Big Data can lead to the extraction of meaningful information and can subsequently drive important decisions (McAfee, & Brynjolfsson, 2012).

In the field of service oriented computing the combination of the cloud computing and Big Data analysis has the potential to lead to the design of flexible and adaptable service-based cloud applications where their functionality and the concrete service providers are determined based on the stimuli from the environment. The criteria based upon the selection of the appropriate services are made, can be based upon several factors such as pricing, quality of service, current availability of the provider, geospatial data etc. In certain cases the selection of the concrete service providers is a time critical operation and is subject to the analysis of big volume of data. Therefore, Big Data approaches can be employed to process and analyse the large volume of data, acting this way as decision supporters in the selection of the concrete service providers. As discussed later in this article the world of E-commerce and electronic (mobile) payments can be used as motivating scenario in this research work. Gartner, the leading information technology research company, reports a 42% average increase in the mobile transaction volume and value in the period 2011-2016 (Gartner Inc., 2012). Therefore, cloud payment services, enabling an application to handle electronic payments, is becoming an indispensable part of a service-based cloud applications. There is a plethora of available payment service providers offering a variety of features regarding the price, the quality of service, the geographical region etc. Different payment providers may at each time serve better the needs of the application. In this case, Big Data analysis techniques may be applied in order to determine the optimal payment provider while the application should be able to choose seamlessly the given provider without disrupting its operations.
Consequently, the combination of Big Data approaches and methodologies of designing service-based cloud applications has the potential to pave the way to reactive applications, which are context aware and able to adopt themselves according to external stimuli. The adaptability lies primarily in the capability to change seamlessly the concrete service-providers without the intervention of a software engineer.

The design and the development of a large scale service-based cloud application involve the consideration of a significant amount of aspects such as: the cloud deployment model, the interoperability aspect, the

quality of service along with the definition of service level agreements (SLAs), the ability to manage and migrate data in a feasible manner and the metering and billing of the cloud application (Rimal, Jukan, Katsaros, & Goeleven, 2010). The current research article focus on a particular aspect of the design cycle, which involves enabling the application to leverage multiple platform basic services provisioned from heterogeneous service providers in a transparent way.

The heterogeneity mainly arises due to (i) the differences in the workflow for the execution of the operations of the services, (ii) the differences in the exposed web APIs and (iii) the various required configuration settings and authentication tokens. The significant number of services that an application may consist of makes the integration and management of the services a strenuous process. The mission of this article is to propose a development framework assisting in the design and execution of the service-based cloud applications by alleviating the three afore mentioned variability issues. Specifically, the target of the proposed framework is twofold: (i) Provide the tools and the methodology to enable the consistent modelling and integration of different categories of platform basic services with the cloud application. (ii) Enable the applications to deploy concrete service providers in a seamless, transparent and agnostic to the software engineer manner.

The rest of the article is structured as follows. Next Section provides background information and related work on the field. Thereafter the motivating example, which is used for the rest of the article, is stated. Subsequently, the main issues, that the article is trying to address, are listed and the high-level architecture of the framework and the components involved in the development process are described. Then, the article focuses on the technical design of the proposed framework and describes how the latter can be used to enable the integration of additional services and providers. Finally, the article concludes with certain limitations of the framework as well as with proposals for future research direction.

BACKGROUND AND RELATED WORK

The emergence of the service-based cloud applications, as a synthesis of various platform basic services, promises to ease and speed up the process of cloud application creation. Applications do not need to be developed from scratch but can rather be constructed using, where appropriate, various platform services, thus increasing rapidly the productivity. Consequently, the barrier of studying the various platform basic services and selecting the one(s) best offered for the task at hand, is now removed. The software engineer has access in a transparent manner to all platform basic services and the selected services are seamlessly incorporated in the service-based cloud application. However, in order for this scenario to be realised, two conditions should be met: (i) the application should be deployable seamlessly in various cloud application platforms and (ii) the variability among the services should be alleviated.

With respect to the second condition, which as mentioned in the Introduction is the focus of this article, a number of challenges arise. The first challenge arises from the fact that there exist multitudes of a particular service, e.g., mail service, since the services are offered by many different providers. The second challenge arises from the need to provide a framework that spans across a number of different kind of services, i.e., mail services, payment services, message queue services and so on. At the same time there is a lack of tools and Integrated Development Environments addressing the issue of proprietary technologies and APIs (Guillen, Miranda, Murillo, & Cana, 2013a).

The constant increase in the offering of platform services has resulted in a growing interest in leveraging services from multiple clouds. Significant work has been carried out on the field. Petcu and Vasilakos (2014) summarises several library-based solutions, which expose a uniform API and can abstract cloud resources such as data storage, compute and message queue services from various providers. Furthermore, several standardisation approaches are listed which aim at the definition of standards enabling the uniform access to cloud resources. Model-Driven Engineering (MDE) techniques are also proposed as enabling

methodology for the creation of cloud applications leveraging services from multiple cloud providers. Guillen et al. (2013) also put forward the idea of model-driven development of cloud applications which are platform agnostic. Moreover, middleware-based solutions are listed aiming at managing the deployment and execution phase of the cloud application.

This article lists representative work from the fields mentioned above, namely: standardisation approaches, b) library-based solutions, c) model-driven engineering techniques, d) and middleware platforms.

Open Virtual Format (OVF) (DMTF, 2013) is a specification for packaging and distributing Virtual Machines (VMs), defined by the Distributed Management Task Force (DMTF) (http://www.dmtf.org/). The architecture of the format is not bound to a particular platform or operating system and thus enables virtual machines to be deployed on different cloud infrastructure providers. Open Cloud Computing Interface (OCCI) (http://occi-wg.org/) created by Open Grid Forum (OGF) (https://www.ogf.org/ogf/doku.php) and Cloud Infrastructure Management Interface (CIMI) (DMTF, 2012) created by DMTF both attempt to standardize the way users access and manage infrastructure resources and therefore to unify the various proprietary APIs that vendors are currently using. They are particularly focus on the compute, storage and network resources. Cloud Data Management Interface (CDMI) (SNIA, 2014) is a cloud storage standard defined by Storage Networking Industry Association (SNIA) (http://www.snia.org/). CDMI attempts to standardize the way users, access and manage cloud storage services offered by storage providers such as Google Storage, Amazon Simple 3 and Windows Azure Storage. Topology and Specification for Cloud Applications (TOSCA) (Binz, Breiter, Leymann, & Spatzier, 2012) is a standardization effort from OASIS (Advancing Open Standards for the Information Society) (https://www.oasis-open.org/) aiming at defining a common representation of the cloud services that a cloud application uses such as application servers and databases. This way TOSCA attempts to automate the deployment process of a cloud application in multiple cloud platforms. While standardisation is an efficient approach to leverage services from multiple cloud environments, as Petcu (2014) and Opara-Martins et al. (2014) highlight, standardisation efforts are still in an immature level and mainly focus on the IaaS level. This is mainly due to the reluctance of the cloud providers to agree in standardised interfaces and specifications as this will lead to an increased direct competition with other providers (Singhal et al., 2013).

Library-based solutions such as jClouds (http://www.jclouds.org.) written in Java and LibCloud (https://libcloud.apache.org/index.html) written in Python, provide an abstraction layer for accessing specific cloud resources such as compute, storage and message queue. While, library-based approaches efficiently abstract those resources, they have a limited application scope which makes it difficult to reuse them for accommodating additional services.

Middleware platforms constitute middle layers, which decouple applications from directly being exposed to proprietary technologies and deployed on specific platforms. Rather, cloud applications are deployed and managed by the middleware platform, which has the capacity to exploit multiple cloud platform environments. mOSAIC (Petcu, 2014) is such a PaaS solution which facilitates the design and execution of scalable component-based applications in a multi-cloud environment. mOSAIC offers an open source API in order to enable the applications to use common cloud resources offered by the target environment such as virtual machines, key value stores and message queues. OpenTOSCA (Binz et al., 2013), is a runtime environment enabling the execution of TOSCA-based cloud applications. TOSCA (Binz et al., 2012) is a specification which enables the description of the deployment topology of a cloud application in a platform independent way. Thus, applications are agnostic with regard to the concrete platform provider resources they use. Both mOSAIC and OpenTOSCA require that applications are developed using the specific technologies and thus impose a restriction in case applications need to leverage platform providers, which are not supported by those environments.

Initiatives that leverage MDE techniques present meta-models or Domain Specific Languages (DSLs), which can be used for the creation of cloud platform independent applications. The notion in this case is that cloud applications are designed in a platform independent manner and specific technologies are only infused in the models at the final stage of the development. MODAClouds (Ardagna et al., 2012) and PaaSage (Jeffery, Horn, & Schubert, 2013) are both FP7 initiatives aiming at cross-deployment of cloud applications. Additionally, they offer monitoring and quality assurance capabilities. They are based on CloudML (Ferry et al, 2013), a modelling language which provides the building blocks for creating applications deployable in multiple IaaS and PaaS environments. Hamdaqa et al. (2011) have proposed a reference model for developing applications which leverage the elasticity capability of the cloud infrastructure. Cloud applications are composed of CloudTasks, which provide compute, storage, communication and management capabilities. MULTICLAPP (Guillen et al., 2013b) is a framework leveraging MDE techniques during the software development process. Cloud artefacts are the main components that the application consists of. A transformation mechanism is used to generate the platform specific project structure and map the cloud artefacts onto the target platform. Additional adapters are generated each time to map the application`s API to the respective platform`s resources. MobiCloud (Ranabahu, Maximilien, Sheth, & Thirunarayan, 2013) is a DSL enabling the design of cloud-mobile hybrid applications which are platform independent. The application is initially designed using the MobiCloud DSL and subsequently specific code generators are used to generate the executable program for each specific cloud platform.

The solutions listed in this Section focus mainly on the cross-deployment of application by eliminating the technical restrictions that each platform imposes. However, they do not support the use of additional platform basic services offered via web API such as payment, authentication and e-mail service. In addition, the client adapters used to address the variability in the providers' APIs are hardcoded and thus not directly reconfigurable in case they are required to be updated. On the contrary, the vision of the authors is to facilitate the use of platform basic services from heterogeneous clouds in a seamless manner. To this end, the proposed solution attempts to alleviate the three variability points described in the Introduction, namely: the differences in the workflow modelling, in the providers' web APIs and in the configuration settings.

MOTIVATING SCENARIO

Electronic payment transactions have gained widespread acceptance in many domains of business and this trend is upwards considering the diffusion of mobile and contactless payments. Forrester Research Inc, a major independent research company, predicts that the US mobile payments growth will accelerate, reaching $90B by the year 2017, a 48% compound annual growth rate from the $12.8B spent in 2012. (Forrester Research Inc., 2012a; Forrester Research Inc., 2013). At the same time the Mobile Commerce in EU is following an exponential growth from €1.7B in 2011 up to €19.2B by 2017 (Forrester Research Inc., 2012b). Therefore it becomes obvious that payment service providers are becoming an essential part of the service-based cloud applications.

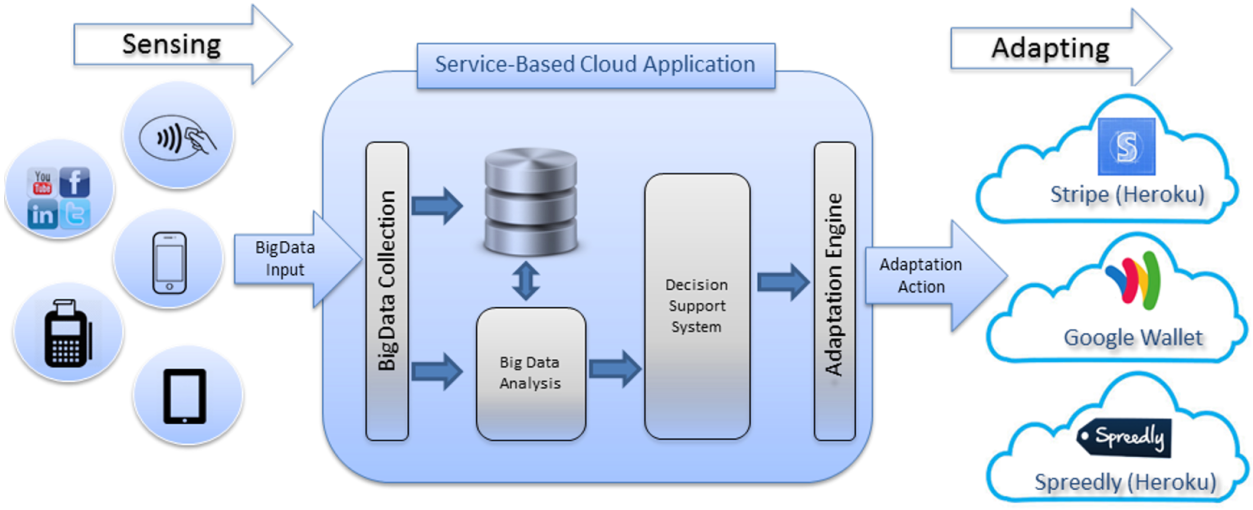
The payment service enables a website or an application to accept online payments via electronic cards such as credit or debit cards by intermediating between the application and the bank. Figure 1 shows a simplified view of the payment process. The added value that such a service offers is that it relieves the developers from handling electronic payments and keeping track of the transactions. Moreover, applications which perform billing transactions needs to be compliant with the Payment Card Industry Data Security Standard (PCI-DSS) in order to maximise its reliability. Acquiring the compliance may be a time consuming and costly process, which may be skipped with the use of a cloud payment service.

Figure 1. Simplified view of the payment process

There are a large number of available payment service providers, which can be deployed by the service-based cloud applications in order to process the payments. Those providers offer different features and may vary in the pricing, the quality of service etc. Large enterprises such as Amazon and Google, operating globally, may require a variable number of payment providers depending each time on certain criteria such as the geographical region and the rate of the payment requests. Therefore, different service providers may at each time better serve the need of the cloud applications. For example in case of an instant peek in the number of transactions in a specific geographical region, the current deployed provider may become unresponsive. Thus the application should be able to predict the failure and deploy a different provider to seamlessly continue its operation.

Figure 2 shows such a scenario where Big Data contribute to the selection process of the payment provider. Data about payment requests are collected globally from a variety of sources. The collected data may expose a heterogeneity in the format since they are generated by diverse sources such as mobile devices, social networks and payment terminals. Moreover, the large velocity and the volume of the generated data impose that traditional data analysis techniques involving relational databases are not sufficient to cope with them. The collected data may indicate a rapid rise in the demand of the payment requests in a particular geographical region. Therefore, Big Data analysis techniques may be deployed to provide a timely prediction about the failure event. Subsequently, the service-based cloud application should be capable of choosing seamlessly a different provider in order to continue its operations in an undisrupted way.

Figure 2. Big Data as enabler for choosing the optimal service provider



The payment service providers are only one example of providers, which can be deployed based on the analysis of Big Data. The same rationale applies when deploying providers offering additional services such as e-mail and message queuing.

The feasibility of the proposed scenario depends on two aspects: (i) The analysis of the collected data through Big Data approaches and the subsequent recommendation of the best alternative service provider,

(ii) The ability of the service-based cloud application to seamlessly deploy an alternative payment service provider so that it continues its operations without disruptions.

As mentioned in the Introduction, the scope of this article is to propose a development framework to cope with the second aspect, namely the design of service-based cloud applications, which are not bound to specific service providers. In order to illustrate how the framework can be used in a real case scenario, the cloud payment service is used. This platform service has been chosen because of its inherent relative complexity compared to other services such as e-mail or message queue service. The complexity lies in the fact that the purchase transaction requires more than one step to be completed and there is a significant heterogeneity among the available payment providers with respect to the involved steps.

Next Sections state the concrete issues that the article aims at addressing and subsequently describe the solution approach.

VARIABILITY ISSUES

Preliminary work of the authors on several platform service providers (Gonidis, 2013) offered by Heroku (http://heroku.com), Google App engine (https://developers.google.com/appengine) and AWS marketplace (https://aws.amazon.com/marketplace) have shown that the following three variability points needs to be addressed in order to decouple application development from vendor specific implementations:

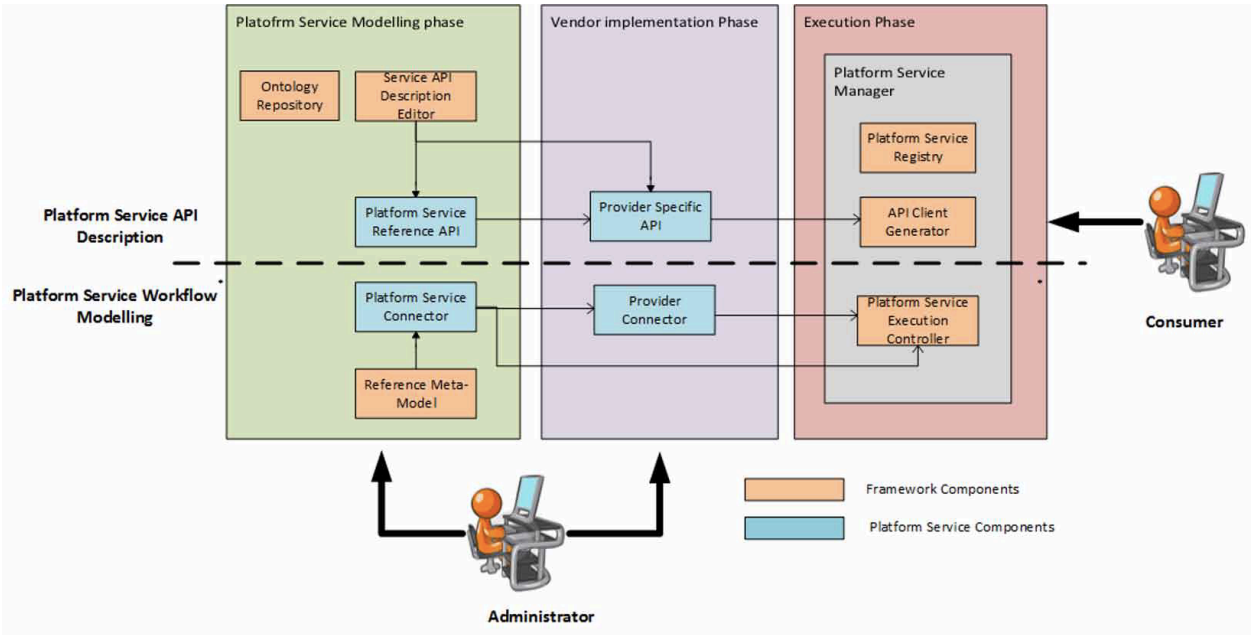
1. **Differences in the workflow:** Stateful services require more than one state in order to complete an operation (Pautasso, Zimmermann, & Leymann, 2008). Such an example is the payment service that enables developers to accept payments through their applications. The process involves two states: (i) waiting for client's purchase request and (ii) submitting the request to the payment provider. However, depending on the concrete payment provider there may be variations in the states involved. Therefore, a coordination mechanism is required to handle the operation flow and additionally to alleviate the differences among the various concrete implementations.
2. **Differences in the web API:** There are several platform providers implementing a given platform service and specific operations. However, they expose a diverse API resulting in conflicts when an application developer attempts to integrate with one or another. The e-mail service and two service providers, the Amazon Simple E-mail Service (SES) (aws.amazon.com/ses/) and the SendGrid (https://sendgrid.com/), an add-on mail service offered via Heroku application platform are considered as an example. Upon the request for sending an e-mail the minimum set of the four following parameters required by Amazon SES is: (i) Source, (ii) Destination.ToAddresses, (iii) Message.Subject and (iv) Message.Body.Text. In the case of SendGrid the anticipated parameters are: (i) from, (ii) to, (iii) subject and (iv) text.
3. **Management of the configuration and authentication variables:** Platform service providers require certain configuration settings and authentication tokens to be present during the interaction with the cloud application. Indicatively, the Google Authentication service and the following set of required variables are mentioned: a) the redirect URL, b) the client_ID, c) the scope and d) the state. The number and the type of the settings vary according to the provider. Considering the large number of services that an application may be composed of, the management of the settings may become a time consuming and strenuous process.

As it became clear a cloud application may interact with several platform basic services in various ways. In order to enable the consistent modelling and integration of services as well as the decoupling from vendor specific implementations, a development framework is proposed.

HIGH LEVEL OVERVIEW OF THE FRAMEWORK

This Section describes the high level architecture of the framework. In particular, it focuses on the components of the framework and the process required to add a new platform basic service or provider. There are two distinct users (roles): the administrator and the developer. The first uses the components in order to enrich the framework with additional services and providers while the latter makes use of its components in order to integrate platform services with the cloud application.
Figure 3 illustrates the components that constitute the development framework. The components are split in two categories, highlighted by the use of two colours. The one highlighted in orange are provided by the framework and are used by the administrator. The one highlighted in blue are the platform service components and are produced by the administrator using the framework components.

Figure 3. High-level architecture of the Framework



As it can be observed in Figure 3, the process of adding a new platform service and provider to the framework can be divided into the following two parts:

1. **Platform Service Workflow Modelling:** As explained in the previous Section, certain platform services require more than one step to complete an operation, such as the authentication and the payment service. Thus, the states that are involved in the execution of an operation shall be defined and modelled in a way that is capable for the framework to handle automatically the workflow.
2. **Platform Service API Description:** One of the main objectives of the framework is to provide the developers with a single API for each platform service, independent of the concrete provider. Therefore, this part involves the definition of the reference API, the description of the web API of each concrete provider supported by the framework and the subsequent mapping of the provider specific web API to the reference one.

Each of the two parts of the development process involves the three following phases: (a) *Platform service modelling phase, (b) Vendor implementation phase, (c) Execution phase.*
In the next subsections, for each of the two parts, the three phases are introduced and the high-level components involved, as depicted in Figure 3, are described.

Platform Service Workflow Modelling

1. **Platform Service Modelling Phase:** During this phase the abstract states of each platform basic service are described. The following components are involved in this phase:
    a. Reference Meta-Model. The Reference Meta-Model contains the concepts required to model the states of the platform service.
    b. Platform Service Connector. The Platform Service Connector (PSC) is the abstract representation of the platform service functionality and hides the specific implementation of the concrete service providers. It contains the states that are involved in each operation provided by the service. It is generated by the administrator of the framework using the concepts of the Reference Meta-Model. The PSC is used by the consumer of the framework to obtain access to the functionality of the service.
2. **Vendor Implementation Phase:** Based on the abstract model defined in the previous phase, the vendor specific implementation is infused. Specifically, the workflow required by each provider is mapped to the abstract one defined for the particular service.
    a. Provider Connector. The Provider Connector (PC) is the module which contains the specific implementation of the concrete service providers. It is constructed by the administrator of the framework based on the PSC which is built during the modelling phase.
3. **Execution Phase:** The execution phase takes place at run-time and coordinates the execution of the states involved in an operation offered by a service.
    a. Platform Service Execution Controller (PSEC). The PSEC handles the execution of the workflow. Particularly, it accepts as inputs the Platform Service Connector (PSC) and the Provider Connector (PC). Then it executes the workflow defined in the PSC.

Platform Service API Description

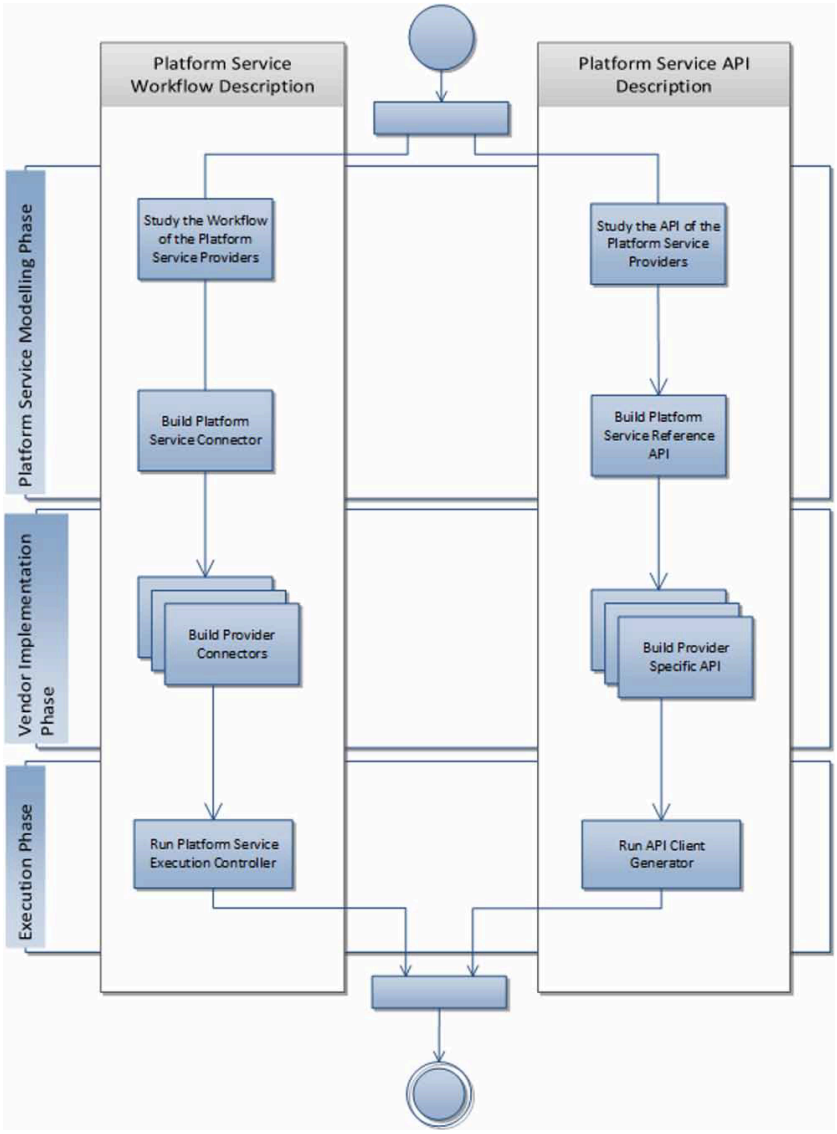
1. **Platform Service Modelling Phase:** As mentioned earlier, the framework shall be capable of addressing the variability in the provider specific web APIs by enabling the definition of a reference API. One reference API is defined, by the administrator of the framework, for each type of platform basic service which is supported by the framework. It contains the set of operations offered by the specific service.
    a. Service API Description Editor. The Service API Description Editor is used to define the reference API. It is implemented as Eclipse plug-in and includes a user interface which is used by the Administrator of the framework.
    b. Platform Service Reference API. The Platform Service Reference API is constructed, by the administrator, for each platform service and is accessible by the consumer of the framework. Its role is to remove the barrier from the consumer to study the various service providers API. Instead the consumer accesses all the supported providers via the Reference API.
    c. Template API Repository. The template API repository contains the collection of the platform service reference APIs which have been defined using the Service API description editor.
2. **Vendor Implementation Phase:** During this phase the specific web API of each of the platform service providers supported by the framework is described and mapped to the Reference API.
    a. Provider Specific API. The Provider Specific API holds the description of the concrete service provider API and the subsequent mapping to the Platform Service Reference API.

3. **Execution Phase:** During the Execution Phase the web clients required for the application to connect to the concrete service providers, are generated. The web clients are source code, which implement the HTTP requests –responses. Additionally, the services that are consumed by the applications are registered.

    a. API Client Generator. The API Client Generator, as the name implies, is responsible for the generation of the web clients for each concrete service provider. It receives the Platform Service Reference API and the Provider Specific API and produces a java library which can be used by the consumer in order to connect to the concrete service provider.

    b. Platform Service Registry: This component is a registry of all the platform services that the application uses. Its role is to keep track of the consumed services and to provide an easy way for the software developer to deploy and release services.

Figure 4. Activity Diagram of the Development Framework



TECHNICAL DESIGN

In this Section the technical design of the development framework is described. Particularly, the Section analyses each of the components mentioned in the High-Level architecture and explains its contribution to the framework. For that reason the narration flow used in the previous Section is followed. The overall process of adding a new platform basic service and service provider using the framework is divided in two parts: *(i) Platform Service Workflow Modelling, and (ii) Platform Service API Description.* In each part the following phases are involved: (a) *Platform service modelling phase, (b) Vendor implementation phase, (c) Execution phase.* The activities, which are required in each phase, are shown in Figure 4.

Action 1 involves the study of the concrete payment service providers and the extrapolation of the common states in which they may co-exist. For that reason 9 major payment service providers have been studied (Gonidis, 2013), provisioned either via a major cloud platform such as Google App Engine and Amazon AWS or via platform service marketplaces such as Heroku add-ons and Engineyard add-ons. These providers can be grouped into three main categories. An exhaustive listing of the characteristics of each payment provider is out of the scope of this article. Rather, the article focuses on demonstrating how concrete providers can be mapped on the abstract model. Therefore, the case of one category is presented, the "transparent redirect". Spreedly (https://spreedly.com/), a payment provider offered via Heroku platform, is used as the concrete service provider.

Transparent redirect is a technique deployed by certain payment providers in which, during a purchase transaction, the client's card details are redirected to the provider who consequently notifies the cloud application about the outcome of the transaction.

Figure 5. Cloud Payment Service

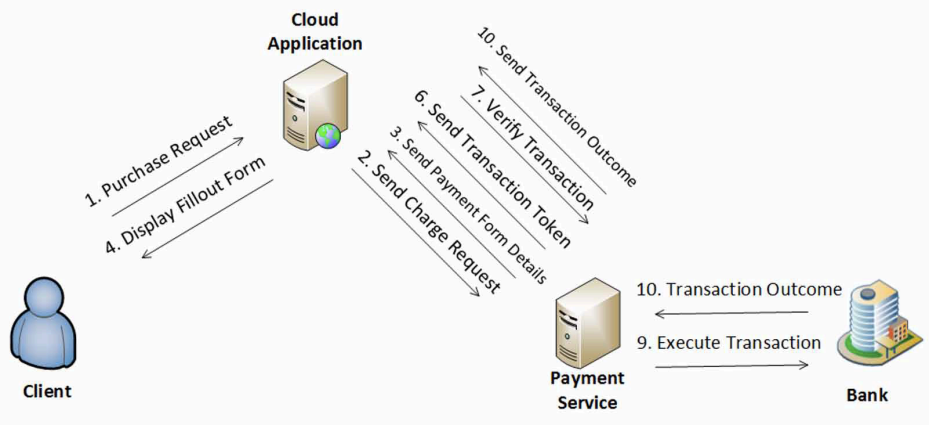
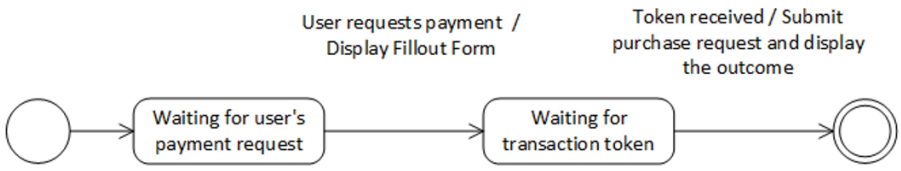


Figure 5 describes the steps involved in completing a payment transaction, while Figure 6 shows the state diagram of the cloud application throughout the transaction. Two states are observed. While the cloud application remains in the first state, it waits for a payment request. Once the client requests a new payment, the cloud application should display the fill out form where the user enters the payment details.

Figure 6. State diagram of the cloud payment service



Subsequently, the cloud application moves to the next state where it waits for the transaction token issued by the payment provider. Once the user submits the form, she is redirected to the payment provider who validates the card details. Then a request to the cloud application is submitted including the transaction token. Once the token is received the application submits a request to the provider with the specific amount to be charged. The provider completes the transaction and responds with the outcome. Depending on the outcome, the cloud application displays a success or failure page to the client.

Next Section describes how the abstract states, depicted in Figure 6, can be modelled utilising the components of the framework and how the concrete provider is mapped to the abstract model.

Platform Service Workflow Modelling

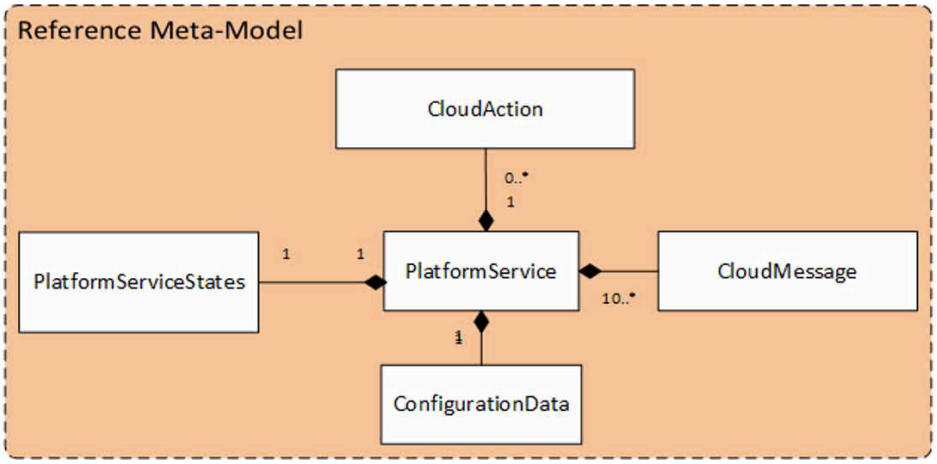
Platform Service Modelling Phase

This phase involves the modelling of the abstract functionality of the platform basic service. Specifically, the states and the workflow required to complete an operation are captured. For that reason, the Reference Meta-Model depicted in Figure 7 is defined.

The Reference Meta-Model consists of the following concepts:

1. **CloudAction:** *CloudActions* are used to model the communication with platform basic services, which require more than one step in order to complete an operation. The whole process required to complete the operation can be modelled as a state machine. Each step in the process can be modelled as a concrete state that the platform service can exist in. For each state a *CloudAction* is defined. When an event arrives the appropriate *CloudAction* is triggered to handle the event and subsequently causes the transition to the next state. The events in this case are the incoming requests arriving either by the application user or the service provider.

Figure 7. Reference Meta-Model



2. **CloudMessage:** *CloudMessages* can be used to perform requests from the cloud application towards the service provider using the web API of the latter. The API usually conforms to the REST principle (Fielding, 2001). *CloudMessages* can either be used in stateless services, where the operation is completed in one step or within *CloudActions* when the latter are required to submit a request to the service provider.

3. **PlatformServiceStates:** The *PlatformServiceStates* is an XML file which holds information about the states involved in an operation and the corresponding *CloudActions* which are initialised to execute the behaviour required in each state.
4. **ConfigurationData:** Certain configuration settings are required by each platform service provider. Example of settings which needs to be defined are the clients' credentials required to perform web requests and authentication tokens.
5. **Platform Service:** As shown from the types of the relationships in Figure 7 the platform service component is composed from all the previously mentioned concepts.

The motivation for the definition of the separate concepts of the *CloudActions* and the *CloudMessages* stems from the basic software design principles of modularisation, separation of concerns and reusability (Hürsch & Lopes, 1995) (Poulin, 1994). Separation of concerns ensures that a software application is composed of distinct units each one addressing a specific issue. In turn software modularisation is enabled which further improves the maintainability of the software. Reusability allows, certain pieces of source code to be reused within the software application improving this way the productivity. In the framework design, *CloudActions* are responsible for defining a template for serving the incoming requests. *CloudMessages* implement a specific web request to the service providers and can be reused by different *CloudActions*.

The reference meta-model is used to construct the Platform Service Connector (PSC). The PSC, as mentioned in the previous Section, is a model which defines the states and the workflow required to complete an operation of a platform service. It is constructed based on two rules. The rules are based on the definition of the *CloudActions* and the *CloudMessages* mentioned earlier where the former are used to handle incoming requests where the latter perform web requests to the service providers.

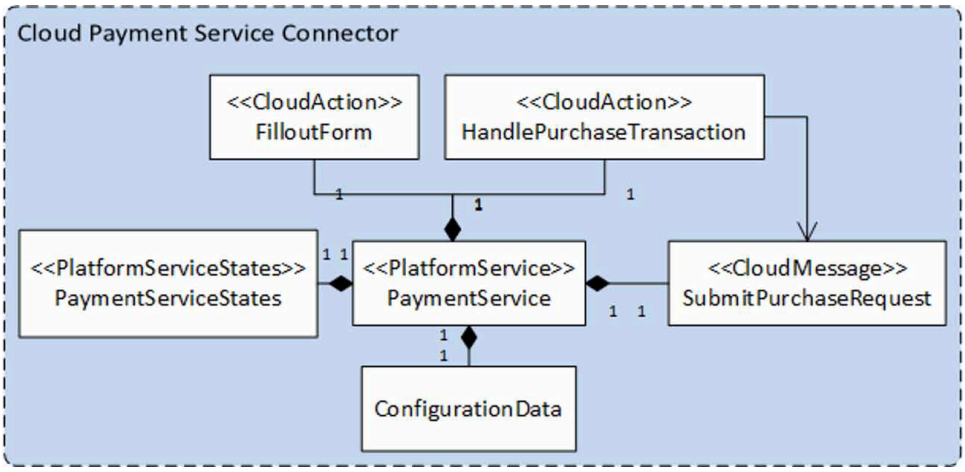
Rule 1: For each state where the application waits for an external request (either from the user of the application or the service provider), a *CloudAction* is defined to handle the request.

Rule 2: For each request initiated by the cloud application towards the service provider, a *CloudMessage* is defined.

In the case of the Cloud Payment Service, Figure 8 shows the Cloud Payment Service Connector. It is constructed based on the state diagram defined in Figure 6 and using the reference meta-model. It consists of the following blocks:

1. **FilloutForm:** The *FilloutForm* is a *CloudAction* which receives the request for a new purchase transaction and responds to the client with the fill out form in order for the latter to enter the card details.
2. **HandlePurchaseTransaction:** The *HandlePurchaseTransaction* is a *CloudAction,* which receives the request from the service provider containing the transaction token. Then, a request is submitted to the provider including the transaction token and the amount to be charged. The provider replies with the outcome of the purchase and subsequently the action responds to the client with a success or fail message accordingly.
3. **SubmitPurchaseRequest:** The *SubmitPurchaseRequest* is a *CloudMessage* used internally by the *HandlePurchaseTransaction* action. Its purpose is to perform the request to the service provider, using the exposed web API, to complete the purchase transaction. It receives the provider's respond stating the outcome and forwards it to the action.
4. **ConfigurationData:** The *ConfigurationData* contains the service settings required to complete the purchase operation.

Figure 8. Cloud Payment Service Connector



5. **PaymentSerivceStates:** In the *PaymentServiceStates* file the states and the corresponding actions involved in the transaction are defined. The file is used by the framework to guide the execution of the actions. A part of the description file is shown below. Two states are defined. For each state the concrete path to the CloudAction responsible for handling the event as well as the next state is described.

```
<StateMachine>
    <State name="PaymentForm"
        action="org.paymentserviceframework.FillOutFormAction"
        nextState="SendTransaction"/>
    <State name="SendTransaction"
        action="org.paymentserviceframework.SendTransactionAction"
        nextState="Finish" />
</StateMachine>
```

6. **PaymentService:** The *PaymentService* is composed of the above mentioned artefacts.

At this point the Cloud Payment Service Connector (PSC) does not contain any provider specific information. Therefore any payment service provider which adheres to the specified model can be accommodated by the abstract model.

Vendor Implementation Phase

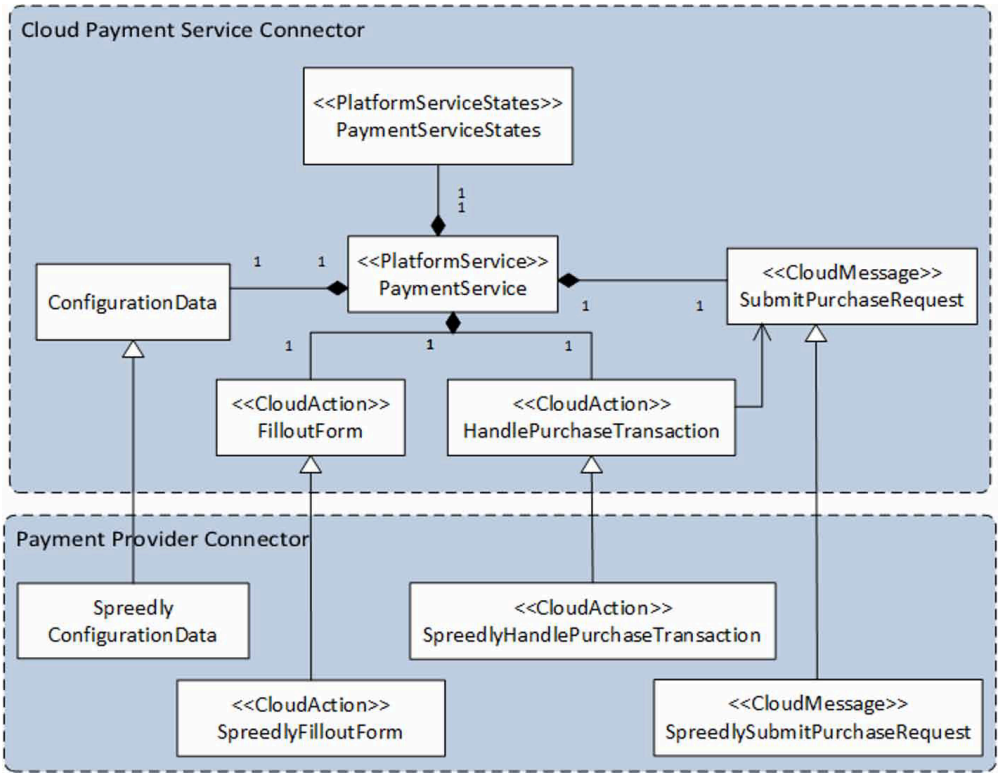
After having defined the PSC, the specific implementation and settings of each concrete provider needs to be infused (Action 3a of Figure 4). For each *CloudAction* and *CloudMessage* defined in the PSC, the respective provider specific blocks should be defined forming the Provider Connector (PC).

In the case of the payment service example, the Cloud Payment Provider Connector for the Spreedly provider is shown in the lower part of the Figure 9. It contains the following blocks:

1. **SpreedlyFilloutForm:** It is a *CloudAction* implementing the *FilloutForm*.
2. **SpreedlyHandlePurchaseTransaction:** It is a *CloudAction* implementing the *HandlePurchaseTransaction*.
3. **SpreedlySubmitPurchaseRequest:** It is a *CloudAction* implementing the *SubmitPurchaseRequest*

4. **ConifgurationData:** This file needs to be updated accordingly in order to match the specific provider.

Figure 9. Cloud Payment Service Provider Connector

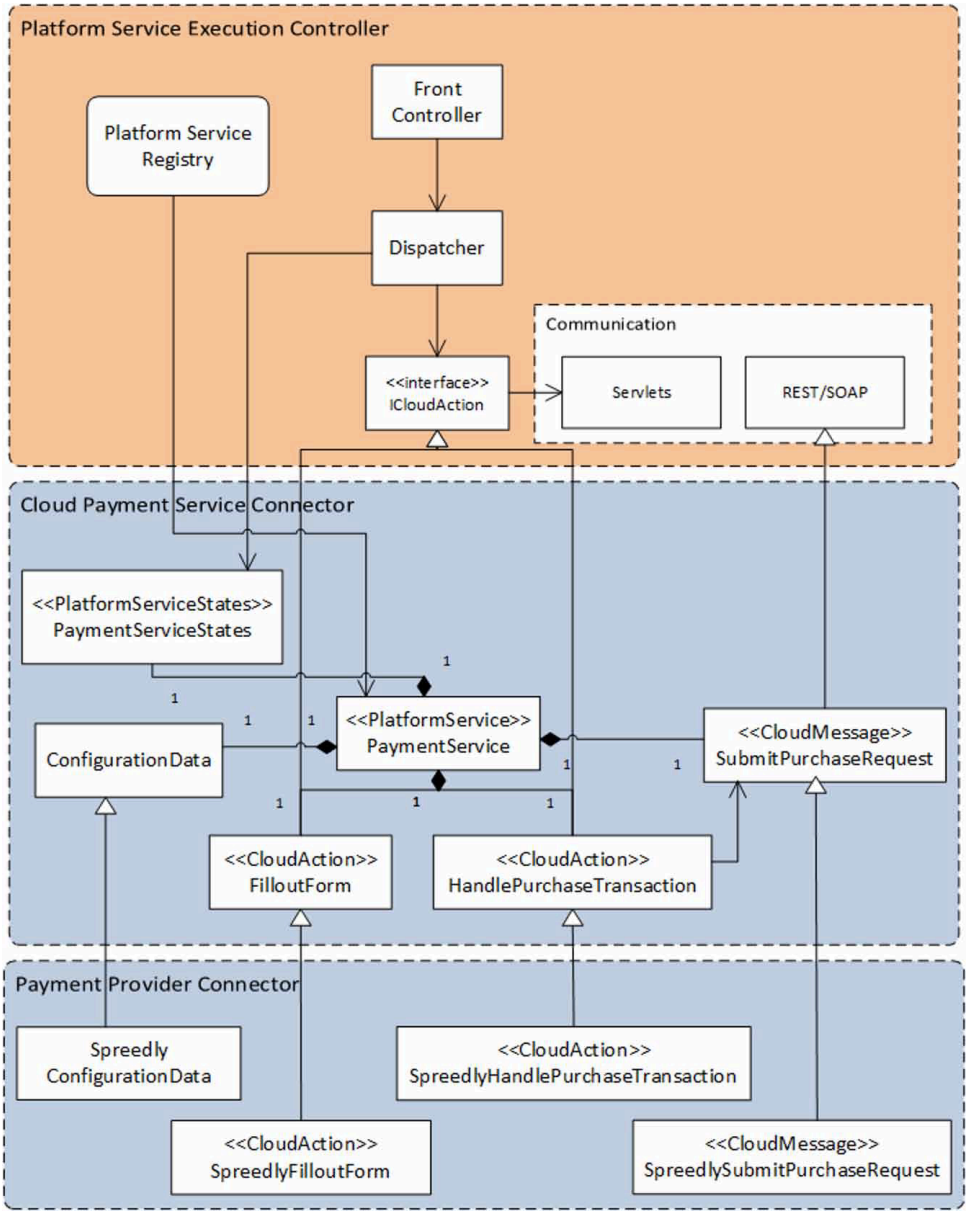


Execution Phase

During the execution phase the PSC and the PC, constructed in the previous phases, are managed by the Platform Service Execution Controller (PSEC) as shown in the Figure 10. The PSEC automates the execution of the workflow required to complete an operation. It consists of the main following components shown in the upper part of the Figure 10.

1. **Front Controller:** The *Front Controller* (Hunter & Crawford, 2001) serves as the entry point to the framework. It receives the incoming requests by the application user and the service provider.
2. **Dispatcher:** The dispatcher (Alur, 2001) follows the well-known request-dispatcher design pattern. It is responsible for receiving the incoming requests from the Front Controller and forwarding them to the appropriate handler, through the I*CloudAction* which is explained below. The requests are handled by the *CloudActions*. Therefore the dispatcher forwards the request to the appropriate *CloudAction*. In order to do so, he gains access to the platform service states description file and based on the current state it triggers the corresponding action.
3. **ICloudAction:** *ICloudAction* is the interface which is present at the framework at design time and which the Dispatcher has knowledge about. Every *CloudAction* implements the *ICloudAction*. That facilitates the initialisation of the new *CloudActions* during run-time through reflection.

Figure 10. Platform Service Execution Controller (PSEC)



4. **Communication patterns:** Two types of communication pattern are supported by the framework: The first one is the Servlets and particularly the Http Servlet Request and Response objects (Hunter & Crawford, 2001) which are used by the *CloudActions* in order to handle incoming requests and respond back to the caller. The second type of communication is via the use of the REST protocol which enables the *CloudMessages* to perform external requests to the service providers.

5. **Platform Service Registry:** The Platform Service Registry, as the name implies, keeps track of the services that the cloud application consumes. Every service which is used by the application is listed in the Platform Service Registry.

Platform Service API Description

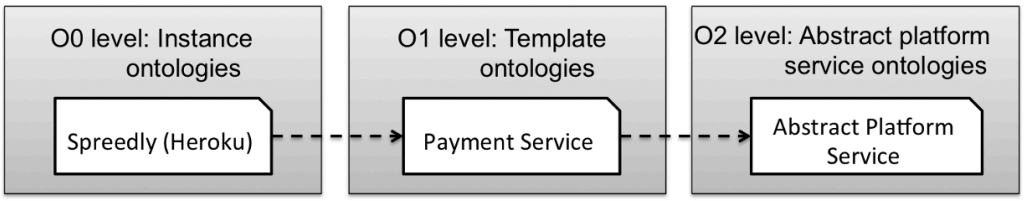
The second part in the process of adding a platform service and providers to the framework constitutes the description of the web API. The second variability point among platform services is the different web APIs that the concrete providers expose. Therefore, the heterogeneity of the web APIs shall be captured by the framework and abstracted by a common reference API exposed to the application developers.

In order to enable the uniform description of the platforms services' API, the benefits of ontologies are exploited. According to Gruber (1993), ontologies are formal knowledge over a shared domain that is standardised or commonly accepted by certain group of people. The advantages here are two-fold. First, ontologies allow to define clearly the domain model of our interest; in our case the domain model is the platform service providers web API. The fact that an ontology can be a shared and a commonly accepted description of a platform service, contributes towards the homogenisation of the latter. The platform vendors can adhere to and publish the description of their service-based on the common and shared ontology.

Moreover, ontologies can be reused and expanded if necessary. Thus, an ontology describing a platform service may not be constructed from the ground up but may be based on an existing one. The intention of the authors is to reuse and expand the Linked USDL (Pedrinaci, Cardoso, & Leidig, 2014) ontology and particularly the extended Minimal Service Model (MSM) as described in the work of Ning et al. (2011). To the best of our knowledge and according to Ning et al. (2011) the MSM is the richest description model capable of capturing the web API and enabling automatic invocation.

The platform service API description is based on an hierarchy of a three level ontologies as shown in Figure 11. Inspiration has been gained by the Meta-Object-Facility (MOF) standard (Gardner, Griffin, Koehler, & Hauser, 2003) defined for the Model Driven Engineering domain. Specifically, the hierarchy of the ontologies resembles the bottom three levels of the MOF structure, namely the meta-models, the models and the instances of the models.

Figure 11. The three levels of the ontology hierarchy



The level 2 Ontology (O2) includes the concepts required to describe a web API. Such concepts are the operations offered by the service providers, the parameters and the endpoint for each operation etc. The level 1 Ontologies (O1) include the concrete description of each of the platform services which are supported by the framework. A dedicated ontology corresponds to each of the platform services and captures information about the functionality that each of the services expose. The ontologies in the O2 level are also referred to as Template ontologies. The level 0 Ontologies (O0) include the description of the specific platform service providers. A dedicated ontology corresponds to each of the providers and describes the native web API. The ontologies in the O0 level are also referred to as Instance ontologies.

During the three phases the Section describes how the ontological service descriptions are formed and used to automatically generate the web clients.

Platform Service Modelling Phase

During this phase, the platform service reference API, is defined. The reference API is exposed to the application developers and describes the operations offered by the particular service. The reference API is captured in the Template Ontology.

Figure 12. Example of Template ontology for the cloud payment service

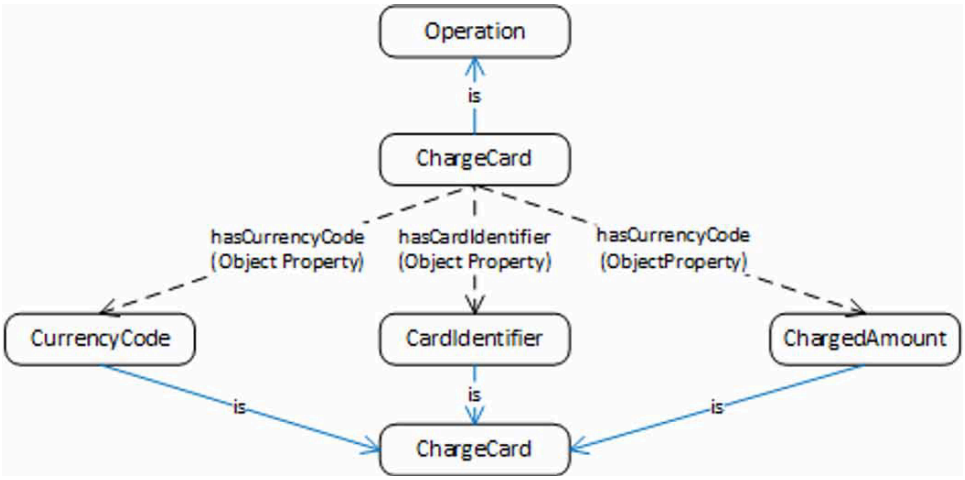


Figure 12 shows a snapshot of the Template ontology for the payment service which describes the operation for charging a card. For the sake of simplicity only the necessary amount of information has been included. The name of the operation is "ChargeCard". It is a subclass of the class "Operation". "Operation" is defined in the Abstract platform service ontology (O2 level) and includes all the operations offered by the service. Figure 12 also includes the following three elements: "CardIdentifier",which denotes the card to be charged, "ChargedAmount", which refers to the amount of money to be charged during the specific transaction and "CurrencyCode" which refers to the currency to be used for the specific transaction. All three elements are subclasses of the class "Attribute". The class "Attribute" is defined in the Abstract platform service ontology and includes all the attributes which are used for the execution of the operations. An attribute is linked to a specific operation with a property. Specifically, the three afore mentioned attributes are linked to the "ChargeCard" operation with the following properties respectively: "hasCardIdentifier", "hasChargedAmount", "hasCurrencyCode".

Vendor Implementation Phase

In this phase the provider specific web API is described and mapped to the reference API. The Service API description editor is used to perform the mapping. The outcome is an Instance ontology (O0 level) for each concrete provider.

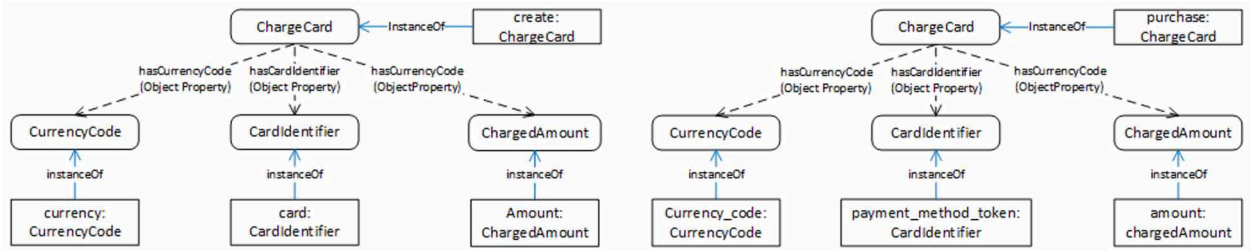
Figure 13 depicts two Instance Ontologies, which correspond to two payment service providers offered by Heroku and Amazon respectively.

Particularly, Figure. 13 shows the description of the charge operation as defined in the API of the Spreedly service offered via Heroku platform. Individuals are created to express each of the specific elements of the provider`s API. An Individual, in the field of Ontologies can be considered as an instance of a class. Specifically, the "purchase" Individual denotes the operation name which is equivalent to the "ChargeCard" operation of the Template Ontology. This justifies the fact that "purchase" individual is of type "ChargeCard". The Individual "amount" denotes the amount to be charged during the transaction and is equivalent to the "ChargedAmount" attribute. Thus it is defined of type "ChargedAmount".

Likewise the Individual “currency_code” is of type “currency” and the “payment_method_token”, which identifies the card to be charged, is of type “CardIdentifier”.

In the same way an Instance Ontology is created (Figure 13) to describe the API of the “Stripe” payment service provider offered via Amazon. The individual “create” denotes the creation of a charge and is equivalent to the “ChargeCard”. Therefore it is of type “ChargeCard”. Likewise, the individuals “amount”, ”currency” and “card” are of type “ChargedAmount”, “CurrencyCode” and “CardIdentifier” respectively.

Figure 13. Example of Instance ontology for the Stripe (left) and Spreedly (right) service provider



In the same way the rest of the functionality of a platform service can be described. At the same time, the differences in the APIs between the various providers can be captured. The proposed structure of the three levels of ontologies can be used to describe the web API of additional platform services such as authentication and message queue service. Initially, a Template ontology is formed to describe the functionality of each of the platform services. Consequently the Instance ontologies are created to capture the vendor specific web APIs.

Execution Phase

During the Execution Phase, the Platform Service Reference and the provider specific API descriptions, which correspond to the Template and the Instance Ontologies respectively, are fed to the API Client Generator (Figure 14). The API Client Generator generates the client code for the web API invocation of each of the concrete providers which implement the platform service.

The process of the code generation is depicted in Figure 14. The API client generator accepts as input the following:

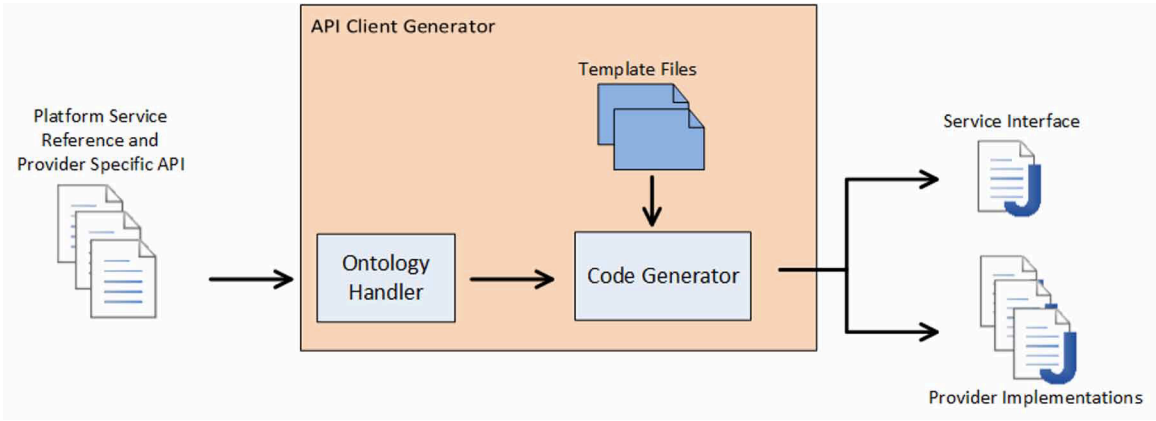
1. The Template and Instance ontologies, which contain the description of the reference API and the mapping of the providers’ specific API to the reference one.
2. The Template Files. These files contain the source code which is common among the generated classes, also known as boilerplate code.

The Ontology Handler parses the ontologies, creates an object representation and forwards it to the code generator. The code generator reads the Template files and fills in the missing information according to the service description obtained by the Ontology Handler. Subsequently, the following Java Classes are generated:

1. A set of Java Interfaces which give access to the platform basic services. One Interface is generated for each service supported by the framework. It contains the operations provided by the services and the reference API as described in the Service Description File.
2. A set of Java Classes which give access to the provider implementations. For each concrete service provider which is supported by the framework a Java Class is generated which implements the service Interface. It essentially includes the provider`s information (URL,

credentials, configuration settings) and the concrete parameters as those are specified in the web API.

Figure 14. Code Generation Process



Therefore, the software engineers can develop their applications against the Service interfaces, which are produced by the API Client Generator and gain access to the supported providers without having to know the underlying specific implementations.

FUTURE RESEARCH DIRECTIONS

As the field of the cloud application platforms and platform basic services gains momentum, the demand for efficient frameworks and methodologies for the design of service-based cloud applications rises. This article proposed a prototype implementation of such a development framework.

The main limitation of the framework is that it is inherently restricted to the abstraction of the common features of the service providers. This means that the reference API contains the operations, which are collectively offered by the supported providers. This is a natural limitation when dealing with API abstraction that is also encountered by similar solutions such as the jClouds, mOSAIC and TOSCA, which are involved with cloud services API abstractions. One solution is to provide the application developers with direct access to the client adapters for the specific provider when they need to use provider specific functionality, which is not addressed by the reference API. In addition, the reference API rather than being static can be continuously updated to reflect the new features offered by the platform service providers. An alternative solution suggests that while developers use the reference API for the parameters that are common among the providers, a secondary mechanism is deployed to enable provider specific parameters to be included in the web requests. The mechanism may involve a call-back operation during which the framework, depending on the concrete deployed provider, requests from the client the additional parameters to be included in the request.

As mentioned earlier, there is a lack of Integrated Development Environments (IDE) to assist in and automate part of the design process of service-based cloud application. Towards this direction, an Eclipse-based plugin editor has been constructed to allow users to define the mappings between the reference and the specific providers' API and drive the code generation process. In the future the editor will be enriched to enable the definition of *CloudActions* and *CloudMessage* and to serve as the main point for the integration of platform services with the application and the management of the already deployed ones.

Future work also involves the expansion of the framework so that it offers functionality for automatic discovery and recommendation of services. Furthermore, it can provide billing information about the incurring costs of the application with respect to the platform basic services that it consumes. Integration

with tools associated with Big Data analysis techniques will also be considered. The long term vision of the authors is to create a development environment which assists the software engineers throughout the whole process of designing, implementing and managing a service-based cloud application.

CONCLUSION

The unprecedented rate in which data are generated nowadays and their subsequent analysis leads to new business opportunities for the enterprises. This fact together with the emergence of service-based cloud applications paves the way for context-aware applications, which are able to adapt themselves according to analysis of the collected data.

This article presented a development framework enabling the design of service-based cloud applications. Particularly, the framework facilitates the integration of platform basic services in a consistent way as well as seamless deployment of the concrete providers implementing those services. It achieves this by alleviating the variability issues that may arise across the platform services, namely: (i) the differences in the workflow when executing an operation, (ii) the heterogeneous web API exposed by the providers and (iii) the various configuration settings and authentication tokens that each provider requires. The main components of the framework are: (i) the reference meta-model, which enables the modelling of the abstract functionality of the platform basic service and an ontology-based architecture for alleviating the differences between the Providers' web APIs and automatically generating the client adapters for the API invocation.

The proposed framework puts forward an approach of developing service-based cloud applications, which are not tightly coupled with specific providers. The software engineers should focus on the services required by the application rather than on the implementers. The selection of the concrete service provider can be a secondary automated process, which is based on external stimuli as explained in the case of the payment service. The collection and analysis of Big Data is able to drive the selection process and pave the way for the design of adaptable and context-aware service-based cloud applications.

REFERENCES

Alur, D., Crupi, J., & Malks, D. (2001). *Core J2EE Patterns: Best Practices and Design Pattern Strategies*. Santa Clara, CA: Prentice Hall / Sun Microsystems Press.

Ardagna, D., Di Nitto, E., Casale, G., Petcu, D., Mohagheghi, P., Mosser, S., Matthews, P., Gericke, A., Ballagny, C., D'Andria, F., Nechifor, C. S., & Sheridan, C. (2012). MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds. In Atlee, J., Baillargeon, R., France, R., Georg, G., Moreira, A., Rumpe, B., Zschaler, S. (Eds), *Proceedings of the 4th International Workshop on Modeling in Software Engineering.* Zurich, Switzerland: IEEE.

Armbrust, M., Fox, A., Griffith, R., Joseph, D. A., Katz, R., Konwinski, A., … Zaharia, M. (2010). A view of cloud computing. *Communications of the ACM,* 53(4), 50-58.

Berson, A., (1992). *Client - server architecture*. New York, NY: McGraw-Hill.

Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F. & Nowak, A. (2013). OpenTOSCA – A runtime for TOSCA-based cloud applications. In Basu, S., Pautasso, C., Zhang, L., Fu, X. (Eds.), *Proceedings of the 11th International Conference on Service Oriented Computing.* Berlin, Germany: Springer Berlin Heidelberg.

Binz, T., Breiter, G., Leymann, F., & Spatzier, T. (2012). Portable cloud services using TOSCA. *IEEE Internet Computing,* 16(3), 80–85.

Cusumano, M. (2010). Cloud Computing and SaaS as new computing platforms. *Communications of the ACM*, 53(4), 27-29.

Demchenko, Y., Grosso, P., de Laat, C., & Membrey, P. (2013). Addressing big data issues in scientific data infrastructure. In *Proceedings of the 2013 International Conference on Collaboration Technologies and Systems.* San Diego, CA: IEEE.

Distributed Management Task Force, Inc. (2012). *Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol.*

Distributed Management Task Force, Inc. (2013). *Open Virtualization Format Specification.*

Ferry, N., Chauvel, F., Rossini, A., Morin, B., & Solberg, A. (2013). Managing multi-cloud systems with CloudMF. In Solberg, A., Babar, A., M., Dumas, M., Cuesta, E., C. (Eds), *Proceedings of the 2nd Nordic Symposium on Cloud Computing & Internet Technologies* .Oslo, Norway: ACM.

R. Fielding, (2000). *Architectural styles and the design of network-based software architectures* (Doctoral dissertation). Available from University of California, Irvine.

Fielding, R., Irvine, UC, Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). Hypertext Transfer Protocol -- HTTP/1.1. Network Working Group.

Forrester Research Inc. (2012a). *US mobile payments forecast, 2012 To 2017.* Cambridge, MA: Carrington, D., Epps, R. S., Doty, C., A., Wu, S., & Cambell, C.

Forrester Research Inc. (2012b). *EU mobile commerce forecast 2023 to 2017.* Cambridge, MA: Gill, M., Poltermann, S., Husson, T., O'Grady, M., Evans, P. F., & Da Costa, M.

Forrester Research Inc. (2013). *US mobile retail forecast 2013 to 2017.* Cambridge, MA: Mulpuru, S., Evans, P., F., Roberge, D., & Johnson, M.

Gardner, T., Griffin, C., Koehler, J., & Hauser, R. (2003). *A review of OMG MOF 2.0 Query / Views / Transformations submissions and recommendations towards the final standard*. Paper presented at the Workshop on Metamodeling for MDA, York, UK.

Gartner Inc. (2012). Forecast: Mobile payment, worldwide, 2009-2016. Stamford, CT: Shen, S.

Gonidis, F., Paraskakis, I., Simons, A. J. H., & Kourtesis, D. (2013). Cloud application portability. An initial view. In Georgiadis, K. C., Kefalas, P., Stamaris, D. (Eds), *Proceedings of the 6th Balkan Conference in Informatics*. Thessaloniki, Greece: ACM.

Gonidis, F. (2013). Experimentation and categorisation of cloud application platform services. South East European Research Centre. Thessaloniki, Greece.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2), 199–220.

Guillén, J., Miranda, J., Murillo, J. M., & Cana, C. (2013a). A service-oriented framework for developing cross cloud migratable software. *Journal of Systems and Software,* 86(9), 2294-2308.

Guillen J., Miranda, J., Murillo, M. J., & Canal, C. (2013b). Developing migratable multicloud applications based on MDE and adaptation techniques. In Solberg, A., Babar, A., M., Dumas, M. & Cuesta, E., C. (Eds) *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies,* Oslo, Norway: ACM.

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*. Noordwijkerhout, The Netherlands: SciTePress.

Hunter, J., Crawford, W. (2001). *Java Servlet Programming*. Sebastopol, CA:O'Reilly & Associates, Inc.

Jacobs, A. (2009). The pathologies of Big Data. *Communications of the ACM,* 52(8), 36-44.McAfee, A., & Brynjolfsson E. (2012). Big data: The management revolution. *Harvard business review,* 90(10), 60-69.

Jeffery, K., Horn, G., & Schubert, L. (2013). A vision for better cloud applications. In Ardagna, E., Schibert, L. (Eds), *Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds*. Prague, Czech Republic: ACM.

Kourtesis, D., Bratanis, K., Bibikas, D., & Paraskakis, I. (2012). Software co-development in the era of cloud application platforms and ecosystems: The Case of CAST. In Camarinha-Matos, L. M., Xu, L., Afsarmanesh, H. (Eds), *Proceedings of the 13th IFIP WG 5.5 Working Conference on Virtual Enterprises*. Bournemouth, UK: Springer Berlin Heidelberg.

Ning, L., Pedrinaci, C., Maleshkova, M., Kopecky, J., & Domingue, J. (2011). OmniVoke: A framework for automating the invocation of web APIs. In O'Conner, L. (Ed), *Proceedings of the Fifth IEEE International Conference on Semantic Computing.* Palo Alto, CA: IEEE.

Opara-Martins, J., Sahandi, R., & Tian, F. (2014). Critical review of vendor lock-in and its impact on adoption of cloud computing. In *Proceedings of the International Conference on Information Society (i-Society 2014).* Bournemouth, UK: IEEE.

Pautasso, S., Zimmermann, O., & Leymann, F. (2008). Restful web services vs. "big"' web services: making the right architectural decision. In Huai, J., Chen, R. (Eds), *Proceedings of the 17th International Conference on World Wide Web*. Beijing,China:ACM.

Pedrinaci, C., Cardoso, J., & Leidig, T. (2014). Linked USDL: A vocabulary for web-scale service trading. In Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, & S., Tordai, A. (Eds), *The Semantic Web: Trends and Challenges: Proceedings of the 11th Extended Semantic Web Conference* (LNCS) (Vol. 8465, pp. 68-82). Crete, Greece: Springer International Publishing.

Petcu, D. (2014). Consuming Resources and Services from Multiple Clouds. *Journal of Grid Computing,* 12(2), 321-345.

Petcu, D., & Vasilakos, V. (2014). Portability in clouds: Approaches and research opportunities. *Scientific International Journal for Parallel and Distributed Computing,* 15(3), 251-270.

Ranabahu, A., Maximilien, E., Sheth, A., & Thirunarayan, K. (2013). Application portability in cloud computing: An abstraction driven perspective. *IEEE Transactions on Services Computing,* PP(99), 1-1.

Rimal, B. P., Jukan, A., Katsaros, D., & Goeleven, Y. (2010). Architectural Requirements for Cloud Computing Systems: An Enterprise Cloud Approach. *Journal of Grid Computing,* 9(1), 3-26.

Singhal, M., Chandrasekhar, S., Ge, T., Sandhu, R., Krishnan, R., Ahn, G. J., & Bertino, E. (2013). Collaboration in Multicloud Computing Environments: Framework and Security Issues. *IEEE Transactions on Cloud Computing,* 46(2), 76-84.

Storage Networking Industry Association. (2014). *Cloud Data Management Interface (CDMI)*.

The Data Warehouse Institute. (2011). *Big Data analytics, TDWI best practices report.* Renton, WA: Russom, P.