The University Of Sheffield.

A Framework Enabling the Cross-Platform Development of Service-based Cloud Applications

By:

Fotis Gonidis

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

The University of Sheffield
Faculty of Engineering
Department of Computer Science

December 2015

South East European Research Centre

The University of Sheffield
Faculty of Engineering
Department of Computer Science

A Framework Enabling the Cross-Platform Development of Service-based Cloud Applications

By:
Fotis Gonidis

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

December 2015

South East European Research Centre

Abstract

Among all the different kinds of service offering available in the cloud, ranging from compute, storage and networking infrastructure to integrated platforms and software services, one of the more interesting is the *cloud application platform*, a kind of *platform as a service (PaaS)* which integrates *cloud applications* with a collection of *platform basic services*. This kind of platform is neither so open that it requires every application to be developed from scratch, nor so closed that it only offers services from a pre-designed toolbox. Instead, it supports the creation of novel *service-based applications*, consisting of *micro-services* supplied by multiple third-party providers. Software service development at this granularity has the greatest prospect for bringing about the future software service ecosystem envisaged for the cloud.

Cloud application developers face several challenges when seeking to integrate the different *micro-service* offerings from third-party providers. There are many alternative offerings for each kind of service, such as mail, payment or image processing services, and each assumes a slightly different business model. We characterise these differences in terms of (i) workflow, (ii) exposed *APIs* and (iii) configuration settings. Furthermore, developers need to access the *platform basic services* in a consistent way. To address this, we present a novel design methodology for creating *service-based applications*. The methodology is exemplified in a Java framework, which (i) integrates *platform basic services* in a seamless way and (ii) alleviates the heterogeneity of third-party services. The benefit is that designers of complete *service-based applications* are no longer locked into the vendor-specific vagaries of third-party *micro-services* and may design applications in a vendor-agnostic way, leaving open the possibility of future *micro-service* substitution.

The framework architecture is presented in three phases. The first describes the abstraction of *platform basic services* and third-party *micro-service* workflows,. The second describes the method for extending the framework for each alternative *micro-service* implementation, with examples. The third describes how the framework executes each workflow and generates suitable client adaptors for the web *APIs* of each *micro-service*.

Acknowledgements

Carrying out a research work and writing the PhD thesis may become a lifelong experience, which to a certain extend has the potential to shape the way you think, react and face the challenges in your life. Above all though, it is a journey through uncharted routes to a destination, which is not clear and by no means visible when you set off. You have a direction but you are not aware of where exactly you will end up and which path you need to follow to reach your goal. And it is this factor of the “unknown” that makes this journey so unique, full of surprises, mystery and unexpected encounters.

Actually, when you start your PhD there is one thing known and certain, that is the emotional fluctuations. There are days, when you are excited and full of motivation, followed by days of disappointment and frustration. There are moments you feel that you found your way to your destination followed by ones that you loose it again. Nevertheless, when you reach the end of your journey, there is only one feeling left, that is fulfilment. You feel that you have fulfilled the promise to yourself, which is to accomplish the assignment you undertook when you started this journey. And it is this feeling, which gives value to all the good and bad moments you have been through.

Carrying out a research work and writing the PhD thesis is primarily a lonely process. Nevertheless, it would be unfair if I didn’t mention that throughout my personal journey there were several people who were always standing next to me and supporting me with their own unique way. And it is thanks to those people that I am in position now to present this thesis. However, the gratitude I feel for those people cannot fit within the limits of this section. Therefore, I would prefer to express my deep gratitude personally and with a special way to each one of those persons.

ΕΛΛΗΝΕΣ ΦΑΣΙ ΠΡΩΤΑΓΟΡΟΥ ΠΡΟΚΑΤΑΡΞΑΝΤΟΣ

ΠΑΝΤΙ ΛΟΓΩ ΛΟΓΟΝ ΙΣΟΝ ΑΝΤΙΚΕΙΣΘΑΙ

Greeks, with Protagoras (c. 485 – c. 415 B.C.) being the first,
say that to every thesis an equal thesis is opposed

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Since the early years of electronic computers and modern computing in the middle of the 20th century [1] several paradigms shifts have taken place to shape the field of software application development and empower developers to produce code faster and better (Figure 1). During the 1960s, software was characterised by complex and tangled control structure, which was mainly a result of the use of "GO TO" statements [2]. As Dijkstra specifically points out, the rampant use of the "GO TO" statements made it extremely complicated for programmers to follow the execution flow of the program and thus understand and maintain the software [3]. This style of programming has been widely known as "spaghetti code" [2].

In response to the need for structured and less complex code, structured and procedural programming appeared in the late 1960s and during the 1970s. These paradigms eliminated the "GO TO" statements and made the code more readable [4]. At the same time software engineering principles were introduced and adopted during software development, such as reusability, the separation of concerns, and modularity [2]. Modularity imposes that the computer programs are built from distinct units of code called modules. Each module implements a concrete task and exposes a well-defined interface via which it communicates with the rest of the computer program. Modularisation reduced development time since separate groups could work in parallel on each module and improved the maintainability of the software by enabling developers to better understand the software systems [5].

Nevertheless, software continued to grow in size and to become more complex. Therefore, alternative development paradigms gained popularity to cope with this complexity and attempted to shift the software development from statement-oriented coding to system building by connecting various components [6]. In other words, as noted by Nierstrasz et al. [7], "*applications can be largely constructed rather than programmed*". This paradigm, which was known since the late 1960s [8], is component-based software development. Components are highly reusable units of software functionality and can act as the building blocks of software systems [9]. At the same time the use of *global variables* is discouraged. Programmers could create better applications by reusing tried-and-tested software components.

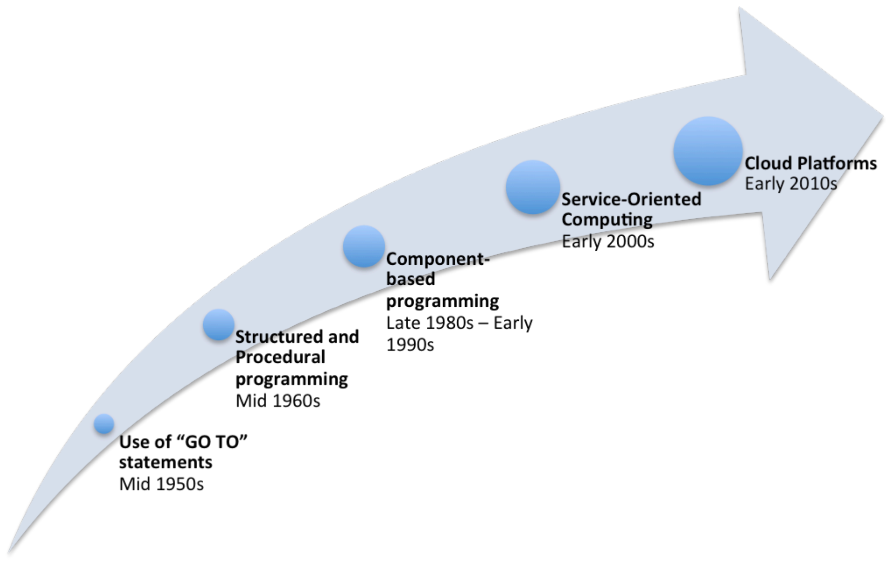

Use of "GO TO" statements
Mid 1950s
Structured and Procedural programming
Mid 1960s
Component-based programming
Late 1980s – Early 1990s
Service-Oriented Computing
Early 2000s
Cloud Platforms
Early 2010s

Figure 1: Representative examples in the evolution of the programming style

From around 2000, the widespread use of Internet protocols such as the *Hypertext Transfer Protocol (HTTP)* [10] and generic data transfer formats, such as the *Extensible Mark-up Language (XML)* [11], paved the way for *Service-Oriented Computing (SOC)* [12]. This paradigm shift decouples component-based software systems into distributed collections of services. The service, which is the core concept of the *SOC*, is considered to be an autonomous, reusable, and portable unit of software, accessed through a standard web *Application Programming Interface* (*API*). Applications could now be built rapidly out of collections of services, which were offered by various providers.

In recent years a new computing paradigm has emerged and attempts to shape the way software applications are developed, namely cloud computing. This paradigm shift has decoupled software services from the platform and hardware resources on which they run, creating a highly distributed web of services, platform, and infrastructure. The continuous emergence of new computing paradigms is evidence of human ingenuity in attempting to satisfy the demand for increasingly complex software systems, for as Wirth says [4]:

"Our society depends to an ever increasing degree on computing techniques."

Keeping this in mind, and having as motivation our continuous striving to empower developers to produce code faster and better, this thesis looks to exploit cloud computing, and *cloud platforms* in particular, to enhance the software development process. Specifically, it proposes a development framework, which will enable the developers to create uniform software applications out of distributed, heterogeneous software services, independently of the concrete provider`s implementation.

The following paragraphs introduce the reader to the field of cloud computing, and *cloud platforms* and emphasise their role in the development of software applications.

Cloud computing refers to a "virtually infinite" number of IT resources, which can be provisioned on-demand automatically, a feature originally introduced by Grid Computing, are charged on a pay-per-use basis, and can be scaled elastically according to the demand [13]. Application developers leverage the development resources provisioned by cloud providers in order to build software applications rapidly and with low-cost. Furthermore, *Independent Software Vendors* (*ISVs*) see an opportunity to develop and launch their software products through major cloud providers, such as *Amazon* [14] and *Google* [15] and thus reach a large number of potential customers [16]. Large organisations and companies exploit cloud computing offerings in order to minimise their IT infrastructure expenses. Instead of owning and maintaining an underutilised data centre, they can lease and be charged only for the resources they actually use. Likewise, small and medium enterprises can reduce drastically their upfront capital costs [17]. End users can access their data and the applications online from anywhere around the world by using web-based

Customer Relationship Systems (CRM) and *Enterprise Resource Planning (ERP)* systems.

Due to the wide range of capabilities involved, which affect a broad target group ranging from large companies to individual users, cloud computing is further divided into three service models [18]:

- **Software as a Service (SaaS):** *SaaS* provides complete software applications such as *CRM* and office solutions for companies and end users who are not necessarily involved in the IT industry.
- **Platform as a Service (PaaS):** *PaaS* offers programming resources such as web servers, run-time containers, and databases and aims at the software developers who seek to reduce the time and effort needed to develop applications.
- **Infrastructure as a Service (IaaS):** *IaaS* refers to low-level hardware resources such as compute, storage, and network resources and mainly targets large organisations and companies.

Due to the wide range of advantages that cloud computing offers to its end users, spanning the whole IT industry, its popularity has grown continuously, and it has already reached a certain maturity [19] and mainstream adoption [20]. Particularly, *Gartner* [21], a leading research and market analysis company, names cloud computing as one of the top 10 strategic technology trends for 2015 [22].

PaaS has received particular attention, due its great appeal to software developers who attempt to harness the offered benefits throughout the whole lifecycle of the application development. The research firm *IDC* [23] estimated a compound annual growth rate of 30% for the *cloud platform* offerings which will reach $22 billion by 2019 [24].

In its basic model a *cloud platform* provides developers with the programming resources required to build and deploy software applications, also referred to as *cloud applications*. Such resources include several programming languages, such as *Java* and *PHP*, databases such as *MySQL* [25], and a selection of web servers, such as *Apache Tomcat* [26] and *Microsoft IIS* [27]. A detailed description of *cloud platforms* and *cloud applications* is provided in the Section 2.3 and 2.4 of the next Chapter.

Due to the popularity of *cloud platforms,* new offerings have been constantly launched, providing additional capabilities such as monitoring, logging and reporting tools.

Apart from the resources offered to assist software engineers in the application development process, certain *cloud platform* providers offer additional functionality, which can be directly integrated into the application. Example of such *cloud platforms* are *Heroku* [28], *OpenShift* [29] and *Engine Yard* [30] . The functionality is provided in the form of many independently-functioning software components which can be accessed by the application using the *Representational State Transfer protocol (REST)* [31].

The additional functionality, which is provided by certain cloud platforms, is referred to in this thesis as a *platform basic service.* Examples of *platform basic services* are the e-mail service, the authentication service, and the payment service. Instead of building applications that offer similar services from scratch, software developers can leverage these *platform basic services* in order to reduce the development effort and time. *Cloud applications* that are based on *platform-basic services* will, in this thesis, be referred to as *service-based cloud applications.* The particular category of platforms which offer the *platform basic services* is also known as *cloud application platforms (CAP)* [32] and this term is adopted for the rest of this thesis[1].

In parallel with the proliferation of the cloud *application platforms* and the *platform basic services* another software development approach has emerged and gains momentum, namely the *micro-services* [33]. The *micro-services* are defined as small and independently deployed services that work together [33]. Their key characteristic is that the communication between the services is achieved via network calls rather than library calls, via an exposed *API.*

Thus, software applications can be built as a combination of collaborating *micro-services*. The benefit of such an approach is that the application can leverage various technologies. Each service can be built using the appropriate technology and tools

[1] The terms *platform basic service, cloud application platform* and *service-based cloud application are further clarified in the Sections 3.6 of Chapter 3.*

without affecting the rest of the application. Furthermore, the application becomes more resilient since if one *micro-service* fails it can be replaced without cascading the failure to other components. Likewise, scaling and deployment can be managed more efficiently since they are restricted to particular *micro-services.* On the other hand *micro-services* may cause efficiency issues especially across the interfaces using network protocols.

A closer consideration of the *micro-service* architecture and the *CAPs* may reveal a strong connection between the two fields. Particularly, the notion of the *platform basic services,* which are offered via the *CAPs,* can be correlated with the concept of *micro-services.* As it has already been described both concepts refer to autonomous pieces of reusable software, which expose certain functionality via an API, and can be accessed through networks calls. As a result, the *service-based cloud applications,* which refer to applications built on *CAPs* and using a number of *platform basic services* can be considered as a realisation of the *micro-services* software development paradigm.

1.1 Motivation for the research work

The growing appeal of *platform basic services* to software developers becomes evident from the ever-increasing number of such services being offered by *cloud application platforms*. Indicatively, it is mentioned, that *Heroku,* one of the leading *cloud application platforms,* now counts almost 150 *platform basic services* [34]. These services may be provided either natively by the owners of the platform, or through the agency of *ISVs*, whose products are hosted by the platform. An increasing number of providers have led to the outcome that multiple providers offer the same category of *platform basic service*. For example, the e-mail service is offered by the following providers: *SendGrid* [35] via *Heroku* and *OpenShift* platform, *Mailgun* [36] via *Engine Yard* and *Heroku*, *Postmark* [37] via *Heroku.* Likewise, there are several providers implementing the payment service such as: *Spreedly* [38] and *Stripe* [39] offered via *Heroku*[1]. Although the various providers may offer the same category of service, they often differentiate themselves in several

[1] The list of the *platform basic service* providers and the *cloud application platforms* at this point of the thesis is provided for explanatory purposes and should not be considered as exhaustive.

features such as the pricing, the quality of the product and the provided functionality. Therefore, application developers can choose from a wide range of available options the one, which best meets the requirement at hand.

The existence of a multitude of *platform basic service* providers should be seen as a positive thing for the consumers, since it increases market competition and thus forces the providers to strive for more quality products and cheaper prices. However, in order for the *cloud platform* consumers to harness the benefits stemming from the pluralism of the providers, they need to be able to deploy seamlessly the ones which each time better meet the requirements at hand. In reality, this cannot be achieved out of the box due to the heterogeneity among the *platform basic service* providers which adds to the already existing heterogeneity among the *cloud platform* offerings. The heterogeneity lies primarily in the resources that the platforms offer. For example, certain programming languages and frameworks, such as the popular *JavaEE* [40] framework, is only supported by certain *cloud platforms* such as *OpenShift*. Likewise, diversity may arise in the provided databases. A detailed list of the differences, which may be encountered across the platform offerings, is stated in chapter 3.

Particularly, in the case of the *cloud application platforms,* an additional level of heterogeneity arises among the providers who offer *platform basic services.* Specifically, the following variability issues, thoroughly discussed in Chapter 4, may arise:

- **The differences in the workflow**: For a specific kind of service, different providers may adopt a different workflow in order to complete an operation. For example, in the case of the payment service various implementations exist that support completing a purchase request, and these can follow different workflows.
- **The differences in the web API:** As explained earlier in this Section, the consumers access the *platform basic services* via their exposed *API.* Depending on the concrete provider, the expected parameters in the *API* may change. For example, in the case of the e-mail service, *SendGrid, Mailgun,* and *Postmark* expose a different *API* for the operation "*send e-mail*".

- **Management of the configuration and authentication variables**: In order to configure a service and authenticate the user, several variables may be required. Indicatively, the *Google* authentication service requires among others the following variables: a) the *redirect URL*, b) the *client ID*, c) the *scope*. The number and the type of the settings vary according to the provider. Considering the large number of *platform basic services* out of which a *service-based cloud application* may be composed, the management of the settings may become a time consuming and strenuous process.

The heterogeneity among the *platform basic service* providers may create an uncertainty to the software developers, willing to leverage *platform basic services,* regarding the optimal provider to be used. In turn, this may reduce the trust towards the *cloud platform* providers and becomes a hindrance to their wider adoption. Representatively, *Gigaom* research [41], a major technology market analysis company, states that despite the steady annual growth of the *PaaS* market, it still enjoys a slower adoption rate compared to the *IaaS* and *SaaS* market [42]. Among the reasons behind this is the confusion that IT managers experience with heterogeneous *PaaS* interfaces and technologies.

To challenge this current state of affairs, we believe that software developers should be empowered to create their applications without having to deal with all the inconsistencies across different *cloud platforms*. In other words, to the extent that this is possible, the underlying technologies that power each platform should be made transparent to the users. Therefore, applications can be built in a platform agnostic manner and only at the deployment stage the target platform is selected.

There are multiple benefits associated with granting *cloud applications* the ability to be deployed across multiple *cloud platforms*. Software developers are not required each time to cope with the peculiarities and the proprietary technology of each *cloud platform*. Instead, the application is only developed once and then it is deployable across multiple platforms. This fact further insulates the developers against the deficiencies of particular *cloud platforms*, by giving them the freedom to switch to another platform offered by a different provider. For example, a provider may fail to meet the pre-determined *service level agreements (SLAs)* either by reducing the quality of the offered service, or increasing the price or even by terminating unexpectedly the services offered [43]. In such a case the software developers will be able to deploy the same applications on a different *cloud platform* and thus minimise the disruption caused to the users of these applications. Eventually, the ability to deploy a *cloud application* securely and robustly across multiple *cloud platforms* will promote a wider adoption of the *PaaS* offerings by the software developers.

1.2 Aim of the thesis

To this end the aim of this research work is to propose a methodology and build a development framework which enables the creation of *cloud platform* agnostic applications. Recapitulating the famous slogan invented by *Sun Microsystems* [44],

"Write once, run everywhere." [45]

,which refers to the fact that Java applications can run on any device with the *Java Virtual Machine* [46] installed, the target of this research work is to empower software developers to code their *cloud applications* once and deploy them on any *cloud platform*. Essentially, developers should remain focused on the creation of the application and not on the process of learning the peculiarities and specific technology of each target platform.

However, although such a vision sounds highly appealing, its feasibility remains disputed. The large diversity across *cloud platform* offerings does not allow the engineering of a solution that abstracts over the whole variety of the platform specific technologies. The field of *cloud platforms* can be ranged between offerings, which provision the basic development resources offerings, which provide additional *platform basic services,* and platforms which follow a concrete high-level development paradigm based on graphical interfaces. Chapter 3 analyses the heterogeneity among the *cloud platforms* and draws the requirements that a platform needs to meet in order to be accommodated by the solution proposed in this research work. Essentially, this thesis focuses on the design of *service-based cloud applications*, namely those *cloud applications*, which are based on *platform basic services.*

Specifically, this thesis proposes the *Service-based Cloud Application Development Framework* (*SCADeF*), which aims to provide the tools and the methodology to support:

i. The consistent integration of different categories of *platform basic services* with a *service-based cloud application*.
ii. The seamless use of the various *platform basic service* providers in a manner that is transparent and technology-agnostic to the software engineer.

Such a vision entails the alleviation of the three variability points across *platform basic services*, namely: (i) the differences in the workflow, (ii) the differences in the web *APIs* of the various providers and (ii) the management of the configuration and the authentication variables. Essentially, software developers shall focus on building the functionality of the application and then will be able to choose automatically the concrete *platform basic service* provider which implements this functionality.

1.3 Standardisation and Intermediation as solution approaches

Towards enabling the development of platform agnostic *cloud applications,* two approaches have become prominent, namely *standardisation* [47] and *intermediation* [48]. The first involves the definition of a common set of standards adopted by all cloud providers. This set of standards could include well-defined *APIs*, which enable the uniform access to the *platform basic services*, and standard formats to store and retrieve data from the databases. An example of the standardisation approach is the *Cloud Data Management Interface (CDMI)* [49] which puts forwards a common *HTTP* interface in order for the consumer to access cloud file storage services. *CDMI* is supported by major IT companies such *IBM* [50] and *Hewlett Packard* [51].

The second approach, intermediation, involves the introduction of an intermediate layer which decouples application development from any specific platform provider technologies. An example of a widely adopted intermediation approach is the *Java Database Connectivity (JDBC)* [52] driver technology, which grants *Java* uniform access to any kind of database by hiding the specific underlying implementation.

While standardisation seems an efficient approach to enable cross-platform development of *cloud applications*, the establishment of a standard is a strenuous and time-consuming process, which requires the consensus of major stakeholders[1] and thus lies beyond the scope and the capabilities of this research work. Therefore, in this thesis work the intermediation approach is adopted.

1.4 Bird eye view of SCADeF (Service-based Cloud Application Development Framework)

Figure 2 shows a high-level view of the *SCADeF* framework, the proposed solution. *SCADeF* mediates between the software developers and the *platform basic service* providers. The developers create their applications using the high level functionality offered by the framework, which then undertakes the tasks to integrate the application with the concrete service providers. As seen from Figure 2, the framework supports a number of different *platform basic services* and providers. For each category it provides a reference implementation, which is exposed to the developer, and the specific implementation of each supported provider. When a *platform basic service* is required, the reference implementation is used while the specific vendor implementation remains transparent.

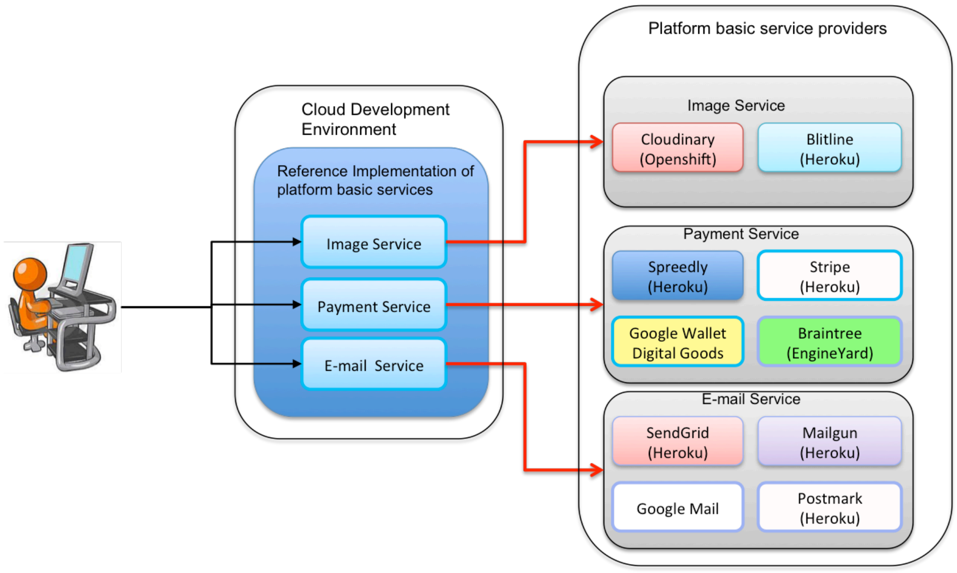


Figure 2: Bird eye view of the SCADeF framework

[1] Such as large companies, organizations and potentially governmental institutions

SCADeF adopts a three phase process in order to support additional *platform basic services* and concrete providers:

1. **Platform Service Modelling Phase:** During this phase, the abstract functionality of a *platform basic service,* which is exposed to the application developers, is modelled.
2. **Vendor Implementation Phase:** During this phase, the concrete implementation of each of the service providers is mapped to the abstract functionality defined in the previous step.
3. **Execution Phase:** During this phase, the development framework will undertake the tasks to mediate between the application and the concrete providers.

The whole process of supporting a *platform basic service* provider with *SCADeF* and the mediation between the application and the concrete providers remains transparent to the software developer. The latter focuses on building the functionality of the application and is not distracted from the specific implementations of the various service providers. Therefore, the proposed framework facilitates the seamless use of *platform basic services* and providers offered by different *cloud application platforms* and with this respect aims to contribute to the cross-platform development of the cloud applications.

1.5 Thesis Objectives

This Section presents the aims and objectives for the work described in this thesis, as well as a summary of contributions, and a synopsis of the rest of the thesis report.

1.5.1 Theoretical objectives

1. Introduce and explain the concept of the *cloud application platform* and the *service-based cloud application* as a specialisation of the concepts *cloud platform* and *cloud application* respectively.
2. Survey the existing *cloud platform* offerings and classify them according to their application development and deployment features.
3. Identify the variability issues which need to be addressed in order to enable the cross-platform development of *service-based cloud applications.*

4. Propose a methodology to alleviate the variability issues.
5. Examine certain means such as the use of *XML* templates and ontologies in order to capture the differences in the web *APIs* of the *platform basic service* providers.

1.5.2 Technical objectives

1. Experiment with a number of *cloud platforms* and *platform basic services* in order to extract the requirements of the development framework.
2. Design the architecture to support the cross-platform development of *service-based cloud applications.*
3. Implement the development framework based on the architecture.
4. Provide the toolset for:
   i. The *administrator* of the development framework to be able to add *platform basic services* and their respective providers to the framework.
   ii. The user of the development framework to be able to make use of it during the development of the *service-based cloud application.*

1.5.3 Experimental objectives

1. Experiment with a set of real case *platform basic services* and a respective number of providers implementing those services.
2. Validate the overall approach of the development framework and demonstrate its capability to contribute to the cross-platform development of *service-based cloud applications.*

1.5.4 Thesis contributions

The main contributions of the thesis are listed below:

C1: Clarification of the notions of *cloud applications platforms* and *platform basic services* and a subsequent exemplification of how these notions could be leveraged to accelerate the cloud-based development process and lead to the creation of *service-based cloud applications*.

C2: The formulation of a methodology which enables the design of *service-based cloud applications* independent of the concrete *platform basic service* providers.

C3: The design of the *Service-based Cloud Application Development Framework (SCADeF)* to support the above methodology.

C4: The construction of a toolset to enable the operation of *SCADeF* by the software developers.

C5: Manifestation of how the *micro-service* architectural style could be applied in the field of cloud computing with the use of *cloud application platforms* and *platform basic services.*

1.6 Thesis Outline

PART A: Literature review on cross-platform development of cloud applications

Chapter 2: This chapter presents a selected overview of the field of cloud computing. Its aim is to introduce the reader to the concepts, which will be used throughout the rest of the thesis. It provides a definition of cloud computing followed by a brief review of the technologies that led to this novel paradigm, namely virtualisation, grid, and distributed computing as well as the *SOC*. Next, it breaks down the term cloud computing into the three service models: *IaaS*, *PaaS* and *SaaS*. The remainder of the chapter focuses on the *PaaS* level and attempts to provide a concise definition of the term *cloud platform* as well as defining the author's notion of a *cloud application*, which will be used throughout the rest of the thesis.

Chapter 3: The chapter presents an analytic survey of current *cloud platforms.* The scope of the survey is to examine the available *cloud platform* offerings and analyse them based on a certain framework of features. The purpose of this survey is to allow the categorisation of the platforms into certain groups, where the platforms in each group expose similar characteristics. This supports the goal to narrow down the research focus onto a specific group, which will benefit most from the framework proposal. According to the analysis of the survey, three clusters can be defined:

1. The first group includes *cloud platforms, which* support widely used and standard technologies, such as *MySQL* and *Java*.

2. The second group offers additional functionality in the form of *platform basic services*.

3. The third group adopts a different development paradigm, which relies on web-based graphical environment.

This thesis focuses on the second group namely, the *cloud application platforms.* The remaining of the chapter further clarifies the terms of the *cloud application platform* and the applications, which are deployed on them namely, the *service-based cloud applications.*

Chapter 4: Chapter 4 surveys the field of cross-platform development of *cloud applications*. The scope of this chapter is to illustrate the approaches followed by the related work on the field. Specifically, two approaches are considered: (i) *standardisation* and (ii) *intermediation*. The latter one is further divided into three sub-categories: a) Library-based solutions, b) Model-driven engineering based solutions and c) middleware solutions. The role of the chapter is two-fold:

1. It aims to identify potential gaps in the area of the cross-platform development of *cloud application,* which are not covered by the existing work.
2. It pinpoints weaknesses stemming from the methodologies and the tools adopted by the related work.

The combination of the outcome of the two aims leads to the precise contextualisation of this research work.

PART B: The Service-based Cloud Application Development Framework (SCADeF)

Chapter 5: This Chapter describes the high level architecture of the proposed *SCADeF* framework, which aims to facilitate the cross-platform development of the service-based cloud applications. Particularly, the framework addresses the variability issues encountered when dealing with multiple *platform basic services* and providers. These are the:

1. The differences in the workflow required completing the operations of the service providers.
2. The differences in the *API* exposed by the various service providers.
3. The management of the configuration and authentication variables required by each provider.

In order to address these issues, as stated in Section 1.4, *SCADeF* consists of two parts. The first one, the *Platform Service Workflow* part, addresses the first variability issue, namely the differences in the workflow across the various *platform basic service* providers. Likewise, the second part, the *Platform Service API*, aims to alleviate the differences in the *APIs* as well as in the various configuration and authentication variables required by the service providers.

For each of the two parts *SCADeF* defines three phases. During the first one, the *Platform Service Modelling Phase,* the abstract functionality of the *platform basic service* is modelled. Subsequently, during the *Vendor Implementation Phase* the concrete implementation of each supported provider is infused. The *Execution Phase* is responsible for executing the workflow, which was defined during the first two phases, and generating the *API* client adapter required to interact with each concrete provider. Therefore, as seen in Figure 2 the application developer is able to access the required *platform basic service* providers by only interacting with the reference implementation provided by the *SCADEF* framework. The Chapters 6-9 describe in details the process of adding a new *platform basic service* and concrete provider to *SCADEF*.

Chapter 6: This Chapter focuses on the *Platform Service Workflow* part, namely the one that addresses the first variability issue regarding the differences in the workflow across the various *platform basic service providers.* The first step towards addressing the workflow variability is to define a reference workflow, which abstracts the provider specific ones. For that reason a *Reference Meta-Model* is provided, including the concepts required to build the reference workflow. The result is the *Platform Service Connector (PSC), which* contains the abstract functionality of the *platform basic service.* Subsequently, the implementation of each concrete provider is built and mapped on the abstract one contained in the.

Chapter 7: After having created the abstract functionality for each *platform basic service* supported by the *SCADeF* framework and having included the concrete provider implementation, Chapter 7 describes how the functionality is executed. Specifically, the aim of the framework is to undertake the task of executing the workflow, which has been defined during the *Platform Service Modelling* and the *Vendor Implementation Phase.* For that reason the *Platform Service Execution Controller* is constructed to manage the execution of each state contained in the workflow. Therefore, the details of the execution of the operations defined in the *platform basic services* remain transparent to the application developers.

Chapter 8: A similar process to the description of the platform service workflow is defined for the description of the *Platform Service API* part. Chapter 8 focuses on the second variability point, which is associated to the differences in the API across the *platform basic service* providers. Similar to Chapter 8, this Chapter describes the way the *Reference API* is constructed. The *Reference API* describes the functionality offered by the *platform basic service* and is unique for each category of service supported by *SCADeF*. The next step is to map each vendor specific *API* to the reference one. Therefore, the application developers interact only with the *Reference API* while the vendors' specific *APIs* remain transparent.

Chapter 9: After having defined the *Reference API* and having mapped the provider's specific ones, Chapter 9 focuses on the *Execution Phase* during which the *API* client adapters are generated. They contain the source code required to invoke the operations offered by the *platform basic service* providers. The *API* client adapters are generated automatically using a code generator technique. Thus the application developers access only the *Reference API* defined in the previous Chapter and they are not required to manually implement the web calls in order to invoke the operations of the service providers. The outcome of the generation is a set of interfaces with the operations offered by the *platform basic service* and the respective implementation of the specific providers supported by the *SCADeF* framework. The software developers access only the service interfaces while the concrete implementation remains transparent. This contributes to the initial aim of this research work, which is to “hide” the providers’ implementation from the software developers.

PART A: Conclusion and future work

Chapter 10: This Chapter concludes the research thesis. Specifically, it summarises the research work and states how contributions, which were defined in Section 1.5 have been fulfilled. Furthermore, the Chapter recommends future work to be carried out as continuation of this research thesis.

PART A

Literature review on cross-platform development of cloud applications

PART A
Literature review on cross-platform development of cloud applications

Chapter 2

Background on cloud computing and cloud platforms

Cloud computing is still an evolving field and new offerings are constantly launched, while the terms and concepts around this paradigm are still being shaped. To this end, the Chapter attempts to provide background information on the field of cloud computing and the concepts which are involved, such as *cloud platforms* and *cloud applications*. Cloud computing has not been developed from ground up but has rather evolved from previous well-established computing paradigms. In order for the reader to understand the underpinnings of this new paradigm, a brief history of its evolution is provided and a comparison with previous computing models, such as the grid and *SOC* is made. The aim of the comparison is to pave the way for clarifying the notions of *cloud platforms* and *cloud applications*.

As mentioned in the Introduction, the research topic of this thesis is the cross-platform development of *cloud applications*. The topic involves two core terms: (i) the *cloud platforms* and (ii) the *cloud application.* Therefore, after having introduced the field of cloud computing the chapter aims at introducing the field of *cloud platforms* and providing the author' notion of the *cloud application,* which will be adopted throughout the rest of the thesis.

Particularly, Section 2.1 attempts a short history of the evolution of cloud computing and provides the relevant information in order for the reader to gain a deeper view on how core concepts of this thesis, such as *cloud platforms* and *applications* have been evolved. To this end it examines previous computing paradigms, such as grid and

SOC. In Section 2.2 the three service models, as defined by the *National Institute of Standards and Technology (NIST)* [18], are introduced namely: the *IaaS*, the *PaaS* and the *SaaS*. The second half of the Chapter focuses on the terms of *cloud platform* and *cloud application.* Thus, Section 2.3 attempts to illustrate the field of *cloud platforms* by mentioning specific characteristics and major providers such as the Google App Engine (*GAE)* and *Heroku*.

Finally, Section 2.4 describes the notion of *cloud application.* The term is often used by different parties to denote applications which are deployed in all three levels of cloud computing (*SaaS, PaaS, IaaS*) [53], [54]. However, in the context of this research thesis a cloud application refers to those applications which are developed using the resources offered by the *cloud platforms* and are subsequently deployed on them.

2.1 Evolution of cloud computing

While cloud computing has emerged as a revolutionary computing paradigm, it does not actually constitute a novel technology, but it has rather evolved from previous established computing paradigms. This Section, as shown in Figure 3, presents a brief timeline of cloud computing and attempts to compare it with previous computing paradigms.

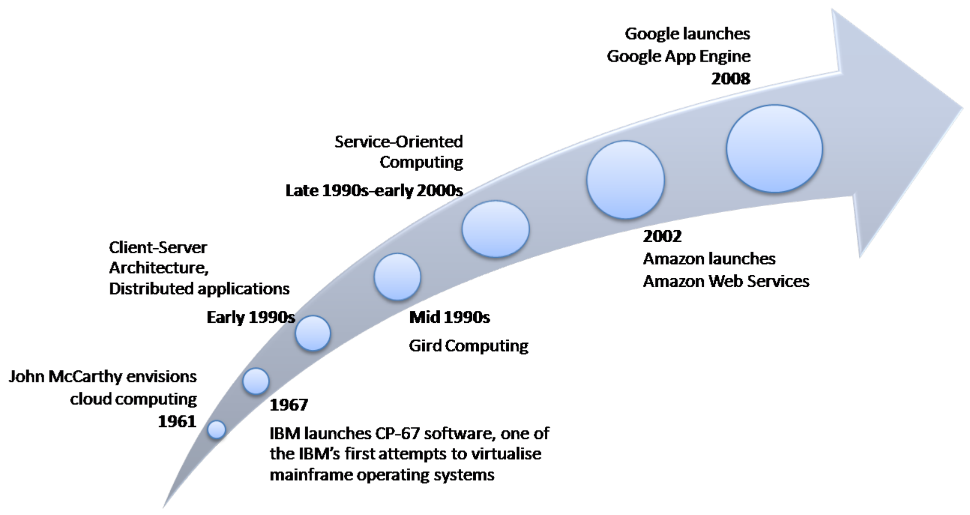

John McCarthy envisions cloud computing
1961
1967
IBM launches CP-67 software, one of the IBM's first attempts to virtualise mainframe operating systems
Client-Server Architecture, Distributed applications
Early 1990s
Mid 1990s
Gird Computing
Service-Oriented Computing
Late 1990s-early 2000s
2002
Amazon launches Amazon Web Services
Google launches Google App Engine
2008

Figure 3: History of cloud computing

Historically, the term cloud computing was first envisaged by the computer scientist McCarthy and dates all the way back to 1961 [55] [56] :

> *"If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility... The computer utility could become the basis of a new and important industry."*

McCarthy was the first one to envision computing being ubiquitous and easily available to the public.

2.1.1 Virtualisation technique

In 1967 *IBM* developed the *CP-67* virtual machine operating system, one of the first attempts at virtualising mainframe operating systems [1]. Particularly, *CP-67* was a hypervisor which enabled the sharing of memory across virtual machines and at the same time provided each user with a dedicated virtual memory space. Virtualisation is a technology which abstracts over the hardware resources and thus allows sharing of the physical resources, such as processing power, storage and networks [57]. While virtualisation has been applied since the 1960s in mainframe computing, it has only been in recent years that it has become a widespread concept in other kinds of computing, thanks to advances in computing power and high-speed networks. Thus, virtualisation constitutes an essential element of cloud computing, which decouples the allocation of the physical resources from their geographical location; and includes the notions of rapid elasticity and resource pooling. [58].

2.1.2 Client-Server architecture

In the early 1990s the client-server computing model emerged and became part of the mainstream [59]. In this computing paradigm a client initiates a request to the server and the latter processes the request and responds with the result [60]. Essentially, it relies on the distributing computing model, which promotes the communication of the various system components via message exchange [59]. The client-server model may be considered as the evolution of the mainframe systems model, which was characterised by centralised computing and restricted access [59] [61]. Cloud computing is still based on the client-server model, but there are certain differences as noted in [58] and [62]. First, cloud computing implements a management layer which monitors the workload and is able to alleviate traffic congestion by deploying and releasing additional resources as needed. Furthermore, via cloud computing the services are offered on demand and they are highly configurable. In addition cloud computing adopts a business model based on the pay-per use approach.

2.1.3 Grid Computing

In the mid-1990s another model of computing emerged as an evolution of distributed computing, namely the *Grid* [63]. Grid computing denotes the large-scale sharing of resources across virtual organisations usually required by compute and data intensive scientific and engineering applications [64]. Cloud computing is sometimes considered to have been a development arising from the grid [58], [65]. However, there are significant differences between the two paradigms. Grid computing mainly focuses on the collaboration among the users in order to share resources for scientific and research purposes, while cloud computing rather targets the business and web-based applications adopting a pay-per-use policy. Additionally, while grid computing focuses on the sharing of virtualized infrastructure resources, cloud computing introduces two additional levels, namely the *SaaS* and the *PaaS*. Furthermore, grid computing aims to combine existing computing resources to provide high performance computer (*HPC*) capabilities to those without access to expensive *HPC* machines. On the other hand cloud computing leverages virtualisation techniques to enable a single physical server to be allocated to many users concurrently [58].

2.1.4 Service-oriented Computing

Moving to the late of 1990s and early 2000s a new computing paradigm arose which became known as (*SOC*). The goal of *SOC* was to push forward the rapid, cost-efficient and easy development of distributed applications as composition of services [66]. A service in this context was a packaged software component capable of being re-used in multiple applications and most importantly it was self-describing, such that it could be discovered and integrated automatically with other similar services [67]. Thus, *service-based applications* [68] were developed from a combination of loosely-coupled services, which each exposed a well-defined interface, hiding at the same time their underlying implementation. Cloud computing exploits the principles of

SOC in order to offer its resources to the public in a similar way. Thus cloud resources, such as storage, computation, and message queuing may be offered in the form of services via standardised technologies, such as the *Web Service Description Language (WSDL)* [69] and the *Simple Object Access Protocol (SOAP)* [70] or the REST [31]. At the same time, widely explored concepts in the domain of *SOC,* such as service governance and *SLAs* are also leveraged by cloud providers [70]. On the other hand, the emergence of the cloud computing paradigm has brought about a number of concepts which were hitherto not extensively explored, such as the pay-per-use business model and automatic resource scaling [67].

2.1.5 Early commercial cloud products

In 2002 *Amazon* launched the *Amazon Web Services (AWS)*, a suite of cloud-based services, which offered computation and storage resources. Cloud computing, in the form which is known today, started after 2006 when *Amazon* launched the *Elastic Compute Cloud (EC2)* [72], which allowed small companies to rent virtual machines [73]. *Google* and *Microsoft* followed shortly afterwards with the release of their own cloud services, namely the *GAE* [74] in 2008 and the *Microsoft Azure* [75] in 2009 respectively.

2.1.6 Cloud computing, why now and not then?

From the above, it is clear that cloud computing has evolved from a number of previous well-established computing paradigms and has exploited technologies which were already established. Then why did cloud computing become widespread only the last couple of years? According to [13] it is the new technology trends and business models, such as the effortless use of credit cards to purchase online services, which has promoted the provisioning of cloud resources. Furthermore, enterprises and organisations have ascertained that their IT resources were historically largely underutilised and they could reduce their costs by renting the resources they actually need [65]. Last but not least the advances in the networking technology offered by the Internet have made feasible the sharing of cloud resources.

Thus, nowadays cloud computing has turned into a significant paradigm serving a wide range of consumers from large enterprises to individual end users. However, there are still barrier to uptake such as security, privacy and interoperability.

2.2 Cloud computing in three service models

While cloud computing initially referred to the provision of low-level hardware resources such as compute and storage, it evolved also to include the sharing of additional IT resources, such as run-time programming environments for software developers and complete software products for end users, such as software for *CRM*. Due to the wide range of heterogeneous resources, which are covered under the umbrella term of cloud computing, the latter is divided into certain service models. The most established classification, shown in Figure 4, is provided by *NIST,* which proposes the following three service models of cloud computing [18]:

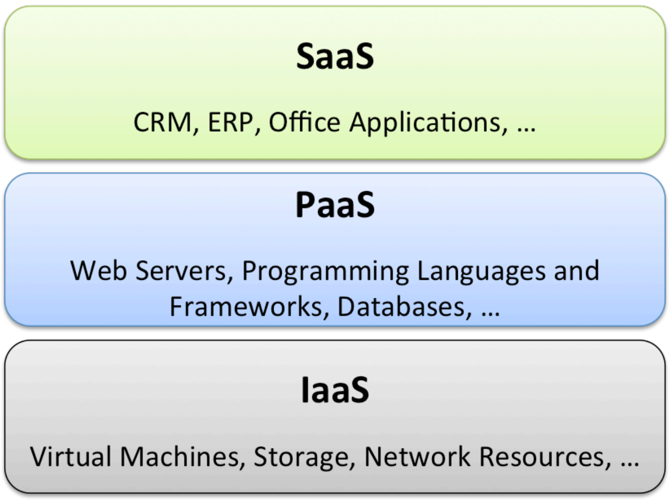

SaaS
CRM, ERP, Office Applications, ...
PaaS
Web Servers, Programming Languages and Frameworks, Databases, ...
IaaS
Virtual Machines, Storage, Network Resources, ...

Figure 4: The 3 service models of cloud computing according to *NIST*

- **Infrastructure as a Service (IaaS):** *IaaS* provides the consumers with low-level IT resources, such as processing, storage and networks, which the latter can use to install and run any arbitrary software including operating systems and applications. Thus *IaaS* allows enterprises and developers to expand their IT resources on demand. However, consumers' access is limited to the operating systems, storage and the deployed applications and they do not have control of the underlying infrastructure. Example of IaaS offerings are: *Amazon* Elastic *Compute (EC2)* [72] and *Google Compute Engine* [76] which offer virtual machines as well as *Amazon Simple Storage* [77] and *Microsoft Azure Storage* [78] which provide dynamically scalable storage solutions.

- **Platform as a Service (PaaS):** Cloud providers in the *PaaS* level offer software programming capabilities such as run-time programming environments, libraries, databases and software tools. Software developers exploit these capabilities in order to create their applications, which are then deployed on the cloud infrastructure. Similar to the *IaaS* level, consumers' control is restricted to the deployed applications while they are restrained from accessing the underlying infrastructure such as the servers and the operating systems. Some representative examples of *PaaS* offerings are: *GAE* [74], a platform to develop and deploy applications in several programming languages such as *Java*, *Python* and *Go*. *Heroku* [28], is another major platform which offers programming resources and third party services, such e-mail service and payment service. *Zoho Creator* [79] offers a web-based programming environment for rapid and specific purpose application development such as the creation of *CRM* applications.
- **Software as a Service (SaaS):** In the *SaaS* level providers make available complete software applications, which are executed on their infrastructure, via a web-browser or a desktop program interface. This model signals a shift from the traditional way that software applications were distributed, installed and upgraded on the consumers' own IT platforms. Rather consumers can directly access an application online via a web client and are charged on a pay-per-use basis. At the same time *SaaS* providers can use a shared application codebase to serve multiple customers via a multi-tenancy architecture [43], which may also occur in the databases. Furthermore, the process of maintaining and upgrading the software is handled entirely by the provider and remains transparent to the consumers. Similar to the IaaS and PaaS level, consumers only control the software application and they do not have access to the underlying resources that the application is relying on. Example of SaaS providers are *Salesforce* [80] which mainly offers *CRM*-oriented applications and *SAP* which provides *ERP* solutions.

2.3 Cloud platforms

As described in the previous Section, at the *PaaS* level, cloud providers offer their solutions via *cloud platforms*. *Cloud platforms* are rapidly gaining momentum and have become particularly popular among software developers, who use them to build and deploy rapidly their web applications. There are two major groups of consumers of *cloud platform* offerings [81]:

1. *Independent Software Vendors* (*ISVs*). By leveraging *cloud platform* resources *ISVs* can quickly develop and launch new software products and at the same time minimise their capital costs. On top of that by offering their software products via a major *cloud platform,* such as the *GAE*, they can reach a far larger global market.
2. IT departments of organisations and companies. Rather than developing from scratch and maintaining the required software on premise, IT departments may choose to leverage the capabilities offered by the *cloud platforms*. Additionally, cloud platforms can be exploited for testing purposes and for building proof of concepts.

2.3.1 Early cloud platform offerings

When the *cloud platforms* first appeared in the late 2000s, they set out to provide the basic programming resources required by the developers in order to build and deploy their applications (Figure 5).

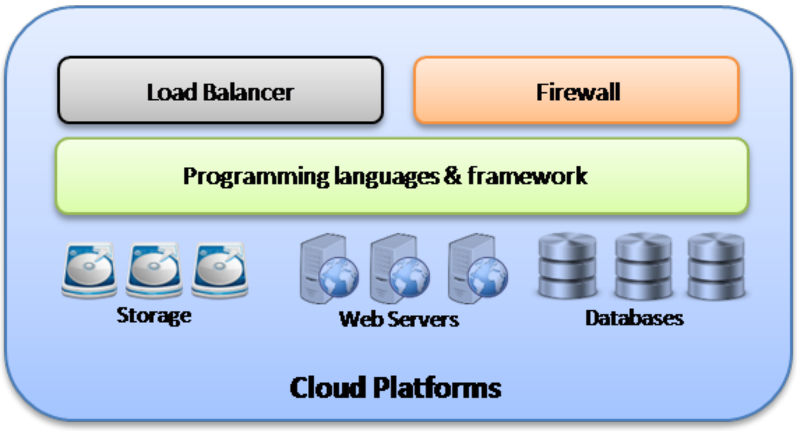


Figure 5: A Cloud Platform

A representative set of these resources is [82] [83] :

1. Programming languages and frameworks. Developers are offered a variety of languages and frameworks such as *Java*, *PHP*, *Java Spring* [84] and *.NET* [85].
2. Web servers. Several pre-configured and ready to be used web servers are offered by the *cloud platforms,* such as *Apache Tomcat* and *Microsoft IIS*.
3. Databases. Software developers can directly connect their applications with one of the offered databases, such as *MySQL* and *PostgreSQL* [86].
4. Storage services. In addition to databases, certain platforms such as *GAE* and *Amazon* offer additional types of block file storage space.
5. Firewalls and load balancers. Apart from the programming resources *cloud platforms* offer capabilities related to the execution of the applications, such as firewalls to ensure a certain level of security and load balancers in order to distribute uniformly the incoming workload.

Exploiting the basic programming resources offered by these *cloud platforms*, software developers are able to reduce the effort and cost, which is required to set up and maintain the programming stack. However, they still need to build the whole functionality of the application from the ground up. This means that *cloud platforms* offer the essential development resources but not any additional functionality such as an e-mail service which could contribute to the reduction of the development time.

2.3.2 Evolution of cloud platforms

Since the first generation of *cloud platforms* and following their rising popularity among software developers [87], a growing number of platform vendors have launched new platform products with extended capabilities. Thus platforms are now extending their functionality beyond traditional programming resources to include tools that support not only the development but also the deployment, execution and management phases of the application. Examples of such tools are the deployment plug-ins for *Integrated Development Environments (IDEs)* like *Eclipse* or *NetBeans*, which automate the deployment process of the application on the platform and also provide tools to perform logging and monitoring of the application.

Gradually the *PaaS* market has experienced a significant shift from the basic technology provided by traditional cloud vendors such as *GAE*, *Microsoft Azure* and *Amazon Elastic Beanstalk (AEB)* [88] to a wide range of capabilities supported by the platform offerings today [89]. In order to meet the consumers' need for ever lower application development times, platform vendors seek to introduce additional capabilities in their offerings. Thus on top of the tools provided up until that point to support the development, deployment, and management of the application, platform vendors attempt now to provide the consumers with software functionality, which can be incorporated directly in the application. This functionality is provided through the platform in the form of an autonomous and reusable software component, which exposes its operations via a standardised interface, using the *REST* protocol. Examples of such functionality are the authentication service, the e-mail service, the message queue service, and the image processing service. Platforms which nowadays offer such functionality include *Heroku*, *OpenShift,* and *Engine Yard*. Thus software developers do not need to build complete applications from the ground up. Rather they can reuse the functionality provided by the *cloud platforms* in order to decrease significantly the required time and effort.

In an attempt to reduce even further the effort and time required to build an application, some platform vendors have launched offerings that adopt a different development paradigm. Rather than expecting the developers to program their applications and deploy the source code on the platform, platform vendors offer web-based graphical environments, where the users can simply create applications by combining drag-and-drop elements and other pre-designed features offered by the platform. The applications which can be created have a specific narrow scope and are usually *CRM* and ERP-oriented. Examples of such *cloud platforms* include *Zoho Creator* [79] and *App Cloud*[1] [90].

2.4 Cloud applications

Having explained and clarified the notion of *cloud platforms* in the previous Section, this Section attempts to define the author's notion of a *cloud application*. As

[1] Formerly known as Force.com

mentioned in the Introduction *cloud platforms* and *cloud applications* are the core concepts of this research work, which concerns the cross-platform development of *cloud applications*.

As described earlier in the Sections 2.1 and 2.3 cloud computing is still an evolving field. Although, there is established terminology that is widely accepted, such as the notions of *IaaS* and *PaaS*, there are still some terms, which mainly due to their generic nature may be used by different parties to denote different concepts. For example the term *cloud service* may often be used as such to denote services provided at the *IaaS*, *PaaS* and the *SaaS* level. While technically the use of the term is not wrong, confusion may be caused among parties with different backgrounds. Therefore, Breiter and Behrendt [91] correctly break down the definition of *cloud service* and refer to the particular subcategories of *infrastructure services, platform services* and *software services.*

Likewise, this Section attempts to clarify the notion of the *cloud application, which* will be adopted in the rest of the thesis. Similar to the term *cloud service,* the term *cloud application* may be used to denote different concepts each time, depending on the viewpoint of a particular cloud computing service model. For instance, looking at the IaaS level, a *cloud application* may refer to an application, which is directly deployed on the infrastructure resource, such as virtual machines leased from an IaaS provider such as *Amazon*. For example *Instagram* [92], a popular online photo-sharing service, had been deployed on *Amazon EC2* infrastructure before it was moved, in 2013, to *Facebook*'s data centre [93]. Likewise, a *cloud application* may also denote applications offered at the *SaaS* level. *Salesforce* provides the following definition for *cloud applications* [53]:

> *"Cloud computing applications, or apps, are the cloud-based services also known as Software as a Service (SaaS)."*

The same understanding of the term is also adopted by *Oracle* and thus they refer to *cloud applications* as *SaaS* applications [54].

However, in this research work the term *cloud application* is specifically used to denote those applications which are developed and deployed on a *cloud platform.*

Therefore, the *cloud application* is built using the platform resources, such as programming frameworks, databases, logging tools and is executed on the platform's infrastructure such as the available web servers.

The author's notion of *cloud applications* is further narrowed down in the next Chapter.

2.5 Summary

This Chapter presented a selective overview of the cloud computing terms associated with this research topic. Particularly, the terms which were introduced are the *cloud computing*, the *cloud platforms* and the *cloud applications*. In order for the reader to gain a deeper understanding of how *cloud computing* and *platforms* evolved, the Chapter presented a brief comparison with preceding computing paradigms such as the grid and the *SOC*.

Subsequently, the Chapter focused on the *Platform as a Service* level and the clarification of the term *cloud platform*. Several types of *cloud platforms* were mentioned followed by their specific characteristics. Finally, the author's notion about *cloud applications* was provided. When referring to *cloud applications* this research thesis denotes the applications which are developed with the use of *cloud platform* resources and which are deployed on them.

The next Chapter continues the examination of the *cloud platforms* and attempts a systematic analysis of their characteristics.

Chapter 3

Survey of cloud platforms

As it is already mentioned in the Introduction of the thesis, the focus of this research work is to contribute to the cross-platform development of *cloud applications*. The aim is to enable the software developers create their applications once and then be able to deploy them on multiple *cloud platforms*. Towards this direction, it needs to be examined whether it is feasible to engineer a solution which is able to "hide" from the software developers the peculiarities and the proprietary technologies of the whole spectrum of available platform offerings. However, Section 2.3 has already introduced the concept of *cloud platforms* and has shown that the field of PaaS imposes a significant heterogeneity in the available offerings with respect to the cross-platform development of *cloud applications*. For example *OpenShift* [29] provides low-level widely used and standardised programming resources as opposed to the *Zoho Creator* [79] which adopts a high-level web-based application development paradigm.

Therefore, before proceeding with the topic of the cross-platform development of *cloud applications*, the nature of the available *cloud platforms* needs to be analysed and their specific technologies needs to be extracted and examined. To this end the Chapter presents a survey of the existing major *cloud platform* providers and examines them based on the development and deployment resources, which they offer to the consumers. The aim is to identify the concrete category of the *cloud platforms,* along with their specific characteristics, where this research work will focus on. The rest of the Chapter is organised as follows:

Section 3.1 attempts to identify and present specific characteristics associated with the *cloud platforms*. The set of features is based primarily on related reports on the field of *PaaS* and additionally on the experience obtained by the author after examining the cloud platforms listed in Table 1. The Section concludes to two categories of features, the first one involves the management and the execution of the *cloud applications,* whereas the second one is related to the development and the deployment process.

Subsequently, Section 3.2 evaluates and compares the *cloud platform* offerings based on the previously defined set of features. Specifically, since the research work focuses on the cross-platform development and deployment of *cloud applications*, the comparison considers the second category of features, which is related to the development and deployment process.

Based on the comparison of the *cloud platforms* Section 3.3 attempts to group them and classify them into certain clusters according to the programming paradigm, which they adopt, i.e. whether they allow the deployment of source code (*OpenShift*) or they solely offer web-based graphical development environment (*App Cloud*). Based on the classification of the *cloud platforms*, the remaining of the Chapter defines the concepts of the *cloud application platform,* the *platform basic service,* and the *service-based cloud application,* which are adopted throughout the rest of the thesis.

3.1 Cloud platforms characteristics

As shown in Chapter 2, *Cloud platforms* are becoming increasingly popular among IT departments and *ISVs*. *ISVs* can rapidly develop new applications and offer them to a large number of customers through the platform. Due to their emerging appeal, a large number of *cloud platform* offerings are already available on the market. The available platforms form a wide spectrum of existing solutions from which a developer may choose. These solutions may vary significantly from each other.

In order to enable a better understanding of the *cloud platform* offerings and the differences among them, the platforms are compared against a concrete list of characteristics (features). The chosen set of characteristics is drawn both from a synthesis of several *cloud platform* surveys and also from the author`s experience of the field. The examined work is divided in two categories. The first one includes work that has been produced by academic organizations or standardisation bodies: the *NIST* [83], Kourtesis et al., [32], Kolb and Wirtz [94], N. Khan et al.[95], M. Rad et al.[96]. The second one includes reports that have been produced by independent research companies: *Forrester* [97] and *Saugatuck Technology* [98].

3.1.1 Cloud platforms reports

NIST has published an extensive report including recommendations for potential cloud computing users [83]. The report focuses on issues related to each of the three service models (*SaaS, PaaS* and *IaaS*) and proposes best practices for minimizing the exposure to the risks imposed by cloud computing. Particularly for the PaaS service model, which is the main interest of this survey, *NIST* stresses the issues of application development and security across the various *cloud platforms*. *NIST* recommends that users choose *cloud platforms* that offer generic interfaces to access infrastructure resources, such as file storage services, message queue service etc., standard languages and standard data access protocols such as *SQL*. However, special attention is required to avoid database compromise through SQL injection. Moreover, *NIST* it suggests that platform users should analyse the security mechanisms of the platforms to secure data and applications and ensure reliable data deletion.

Kourtesis et al. [32] focus on the concept of software co-development in relation to cloud platforms. With the term “co-development” they refer to the characteristic of a *cloud platform* that allows *ISV*s to develop services and offer them via the platform. These services may also be referred to as *cloud platform* services. The authors are particularly interested in the way software co-development is addressed by *cloud platforms* as a mean for enriching the platform`s core functionality and making a service publicly available to a large number of potential clients. The third-party service can either reside on the platform`s infrastructure or be provisioned by a third-party infrastructure. A cloud-based application can make use of these cloud platform services in order to enrich its functionality. Popular *cloud platform* which are addressed in this thesis and provide software co-development capabilities are: *App Cloud*, *Heroku*, *GAE,* and *Microsoft Azure*.

Kolb [94] attempts to address the issue of application portability across the *cloud platforms*. Towards this direction, a taxonomy of the *PaaS* model based on available *cloud platform* offerings is created. The identified platform characteristics are, among others, the basic programming capabilities such as the run-time environment's and the frameworks, the ability of the platform to scale horizontally and vertically as well the provisioning of additional platform specific functionality either natively or via *ISVs*.

Khan et al. [95] attempt to pinpoint how cloud computing differs from cluster and grid computing. In this context, the three computing paradigms are evaluated against several characteristics. Specific features that are considered are: scalability, ability to negotiate *SLAs,* and pricing models. Moreover, the report takes into account the issues of security and privacy, of standard used technologies and the possibility for third party service integration. Then the authors focus on cloud computing and discuss the three service models (*SaaS, PaaS, IaaS*). Particularly for *PaaS*, they evaluate *GAE* and *Microsoft Azure* against the ability to negotiate dynamically the Quality of Service (QoS), the use of web *APIs* and interfaces to access the services, and the available programming frameworks.

Rad et al. [96] focus on the basic characteristics of *cloud platforms*. They are particularly interested in those characteristics which are related to the development of *cloud applications*. They specifically consider the programming languages and frameworks that the platforms support, the database and file storage offerings, the ability to integrate platform services with the client applications and the deployment methods. The authors evaluate commercial *cloud platforms* such as *Microsoft* Azure, *Salesforce* and *GAE*. They also mention certain issues related to the management of the applications that need to be addressed by the *cloud platforms*. Such issues are: security, performance and availability of the services that the platform offers.

Forrester, a research and IT market analysis company, has issued an enterprise-oriented report [97] to help *ISVs* choose the best *cloud platform* for their partnerships. They attempt a high level classification of the *cloud platforms* according to the type of *IDE* that they provide to the developers and the application development paradigm that they adopt. Then several *cloud platforms* such as: *GAE*, *Microsoft* Azure, *App Cloud*, *Heroku,* and *Zoho Creator* are evaluated against the following three categories of features: development and management of *cloud applications*, cloud vendor`s strategy and product roadmap, and cloud vendor`s market presence.

Saugatuck, an IT research and strategy consulting company, has produced a report [98] to enable *ISVs* and software developers to evaluate the different *cloud platform* offerings. The report pinpoints the following set of characteristics that should be considered when choosing a cloud solution: performance oriented features, such as scalability, reliability and availability, flexible deployment methods, industry standard technologies and methodologies, data and application security, and application integration capability with third party applications. Then the report proposes a cloud development stack and narrows down to specific characteristics for each stack. Some of the features which are considered for the layers directly related to the application development are: security, database offerings, a file storage service, development and deployment tools and methodologies. For all the layers of the proposed cloud development stack, the management capabilities should be considered.

3.1.2 Cloud platform framework of features

After reviewing and analysing the previously mentioned reports about *cloud platforms* and based on the author's own experience of the field, a set of characteristics have been compiled that is considered necessary to be evaluated when comparing the *cloud platform* providers. The characteristics are grouped into two main categories, as shown in Figure 6:

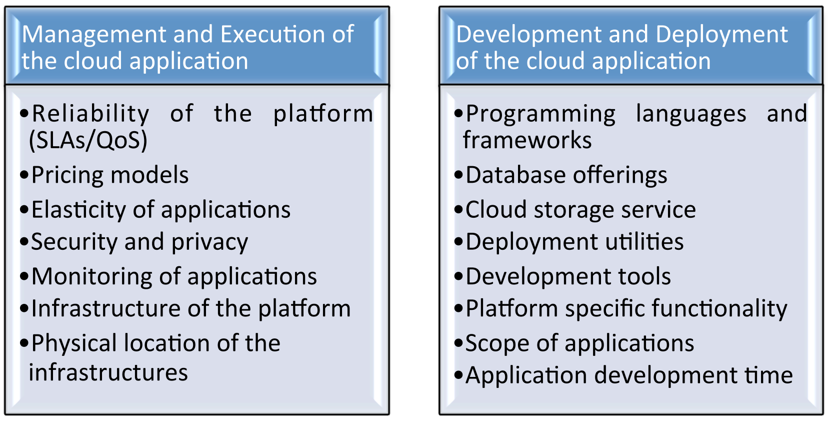

Management and Execution of the cloud application
•Reliability of the platform (SLAs/QoS)
•Pricing models
•Elasticity of applications
•Security and privacy
•Monitoring of applications
•Infrastructure of the platform
•Physical location of the infrastructures
Development and Deployment of the cloud application
•Programming languages and frameworks
•Database offerings
•Cloud storage service
•Deployment utilities
•Development tools
•Platform specific functionality
•Scope of applications
•Application development time

Figure 6: Cloud platform framework of features

1. **Management and execution of the cloud application:** A *cloud platform* acts as the middleware on which the application is running. Therefore features in this category are related to the behaviour of the *cloud platform,* once the application is deployed and is executing on it.
2. **Development and deployment of the cloud application:** Features that are included in this category are related to the development and the deployment stage of the application.

Therefore these two categories can serve as a starting point, for the evaluation of the *cloud platforms*.

3.1.2.1 *Management and execution of the cloud application*

1. **Reliability of the platform**: Reliability ensures that the platform does not violate the agreed *SLAs*. Platforms may suffer from outage resulting in unavailable services, which in turn may lead to unacceptable profit loss for the clients.
2. **Pricing models**: *Cloud platforms* charge the users based on the consumed resources such as the storage, the number of database instances and the computing capacity. They may also apply charges for custom services that they offer. The pricing models may vary significantly across platforms and therefore should be well considered when choosing the target platform.
3. **Elasticity of applications**: Elasticity is related with the ability of the platform to scale up an application to multiple servers when the load increases. It also implies the release of the idle resources when the load decreases. Elasticity represents how efficiently a platform can respond to load fluctuations. This characteristic may be of major significance in applications where the load varies unpredictably.
4. **Security and privacy**: A major impediment for the wide adoption of *cloud platforms* by companies and organizations are the issues of security and privacy. Security refers mainly to the fact that the hosted application and data should be well secured against external hackers' attacks. Privacy, on the other hand, refers to the fact that the platform provider will not exploit clients` hosted data for own profit by revealing them for commercial or other

purposes or that information on European citizens is not processed outside Europe.

5. **Monitoring of applications**: *Cloud platforms* monitor the resources consumed by the hosted applications. This way they can ensure the timely provisioning of additional resources, or the release of idle resources, depending on the workload of the application.
6. **Infrastructure of the platform:** Certain *cloud platforms* own the infrastructure where the clients' applications are running whereas other platforms are renting the infrastructure resources from an *IaaS* provider. The users of the *cloud platforms* may be interested in knowing where the applications are physically hosted since this affects the reliability, security, and privacy factor of the platform.
7. **Physical location of the infrastructure:** A characteristic, closely related to the previous one, is the physical location where the application is hosted. The physical location of an application may impact the performance of an application. Consider, for example, that an application is hosted in a data centre in the USA and the majority of the users come from China. There is a time overhead in accessing the remote server in the USA. For that reason major platform providers are building data centres all around the world. There may also be legal issues related to the physical server location that hosts an application.

3.1.2.2 *Development and deployment of the cloud application*

1. **Programming languages/frameworks:** There is a wide variety of programming languages and frameworks from which a developer can choose to develop an application. Depending on the selection a *cloud platform* can be chosen accordingly to provide the selected languages and frameworks.
2. **Database offering**: A database is an essential part of many applications. There is a wide range of database implementations offered by the *cloud platforms*. They range from the popular *SQL* to the emerging *NoSQL* [99] databases.

3. **Cloud storage service:** Apart from the use of a database, an application may be required to store other files as binary large objects ("Blobs"). Some platforms offer this possibility via a storage service.
4. **Deployment utilities:** Once the application is developed, it needs to be deployed on the *cloud platform*. Platforms may offer a command line tool or, additionally, a plug–in for a popular development tool such as *Eclipse*. In some platforms, where the application is developed online via a web-browser, the deployment is realised automatically through the web-browser.
5. **Development style:** Depending on the characteristics of the platform and especially on the platform`s application scope and the application development time, the development tools that are available to the users may vary. In general, platforms with generic application scope integrate their *Software Development Kit (SDK)* with a popular *IDE*, such as *Eclipse*. Platforms with specific application scope usually offer an online development environment via a web-browser.
6. **Scope of applications**: *Cloud platforms* may vary according to the scope of application that a developer can create. There are generic platforms where the developer can deploy any source code provided that it is compatible with the technologies offered by the platform. On the other hand there are application specific platforms that are specialised in certain application domains such as platforms devoted to *CRM* systems. Such platforms do not require any source code. Instead developers use the available tools provided by the platform.
7. **Platform specific functionality via *APIs*:** In some *cloud platforms* the functionality offered by applications can be enhanced in planned ways by integrating available 3rd party applications via *APIs*. In this case the platforms act as a framework and a marketplace where *ISVs* can offer their 3rd party applications. The users are able to combine these applications in order to build their own products.
8. **Application development time:** The time that a user needs in order to create an application varies across the *cloud platforms*. It is closely related to the previous characteristic, namely the platform`s application scope. In the case where the platform has a generic scope, application development normally takes more time because the user needs to code all the functionality from

scratch. In the case where the platform is application specific, it may not be necessary for the user to write any source code at all. There are available functionality blocks or templates that the user can customise and integrate in his application via a graphical interface. In the latter case the application development time is dramatically decreased.

In this Section several features were listed that are related to the *cloud platforms.* This list is not exhaustive. However, it may provide an adequate knowledge and serve as starting point for the evaluation and comparison of the *cloud platforms*.

3.2 Cloud platforms comparison

There is a wide range of *cloud platforms* that are commercially available and new offerings are continuously emerging. The scope of this survey is not to present an exhaustive list with all the available offerings but rather to provide the reader with an insight into the different types of *cloud platform.* The aim of this research work lies in the creation of *cloud applications*, which are agnostic to the underlying target platforms. In this context the comparison of the *cloud platforms* will contribute to the understanding of the nature of the platforms and eventually lead to the selection of the *cloud platforms*, exposing similar characteristics, where this thesis will focus on.

The research interest of the author lies in the application development domain rather than in the management and the provided quality of service of the platform. Therefore, the description of the platforms provided in this Section is based on the various application development paradigms that they adopt and are evaluated against the development and deployment characteristics presented in 3.1.2.1.

The *cloud platforms* which are evaluated in the next Sections are: 1) *Rapidcloud* [100] , 2) *Shelly Cloud* [101] , 3) *OpenShift* [29], 4) *AEB* [88], 5) *GAE* [74], 6) *Heroku* [28], 7) *Engine Yard* [30], 8) *Appfog* [102], 9) *Bluemix* [103], 10) *Zoho Creator* [79], 11) *App Cloud* [90].

The selection of the platforms is representative and serves the purpose of demonstrating the diverse set of technologies and programming paradigms that the platforms may offer.

Table 1 lists the *cloud platforms,* considered in this survey, according to their development and deployment characteristics.

Specifically, regarding the programming languages and frameworks different platforms may support different languages and frameworks such as *Java, Ruby* and *.NET.* In addition, platforms such as *Zoho Creator* and *App Cloud* do not offer any programming support. Instead they choose to provide a graphical environment that developers can exploit to developer their *cloud applications*.

A similar diversity is observed in the database offerings. Specifically, different platforms support different set of *SQL* and *NoSQL* databases. For example *AEB* offers, among others, *SimpleDB* [122], while *GAE* offers the proprietary *Cloud SQL* [124] and *App Engine Datastore* [125].

Regarding the Cloud storage service, there are platforms, such as *AEB* and *GAE,* which provide the developers with storage space. By contrast platforms such as *Rapidcloud* and *Appfog* do not offer any storage service.

With respect to the deployment utilities examples of available options are an *IDE* plugin and a *Command Line Interface (CLI).* For example *Openshift, GAE and AEB* provide both a *CLI* and an *IDE* plug-in. By contrast, *App Cloud* and *Zoho Creator* do not offer any deployment utilities since developers are not expected to deploy any source code.

The development style may also vary depending on the *cloud platform.* The majority of the offerings, such as *Heroku, Openshift, GAE,* and *AEB* enable developers to create applications by writing source code using a programming language. On the other hand, platforms such as *Zoho Creator and App Cloud* offer the developers a graphical interface that the latter can use to create *cloud applications.*

Depending on the development style the scope of the *cloud applications,* which can be developed in each platform, is determined. For example platforms, such as *Zoho Creator* and *App Cloud,* which offer a graphical development environment, support the creation of CRM-oriented applications.

In addition, certain platforms may choose to offer platform specific functionality via *APIs* that developers can exploit, such as e-mail service, image service and payment service. Examples of such platforms are *Openshift, Heroku* and *EngineYard.*

Depending on whether the *cloud platform* offers platform specific functionality the expected application development time may also vary.

3.3 Classification of Cloud Platforms

Table 1 summarises the classificatory features that are related to the development and deployment of a *cloud application* and evaluates the *cloud platforms* against these features. Whereas in Section 3.2 a selection of the most widely used platforms was presented, Table 1 adds to the list of available *cloud platforms* with further less well-known offerings. Subsequently, a classification of these *cloud platforms* is attempted.

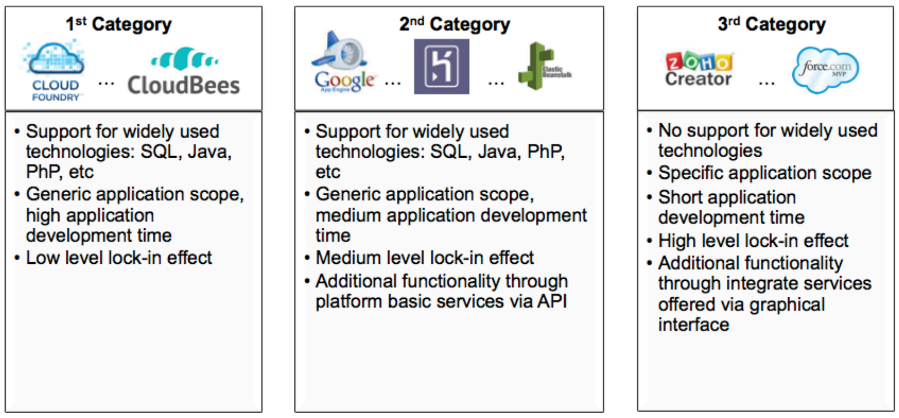
Table 1: Cloud platforms with respect to the application development features

Features / Cloud Platforms	Programming languages and frameworks	Database offerings	Cloud storage service	Deployment utilities	Development via graphical user interface	Platform application scope	Platform specific functionality via APIs	Application development time
Rapidcloud	Java, Play	MySQL, Memcached, Redis	No	GitHub, Bitbucket	No	Generic	No	High
Shelly Cloud	Ruby, Grape, Rack, Rails, Sinatra	MySQL, PostgreSQL, MongoDB, Redis		CLI				
Apprenda	Java, .NET	Microsoft SQL, Oracle RDBMS	No	CLI, Visual Studio plugin	No	Generic	No	High
ConPaas	Java, PHP	MySQL, Scalarix NoSQL	No	CLI	No	Generic	No	High
OpenShift	Java, PHP, Ruby, Node.js,Python, Perl	MySQL, PostgreSQL, MongoDB	Yes	CLI, Eclipse plugin	No	Generic	Yes	Medium
Amazon Elastic Beanstalk	Java, PHP, Python, Ruby, Node, .NET	Amazon RDS PostgreSQL, DynamoDB, SimpleDB	Yes	CLI, Eclipse plugin	No	Generic	Yes	Medium
GAE	Java, Python, PHP, Go	Cloud SQL, App Engine Datastore	Yes	CLI, Eclipse plugin	No	Generic	Yes	Medium
Microsoft Azure	.NET, Node.js, Java, Python, Ruby, PHP	AzureSQL, DocumentDB, Redis	Yes	CLI, Eclipse plugin, Visual Studio plugin	No	Generic	Yes	Medium
Heroku	Java, Node.js, Python, Ruby, PHP	PostgreSQL	Yes	CLI, Eclipse plugin	No	Generic	Yes	Medium
Engine Yard	PHP, Ruby, Node.js	PostgreSQL, Redis, MySQL	Yes	CLI, GitHub	No	Generic	Yes	Medium

Features / Cloud Platforms	Programming languages and frameworks	Database offerings	Cloud storage service	Deployment utilities	Development via graphical user interface	Platform application scope	Platform specific functionality via APIs	Application development time
Appfog	Java, Node.js, PHP, Python, Ruby	PostgreSQL, MySQL, MongoDB, Redis	No	CLI	No	Generic	Yes	Medium
Bluemix	Java, Node.js, PHP, Python, Ruby,	MySQL, PostgreSQL, MongoDB, Redis	Yes	CLI, Eclipse plugin	No	Generic	Yes	Medium
Jelastic	Java, PHP, Ruby, Node.js, Python,	MySQL, MongoDB, Neo4j, Redis, PostgreSQL	Yes	Git, Bitbucket, Eclipse plugin,	No	Generic	Yes	Medium
AppHarbor	.NET	MySQL, Microsoft SQL,	No	Git, Bitbucket,	No	Generic	Yes	Medium
Pivotal Cloud Foundry	Java, Ruby, Python, Go, PHP, Node.js	MySQL, Redis, Cassandra, MongoDB, Neo4j	Yes	CLI, Eclipse plugin	No	Generic	Yes	Medium
Standing Cloud	Java, PHP, Ruby, Python	MySQL, PostgreSQL, Redis, MongoDB, Memcached	Yes	CLI	No	Generic	Yes	Medium
OracleCloudPaaS	Java, Node.js	Oracle SQL, Oracle NoSQL	Yes	CLI, Eclipse plugin	No	Generic	Yes	Medium
Zoho Creator	Deluge	Custom database via GUI	Yes	None	Yes	Specific	No	Low
App Cloud	Apex	Custom database via GUI	Yes	None	Yes	Specific	No	Low
Caspio	-	Custom database via GUI	Yes	None	Yes	Specific	No	Low
Rollbase	-	Custom database via GUI	Yes	None	Yes	Specific	No	Low

Based on Table 1, we anticipate that some platforms may achieve short application development times, while in some others the developer is expected to spend a relatively longer amount of time in order to create the application. The platforms with shorter expected development time offer custom proprietary technologies, functionality via a graphical interface and may not support the deployment of generic applications written in a standard programming language. On the other hand the platforms with relatively longer development time provide support for open source technologies and they do not offer additional functionality via platform specific services.

Therefore the surveyed *cloud platforms* may be classified into three categories (Figure 7). The classification is primarily based upon the application development time and whether the *cloud platforms* provide additional functionality in order to speed up the creation of the applications.



1st Category
- Support for widely used technologies: SQL, Java, PhP, etc
- Generic application scope, high application development time
- Low level lock-in effect

2nd Category
- Support for widely used technologies: SQL, Java, PhP, etc
- Generic application scope, medium application development time
- Medium level lock-in effect
- Additional functionality through platform basic services via API

3rd Category
- No support for widely used technologies
- Specific application scope
- Short application development time
- High level lock-in effect
- Additional functionality through integrate services offered via graphical interface

Figure 7: Classification of Cloud platforms

1. The first category includes the platforms that provide support for widely used and open source technologies. Developers can code their applications using standard programming languages and databases offerings. Platforms in this category have a generic application scope and users can upload the source code of their application. They do not provide additional custom functionality via *APIs*, which in turn increases the application development time. However, the fact that they offer only standard programming technology without native *APIs* minimises the dependency of the application on any specific platform and thereby the vendor lock-in effect. Furthermore platforms, which fall in

this category, may be preferred by experienced developers, who are familiar in developing applications using traditional programming tools and languages. Moreover, they may be preferable in the case of existing applications which are deployed on a *cloud platform* in order to save management time and cost. Examples of *cloud platforms* in this category are *Rapidcloud* and *Shelly Cloud*.

2. The second category includes platforms that, similar to the first category, offer standard programming languages and databases, such as *Java* and *MySQL*. However, in order to decrease the application development time they also offer platform specific services via *APIs,* such as the payment and the e-mail service. Developers may exploit these services to speed up the creation of the application. However, the more such services are used by an application, the bigger the dependency is upon the platform. Cloud solutions in this category may suit developers with coding experience that need to rapidly develop new applications and therefore use ready functionality offered by the platforms. Examples of platforms in this category are *Heroku*, *Engine Yard* and *Openshift*.

3. The third category includes platforms that adopt a different application development paradigm, characterised by tools for online development via a web browser, using visual interfaces and design templates. Developers are provided with a generic graphical application framework that they can customise in order to meet their requirements. These platforms have a specific application scope that is oriented in *CRM* systems and similar business applications. The development time can be dramatically decreased due to the automated development processes. However, this is done at the expense of a high dependency of the application upon the platform and the limited scope of applications. Since little or no coding is required in order to create an application, these platforms are suitable for business experts that need to develop rapidly office applications, from scratch, with little or no coding experience at all. Examples of *cloud platforms* in this category are *App Cloud* and *Zoho Creator*.

3.4 Determining the target cloud platform category of this research work

It becomes obvious that there are significant variations between *cloud platform* offerings available on the market. As it has already been mentioned in the Introduction of the thesis, the large heterogeneity among the platform providers may hinder the engineering of a solution enabling the development of platform agnostic applications across the whole spectrum of the available platforms. Therefore, the effort of this research work needs to be concentrated on a specific cluster of platforms that present similar characteristics.

The first category of *cloud application* platforms consists of offerings that are strongly characterised by the use of standard and widely adopted technologies. Therefore, the *cloud application* development process does not deviate from the traditional programming style, which imposes the use of established programming languages and tools such as *Java*, *MySQL* and the *Eclipse IDE*. On the other side of the spectrum, the third category of platforms comprises offerings adopting highly proprietary environments, which do not allow the deployment of any source code. Instead the whole development phase takes place via web-based graphical tools. As a result no programmatic solution can be engineered to address the development of *cloud applications* targeting this category of platforms.

Consequently, the focus of the presented research work is on the second category of *cloud platforms*, namely the ones that offer additional platform services via proprietary *APIs* allowing at the same time developers to create and deploy their own source code. Platforms in this category are also known as *cloud application platforms* [32].

3.5 Cloud application platforms and platform basic services

As mentioned in the previous Section and also noted in [32], a *cloud application platform* is a special category of *cloud platforms*. As shown in Figure 8, the key characteristic of the platforms of this kind is that, apart from the basic platform resources such as run-time environments and databases, they offer additional platform specific functionality also referred to as *platform basic services*.

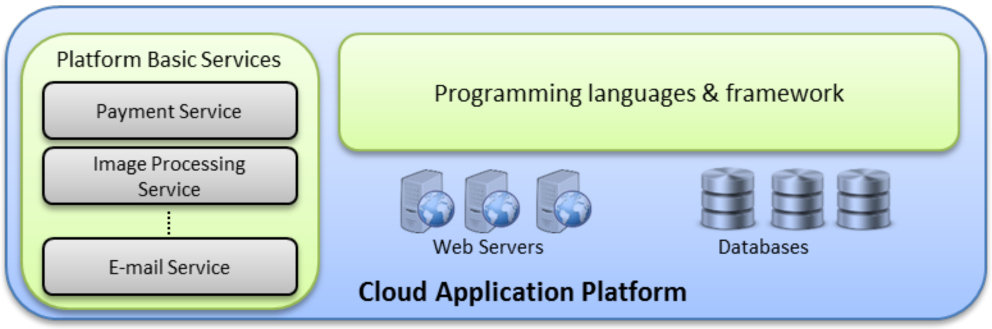


Figure 8: Cloud application Platform

The *platform basic services* expose a certain functionality, which can be exploited by software developers in order to speed up the process of application development. They are usually provisioned via a *REST API* via the marketplace of the platform. A marketplace allows *ISVs* to create their own services and provision them through the *cloud platform*. Such an example is the marketplace of *OpenShift* and *Heroku*.

Examples of *platform basic services* are the payment service, the image processing service and the e-mail service:

1. **Payment service**: A payment service undertakes the task to perform electronic transactions via credit or debit cards. This service can be used by *cloud applications*, which sell products or services online. Instead of having developers create from scratch the functionality, which handles electronic transactions, a payment service can be exploited to complete the operation. Example of payment service providers are *Stripe* [39] and *Spreedly* [38] offered via *Heroku* platform
2. **Image processing service**: This service offloads the task for the application developers to perform image transformations, such as crop, resize, apply a filter etc. Instead these operations are carried out by the service, in this way saving application development time and processing power. Example of image service providers are *Google* and *Cloudinary* [130], which is offered via *Heroku* and *Engine Yard*.

3. **E-mail service**: As the name implies, the E-mail service enables the application to send, receive, and store e-mails without the need for the developers to set up and maintain a dedicated e-mailing server. Rather this task is outsourced to the e-mail service providers. Example of e-mail providers are *Google*, *Amazon,* and *SendGrid* via *Heroku*.

The services mentioned above are only representative examples of what a *platform basic service* may look like. There are several *cloud application platforms* offering a constantly growing number of *platform basic services* as mentioned in the previous Sections. For an extended list of available *platform basic services* the reader may look up the *Heroku* marketplace [131] as well as the marketplace offered by *Engine Yard* [132] and *OpenShift* [133]. Along with the increase in the number of available *cloud application platforms*, there is a steady proliferation in the number of *platform basic services* offered via the platforms. Currently, *Heroku* offers almost 150 services, *OpenShift*, which recently (2014) launched its own marketplace, counts almost 30 services, and *Engine Yard* provides 64 *platform basic services*. The on-going increase of the available *platform basic services* and their growing popularity among application developers is leading to a new paradigm of application development, where applications are not created from the ground up, but rather are synthesised from a number of *platform basic services*. Figure 9 introduces the term *service-based cloud application,* which describes a software application, which is deployed on a *cloud application platform (CAP)* and utilises a number of available *platform basic services*.

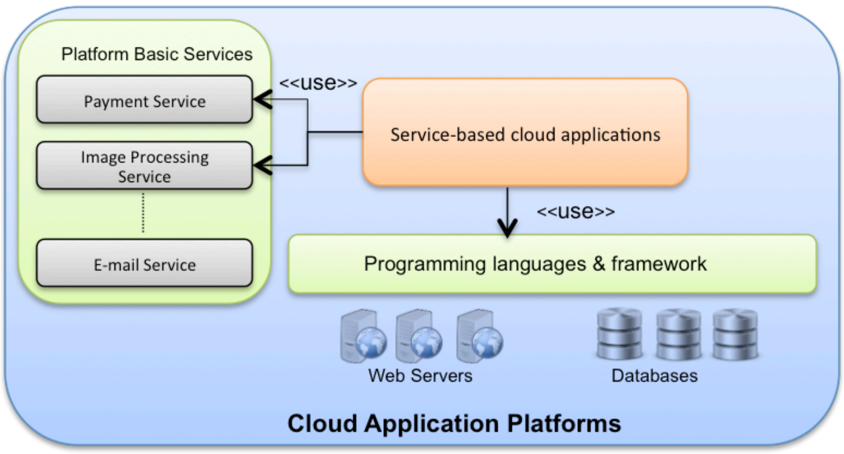


Figure 9: Service-based cloud applications

At the initial stage of this research work, there were a limited number of *CAPs*, such as the *GAE* and the *Heroku*. However, over the course of the research work, additional *CAPs* have constantly been launched such as *Engine Yard*, *Appfog*, *Jelastic* and *AppHarbor*. This fact highlights the momentum that *platform basic services* gain and their impact in the domain of *cloud application* development. Therefore, the presented research work focuses on these *cloud application platforms*, namely the second category of the classification performed in Section 3.3 and specifically on methodologies and techniques which enable the development of *service-based cloud applications* as those that have been defined in this Section.

3.6 Summary

This chapter presented a survey of *cloud platform* offerings. Due to the large heterogeneity among the available offerings, the survey contributed to the clarification of the differences between the various platforms and to further contextualization of the presented research work. Since the research focus lies on the cross-platform development and deployment of *cloud applications,* the survey was based on certain platform characteristics related to the development and deployment phase.

Based on the analysis which followed, the survey concluded that there are three relevant categories of *cloud platforms*, when considering the problem of cross-platform development. The first one includes *cloud platforms, which* adopt widely used programming technologies and have a generic application scope. They do not offer any *platform basic services* and thus they have a relatively high application development time. The second category presents similar characteristic with the first one. However, in order to speed up the development process, platforms in this category offer additional *platform basic services* via *APIs*, which can be exploited by the developers. The third category includes platforms, which adopt a different application development paradigm. Rather than deploying their source code, developers create their applications based on web-based graphical environment.

This research work chooses to focus on the second category of platforms, also referred to as *cloud application platforms.* The reason is that they promote the traditional programming style and the deployment of source code, while at the same time they attempt to provide additional proprietary functionality via the use of so-called *platform basic services*. The combination of both open source and proprietary technology and the way that application developers can exploit it, motivated the choice of the presented research work. Moreover, as mentioned in Section 3.5, *cloud application platforms* gain momentum and new offerings are constantly launched.

After having surveyed the field of *cloud platforms* and concluded to the concrete category where this research work will focus on, the next step is the survey and analysis of the related work on the field of the cross-platform development. The aim of the literature review is to identify potential gaps in the research areas covered by the related work and to expose any weaknesses involved. Thus the review will further contribute to the precise contextualization of this research work with respect to the concrete area that the work will focus on and the approach to be adopted in order to address the issue of cross-platform development.

Chapter 4

Related work on the field of cross-platform development of service-based cloud applications

The preceding Chapters have introduced the scope of this research work, which is the cross-platform development of *cloud applications*. Particularly, specific concepts related to the research work have been defined, such as the *platform basic services, the cloud application platforms,* and the *service-based cloud applications.* Due to the diverse ecosystem of the *cloud platforms*, as shown in Chapter 2 and Chapter 3, this research work focuses on a specific cluster of platforms exposing similar characteristics with respect to the cloud application development. For this reason, Chapter 3 presented a survey of the field of the *cloud platforms*, which in turn led to the classification of the platforms into three categories based on their development paradigm and on the additional functionality, which they may offer to the developers. As discussed in Section 3.3, the first category includes the platforms, which adopt widely used technologies and basic development capabilities without offering additional custom functionality. At the other end of the spectrum, the third category contains offerings, which adopt a high-level graphical development paradigm including narrowed scope and proprietary functionality. The middle category, where this research thesis focuses on, includes the platforms, which provide support for widely used programming technologies and additionally offer custom functionality via the use of the *platform basic services.* These platforms are also known as *cloud application platforms.*

In order to proceed with the exploration of the field of cross-platform development of *cloud applications*, a review of existing work needs to be carried out. The aim of the review is to reveal the precise context of the relevant work on the field including the methodologies and tools that have been used. At the same time the strengths and limitations of the examined work will lead to the refinement of the context of the presented research and contribute to the selection of the appropriate approach and tools in order to advance the presented research topic. Therefore, the aim of this Chapter is two-fold: First, to identify potential gaps in the areas explored by related work and second to detect any weaknesses in the adopted methodologies.

Two general approaches have been adopted in order to address cross-platform development and enable the software engineers to leverage resources offered by the various *cloud platforms*, namely the standardisation and the intermediation approach.

Section 4.1 examines the standardisation approach. This approach focuses on the definition of common set of standards for the cloud offerings. The adoption of such standards by all cloud providers would enable developers to create their applications independently of specific cloud environments and then deploy them on the cloud provider of their choice. However, for reasons not necessarily related to technology, it is very difficult for all cloud providers to eventually agree on a common set of standards. Therefore, as it has already been mentioned in the Introduction of the thesis, this research work focuses on the intermediation approach. The field of standardisation is briefly examined in order to give the reader an insight on the reasons that led this research work adopt the intermediation rather than the standardisation approach.

Section 4.2 examines the alternative approach towards enabling the development of platform agnostic *cloud applications*, namely the intermediation. Contrary to the standardisation approach where the consensus of the platform vendors is required, intermediation focuses on the introduction of an intermediate layer that decouples application development from specific platform *APIs* and supported formats. This approach can be applied using mainly three techniques: (i) library-based solutions, (ii) middleware platforms and (iii) model-driven engineering (MDE) techniques. The techniques and related work adopting each of the techniques are discussed in the Sections 4.2.3, 4.2.4 and 4.2.5 respectively. Next, Section 4.2.6 positions the examined work with respect to the three service models, the *IaaS*, the *PaaS*, and the *SaaS*. The conclusion, which is drawn in Section 4.2.7, is that the work carried out in the *PaaS* model is significantly less compared to the *IaaS*. Moreover, little or no work has been done towards addressing the variability across the various *platform basic services* and the respective providers.

4.1 Standardisation approach

Cloud computing covers a wide area of heterogeneous offered services. It is not yet clearly defined what are the exact services and in which form they are offered to the users. Moreover, cloud services are provided to the users via proprietary *APIs* and technologies that vary across the cloud providers. Consequently, confusion may be caused to the users regarding the most suitable service and the feasibility to move across cloud providers without being enforced to comply by the specific technology and *APIs* of each provider. In an effort to reduce the confusion and establish trust towards cloud computing services, several organizations undertake the task of putting forward standards related to the way that cloud services shall operate and be offered to the users.

4.1.1 Cloud Standardisation efforts

4.1.1.1 *Open Virtualisation Format (OVF)*

Open Virtualization Format (OVF) is a specification for packaging and distributing Virtual Machines (VMs), defined by the *Distributed Management Task Force (DMTF)* [134]. Conflicts may occur when trying to port a VM from one vendor to another due to the proprietary formats. *OVF* attempts to bridge the differences among the vendors by putting forward a standard format for VMs. Among the key properties of *OVF* is that it is platform independent. Thus the architecture of the format is not bound to a particular platform or operating system and so enabling virtual machines to be deployed on different cloud infrastructure providers.

4.1.1.2 *Open Cloud Compute Interface (OCCI)*

Open Cloud Computing Interface (OCCI) is a standard created by *Open Grid Forum (OGF)*. *OCCI* attempts to standardise the way users access and manage infrastructure resources and therefore to abstract the various proprietary *APIs* that vendors are currently using. *OCCI* defines the following infrastructure resources, which can also be viewed in the technical report "Open Cloud Computing Interface – Infrastructure" [135]:

1. **Compute:** Compute resources refer to VMs. Operations defined in the *OCCI API* include: start, stop, restart, and suspend a virtual machine.
2. **Storage:** Storage resources refer to data storage devices. The actions that can be performed are: set the device online or offline, create a backup, take a snapshot or resize the storage space.
3. **Network:** Network interconnects the available resources. Users can set the network to active or inactive.

4.1.1.3 *Unified Cloud Interface (UCI)*

The *Unified Cloud Interface (UCI)* [136] was proposed by the *Cloud Computing Interoperability Forum (CCIF)* [137] in order to create an *API* for abstracting resources offered by various cloud providers. Similar to *OCCI* the goal of the *UCI* was to abstract infrastructure resources. On top of that, it aimed at the higher service levels such as the *PaaS*. The use of semantic technologies such as *OWL* and *RDF* would enable the definition of a standardised model of the cloud computing stack. However, *UCI* remains inactive since 2010 and no further information is provided. It has been included in this chapter for historic reasons and for the sake of completeness.

4.1.1.4 *Cloud Infrastructure Management Interface (CIMI)*

The *Cloud Infrastructure Management Interface (CIMI)* [138] has been proposed by the *DMTF* [139] as a standardised management interface for Infrastructure resources. Cloud Infrastructure consumers can leverage *CIMI* in order to manage machine, network, and monitoring resources. Specifically, operations that are supported by the standard include creation of VMs, start and stop the VMs as well obtaining credentials for logging on. Consumers can also manage the network resources for creating virtual networks and connecting storage devices.

The focus of *CIMI* is on the Infrastructure service level and does not extend beyond that to other levels such as the *PaaS* or *SaaS*.

4.1.1.5 *Cloud Data Management Interface (CDMI)*

Cloud Data Management Interface (CDMI) is a cloud storage standard defined by *Storage Networking Industry Association (SNIA)* [140]. Cloud storage service can be used to store files. Several cloud providers offer storage service, such as *Google Storage*, *Amazon Simple 3,* and *Azure Blob*. However, each provider is using a proprietary *API* to let developers use the service. Consequently, users need to comply each time with the provider`s specific *API*. *CDMI* attempts to standardize the way users, access and manage cloud storage services. It defines a restful *HTTP* interface through which clients can access and manage the resources of the cloud storage providers.

4.1.1.6 Topology and Specification for *Cloud applications (TOSCA)*

Contrary to the previously mentioned standards, which mainly focus on the *IaaS* level, *TOSCA* is a standardisation effort from *OASIS (Advancing Open Standards for the Information Society)* [141], which aims at the cross-deployment of *cloud applications* Even cloud platform services that consist of standard technologies, widely supported by multiple *cloud platforms*, require a certain level of configuration and human intervention before being deployed across *cloud platforms*. *TOSCA* envisions automating the process of deploying an application across platforms. Essentially the proposed standard attempts to specify a uniform way to define the resources that a cloud application relies on, such as web servers and databases, so that the latter are automatically instantiated and configured by the target *cloud platform*.

4.1.1.7 *Cloud application Management Platform (CAMP)*

The *Cloud application Management Platform (CAMP)* [142] is a standard proposed by *OASIS*, aiming at the management of the *cloud applications* across various *cloud platforms*. Specifically, it defines a RESTful *API* which is language and framework agnostic and allows users to perform certain management operations independent of the target platform [143]. The *API* covers the whole lifecycle of the application and can be utilised by the users in order to deploy the application on the platform, start and stop it, as well as monitor and update the application with new versions [144].

CAMP is closely related with *TOSCA* described in Section 4.1.1.6 since they have been both initiated by the same organisation, *OASIS*. While *TOSCA* defines a standard way to describe the resources of a *cloud application* such as the web servers, the database etc. and its topology, *CAMP* aims at the definition of a uniform *API* in order to manage the deployment and execution of the application.

4.1.2 Positioning of the cloud standards based on the cloud computing service model

As described in the previous Sections, there are several standardisation efforts, which span across the levels of cloud computing (*IaaS, PaaS, SaaS*). This Section positions the described standards with respect to the service level that they focus on and subsequently Section 4.1.3 attempts to identify whether there is an established cloud standard addressing the cross-platform development of *cloud applications* which can be exploited by the presented research work.

Figure 10 positions the standardisation approaches across the three service levels. Specifically, *OVF* is positioned in the *IaaS* level since it addresses the virtual machine image migration.

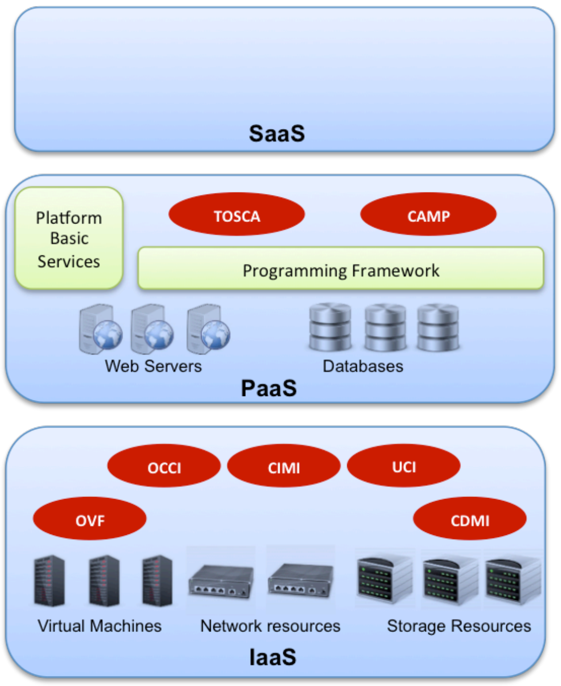


Figure 10: Cloud computing standards classification

CDMI addresses the cloud storage resources migration and management and thus is also placed in the *IaaS* level. At the same level are positioned *OCCI*, *CIMI,* and *UCI* which all focus on the management of the infrastructure cloud resources such as compute and networking.

TOSCA and *CAMP* are both related with the deployment of the *cloud applications* on the *cloud platforms* and are placed on the *PaaS* level. The first one defines a standard way to describe the application in order to make it deployable across multiple cloud providers while the latter addresses the *cloud application* management operations.

4.1.3 Standardisation approach with respect to cross-platform development of cloud applications

The adoption of the standards, described in the previous Sections, by all the cloud providers would enable users to utilise cloud resources in a uniform way independently of the specific provider environment. Particularly, in the context of this research work standardisation could enable the development of *cloud applications* based on standards and therefore "shield" them against platform proprietary technologies. However, there are two main reasons why standardisation approach could not be leveraged by this research thesis:

i. The establishment of a standard is a strenuous and time-consuming process requiring the consensus the interested stakeholders.
ii. The majority of the existing cloud standards focus on the *IaaS* rather than on the *PaaS* level.

Cloud providers may appear reluctant to adopt and agree in common standards since the direct competition among the providers will be increased [145] [146] [147]. Proprietary *APIs* and technologies is a way to increase clients' reluctance to move to another provider (i.e. lock-in). Furthermore, a detailed definition of a standard requires the consensus of the providers on technical details. Even when* providers are willing to collaborate, the whole process is strenuous and time consuming due to the complexity at the technical level. Finally, the definition of a quality standard, which brings benefit to the stakeholder, requires skills, judgement and experience of the subject matter. Lack of these qualities may result in the definition of poor standards which act as an impediment rather than catalyst for the exploitation of the subject technology [145].

In addition to the issues related with the creation of standards, the existing standardisation efforts have not reached a maturity level sufficient to overcome the heterogeneities which rises due to the proprietary vendor *APIs* and technologies [146]. Similar to the web service standards which had been formulated during the first decade of 2000, time is needed for a well-defined and established set of cloud standards to emerge [47].

As seen in Figure 10 and also pinpointed in several articles [47] [148] [149], most standardisation efforts are taking place at the IaaS level. *NIST* further highlights this issue by identifying the areas of standardisation gaps. It is clearly shown a gap in the *PaaS* and *SaaS* level with respect to the available standards. Only in the recent years standards start to emerge at the *PaaS* level, whereas *SaaS* is still largely unexplored.

Lewis [47] tries to explain the fact why the majority of standards cover the *IaaS* level. It is primarily mentioned that the complexity of the provisioning of the cloud offerings at the *IaaS* level is not insurmountable and therefore an IT department can extend its resources with additional ones offered by the cloud providers. Moreover, there is not much heterogeneity among the *IaaS* offerings other than the pricing and the *SLAs*. Therefore, the definition of standards at the *IaaS* level is feasible. However, while moving upwards in the cloud computing stack the complexity in the offerings rise. Thus, at the *PaaS* level an organisation needs to make a decision on programming languages and frameworks, deployment options, specific functionality offered via *platform basic services*. As a result the heterogeneity among the platform providers rises. Therefore, standardisation efforts at this level become more challenging.

This Section discussed the standardisation efforts at the three service levels of cloud computing. It became clear that although standardisation is an efficient approach to promote the wide exploitation of cloud offerings and enable the cross-platform development of *cloud applications,* there are still not established and widely supported set of standards. Moreover, the majority of the approaches focus on the *IaaS* level. Efforts at the *PaaS* level are limited and focus mainly on the deployment process of the *cloud applications*. With respect to the Figure 10 there are no standards aiming at the exploitation of the *platform basic services* as those offered by the *cloud application platforms*.

4.2 Intermediation approach

While standardisation approach constitutes an efficient solution, which allows users to exploit cloud services in a uniform way, it was shown in the previous Section that there are significant limitations, which dictate the enforcement of alternative directions. Such an alternative direction is the intermediation approach. This is, introducing an intermediate layer, which decouples application development from specific platform provider technologies. The intermediate layer prevents developers from being bound to specific platforms *APIs*, programming languages, or *platform basic services*.

Prior describing specific work exploiting the intermediation approach, the scope of the presented research work needs to be further narrowed down.

4.2.1 Position of the research work with respect to the cloud application lifecycle

In the literature there is extensive work [148] [150] [151] carried out describing the various phases of the *cloud application* development and management. According to this the lifecycle of the *cloud application* primarily consists of three main *phases: (i)* The development phase, (ii) the deployment phase, and (iii) the execution of the application. For each phase certain aspects have been identified (Figure 11). This Section attempts to illustrate the *cloud application* lifecycle and subsequently to identify the concrete aspect where this research work focuses on.

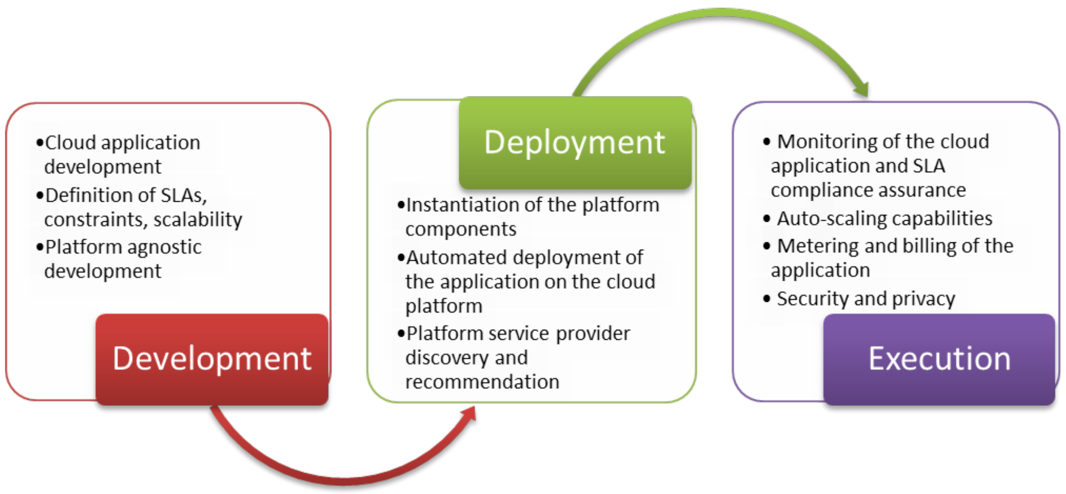

•Cloud application development
•Definition of SLAs, constraints, scalability
•Platform agnostic development
Development
Deployment
•Instantiation of the platform components
•Automated deployment of the application on the cloud platform
•Platform service provider discovery and recommendation
• Monitoring of the cloud application and SLA compliance assurance
• Auto-scaling capabilities
• Metering and billing of the application
• Security and privacy
Execution

Figure 11: Aspects during the cloud application lifecycle

Regarding the development phase, Maximilien et al. [150] from *IBM* focus on the creation of platform agnostic application able to leverage multiple concrete providers. *PaaSage* [151], a European research project, highlights the definition of constraints such as scalability rules. Petcu [148] compiles a comprehensive list of non-functional requirements. Similar to the previous work the main focus lies on the independence of the *cloud application* from the specific providers' resources as well on the ability to define functional and non-functional requirements.

With respect to the deployment phase, Maximilien et al. [150] put forward the need for recommendation systems which propose the best deployment topology. *PaaSage* additionally mentions the ability to select the optimal cloud providers while Petcu, among others, pinpoints the need for automated deployment procedures and for service authentication using single sign-on.

The execution phase, according to Maximilien et al. [151], involves the requirements for automated scaling up and down of the *cloud application*, the ability to perform management operations such as back-up and restore, and the requirements related to the security and privacy issues. *PaaSage* additionally mentions the monitoring of the application and the ability to adapt the provisioning of the resources and the deployment of the application components in order to meet the constraints as those have defined during the development phase. Petcu [148] also pinpoints the above mentioned issues.

Figure 11 summarises the main aspects involved in each of the three phases of the *cloud application* lifecycle as those are mentioned in the literature and also arose based on the authors' experience on the field.

Indicatively, in the development phase the following aspects are listed:

- The *cloud application* development. It is self-explanatory and implies the design and implementation of the application taking into account the capabilities and the offerings of the *cloud platforms* such as: the available runtime environments, the data stores, the databases, and the *platform basic services*.
- The definition of *SLAs* and constraints. In order for the *cloud application* to meet a certain quality level, rules and *SLA*s needs to be defined and embedded in the application. Examples of such rules may regard the response time of the application and the resources up to which the application may scale to keep up with the quality standards.

- The development of the application in a platform provider agnostic manner. This aspect ensures that the application implementation is not bound to a specific provider technology and *API* and thus it is able to leverage multiple cloud providers.

In the deployment phase the following aspects are defined:

- Instantiation of the platform components. Based on the requirements of the application for resources, the platform components are instantiated. Such components may be the runtime container and the database instances.
- Automated deployment of the application on the *cloud platform.* The various components of the *cloud application* are automatically deployed on the platform resources, which have previously been instantiated.
- Discovery and recommendation of platform service providers. There is a plethora of available platform service offerings [48]. Recommendation systems undertake the task to find and recommend the optimal offering which best meets the requirements of the application.

In the execution phase the aspects, which arise, are:

- Monitoring of the *cloud application* and reassurance of compliance to the *SLA*s. This implies monitoring the application performance and checking whether certain constraints or *SLA*s are violated.
- Based on the monitoring, which takes place, the application should be able to adapt itself in order to ensure compliance to the defined constraints. Example of adaptation action is the ability of the application to scale up and down according to the incoming workload.
- Metering and billing of the application. The resources that a *cloud application* consumes are metered and the total incurring cost is calculated.
- Security and Privacy. Since the advent of the cloud computing, security and privacy are important impediments to its widely adoption [152]. Cloud users are concerned with the level of security and privacy that is applied to their applications and their data.

The scope of this research work lies primarily in the development phase and particularly in the platform agnostic development of the *cloud applications*. This means that the applications are not bound to a specific platform provider during the design and implementation. The primary motivation stems from the need to alleviate the burden from the developers to study and cope with the peculiarities of each platform provider. The main focus of the developers should be the development of the business logic of the application and thus the specific platform technologies and APIs should be transparent to them.

In addition, Petcu [148] summarises the benefits of creating platform agnostic applications which can leverage resources from multiple platform environments.

- Quality assurance may be easier to be achieved and maintained. In case that a platform service provider fails to live up to the agreed *SLA*s, the application may be deployed on a different platform provider.
- The fluctuation of the prices may be another determinant for the concrete service provider. Sudden increase in the price of a service may lead the application developers to choose an alternative service provider.
- Legal constraints may impose that different service providers are selected in various geographical regions.
- In case of a sudden increase in the workload the application should be capable of consuming resources from an alternative platform provider.

In the next Sections, representative work aiming at enabling the development of *cloud platform* agnostic applications is presented.

4.2.2 Clustering of work promoting cloud platform agnostic applications

In the literature there is a variety of research efforts contributing to the development of platform agnostic applications by following various solution directions. Petcu and Vasilakos [153], among others, classify the research efforts in library-based and model-driven oriented solutions. The former ones provide the developers with an intermediate *API* which is provider agnostic and abstracts various concrete platform providers. The latter solutions deploy model-driven engineering (MDE) techniques in order to enable the creation of platform independent models, which are then undergoing model transformations to target specific platform environments.

Similar to Petcu and Vasilakos [153], Ferry et al. [155], among others, highlight approaches based on MDE techniques and library-based solutions.

Guillen et al. [154] also put forward the idea of exploring MDE techniques in the development of *cloud platform* agnostic applications. In addition, they refer to middleware solutions, which create an abstraction layer between the deployed application and the target platform environment. Based on the previously mentioned solution directions the work presented in the next Sections are clustered in three categories, namely: *i) Library-based solutions, ii) middleware solutions,* and *iii) MDE-based solutions.*

4.2.3 Library-based solutions

Library-based solutions are code wrappers, which as the name implies wrap the specific provider implementations and provide a single common *API* to the developers.

Code or library wrappers, as it is also mentioned in [156], are objects or piece of code which encapsulates other objects or piece of code respectively controlling this way the methods execution. In other words library wrappers hide the detailed implementation of the specific technology and expose to the user a single interface, which is independent of the underlying implementation details. The ability of "hiding" the implementation details is what promotes the platform independence since the user is able to take advantage of multiple different implementations without the need to adjust the source code each time to the respective implementation. The concept of wrapper libraries has been extensively used by software engineers and also has been endorsed as best practices in relevant literature for the creation of software which is independent of the deployed environment [156] [157] [158] [159].

Example of code wrappers are database gateways, which have been widely used to enable data wrapping [156]. Data wrapping refers to the uniform access to multiple databases using a single interface. There are multiple database offerings, such as

MySQL, *PostgreSQL* etc., each one providing different native access to the database. Therefore a data wrapper offers a single interface for the developer to use while it "hides" the detailed implementation for accessing each of the databases. Widely adopted data wrappers are the *Java Database Connectivity (JDBC)* drivers [52]. *JDBC* is an *API* for *Java* language, which specifies the interface that the developers can use to access a database. A *JDBC* driver exposes the *JDBC API* and implements the access to the database. Thus, developers who adopt the *JDBC API* and use *JDBC* drivers can switch databases without the need to modify the source code of the program. Another example of data wrapper is the *Open Database Connectivity (ODBC)* drivers [160]. Similar to *JDBC, ODBC* is an *API* that defines the way users can access various databases. Contrary to *JDBC, ODBC API* was developed by *Microsoft* and primarily targets applications created with the *.NET* framework [161].

Library wrappers can be implemented using the widely adopted design patterns [162]. Design patterns describe certain classes and their interrelationships in order to address a general design problem. The use of such patterns promotes composability, maintainability, and portability. Particularly for library wrappers the following design patterns can be used:

- **Adapter:** The adapter design pattern can be used to adjust an interface of a class in order to be compatible with the one that the client expects to use. For example the client needs to perform a certain request using a specific interface. Instead of implementing the request manually, an existing library can be reused. However, the interface that the library exposes is incompatible with the client`s one. Adapter undertake the tasks to call the library`s interface on behalf of the client.
- **Strategy:** Strategy is a design pattern where the concrete implementation of an algorithm can be determined during run-time. This pattern is used when there are several concrete implementations for a specific operation and the user does not know at design time, which one will be invoked during run-time.

Next we discuss existing work in the field of cross-platform development of *cloud applications*, which adopts the approach of the library wrapper.

4.2.3.1 *jClouds*

jClouds [163] is an open source library that can be used by application developers in order to abstract cloud vendors` specific *API*. It offers two types of services: a file storage service and a compute service.

File storage service allows an application to store and read files from a remote store provided by a cloud provider. *jClouds`* storage service, called *blobstore*, consists of the following structure:

- Container: Container is the namespace for the files to be stored. It can be perceived as the top level directory.
- Blob: Blob is the unstructured data that is stored in the container. More specifically blob refers to files.
- Folder: Similar to the folders that can be created in the filesystem of a PC, *jClouds`* storage service allows the user to organise the blobs into folders.

Major cloud storage services that *jClouds* can abstract are: *Azureblob* by *Microsoft Azure* and *Amazon S3* by *Amazon*.

Regarding the Compute Service, *jClouds* provides abstraction for managing server instances. Using *jClouds* Compute API, developers can get information about running instances or create new ones. Major compute service providers that are supported by *jClouds* are: *Amazon EC2, Eucalyptus* and *Rackspace.*

From a technical point of view *jClouds* is a client implementation of the *REST API* that cloud providers expose to allow developers to use the cloud storage service. To further illustrate the internal mechanism of *jClouds* let us consider a case example. A developer needs to use the cloud storage provided by *Amazon S3*, *Google* cloud storage, and *Microsoft Azure* in order to store a file. There are three approaches to achieve this. The first one is to manually implement and perform an *HTTP* request using the properties required by each provider. For example *Amazon*`s *HTTP* request syntax is the following:

```
PUT /myfile.txt HTTP/1.1
Host: myBucket.s3.Amazonaws.com
```

```
Date: <Date>
Authorization: <AWS Authentication String>
Content-Type: text/plain
Content-Length: 11434
Expect: 100-continue
```

The equivalent *HTTP* request to *Google Cloud Storage* should look like the following:

```
PUT /myfile.txt HTTP/1.1
Host: bucket.storage.googleapis.com
Date: <Date>
Content-Length: <request body length>
Content-Type: <MIME type>
Authorization: <authentication string>
```

This approach however, requires high development effort and can be error prone. The second approach to store the file to the cloud storage is to download and use a client library from each respective cloud provider, provided that there is one available. A client library implements the previously mentioned *HTTP* requests and wraps the implementation around a method call. So the same request in *Amazon S3*, using the client Java library, looks like the following:

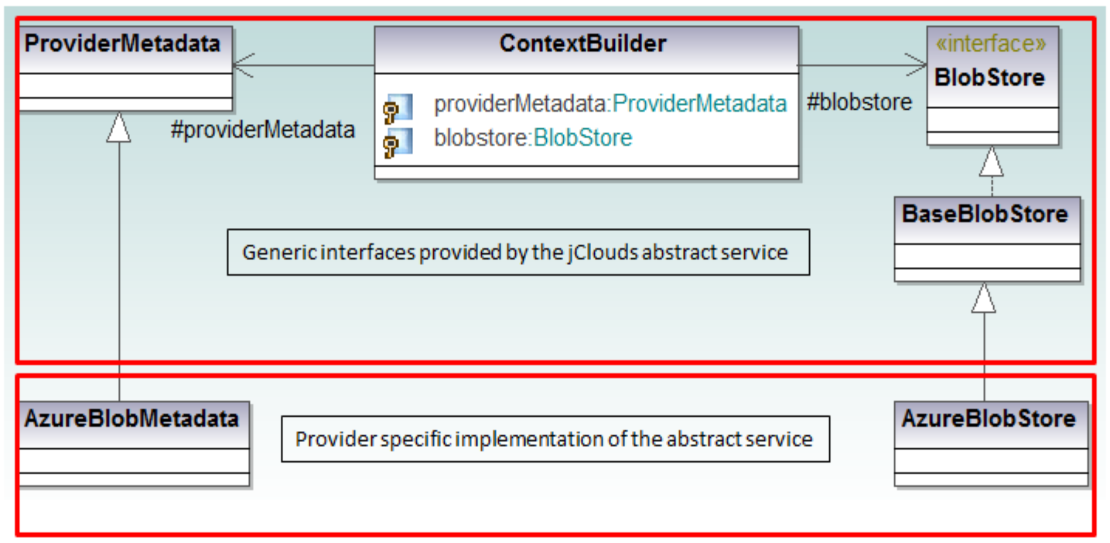
```
s3.putObject(new PutObjectRequest(<bucketName>,
<FileName>, <FileContent> ));
```

A similar method call is available in the equivalent *Google Cloud Storage* client library. Obviously the second way is much easier and faster to develop that the first one. However, for each different cloud storage service the equivalent library needs to be installed. This fact increases the library dependencies, the total size of the application, and potentially adds a performance overhead. The third way to store the file is to use *jClouds* library. This library implements the *HTTP* requests for each of the supported cloud provider such as *Amazon S3* and *Windows Azure*. So the same operation using *jClouds* could be performed using the following method call:

```
<CloudProvider>.putFile(<FileName>,<FileContent>);
```

The advantage over the second mentioned way is that developers can use the same method call to store the file in each of the supported cloud providers. Additionally, there is no need to install each cloud provider`s client library since JClouds accesses each cloud storage using direct *HTTP* requests. The deployment of *jClouds* is straightforward and requires to include the respective libraries in the developer`s project. There is relatively extended documentation that allows the developers to easily familiarize themselves with the *API*.

In Figure 12, a simplified view of the *jClouds* internal design is shown. The framework consists of the main components, which are platform independent and describe an abstract cloud service. Additionally, it contains the components that are platform specific and implement the abstract service. In this case the cloud storage, also known as blobstore service is described.



ProviderMetadata
ContextBuilder
providerMetadata:ProviderMetadata
blobstore:BlobStore
#providerMetadata
#blobstore
«interface»
BlobStore
BaseBlobStore
Generic interfaces provided by the jClouds abstract service
AzureBlobMetadata
Provider specific implementation of the abstract service
AzureBlobStore

Figure 12: Simplified view of jClouds blobstore service internal design

The main elements that are depicted in Figure 12 are the following:

- **ContextBuilder:** *ContextBuilder* is the main class that gives access to each of the supported services, namely the compute and the blobstore service. It is associated with the *ProviderMetadata* and the *Blobstore*.

- **ProviderMetadata:** *ProviderMetadata* is an interface defining a template with all the properties required by *jClouds* in order to generate the *REST* request for the providers. Examples of properties included in the interface are: provider`s name, endpoint, credentials` names, and custom properties defined for each provider.
- **BlobStore:** *BlobStore* is an interface defining the common operations that are offered by all supported providers. Examples of operations are: *putblob, getblob* and, *removeblob*
- **BaseBlobStore:** *BaseBlobStore* in an abstract class, which implements some of the operations defined in the *BlobStore* interface. Particularly it implements the ones that are commonly implemented by the providers.
- **AzureBlobStore:** *AzureBlobStore* is a concrete class, which inherits the *BaseBlobStore* class. It implements the operations that are defined in the *BlobStore* interface and are not implemented by the *BaseBlobStore* class. Additionally, it implements certain operations, which are provider specific.
- **AzureBlobMetadata:** *AzureBlobMetadata* is a concrete class that implements the *ProviderMetadata* class. It basically fills in the information about the specific provider that is needed by *jClouds* in order to generate the *REST* requests.

While *jClouds* proves to abstract efficiently the various cloud storage providers, the design solution imposes certain limitations. The implementation of the *HTTP* calls to the storage providers are hardcoded. Therefore if providers change the syntax of their *APIs*, then the solution immediately becomes outdated and potentially not functional. There is no flexibility to allow any dynamic reconfiguration of the *API.* Moreover, the design does not provide a generic architecture in order to allow future expansion and integration of more platform specific services. In terms of programming language supportability *jClouds* is limited to *Java* and *Closure* implementation. Moreover, it does not support *Google* Storage, which is one of the major file storage cloud providers. In the context of a *cloud application*, *jClouds* is limited in the file storage field, compared to other similar solutions that also provide abstraction for database and message queue services.

4.2.3.2 *LibCloud*

Similar to *jClouds*, *LibCloud* [164] offers abstraction for certain cloud resources. Particularly, using *LibCloud* application developers can access the following cloud resources:

- Cloud Servers such as *Amazon EC2* and *Rackspace* CloudServers
- Cloud Storage such as *Amazon S3* and *Rackspace* CloudFiles
- Load Balancers such as the *Amazon Elastic Balancer* and *GoGrid* LoadBalancers
- Cloud Domain Name System (DNS) services such as the *Amazon Route 53* and the *Zerigo*.

From a technical perspective *LibCloud* adopts the same design principles as *jClouds*. However, contrary to the previous solutions, *LibCloud* targets *cloud application* written in Python programming language.

4.2.3.3 *Other related cloud libraries*

The solutions listed above, namely *jClouds* and *LibCloud* are among the most prominent ones in the domain of library-based solutions and among the earliest to appear. However, in the recent years further libraries have been created to serve additional programming languages. Their design principle is similar to the above listed solutions.

For the sake of completeness of this survey we list the additional libraries.

1. **Fog** [165]: *Fog* is a library targeting applications written in *Ruby* programming language. The supported cloud services and some major respective providers are:
   a) Compute service supporting *AWS*, *Google,* and *Openstack*.
   b) Storage service supporting *AWS*, *Googl,e* and *Rackspace*.
   c) DNS service supporting *AWS*, *Rackspace,* and *IBM*
   d) Content Distribution Network (CDN) service supporting *AWS*, *Rackspace,* and *HP*.
2. **Pkgcloud** [166]: *Pkgcloud* is a library targeting applications written in node.js programming language. *Pkgcloud* provides abstraction for the following

services: (i) compute, (ii) storage, (iii) database, (iv) DNS, (v) load balancers, and (vi) network. Some major supported providers are: *Amazon*, *Windows Azure* and *Google*. A complete list of the supported services and the respective providers can be found in the [166].

3. **Elibcloud** [167]: *Elibcloud* is a wrapper around *LibCloud* and targets applications written in *Erlang* programming languages. *Eliblcoud* provides abstraction for cloud services such as: compute and key value stores. Major providers that are supported are *AWS, HP* Cloud, and *Rackspace*. A complete list of the supported services and the respective providers can be found in the [167].

4.2.3.4 *Discussion on the Library-based solutions*

Library-based solutions provide an efficient way to abstract traditional cloud resources such as cloud storage and compute service. They can be integrated easily with the *cloud application* by including the respective packages of the libraries in the software project, which is under development. In addition, they have a significantly low learning curve since the developers are only required to examine the exposed *API*. Furthermore, there are available libraries supporting the popular programming languages such as *Java, Pytho,n* and *Ruby*.

On the other hand in the library-based solutions the provider specific implementation is usually hardcoded in the source code. This means that such solutions are not easily updated and expanded with additional services. Moreover, they do not provide a widely shared description of their API, which makes their integration with the *cloud applications* and with additional providers a more challenging procedure. Furthermore, a certain performance overhead is imposed which may not be negligible in time critical applications. The overhead is imposed due to the run-time translation from the abstract API to the provider specific API.

4.2.4 Middleware solutions

As noted in [168], in the early years of 2000 the term middleware used to refer to the software layer abstracting the distributing applications from the underlying operating systems. Middleware primarily focuses on hiding the complexity in the networks environment by abstracting the applications from handling operations such as protocol handling, data replication and network faults. Even further a middleware seek to hide the heterogeneities of the computer architectures, operating systems, and programming languages and thus enabling the development and management of applications independently of the underlying infrastructure technologies.

However, the notion of "middleware" is not a recent concept. As mentioned in [168], the first middleware elements were built by researchers circa 40 years ago. At that time the middleware components were following the advances in hardware technology and in the networking of the workstations. They were mainly deployed for providing remote procedure calls, file, and directory service. According to *Gartner* the earliest Unix-based middlewares were the Transaction Processing Monitors (TPM) [169] such as the *Encina* developed by *IBM* and Tuxedo developed by *BEA Systems*.

In the recent years, *Gartner* refers to platform middleware as being the runtime hosting environment to execute application program logic [170]. It provides the applications with means of communications with other applications, which are potentially deployed in different hosting environments. It also manages the execution of the application by controlling the memory and the operating system processes as well as by providing security, monitoring, and load balancing mechanisms. Examples of modern middleware systems are application servers supporting the *.NET* and *J2EE* [171] programming models as well message-based systems such as *IBM*'s Message Queue Service (MQS) [172]. However, the applications of middleware are not restricted to enterprise computing but they have expanded to other domains such as smart devices and networking equipment which entail additional requirements such as high performance and high availability [173].

Therefore, middleware may have two primary scopes: i) they abstract the heterogeneities of the underlying resources that the application is running on. ii) They handle the execution of the application and its communication with components, which are hosted in different run-time environments.

With respect to the presented research work, middleware platforms are examined in the domain of cloud computing. Middleware solutions primarily seek to abstract the *cloud applications* from directly accessing the underlying resources, which are provided by the cloud vendors in the *IaaS* and *PaaS* level. This way the applications remain agnostic to the heterogeneities among the cloud providers. As it was analysed in the previous chapter, the heterogeneity mainly arises due to the differences in the programming languages and frameworks, the data stores, the *APIs* through which *platform basic services* are provisioned as well as the differences in the underlying virtual machines.

Middleware solutions span across all three phases of the *cloud application* lifecycle namely: (i) the development, (ii) the deployment and (iii) the execution. Certain requirements are identified in each phase [148] [150]. Specifically, in the development phase the middleware should provide the means for the creation of applications, which are agnostic to specific programming frameworks and providers as well as enabling the definition of constraints and scalability rules. During the deployment phase, support for automatic deployment on the target *cloud platform* should be provided. During the execution phase, the middleware should monitor the applications and perform adaptation actions when required.

In the next subsections concrete middleware solutions are examined. As mentioned in 4.2.1 the scope of the presented research work lies in the development of *cloud applications* in such a way that they remain agnostic of the target platform environment. Therefore, the development aspect will be the focus of the analysis of the middleware solutions.

4.2.4.1 *mOSAIC*

mOSAIC [174] is an EU research project aiming at developing a middleware platform which abstracts *cloud application* from specific provider technologies. Furthermore, *mOSAIC* offers monitoring and scalability capabilities.

Developers need to follow a specific application development paradigm in order to be compliant with the platform. A *mOSAIC* application consists of one or more components, which can use cloud resources via the *mOSAIC API*. The components are only allowed to communicate with each other indirectly via an abstracted message queue. The cloud resources, which the application has access to, are related to storage, such as database and file stores and communication, such as message queues.

With respect to enabling provider agnostic *cloud applications*, *mOSAIC* API is the core feature that allows application decoupling from native *APIs* [175]. The *API* consists of several layers and each one of those increases the abstraction, until the developer sees a single consistent interface. At the lowest layers there is the native protocol and the native *API* provided as a library by the cloud vendor for a certain programming language. At this level there is no uniformity since each vendor may use a proprietary *API*. One layer upwards there is the driver *API*, which wraps the native *API* thus providing the first level of uniformity. Interoperability *API* aims at providing programming language interoperability. This level ensures that *Java* and *Python* code communicates with the Drivers in the same way, using similar messages. The first layer that the developer is expected to use is the Connector *API*. The applications use specific connectors to access cloud resources, e.g. key-value store. Connectors are cloud independent, which means that same key-value store connector type can intermediate the access to specific platform stores. The last level of abstraction is the Cloudlet *API*, which is similar to the existing Java Servlet technology that provides standard programming components in *J2EE* environment. A Cloudlet is a component through which the developer may invoke the different functionalities offered by the Connector *API*. Cloudlets live inside the cloudlets container, which are managed by the *mOSAIC* platform. This way the platform can manage the applications and ensure scalability.

Contrary to *jClouds* that was discussed in Section 4.2.3.1, *mOSAIC* provides a complete application stack, which allows the users to create applications that are managed by the platform. However, *mOSAIC* platform requires that developers use certain programming concepts such as cloudlets and a specific programming paradigm, which is event-driven. Therefore it imposes a significant learning curve for the developers. Moreover, the use of a specific programming paradigm creates a lock-in effect to the specific middleware. Furthermore, *mOSAIC* acts as a middleware platform and mainly focuses on the abstraction of *IaaS* rather than *PaaS* resources.

4.2.4.2 *PaaSage*

PaaSage [176] is an EU research project aiming at the development and deployment of platform agnostic *cloud applications* which are able to leverage multiple cloud providers. The focus of *PaaSage* is to assist in all the stages of the *cloud application* lifecycle, namely: (i) the development stage by enabling the modelling of the application and the specification of requirements and *SLAs*, (ii) the deployment stage by choosing the optimal cloud providers, and (iii) the execution stage by monitoring the performance of the application and taking adaptation actions when required.

PaaSage consists of two parts, a family of Domain Specific Languages (DSLs) that allows the modelling of the application and the platform components, which undertake the tasks of deploying, monitoring and adapting the application.

The available DSLs cover the various stage of the application lifecycle. Specifically, *CloudML* [155] enables the modelling and deployment of the application in a provider independent manner. Saloon allows the developers to specify goals and requirements and select compatible cloud providers. WS-Agreement is used for creating *SLA*s and monitoring the application at run-time while *SRL* [177] is used for specifying scalability rules.

Regarding the platform components, the reasoner takes into account the requirements, which have been defined during the modelling stage, and tries to find the optimal deployment plan for the *cloud application*. Subsequently, the Executionware is responsible for deploying the application, monitoring its performance, and taking any required adaptation action.

With respect to enabling the creation of platform agnostic applications and their deployment to multiple cloud providers, the *PaaSage* offers two main components. The first is the *CloudML* [155] language which enables the modelling of the application in a provider independent manner. The latter is the Execution Engine, which is responsible for the deployment of the application in multiple cloud providers. In order to support multiple providers *PaaSage* leverages the *jClouds* library, described in Section 4.2.3.1. Therefore, up until now *PaaSage* mainly focus on the *IaaS* level rather than the *PaaS*.

4.2.4.3 *MODAClouds*

MODAClouds [178] is a EU research project addressing the issue of developing and managing *cloud applications,* which are independent of the target cloud provider. It covers several phases of the lifecycle of the *cloud application* such as the provider-agnostic design of the application, the semi-automatic translation of the models in to source code and the deployment in the target environments. Moreover, it provides run-time support by monitoring the application and performing adaptation actions.

MODAClouds comprises three main tools [179]: (i) The *Creator4Clouds* which offers an *IDE* for modelling and deploying provider agnostic *cloud applications*, (ii) the *Venues4Clouds*, which is the decision support system to identify the optimal cloud resources based on the application requirements and (iii) the *Energizer4Clouds* which consists of the run-time execution environment.

With respect to the development and deployment of platform provider agnostic applications, *MODAClouds* relies on two components: (i) The *CloudML* modelling language, which is also leveraged by *PaaSage* project described in the previous Section and (ii) the execution platform. The execution platform contains the adapters, which enables the application to leverage multiple cloud providers. Specifically, the platform reuses existing libraries and middlewares such as *jClouds*, described in Section 4.2.3.1 and *mOSAIC*, described in Section 4.2.4.1.

4.2.4.4 *JCloudScale*

JCloudScale is a Java-based middleware which contributes to various phases of the *cloud application* lifecycle such as the development, deployment and monitoring [180]. Specifically, the middleware enables the creation of applications which are cloud provider agnostic. It undertakes the task of instantiating the required virtual machines, deploying the application and monitoring it in a way that the application is unaware of the concrete cloud provider that it is executed on.

The basic notions of *JCloudScale* are the *CloudObjects* and the *CloudHosts*. The latter represent the virtual machines offered by the cloud provider. The *CloudObjects* can be *Java* classes implementing methods of the applications, which execute on the

CloudHosts. The middleware intercepts the execution of the application by invoking the methods defined in the *CloudObjects*, assigning their execution to specific *CloudHosts* and eventually returning the results back to the application. The whole process is transparent to the developers and to the application.

The developers create the application in *Java* without including any information about specific cloud providers. Then, annotations are used to define the *CloudObjects* to be executed on the *CloudHosts* and the scaling policies of the application. The middleware undertakes the task to distribute the code to the selected target applications and coordinate the execution of the application.

Contrary to the previous solutions, *JCloudScale* follows a declarative programming paradigm, which is based on annotations. Declarative programming may empower separation of concerns since the annotations are used for the deployment process and are not mixed with the source code containing the business logic of the application. Moreover, it reduces the dependence of the application to the middleware since the developers can decide whether they execute the application without the *JCloudScale* middleware by disabling the aspect oriented mechanism responsible for the annotation handling. However, *JCloudScale*'s focus is on the abstraction of *IaaS* providers rather than the *PaaS*. Specifically, it is limited to supporting *Openstack*-based private clouds and *Amazon EC2*. Furthermore, as reported in [180], the middleware adds a performance overhead to the application, which is proportional to the number of used *CloudHosts* and can be significant in time-critical applications.

4.2.4.5 *Multiclapp*

Multiclapp is a framework enabling cross-deployment of *cloud applications*. The main concept is the *cloud artifacts* [181]. These are software components, which encapsulate a piece of the business logic of the application and are deployed in a specific provider. *Cloud artefacts* consists of the source code of the application, the adapters which enable the application to consume some of the services offered by the platform such as the database and the client interoperability which allows *cloud artefacts* deployed in different cloud environments to communicate with each other.

Information about the deployment of the application is included in the deployment plan. Based on the plans, the main component of the framework, which is the source transformation engine, generates the *cloud artefacts*. The service generation engine produces the service clients required for the communication of the *cloud artefacts* while the cloud adaptation engine generates the adapters required by each *artefact* to use the service of the cloud provider.

In order to enable the creation of provider agnostic applications, *Multiclapp* uses the deployment plans to gather the information related to the deployment topology and concrete providers and to separate it from the source code of the applications. Contrary to *JCloudScale* where it is based on annotations, the *Multiclapp* uses *XML* files to hold the deployment plans. Moreover, an *Eclipse*-based *IDE* is available. This makes it more developer friendly contrary to the *JCloudScale,* which uses a Command Line Interface (CLI) tool. Furthermore, *Maven* [182], a popular dependency management tool, is used by both solutions to manage the project dependencies and libraries.

4.2.4.6 *OpenTosca*

OpenTosca is a middleware, which has been developed to support the deployment and execution of the *TOSCA*-based *cloud applications*. It comprises three tools. The first one called Winery [183], is a graphical environment which supports the modelling of the application. *OpenTosca* Container [184] is responsible for deploying and executing the application while *Vinothek* is a self-service portal where developers can inspect the deployed applications.

The *OpenTosca* Container, which is the middleware and run-time environment, further consists of three components: (i) the *Implementation Artifact Engine*, (ii) the *Plan Engine* and (iii) the *Controller*. The *Implementation Artifact Engine* runs any artifacts defined in the *cloud applications*. An artifact may represent a *SOAP* web service, which is required by the application. The *Plan Engine* is responsible for executing the managing plan, which contains the topology of the application, namely the components such as web servers, virtual machines, databases that need to be instantiated to host the application. The third component, the *Controller*, offers functionality for managing, installing and un-installing components of the application.

In order for *OpenTosca* to be able to coordinate the deployment of the application, the application is written and packaged in a particular way defined by *TOSCA* standard. Thus the *cloud application* becomes heavily dependent on the *TOSCA* specification and cannot be deployed on a non-*TOSCA* compatible environment. This contradicts the *JCloudScale* approach described in Section 4.2.4.4, which is based on annotations that developers can easily disable. Moreover, application developers need to become familiar with the specific application modelling and packaging style imposed by *TOSCA* as well as with the development tools, which results in a significant learning curve.

4.2.4.7 *Discussion on the Middleware solutions*

Table 2 summarises the middleware solutions presented in the previous Sections.

Table 2: Summary of Middleware solutions

Related Work	**Description**	**Comments**
mOSAIC	• Multi-layer abstraction API. • Coudlet is the core component for each cloud resource. • Event-based programming.	• Specific programming paradigm. • Lock–in effect to the platform. • Abstraction of IaaS resources.
PaaSaage	• Asist in the development, deployment and execution (monitoring) of the application. • Family of DSLs and platform components. • DSLs cover the modelling, deployment, monitoring rules. • Executionware handles deployment, monitoring and takes adaptation actions.	• Reuse of jClouds library to enable abstraction of cloud resources. • Abstraction of mainly IaaS resources.
MODAClouds	• Provider agnostic modelling of cloud applications. • Semi-automatic translation into source code and performance monitoring. • 3 main tools: 1. Creator4Clouds, an IDE for modelling and deployment. 2. Venues4Clouds decision support system for finding the optimal cloud resources, 3. Energizer4Clouds run-time execution environment.	• Regarding the abstraction of cloud resources, MODAClouds adopts similar techniques as PaaSaage. • Reuse of jClouds and mOSAIC. • Abstraction mainly of IaaS resources.
JCloudScale	• Java-based middleware for development, deployment and monitoring of applications. • Application composed of CloudObjects,	• Declarative programming based on annotations. Separation of concerns since annotations are not mixed with the source code

	which run on CloudHosts (VMs). • Declarative programming style.	• Performance overhead proportional to the number of CloudHosts used. • Limited to abstraction of VMs from IaaS providers
MultiClapp	• Framework enabling cross-deployment of cloud applications • Application is composed of: 1. Cloud Artefacts, containing the business logic, 2. Adapters, allowing the application to consumer cloud resources and 3. Client interoperability, allowing artefacts from different clouds to communicate. • Deployment plans are created to support the deployment of the application.	• Contrary to annotations used by JCloudScale, MultiClapp uses XML files to create the deployment plans. • Cloud adaption engine generates the adapter required for consuming the cloud resources.
OpenTosca	• Middleware for development, deployment and execution of cloud applications based on TOSCA standard. • Consists of 3 tools: 1. Winery, graphical tool for modelling the application, 2. OpenTosca container enables deployment and execution, 3. Vinothek, allows the monitoring of deployed applications.	• Developers need to become familiar with TOSCA standard and all the relevant tools. • Potential Lock-in to TOSCA way of packaging and modelling the application

Middleware solutions intermediate between *cloud applications* and cloud providers and thus contribute to the development of applications, which are agnostic to the concrete cloud environment. Similar to library-based solutions, middleware attempts to abstract cloud resources such as virtual machines, storage and message queues. However, contrary to the cloud libraries, middleware platforms often handle several aspects of the *cloud application* lifecycle such as the discovery and recommendation of the optimal cloud providers, the deployment of the application, as well as the monitoring and scaling of the application during the execution phase.

On the other hand the middleware platforms require a much higher installation and configuration time due to the fact that the majority are still in a research stage (*OpenTosca*, *mOSAIC*, *PaaSage*) and also due to the several components that they consist of (deployer, recommender, monitor etc.). The complexity of the middleware platforms also contribute to the significantly higher learning curve compared to the one of the cloud libraries. On top of that, the specific programming model that is usually adopted adds to the complexity and to the time required by the users to become familiar with the middleware. For example *mOSAIC* uses an event-driven programming model while *OpenTosca* requires that *cloud applications* are packaged with a specific structure and that specific configuration files are included. The fact that most of the middleware platforms expect that *cloud application* are written and packaged in a specific way creates a lock-in effect to the particular middleware. Therefore, while applications try to remain agnostic to the concrete cloud providers, they become aware and potentially locked in to the middleware platforms. Moreover, as it is specifically mentioned by *JCloudScale* approach described in Section 4.2.4.4, middleware solutions impose a performance overhead, which may be considerable in performance-critical production applications. In addition, contrary to the cloud libraries where there are available solutions for each popular programming language, approaches in this category mainly focus on Java oriented *cloud applications.*

4.2.5 Model-driven Engineering (MDE) based solutions

In the previous Sections, a number of approaches were listed that attempt to enable the creation of platform agnostic *cloud applications*. Apart from the use of library wrappers and middleware platforms, model-driven engineering (MDE) techniques can also be exploited.

MDE is an approach to system and software development in which software models play an indispensable role [185]. MDE is based on two core ideas: Abstraction and Automation. As seen in Figure 13, abstraction enables the engineers and software developers to begin with the creation of applications independently of the target platform by creating a *platform independent model (PIM).* This intermediary model does not include specific platform characteristics. Consequently, engineers can focus on the system-level application development ignoring low-level and error prone details. An inherent benefit of abstraction, which is particular interesting for this research proposal, is the potential to improve cross-platform development of *cloud applications* by decoupling the development from the specific platform technology.

Automation refers, among others, to the ability to change the level of abstraction automatically using model transformations. While application development may begin creating the *PIMs,* that abstract specific implementations, the *PIMs* are used to subsequently generate the *platform specific models (PSMs),* targeting a specific platform implementation and eventually generate the source code for the target *cloud* *platform*. Model transformations can automate the whole process of generating the platform specific implementations.

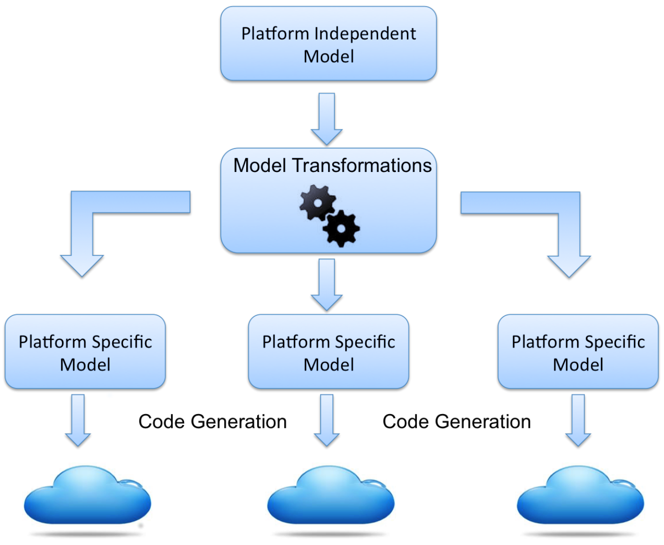


Figure 13: MDE approach in developing cloud portable applications

Particularly, this research work focuses on a special type of model transformations, namely the code generators. Code generators are a subcategory of the Model-to-Text transformations, as proposed by Czarnecki and Helsen [186] and Mens and Gorp [187].

Code generators include transformations where the output is a set of strings. Typical example of this category is the code generators where the input can be a UML model and the output can be *Java* code. The Model-to-Text approach can further be divided into two sub-categories: visitor-based and template-based approach.

Visitor-based approach: The visitor-based approach includes a visitor mechanism to traverse the internal structure of the input model and generate output text in a text stream. An example of this approach is *JAMDA* [188], a Model (UML) to *Java* generator. *JAMDA* provides users with a set of *APIs* for manipulating the models and a visitor mechanism for generating code.

Template-based approach: Contrary to the visitor-based approach where the whole output text is generated during the transformation, in the template-based approach there are pre-defined templates. During the transformation the source model is read to fill in the information missing from the output template. Listing 1, depicts part of a code generation template. Keywords like *public, class*, etc. are part of the template. The source model is traversed in order to fill in the missing information like the name of the class etc. *XPand* [189] is an example of Model-to-Text approach based on the template approach. *XPand* is part of the OpenArchitectureAware MDD generator framework.

Listing 1: Code generation template [178]

```
public class <<name>> {
     <<FOREACH attrs AS>>
     private <<a.type.name>> <<a.name>>;
     <<ENDFOREACH>>
```

As it was mentioned earlier in this Section there are two phases in MDE. The first one involves building the abstract model of the software system while the second one, involves automating the process of the code generation for a specific platform. In the coming Sections existing work that covers both phases are examined.

4.2.5.1 *Reference model for developing cloud applications*

Traditional software architectures lack the ability to describe concepts, such as scalability and resources requirements, which a *cloud application* consists of. Moreover, platform providers often introduce their own standards that influence the way applications are written. Therefore, there is a particular interest towards the direction of defining a generic model, which can capture the concepts that *cloud applications* comprise. Such a model can be used as a blueprint by developers for building uniform *cloud applications*.

Hamdaqa, Livogiannis and Tahvildari from University of Waterloo [190] propose a *cloud application* meta-model which is able to describe the main *cloud application* components.

According to the meta-model, a *cloud application* "has" one or more *cloud tasks*. A *cloud task* is a composable unit, which consists of a set of actions that provide a specific functionality. *Cloud tasks* can be classified into:

1. **CloudFrontTask**: *CloudFrontTask* is the entry point to the *cloud application* that handles the user requests, which are distributed by a load balancer. It is usually a web application.
2. **CloudCrossCuttingTask**: *CloudCrossCuttingTask* is responsible for monitoring cloud resources (compute, storage and load balance). It is also responsible for logging, maintaining quality of services of the *cloud application*, deployments of tasks, launching instances etc.
3. **CloudRotorTask**: *CloudRotorTask* runs in the background and is responsible for general development work or for helping other tasks by performing particular functionality.
4. **CloudPersistenceTask**: *CloudPersistenceTask* manages the access control and the login to cloud storage.

Each *cloud task* has some *Configuration Data*. *Configuration Data* contains information about the size of the *VM*, number of instances, database size, bandwidth etc. *Cloud tasks* can communicate with each other using *EndPoints*. An *EndPoint* may be internal or external (publicly visible) and uses an access mechanism that defines the way messages can be exchanged. Finally 3 types of storage are described: *Blobs, Table,* and *Queues*.

The *Reference Meta-Model* defines a relative adequate amount of concepts in order to enable the modelling of the application and the description of *IaaS* resources such as the provisioning of virtual machines. However, there is no information regarding the *PaaS* resources such as specific web servers, programming frameworks and *platform basic services*. Moreover, up until now there are no complimentary tools developed in order to enable the creation of the models and the generation of source code .

4.2.5.2 *CloudML*

CloudML [155] is a DSL used within the *PaaSage* EU project [176] to enable the modelling of *cloud applications* in a provider independent way and their subsequent deployment in multiple cloud providers. The *meta-model* of the DSL is *Ecore* models, which conform to the *Ecore meta-model*. The *Ecore* is provided by the

Eclipse Modelling Framework (EMF), which is, the modelling framework supported by *Eclipse* community.

CloudML defines the following core concepts:

- **Virtual Machines**: The *Virtual machine* type represents a generic virtual machine. It can be parameterised by defining certain values, such as the number of cores, the size of the memory, and the operating system.
- **Application components**: The *application component* type represents a component to be deployed on a virtual machine. Such components may be a Tomcat servlet container or a *MongoDB* database.
- **Ports**: A *port* refers to the interface of a feature of an application component. There are two cases for a port. A port can be provided, which means that it can be used to access a feature of an application component. Alternatively, the port can be required meaning that that the application component depends on the one which offers the feature.
- **Relationship:** A *relationship* is formed between two ports. It can either be a form of communication, via *HTTP* or a form of containment, meaning that one component contains the other one.

CloudML allows the modelling of the application at two levels of abstraction. During the first level, the cloud provider independent models are created (CPIM). These models define the deployment of the application in a provider independent way. During the second level, the cloud provider specific models (CPSM) are formed which contain information about the specific cloud providers where the application will be deployed on.

Compared to the reference model presented in 4.2.5.1, *CloudML* defines significantly less concepts and thus allows the definition of a *cloud application* only at a higher level of abstraction. For example, to the best of our knowledge, *CloudML* does not offer any concepts to describe the particular storage mechanisms of the *cloud application*. However, contrary to the reference model, *CloudML* is accompanied with a set of tools, which leverage the created models and can automate the deployment of the application. Moreover, the scope of *CloudML* overlaps with the

TOSCA standard, described in Section 4.1.1.6, which focus on the modelling of the deployment topology of a *cloud application*. However, *TOSCA* has reached a mature level and constantly gains supporters from industry and academia. As a result *CloudML* joined forces with the technical committee of *TOSCA* and attempts to become a *TOSCA*-compliant language.

4.2.5.3 *Pim4Cloud*

Pim4Cloud [191] is a modelling language which supports the modelling of the deployment of the *cloud application* and the relationships between its components in a provider agnostic way. It is developed in the context of the *REMICS* EU research project [192].

The main concepts of the *Pim4Cloud* modelling language are: (i) the *cloud application* which describes the application to be deployed in a cloud provider, (ii) the cloud resources such as, processing, storage and communication which are offered by the providers and the (iii) the cloud providers which describe the entity which offers the resources.

The concepts defined in the *Pim4Cloud* are capable of describing the *IaaS* resources. Regarding the *PaaS* level, the language does not provide any supporting elements mainly due to the high degree of heterogeneity in this level. Moreover, *Pim4Cloud*s focuses on the description of the resources required by a *cloud application* and can be used for the discovery of the provider which offers those resources. However, it does not cover the rest of the application lifecycle such as monitoring, adaptation and *SLAs* definition [193]. In addition, contrary to *CloudML*, *Pim4Cloud* does not provide run-time support of the created models. This means that fully automatic deployment of the applications is not supported. On the other hand, the language is developed as a UML profile, meaning that it extends the UML, a widely used modelling language. This fact eases the integration of *Pim4Cloud* models with the UML models and makes the language easy to be used by the software developers.

4.2.5.4 *MobiCloud*

One step beyond the modelling languages, which allows the description of the generic architecture of *cloud applications* is to create a DSL for developing applications at a higher level of abstraction. Ajith Ranabahu et al. have been working on such a DSL called *MobiCloud* [194].

MobiCloud DSL is a modelling language that closely resembles the Model-View-Controller (MVC) design pattern [195] by providing constructs for each of the three key components: model, view, and controller. This approach allows developers to reuse the same model to generate source code targeting different platforms. Code is automatically generated to target specific platforms.

On top of the DSL a graphical interface has been created. Using drag and drop components, users can create their applications, while the concrete syntax of the language is automatically generated.

Contrary to the previous approaches, *MobiCloud* provides full automation for the generation of source code. This means that developers only need to create the models using the user interface and then the whole source code is generated and is ready to be deployed in the target platform. However, *MobiCloud* has a very limited scope. Developers, using the user interface can only create simple CRUD (Create, Read, Update, and Delete) applications. Moreover, it cannot exploit any functionality, which is provided by the *cloud platforms* such as cloud storage, *noSQL* databases, and platform specific services. The code generator is limited to *Java* source code and the supported platforms are *GAE* and *Amazon EC2*.

4.2.5.5 *Discussion on Model-driven engineering based solutions*

Table 3 summarises the MDE-based solutions presented in the previous Sections.

Table 3: Summary of Model-driven engineering based solutions

Related Work	**Description**	**Comments**
Reference model for developing cloud applications	• Meta-model describing the components of the cloud application. • Application consists of cloud tasks. Each cloud task has some configuration data (VM size number of instances etc.) and communicates through Endpoints.	• Relative adequate concepts to model a cloud application. • No information regarding the description of PaaS resources. • No tools for source code generation
CloudML	• DSL used within PaaSaage project. • CloudML uses Ecore models provided by Eclipse Modelling Framework. • CloudML defines the following concepts: Virtual Machines, Application components, Ports, Relationships.	• Contrary to the previous work, CloudML defines significantly less concepts. • Tools to support source code generation. • CloudML overlaps with TOSCA and thus attempts to become TOSCA-compliant language.
Pim4Cloud	• Modelling language for the deployment of cloud applications. • Main concepts of Pim4Cloud are: 1 Cloud application, 2. Cloud resources (processing, storage and communication) and 3. Cloud providers.	• The defined concepts target the IaaS resources. • Contrary to CloudML, Pim4Cloud does not provide run-time environment. • Language developed as UML profile.
MobiCloud	• DSL providing constructs for the MVC pattern. • Graphical environment is provided.	• MobiCloud supports generation of source code. • Very limited scope. Only CRUD applications are supported.

Model-driven engineering technique involves the creation and manipulation of models, which at an initial stage are language and provider independent. Therefore, the same models can be reused for multiple programming languages. As it was discussed in 4.2.5, code generation is a kind of model transformation. Code generations are particularly useful in the domain of MDE because of their ability to automatically generate code from abstract models in any programming language. Thus the number of code lines that the developer needs to write can be reduced significantly. Furthermore, model transformations can be implemented to target a particular platform each time. Therefore, the performance overhead, which was imposed to the deployed application, by the library wrapper or the middleware platforms, is eliminated. Consequently the size of the application can be also reduced.

On the other head, the application lacks the ability to switch providers at run-time. The models need to be adjusted to the new target provider and the updated source code needs to be generated before it is redeployed on the *cloud platform*. This task takes place at design-time. Moreover, software engineers need to become familiar with the model-driven approach of application development, which can be a strenuous process. In addition, while in principle tools such as code generators are available to exploit the abstract models and perform transformations to target specific execution environments, in reality this is not always the case. Most of the existing work, which was presented, does not provide fully support for the automatic deployment and execution of the *cloud applications.*

4.2.6 Positioning of the related work with respect to the cloud computing service levels

Throughout the previous Sections existing work has been presented aiming at the development of *cloud applications*, which are agnostic to their actual deployment and execution environment. Based on the solution approach that is adopted each time, the work has been clustered in three categories, namely: (i) the library-based solutions, (iii) middleware solutions and (iii) the Model-driven engineering based solutions. As seen from Figure 14, this Section positions the related approaches with respect to the service level of cloud computing that they focus on and attempts to identify any potential gaps.

Specifically, the library-based solutions are mainly focused on the abstraction of *IaaS* resources and particularly on the provisioning of virtual machines. Additionally, they offer an abstract *API* for the users to leverage cloud storage services from multiple providers. Only specific cloud libraries such as the *Pkgcloud* provide support for a certain number of databases such as the *Azure* tables, the *MongoDB* and the *CouchDB*. Still the *Pkglcoud* only supports a limited set of operations offered by those databases.

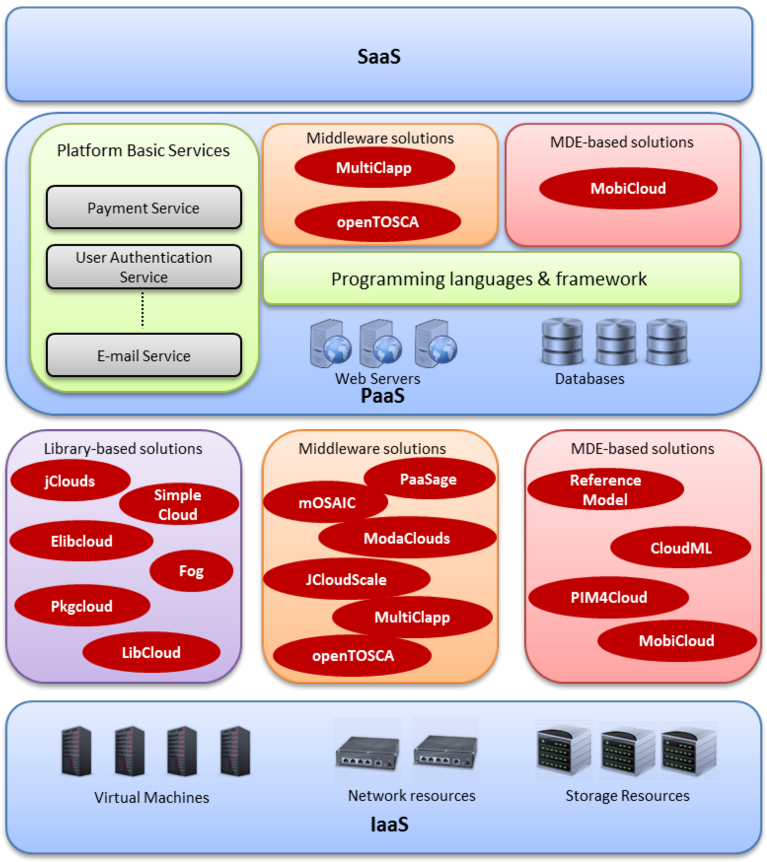


Figure 14: Positioning of the work with respect to the three levels of cloud computing

Similar to the cloud libraries, the middleware platforms primarily target resources offered by *IaaS* providers. Specifically, their main purpose is to abstract the provisioning of virtual machines from multiple cloud providers. Certain middleware extend their scope to the abstraction of databases such as *mOSAIC*, which supports *Riak*, a key-value pair database. With respect to the basic *PaaS* resources such as web servers and programming frameworks, *Multiclapp* [181], supports among others the *Windows Azure* platform and specifically the *Apache Tomcat* web server [17] and the *Glassfish* application server [196]. *OpenTosca* [184], on the other hand, provides support for the *PHP* programming language, the *Apache Tomcat* and *MySQL* database.

Regarding the MDE-based approaches, the majorities allow the description of the *cloud application* in an agnostic way with respect to the compute and storage services that the applications consume. For example software engineers are enabled to define the number of virtual machines instances required along with their characteristic such as the size of memory and the processing power. Additionally, the modelling languages offer the capability of defining certain constraints and scalability rules to be considered during the monitoring and adaptation phase of the *cloud applications*. With respect to the *PaaS* resources, *MobiCloud* is able to generate automatically, the complete source code required to deploy the *cloud application* to the *GAE* platform. The source code contains the configuration files required by the *GAE* platform as well as the files needed to connect to the *SQL* database. However, as mentioned in 4.2.5.4, the applications supported by *MobiCloud* have a very limited scope.

4.2.7 Focus of the presented research work

As it became obvious in the previous Section, the majority of the research effort focuses on the abstraction of cloud resources offered in the *IaaS* level such as, computational and storage resources. Significantly less work has been carried out in the *PaaS* level and this work mainly targets the various databases and to a certain extend the programming frameworks.

However, little or no work has been conducted regarding the *platform basic services* offered by the *cloud application* platforms as those were discussed in the previous chapter. As mentioned in Section 3.5, a *platform basic service* is a piece of software offering a certain functionality which can be reused by multiple users during the development of a *service-based cloud application*. The service is usually offered via a web *API*. Examples of such services are the payment service, the e-mail and the authentication service. The *platform basic services* have the ability to speed up the application development since the latter rather than being developed from the ground up they can be synthesised from services offered by the *cloud application* platform.

To this end this research thesis focuses on the *platform basic services*. It primarily attempts to facilitate developers to exploit those services in a consistent way independently of the concrete category of service, such as payment or e-mail service. At a second level the work focuses on enabling developers to use *platform basic services*, which are not bound to a specific service provider. In other words the application should be agnostic to the specific providers that are used each time. This is achieved via creating an abstraction between the applications and the *platform basic* *service* providers. The decision of the author to focus on the *platform basic services* has been based primarily on two facts:

i. The concept of the *platform basic services* has not been addressed by the related work examined in this Chapter.
ii. The *platform basic services* gain constant momentum, as seen in Section 3.5, and have the potential to implement the *micro-service* development paradigm [33], as briefly mentioned in the Introduction of this thesis and further analysed in Section 4.2.7.2.

4.2.7.1 *Limitations of the related work with respect to the platform basic services*

The solutions described throughout this Chapter entail certain limitations that were described both in each of the solutions separately and collectively in Sections 4.2.3.4, 4.2.4.7 and 4.2.5.5. In addition they have a different scope as discussed in 4.2.6. On top of that they lack certain characteristics, which the presented work attempts to accommodate and which are the following:

- ➢ Enable the description of the functionality of the service in a way that can be publicly available and shared among both the service providers and application developers. Consequently, the latter are eased during the integration of the service whereas the former can adhere to the commonly used and agreed service description. This will further pave the way for the homogenisation of the offerings of the various service providers.
- ➢ Allow reconfiguration and expansion of the service description. The domain of *cloud application* platforms and *platform basic services* is highly dynamic and the providers constantly offer new functionality. Therefore the proposed solution should be able to reflect these changes in a straightforward and rapid manner.
- ➢ Provide a generic methodology to enable integration of additional *platform basic services* and providers. Following the previous characteristic, the proposed solution should offer a concrete methodology to enable new services and providers to be supported.

4.2.7.2 *Micro-service development paradigm and the connection to the platform basic services*

Micro-service is a relatively new approach to software development which promotes the creation of applications composed of single purpose, independently deployed services [33]. These services are running independently in their own process and are also deployed independent of the rest of the application. They usually communicate through a lightweight mechanism, which is *HTTP*-based [197].

The use of services in the development of software applications is not a new concept. The notion of services and components have been exploited and implemented in the monolithic applications. However, in that case the services are usually packaged in a single *Web Application Archive (WAR)* file and communicate with each other via library calls. This fact deteriorates considerably the scalability and the maintenance of the application.

Service-oriented Architecture (SOA) [12] focused on the design of more flexible applications by enabling the use of loosely-coupled services. However, the complexity of its architecture and the fact that it is based on a heavyweight communication protocol (*SOAP*) may have acted as impediment to its wide adoption. On the other hand as Oracle states, *micro-services* are emerged as a mean of regaining flexibility, which may have been lost in *SOA* as a result of the latter becoming rigid [198].

There are several benefits associated with the micro-service approach of application development [33]. First, the fact that the services are completely independent of each other eases the deployment process and improves its scalability. Rather than scaling everything together, as it mainly happens in monolithic applications, *micro-services* allow to focus on scaling only the required resources. Furthermore, the applications become more resilient. When a failure occurs, it affects only a specific part of the system since the services are isolated and run on independent hardware resources. This fact further implies that the replaceablity of the application is optimised. A service can be replaced independently of the other parts of the application.

The prospects of the *micro-services* approach also become evident from the fact that prominent software enterprises, such as *Oracle* and *IBM*, embraces and promote this new software development architectural style.

As it has already been mentioned in the Introduction of this thesis, the advent of cloud computing has paved the way and provided the means for the realisation of the micro-service development approach. Specifically, the *service-based cloud applications* and the *platform-basic services* may constitute an instantiation of the micro-service based applications and the *micro-services* respectively. The *platform basic services,* as they have been defined in Section 3.5, are pieces of reusable software which expose a certain functionality. They are primarily accessible over the *HTTP* protocol via a *RESTful API*. However, the same notion of small autonomous piece of functionality applies for the *micro-services* as described earlier in this Section. Therefore, the author believes that the *cloud application platforms* via which the *platform basic services* are provided can realise the *micro-service* approach and offer to the developers the benefits associated with it.

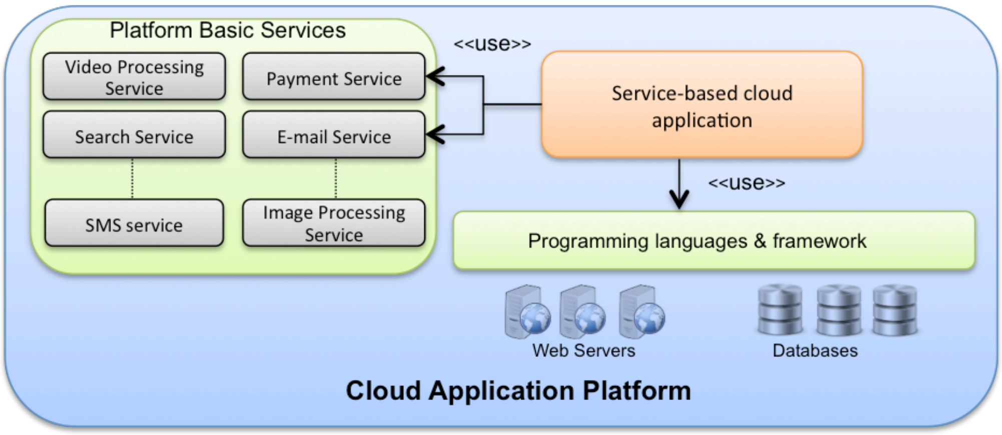


Figure 15: Example of micro-service approach using the Heroku platform

Heroku, as mentioned in Section 3.5, is a major *cloud application platform, which* provides developers with *platform basic services.* Figure 15 shows how the developers can leverage the *platform basic services* in order to build *service-based cloud applications.* Rather than creating the whole functionality from the ground up the developers may use the offered services in order to decrease the development time and effort. Specifically, Figure 15 depicts an application, which utilises the payment and the e-mail service. There are several additional services that may be used such as the image and the *Simple Messaging System (SMS)* services.

However, there are various categories of *platform basic services* and multiple providers. The developers in order to gain the complete benefits of the *cloud application platforms* and the *micro-service* approach, as discussed in Section 4.2.7, they need to be able to exploit the services in a consistent way and choose seamlessly the concrete providers.

Towards this direction, the next part of this thesis, Part B, further strengthens the argument for a solution targeting the abstraction of *platform basic services* and clarifies the concrete variability points among the services and the providers, which need to be addressed. Subsequently, a methodology is proposed and the *SCADeF* development framework is described to enable the *service-based cloud applications* integrate with *platform basic providers* without being exposed to the concrete providers' implementation.

PART B

The Service-based Cloud Application Development Framework (*SCADeF*)

Chapter 5

High-level architecture of the *SCADeF* framework

In the previous chapters the domain of cloud computing was introduced and the field of *cloud platforms* was examined and analysed. Special attention was paid to the *cloud application platforms*, which focus on the provisioning of *platform basic services*. As explained in detail earlier in Chapter 3, the *platform basic services* can be considered as piece of software, which provides certain functionality and can be reused by multiple users. It is typically accessible via a *HTTP*-based *API*. Examples of *platform basic services* include the e-mail, authentication and the payment service.

As proposed in Section 4.2.7.2, the rise of the *cloud application platforms* has the potential to lead to a paradigm shift in software development where the *platform basic services* act as the building blocks for the creation of *service-based cloud applications*. However, due to heterogeneity among the offered solutions, software engineers are required to spend a considerable amount of time on examining the various services and the providers, before selecting the one, which better meets their requirements.

In Chapter 4 we reviewed several initiatives proposing different methodologies and frameworks for the design of *service-based cloud applications* leveraging heterogeneous resources. Most of the approaches focus on the abstraction of traditional cloud resources, such as the abstraction of virtual machines and cloud storage at the *IaaS* level. By contrast, the focus of this PhD thesis is the abstraction of the *platform basic services* offered at the *PaaS* level by the *cloud application platforms*.

Moreover, previous work has been done on the abstraction of the deployment infrastructure such as web servers. Rather than focusing on the deployment process, our efforts concentrate on the integration process of the *platform basic services* with *cloud application*s.

Furthermore, the proposed solutions in earlier work are primarily either in the form of software libraries acting as code wrappers or in the form of middleware platforms mediating between the *cloud application* and the cloud resources. In the former case, the wrappers offer limited application scope, as mentioned in Chapter 4, since they are designed to mediate between specific software interfaces. In the latter case specific programming paradigms are adopted with a considerable learning curve for the software engineers. On top of that middleware solutions are complex environments, which may impose a performance overhead.

Work has been also carried out in the field of automatic generation of the web client adapters required to communicate with the service providers. However, no capabilities are offered for constructing and providing a *Reference API.* Therefore, using the existing solution it is not possible to abstract the various providers` specific web *APIs*. In a broader view, there is a lack of a concrete methodological approach to the development of *service-based cloud applications.*

Therefore, the *SCADeF* framework described in this Chapter attempts to address the limitations of the existing solutions in the field of the cross-platform development of service-based cloud applications. Specifically, the focus of the proposed work is on the *platform basic services* offered by the *cloud application* platforms and on engineering a methodology to enable heterogeneous *platform basic services* and providers to be integrated seamlessly with the *service-based cloud applications*.

In order to achieve this, the differences across the *platform basic services* and the providers need to be alleviated. As it is analysed in Section 5.1, these differences primarily arise from the diverse workflow and *API* exposed by each provider as well as the multiple configuration and authentication variables required to set up the service.

Towards addressing these variability issues the framework splits in two parts, the *Platform Service Workflow Description* and the *Platform Service API Description.* As the names imply the first part addresses the differences in the workflow of the operations of the providers while the second part involves the differences in the *API* and configuration and authentication variables.

Each part contains three phases. The *Platform Service Modeling Phase* defines the abstract functionality of the *platform basic service,* including the reference workflow and the *Reference API.* Subsequently, the *Vendor Implementation Phase* describes the concrete provider functionality and the *Execution Phase* makes the service available to the developer. Section 5.2 states the high-level solution direction while the remaining Sections of the Chapter describe in detail the parts and the phases of the *SCADeF* framework.

5.1 Variability points across the platform basic service providers

Prior to the description of the proposed development framework, the motivation and the specific focus of the solution should be stated.

Preliminary work of the author on several *platform basic service* providers offered by *Heroku*, *GAE*, *AWS marketplace,* and *OpenShift* have shown that the following three variability points needs to be addressed in order to decouple application development from vendor specific implementations:

i. Differences in the workflow for the execution of the operations offered by the *platform basic service* providers.
ii. Variability in the web *API* exposed by the providers.
iii. Management of the configuration and authentication settings required during the interaction with the services.

5.1.1.1 *Differences in the workflow*

Stateful services require more than one state in order to complete an operation [199]. The number and the implementation of the states may differ both across the various types of *platform basic services* and the concrete service providers.

Such an example is the payment service that enables developers to accept payments through their applications. As it is explained in Section 6.2, 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. Additionally, the authentication service is a different type of *platform basic service*, which enables a *cloud application* to authenticate its users. The service involves two states: (i) redirecting the user to the authentication service provider and (ii) receiving the access token to collect the user`s information.

We believe that software engineers require support to overcome this degree of heterogeneity. They should be able to model the interaction of *cloud application*s with *platform basic services* irrespective of the type of service or concrete provider. For that reason the commonalities when integrating various types of services and providers should be extracted. Furthermore, a coordination mechanism is required to handle the operation flow and automatically execute the actions defined in each state.

5.1.1.2 *Differences in the API*

There are several platform providers implementing a given *platform basic service*. However, they expose different *APIs* resulting in conflicts when an application developer attempts to integrate with one or another. As an example we consider the e-mail service and two service providers, the *Amazon Simple E-mail Service (SES)* [200] and *SendGrid* [35], an add-on mail service offered via the *Heroku* application platform. Upon the request for sending an e-mail the four following parameters are required: (i) the recipient, (ii) the sender, (iii) the content and (iv) the title of the e-mail. The concrete naming of the parameters as required by *Amazon SES* is respectively: (i) *Source*, (ii) *Destination.ToAddresses*, (iii) *Message.Subject,* and (iv) *Message.Body.Text* whereas regarding the *SendGrid* the anticipated parameters are: (i) *from*, (ii) *to*, (iii) *subject,* and (iv) *text*.

Therefore, for the same category of platform service, namely e-mail service, software engineers are required to adjust their code to the web *API* of each of the respective service providers.

5.1.1.3 *Management of the configuration and authentication variables*

In addition to the construction of the web calls and the operation workflow handling, *platform basic services* require certain configuration settings and authentication tokens to be present during the interaction with the *cloud application*. Indicatively, we refer to the *Google Authentication* service and the following set of required variables: 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 explained above a *cloud application* may interact with several *platform basic services* in various ways. If we count up the large number of services out of which an application may be composed of, it is clear that the integration and management of the services may become a time consuming and strenuous process. In order to support the consistent modelling and integration of services as well as the decoupling from vendor specific implementations, the *SCADeF* framework is proposed.

5.2 Solution Direction

As mentioned earlier in this Chapter and also seen in Figure 16, there exist multitudes of a particular type of *platform basic service*, since the services are offered by many different providers. For example, there are 4 different service providers offering the E-mail service, the *SendGrid* [26] and the *Mailgun* [27] offered by *Heroku*, the *Google Mail* [201] and the *Amazon Simple E-mail Service* [200]. On top of that, there exist various categories of *platform basic services* such as the payment, the e-mail, and the message queue service.

To support developers in using *platform basic services* from various environments, a new approach to creating *service-based cloud applications* should be adopted. The key concept is for users not to develop applications directly against the proprietary cloud provider's environment. Rather, they should use either standard and widely adopted technologies or abstraction layers, which decouple application development from specific target technologies and *APIs*. Taking that into account this thesis proposes the *SCADeF* framework, which promotes uniform access to *platform basic service* and providers.

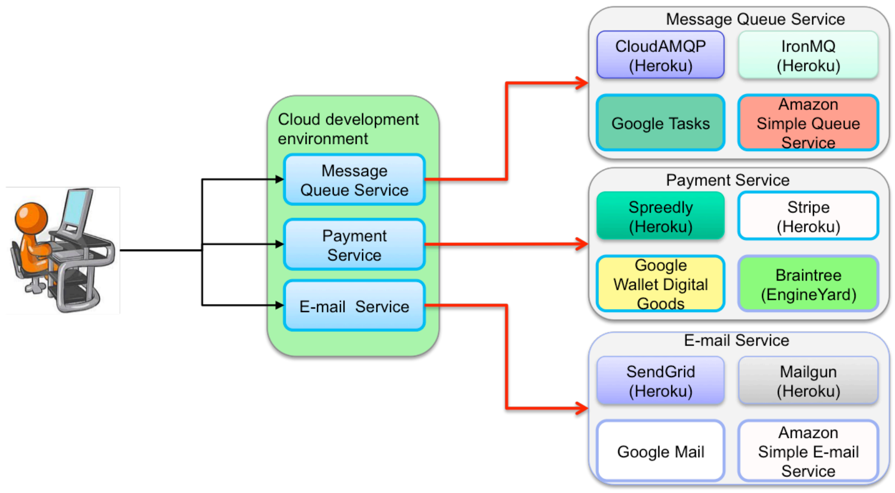


Figure 16: Conceptual View of the Development Framework

The objective of the framework is two-fold: (i) First to enable the integration of *platform basic services* in a consistent way and (ii) second to facilitate the seamless use of the *platform basic service* providers by alleviating the heterogeneities among them. Thus application developers can focus on the design of the application without dealing with the peculiarities of each provider.

The main components of *SCADeF*, as it will be explained in the next Chapters, are the *Reference Meta-Model* and the *Service API editor*. The former includes the concepts enabling the consistent modelling of the various *platform basic services* while the latter enables the description and the abstraction of the provider specific web *APIs.*

Inspired by the Model-driven Engineering [202] design approach, described in Section 4.2.5, the framework adopts a three phase process in order to enable the abstraction of the *platform basic service* providers. First the abstract functionality of the *platform basic service* is described. During this phase the workflow of the service is modelled and the *Reference API* is defined. In the next phase the concrete vendor implementation is infused. The specific workflow and web *API* is mapped on the reference one defined in the first phase. During the third phase, the framework handles the execution of the workflow and automatically generates the client adapters to invoke the providers' web *API*.

In the next Sections the requirements of the proposed framework are listed.

5.3 Requirements of the *SCADeF* framework

There are certain requirements identified for the development framework as listed in Table 4. They have primarily been defined upon the objective of addressing the variability points, which were listed in 5.1, namely, the differences in the workflow, in the web *API* and in the settings and tokens required by each concrete *platform service* provider. Furthermore, the definition of the requirements has been based on those defined in related development frameworks [181], [203], as well as on the author's personal experience on the field of *cloud application platforms* and *platform basic services*.

Table 4: Requirements of the *SCADeF* framework

R1	***SCADeF* should provide workflow modelling capabilities**
R2	***SCADeF* should automate the execution of the workflow**
R3	***SCADeF* should address the API variability**
R4	***SCADeF* should automatically generate the client adapters**
R5	***SCADeF* should be generic enough, so that additional platform basic services and providers can be supported.**
R6	***SCADeF* should be able to substitute the platform basic service providers.**
R7	***SCADeF* should accommodate two distinct user roles, namely the *administrator* and the *consumer.***
R8	***SCADeF* should manage the *platform basic services* and the configuration variables**

5.3.1 Support of workflow modelling capabilities

As mentioned in Section 5.1, the first variability point across the various *platform basic service* providers is the differences in the workflows for the completion of the operations. Therefore, the development framework should provide the application developers with the necessary building blocks to enable the workflow modelling in a consistent way. Independent of the type of the *platform basic service* or the concrete provider, two basic request types are present, as seen in Figure 17:

i. The outgoing request from the application to the *platform basic service* provider using the web API of the latter.
ii. The incoming requests from the client or the *platform service* to the application which needs to be received and handled by the latter. The framework should enable the modelling of these request types.

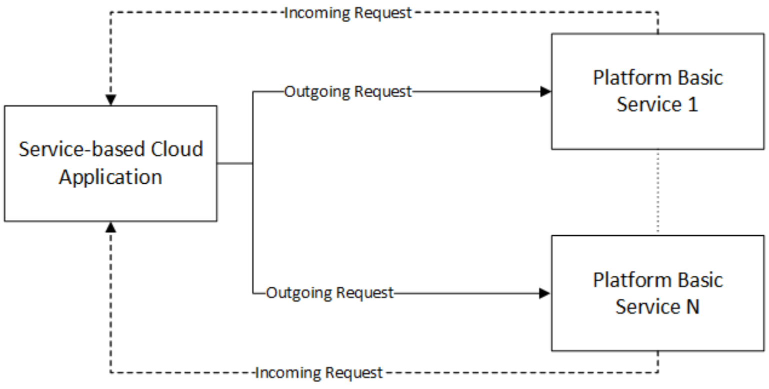


Figure 17: Interaction between the Service-based Cloud Application and the Platform Basic Services

5.3.2 Automating the execution of the workflow

In the previous requirement, it was stated that the framework needs to provide the tools for modelling the states of the *platform basic service* providers. Following that requirement the framework should also facilitate the execution of the states. Thus, rather than enforcing the developer to coordinate manually the execution of an operation, an execution engine should be able to handle the operation workflow automatically and thus decoupling the application developer from directly accessing the provider specific implementation.

5.3.3 Addressing the API variability

The second variability point, as mentioned in Section 5.1 is the differences in the *APIs* exposed by the *platform basic service* providers. In order to effectively abstract the vendor specific implementations, the framework should address the peculiarities in the various web *APIs* exposed by the providers. Two further dimensions are implied:

i. The capability of defining a *Reference API* for each category of *platform basic service.* The *Reference API* is exposed to the application developers and can be used to access all the supported providers.
ii. The mapping of the vendor specific *API* to the equivalent reference one. The framework should be capable of mapping each vendor specific *API* to the reference one. Thus the application developers are not required to interact with the specific *APIs.*

5.3.4 Automatic generation of the client adapters

Following the previous requirement regarding the variability in the *API,* the framework should additionally be able to generate the code required to perform the invocation requests to the provider specific *API.* The majority of the providers expose a web *API* based on *HTTP* requests. By offering code generation capabilities, application developers are alleviated from the task of having manually to code the invocation requests each time integration with a new service provider is required.

5.3.5 Generic nature of the framework

One of the main requirements of the framework is its capability to support new *platform basic services* and providers. Rather than being static our objective is to ensure its flexibility so that it is continuously expanded and updated with new types of *platform basic services* and providers. This is partially achieved by the first and third requirement, namely by providing the generic building blocks to model the workflow of the *platform basic service* and also the capability of defining the *Reference API* for the service which is supported by the framework.

5.3.6 Ability to substitute the platform basic service providers

As mentioned earlier in this Chapter, there exist multiple providers implementing a particular *platform basic service.* Under certain cases an application may require to switch to a different provider at run-time. For example in the scenario of a sudden increase in the workload a particular provider may become unresponsive. In this case the application should be able to choose an alternative provider in order to continue its operations without disruptions. Therefore, the framework should enable the users to determine the concrete service providers at run-time and be able to switch providers according to which one best meets the requirements at hand each time.

5.3.7 Distinct user roles

The framework should support the following two distinct users:

i. The *administrator*: The *administrator* should be capable of enhancing the framework with new *platform basic services* and providers. As stated in the Section 5.3.5, the framework rather than being static needs to be expandable and be able to support additional services and providers. This is an essential characteristic in order for the framework to keep up with the increasing number of *cloud application platforms.*
ii. The *consumer*: The *consumer* is the person who is using the framework. Usually, this is the application developer who needs to exploit the *platform basic services* during the application development.

Therefore, the framework should provide the required tools for the above two mentioned users.

5.3.8 Management of the platform basic services and the configuration variables

As proposed in Section 5.1.1.3 the third variability point when dealing with multiple *platform basic services* and providers is the large number of configuration settings that the developer needs to manage. This may become a tedious and error prone process. Therefore, the framework should enable the application developers both to add or remove services seamlessly from the application and also manage the configuration settings and the authentication tokens required by each of the concrete providers.

5.4 High-level architecture of the *SCADeF* framework

This Section describes the high level architecture of *SCADeF*. In particular, it focuses on the components of the framework and the processes, which are required to execute the supported *platform basic services* and to add new services and providers.

Figure 18 shows that the processes split into two stages, namely the *Modelling* and the *Execution Stage*. As it is described in the next Section, the first stage involves the phases and the components required to add a new *platform basic service* and provider into the framework. The *Execution Stage* involves the components required to execute the *platform basic services* supported by the *SCADeF* framework and thus gives the *cloud application* access to the concrete service providers.

Following the two stages described in the previous paragraph, two distinct users of the framework (roles) are defined: the *administrator* and the *consumer*. The first uses the components, available during the *Modelling Stage*, in order to enrich the framework with additional services and providers. The latter makes use of the framework during its *Execution Stage* in order to integrate *platform basic services* with the *cloud application*.

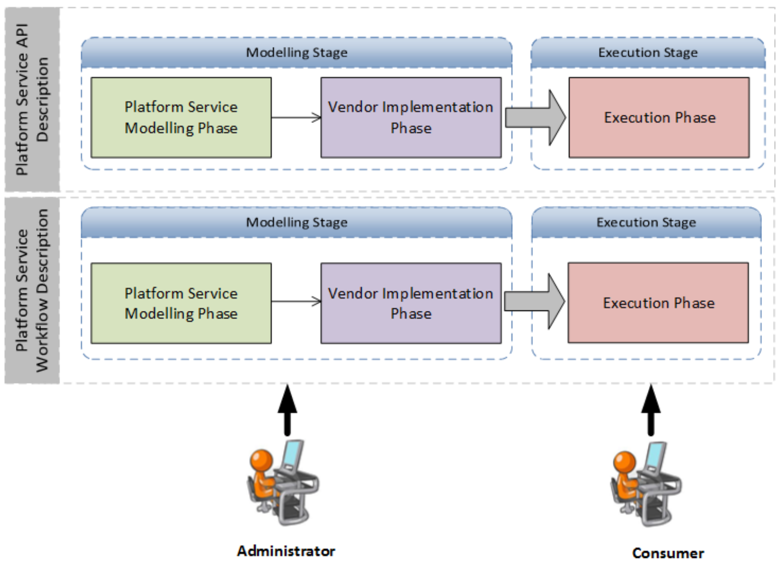

Platform Service API Description
Modelling Stage
Platform Service Modelling Phase
Vendor Implementation Phase
Execution Stage
Execution Phase
Platform Service Workflow Description
Modelling Stage
Platform Service Modelling Phase
Vendor Implementation Phase
Execution Stage
Execution Phase
Administrator
Consumer

Figure 18: Modelling and Execution Stage of the *SCADeF* framework

As it can be observed in Figure 18, the process of adding a new *platform basic service* and provider to the framework can be divided into the following two parts [204]:

i. ***Platform Service Workflow*** **Description**: As explained earlier in 5.1, certain *platform basic 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 described in a way that is capable for the framework to handle automatically the workflow.
ii. ***Platform Service API*** **Description**: One of the main objectives of the framework is to provide the developers with a single *API* for each *platform basic service*, independent of the concrete provider. Therefore, as it is described in details in Chapter 8, 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 API.*

Each of the two parts of the development process involves the *Modelling* and the *Execution Stage*. As shown in Figure 18 the two stages consist of the following phases: (a) *Platform Service Modelling Phase, (b) Vendor Implementation Phase, (c) Execution Phase.*

In the next Sections for each of the two parts, namely the *Platform Service Workflow Description* and the *Platform Service API Description*, the three phases are introduced and the high-level components involved are described. Figure 19 illustrates the components that constitute the *SCADeF* framework. The components are split into two categories, highlighted by the use of two styles, according to whether they are used, or created, by the *administrator*. The ones highlighted in orange colour are provided by the framework and are used by the *administrator*. The one highlighted by stripes are the *platform service* components and are produced by the *administrator* using the framework components.

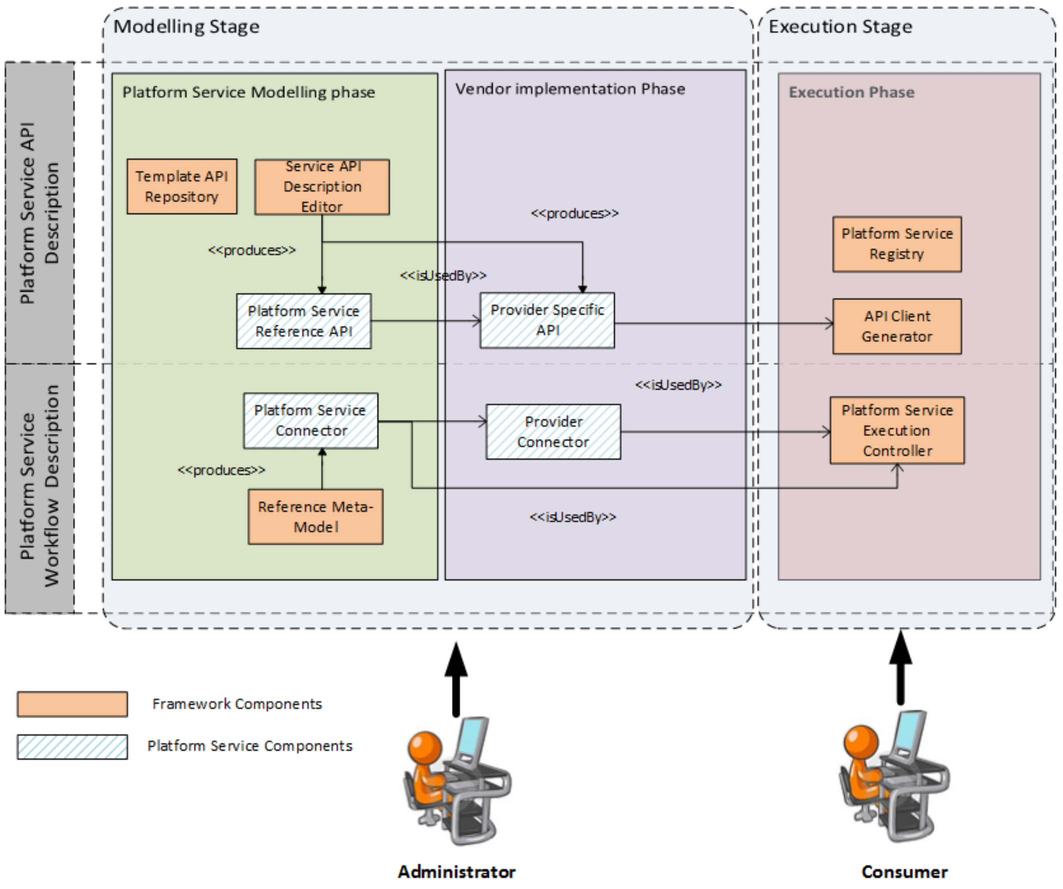

Modelling Stage
Execution Stage
Platform Service API Description
Platform Service Workflow Description
Platform Service Modelling phase
Vendor implementation Phase
Execution Phase
Template API Repository
Service API Description Editor
<<produces>>
<<isUsedBy>>
Platform Service Reference API
Provider Specific API
Platform Service Registry
API Client Generator
Platform Service Connector
Provider Connector
Platform Service Execution Controller
Reference Meta-Model
Framework Components
Platform Service Components
Administrator
Consumer

Figure 19: High-level architecture of the *SCADeF* framework

5.4.1 Platform Service Workflow Description

5.4.1.1 *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:

1. **Reference Meta-Model:** The *Reference Meta-Model* contains the concepts required to model the states of the *platform basic service*. Section 6.2.2.1 of discusses in details the involved concepts.
2. **Platform Service Connector:** The *Platform Service Connector* (*PSC*) is the abstract representation of the *platform basic 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.

5.4.1.2 *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.

1. **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.

5.4.1.3 *Execution Phase*

The *Execution Phase* takes place at run-time. During that phase the *Platform Service Execution Controller* (Figure 19) handles the execution of the workflow. Particularly, it accepts as inputs the *PSC* and the *PC*. Then it executes the workflow defined by the states and transitions in the *PSC*. For each state it checks whether a specific provider implementation exists in the *PC*. If it does so, then it executes the specific implementation. Otherwise it executes the one specified in the *PSC*.

Figure 20 shows the interrelations of the various components of the *Platform Service Workflow Description* part of the *SCADeF* framework in order to achieve the abstraction of the concrete service providers from the *cloud application*. The *PC* communicates directly with the service provider. It implements the abstract model inherited from the *PSC* for the specific provider [205]. The *cloud application* developer uses the *Platform Service Execution Controller* (*PSEC*) in order to execute the workflow for the specific platform service, as it is defined in the *PSC*.

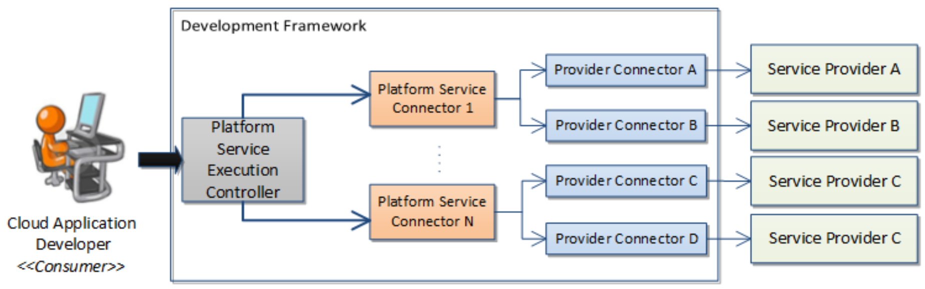


Figure 20: Workflow description part of the *SCADeF* framework

In the following Chapters, each of the components depicted in Figure 20 is analysed and its role in the *SCADeF* framework is examined.

5.4.2 Platform Service API Description

5.4.2.1 *Platform Service Modelling Phase*

As mentioned in Requirement 3 (R3) in Section 5.3, 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. *Consumers* leverage *the Reference API* in order to gain access to the specific service providers, which are supported by the framework. In Chapter 8 a detailed description of the definition process of the *Reference API* is presented.

1. **The Service Description Editor:** The *Service Description Editor (SDE)* is used to define the *Reference API* and configuration and authentication settings required by the *platform basic service* providers. It is implemented as an *Eclipse* plug-in and includes a user interface, which is used by the *Administrator* of the framework. Further information regarding the *SDE* can be found in the Appendix B.
2. **Platform Service Reference API:** The *Platform Service Reference API* is a template describing the web *API* of the specific *platform basic service*. A *Reference API* is constructed, by the *administrator*, for each *platform basic service* and is accessible by the *consumer* of the framework. Its role is to remove the barrier from the *consumer* to study the various providers' specific APIs. Instead the *consumer* accesses all the supported providers via the *Reference API*.
3. **Template API repository:** The *Template API Repository* contains the collection of the *Platform Service Reference API*s that have been defined using the *SDE*.

5.4.2.2 *Vendor Implementation Phase*

During this phase the specific web *API* of each of the *platform basic service* providers supported by *SCADeF,* is described and mapped to the *Reference API.*

1. **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.*

5.4.2.3 *Execution Phase*

During the *Execution Phase* the web clients [206] 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.

1. **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 application developer in order to connect to the concrete service provider.
2. **Platform Service Registry:** This component is a registry of all the *platform basic 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.

5.4.3 Modelling and Execution Flow of the Development Framework

This chapter stated the variability points that may arise when application developers deal with multiple *platform basic services* and providers, namely: (i) the variation in the workflow of the execution of the various *platform service* providers, (ii) the variation in the exposed web *APIs,* and (iii) the management of the configuration and the authentication variables that the various service providers require.

Thereafter, the *SCADeF* framework was proposed to address the above mentioned issues and assist the application developers in the design and execution process of the *service-based cloud applications*. The development framework consists of two parts: (i) the description of the *Platform Service Workflow*, which attempts to alleviate the heterogeneity in the workflow of the various *platform service* providers and (ii) the description of the *Platform Service API*, which aims at the homogenisation of the various web *APIs*.

For each part two stages are defined, namely the *Modelling* and the *Execution*. The first stage involves the integration of new *platform basic services* and providers in the framework and is handled by the *administrator* of the framework as mentioned earlier in Section 5.4. During the *Execution Stage* the application developers are able to make use of the supported services and providers. A task flow is defined in order to enable the *administrator*s to enrich the framework with services and the application developers to utilise those services [207].

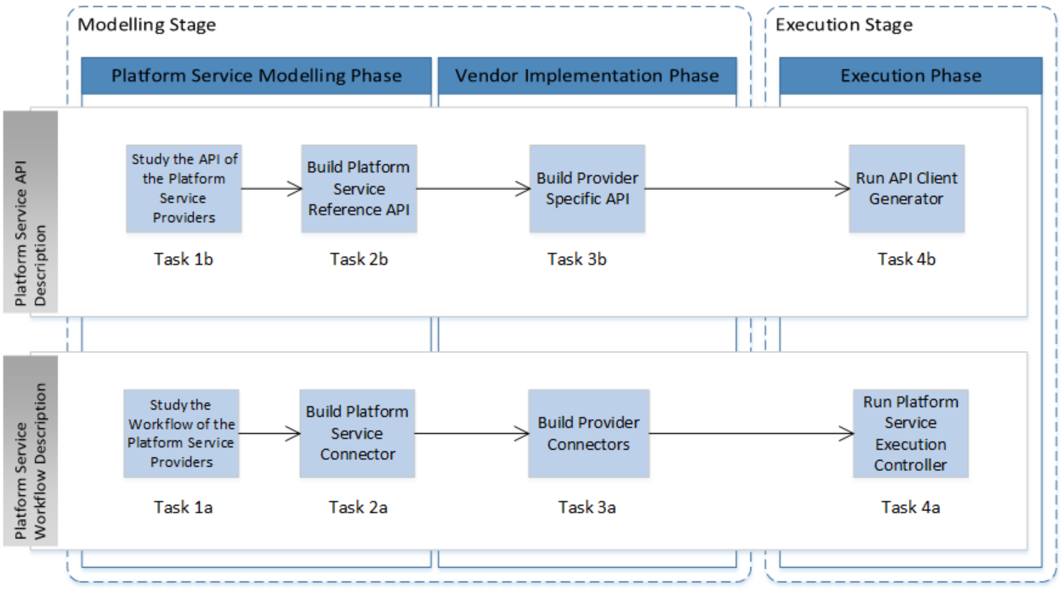

Modelling Stage
Execution Stage
Platform Service Modelling Phase
Vendor Implementation Phase
Execution Phase
Platform Service API Description
Platform Service Workflow Description
Study the API of the Platform Service Providers
Task 1b
Build Platform Service Reference API
Task 2b
Build Provider Specific API
Task 3b
Run API Client Generator
Task 4b
Study the Workflow of the Platform Service Providers
Task 1a
Build Platform Service Connector
Task 2a
Build Provider Connectors
Task 3a
Run Platform Service Execution Controller
Task 4a

Figure 21: The task flow of the *SCADeF* framework

Figure 21 depicts the task flow of the *SCADeF* framework. There are two separate task flows defined, one for the description of the workflow of the *platform basic service* and one of the *API*. For each part the following tasks are defined:

1. ***Platform Service Workflow* Description:**

Task 1a. Examination of the workflow of the available service providers for a given *platform basic service*. Particularly, this task involves the description of the states required in each of the providers in order to execute an operation. The output consists of the definition of an abstract state machine diagram, which accommodates all the examined providers.

Task 2a. Based on the abstract state machine diagram the *PSC*, defined in 5.4.1, is constructed.

Task 3a. Based on the *PSC* the *PC,* which contains the vendor specific implementation, is built.

Task 4a. The last task involves the execution of the *platform basic service* and is performed by the application developer in order to integrate the service in the *cloud application*. For that reason the *Platform Service Execution Controller* (*PSEC*) is used.

2. ***Platform Service API* Description:**

Task 1b. Examination of the web APIs exposed by the available service providers for a given platform service. Similar to **Task 1a** the various web APIs need to be examined in order to create a *Reference API* sufficiently generic to abstract the provider specific APIs.

Task 2b. Based on the API analysis of **Task 1b**, the *Platform Service Reference API* is defined.

Task 3b. This task involves the mapping of the provider specific API to the reference one.

Task 4b. During the *Execution Phase* the *API Client Generator* is invoked and the web clients are generated. The application developers use the latter in order to invoke the specific service providers.

5.5 Summary

This Chapter introduced the reader to the *SCADeF* framework, which aims to facilitate the application developers to build *service-based cloud applications* by seamlessly utilising the *platform basic services* and the concrete providers. To this end, the framework adopts a concrete methodology, as stated in Section 5.4.3, in order to address the three variability points, as proposed in Section 5.1 namely, the differences in the workflow and in the *API* across the various *platform basic service providers,* as well as the management of the various required configuration and authentication settings.

Specifically, the framework provides the components and the tools to enable the description of two parts, namely: (i) the workflow and (ii) the *API* of the *platform basic services*. For each part two stages are defined: (i) the *Modelling* and (ii) the *Execution*. During the first stage the functionality of new *platform basic services* and providers are modelled and inserted in the framework. The latter stage enables the application developers to utilize the framework by integrating the supported services to their *cloud application*s. The *Modelling Stage* further includes the *Platform Service Modelling* and the *Vendor Implementation Phase*, while the *Execution Stage* involves the *Execution Phase*. Each phase requires certain tasks to be performed.

The next Chapters examine in details the tasks that have been defined in each phase (Figure 21) and the components, which are involved (Figure 19). Particularly, Chapter 6 and Chapter 7 analyse the *Modelling* and the *Execution Stage* of the *Platform Service Workflow Description* part respectively. Subsequently, Chapter 8 and Chapter 9 examine the Modelling and the *Execution Stage* of the *Platform Service API Description* part respectively.

Chapter 6

Modelling Stage of the Platform Service Workflow

The preceding Chapter introduced the reader to the *SCADeF* framework, which is proposed in this research work. The aim of the framework is to provide the developers with the functionality of the *platform basic services while* at the same time hides the specific vendor implementations. To this end the process of providing a new service and hiding the provider implementation consists of two parts namely the description of the workflow and the *API*. Each part further involves two stages, as described in Section 5.4 and shown in Figure 18 namely, the *Modelling* and the *Execution Stage*.

The aim of this Chapter is to describe the *Modelling Stage* of the *Platform Service Workflow*. This part contributes to the alleviation of the variations in the workflow observed among the concrete service providers. For example in the case of the cloud payment service, as mentioned in Section 5.1, there are several providers such as *Spreedly*, *PayPal* and *Braintree*, adopting a different workflow during the operation of charging a card. This Chapter describes how an abstract workflow can be defined and subsequently how the abstract one can accommodate the providers' specific workflows. This process takes place during the *Workflow Modelling* and the *Vendor Implementation Phase* respectively.

The structure of this Chapter is as follows:

Section 6.1 describes the cloud payment service. This service will be used as an example throughout the rest of this Chapter.

Then Section 6.2, describes the *Platform Service Modelling Phase.* Particularly, this phase involves studying the workflows of the various service providers in order to conclude to the reference workflow, which is exposed to the developers. For that reason several payment service providers have been examined. Subsequently, the reference workflow needs to be modelled so that the framework can handle it. The Section describes the tools and the components involved in the modelling phase.

Once the reference workflow is modelled, Section 6.3 discusses how the vendor implementation is modelled and inserted in the framework. This process involves the construction of the *PC.*

6.1 The cloud payment service example

The payment service enables a website or an application to accept online payments via electronic cards such as credit or debit cards. The added value that such a service offers is that it relieves the developers from handling electronic payments and keeping track of the transactions by intermediating between the *cloud application* and the payment authorities (Figure 22 and Figure 23). The payment provider undertakes the task to verify the payment and subsequently informs the application about the outcome of the transaction.

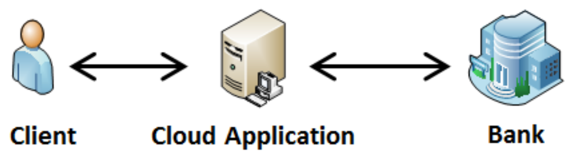


Figure 22: Simplified view of the payment process

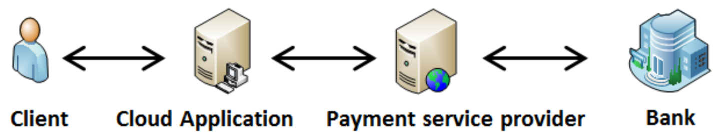


Figure 23: Simplified view of the payment process including the payment service provider

Electronic payments tend to become an essential part of a *cloud application*. The E-commerce field is already well established and has gained wide acceptance in many domains of business. As a result several *cloud platforms* offer a payment service. For example *Heroku* offers *Spreedly*, a payment service created by an *ISV*, *Amazon* offers *Stripe* via its marketplace and *GAE* offers its own native payment service called *Google Wallet* [208].

The payment service has been chosen because of its inherent relative complexity compared to other services such as e-mail or image processing service. The complexity lies in the fact that the purchase transaction requires more than one state to be completed and there is a significant heterogeneity among the available payment providers with respect to the involved states. Moreover, any application which performs billing transactions requires compliance with the *Payment Card Industry Data Security Standard (PCI-DSS)* [209] in order to maximise its reliability. Acquiring the compliance may be a time consuming and costly process. Therefore, by using an existing cloud payment service provider, developers may skip the process of becoming *PCI-DSS* compliant.

6.2 Platform Service Modelling Phase

As depicted in Figure 21, the *Platform Service Modelling* Phase involves two Tasks: (i) the analysis of the workflow of the available *platform service* providers and (ii) the construction of the *PSC*.

6.2.1 Study the Workflow of the Platform Service Providers

The first step towards describing the workflow of the payment service is to explore the concrete payment providers and extrapolate the common states in which they may co-exist. For that reason several providers, have been studied and of those 9 major payment service providers, listed in the Appendix A. They are primarily provisioned either via a major *cloud platform* such as *GAE* and *Amazon AWS* or via *platform service* marketplaces such as *Heroku* add-ons and *Engine Yard* add-ons. After studying the payment process of each of those payment providers we have been able to cluster them in three categories, namely: (i) the transparent redirect, (ii) the server to server, and (iii) the hosted payment process category.

The clustering has been performed according to the payment process that each of the providers adopt. A detailed description of the process per provider can be found in the Appendix A.

6.2.1.1 *Transparent Redirect*

The transparent redirect process is depicted in Figure 24. The following steps are applied in order to complete the payment process.

1. The client requests a new purchase from the application (e.g. an e-shop)
2. The application displays the fill out form, where the client is required to fill in his personal and card details.
3. Once the form is submitted, the client is redirected to the payment provider and the latter receives the card details.
4. The payment provider sends to the application a transaction token which corresponds to the client`s card details.
5. The application uses the transaction token to request a new charge for the amount of the purchase.
6. The payment provider executes the transaction.
7. The bank or the payment processor responds to the payment provider with the outcome of the transaction.
8. The payment provider responds to the application with the outcome of the transaction.
9. The application displays the outcome to the client.

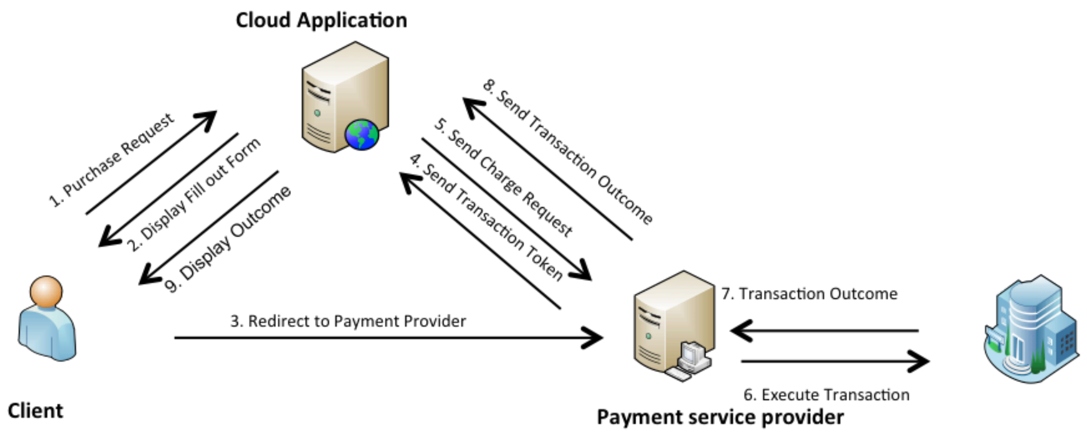


Figure 24: "Transparent redirect" Payment Process

Figure 25 shows the UML state machine diagram of the *cloud application* throughout the transaction. Two states are observed[1]. 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.

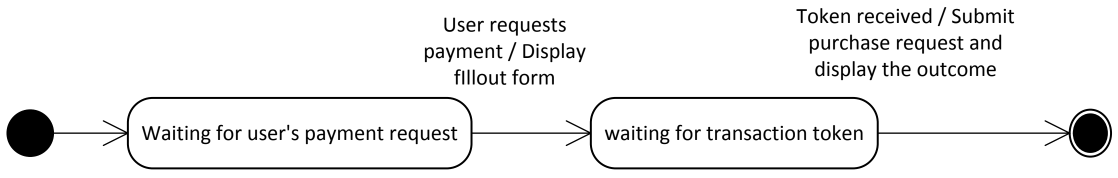

User requests payment / Display fIllout form
Token received / Submit purchase request and display the outcome
Waiting for user's payment request
waiting for transaction token

Figure 25: State machine diagram of the "transparent redirect" payment process

Subsequently, the *cloud application* moves to the next state where it waits for the transaction token issued by the payment provider. The transaction token uniquely identifies the current transaction and can be used by the *cloud application* to complete the purchase. Once the user submits the form, the user 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.

[1] The number of requests is determined by the number of incoming requests to the *cloud application.* In this example, the application receives two requests and thus two states are defined.

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. The main advantage of this payment process is that the *cloud application* does not handle any electronic card data and therefore it is not required to be *PC*I compliant [209].

Payment providers, which adopt the transparent redirect technique, can be accommodated by the state machine diagram shown in Figure 25.

6.2.1.2 *Server to server*

The server to server process is depicted in Figure 26. The following steps are applied in order to complete the payment process.

1. The client requests a new purchase from the application (e.g. an e-shop).
2. The application displays the fill out form, where the client is required to fill in his personal and card details.
3. The client fills in the card details and submits the form. The card details are received by the *cloud application*.
4. The application uses the card details to request a new charge for the amount of the purchase.
5. The payment provider executes the transaction.
6. The bank or the payment processor responds to the payment provider with the outcome of the transaction.
7. The payment provider responds to the application with the outcome of the transaction.
8. The application displays the outcome to the client.

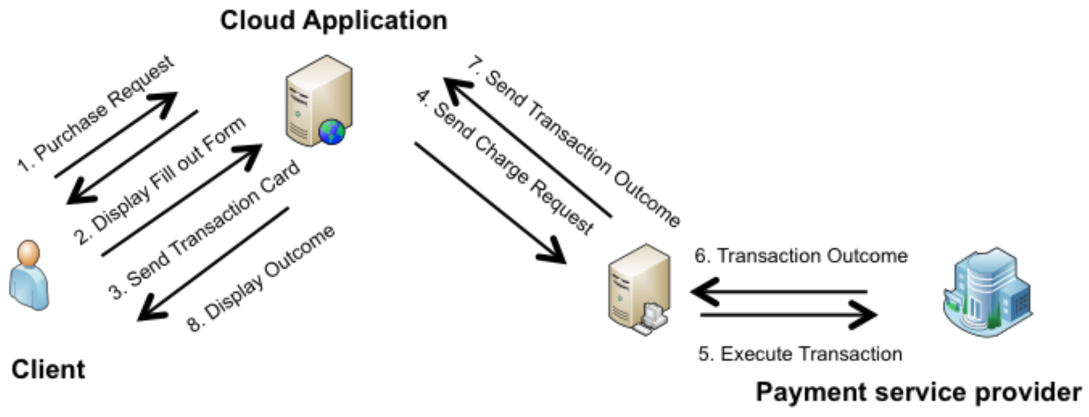


Figure 26: "Server to server" payment process

Figure 27 shows the state machine diagram of the *cloud application* throughout the transaction. Similar to the transparent redirect process, 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.

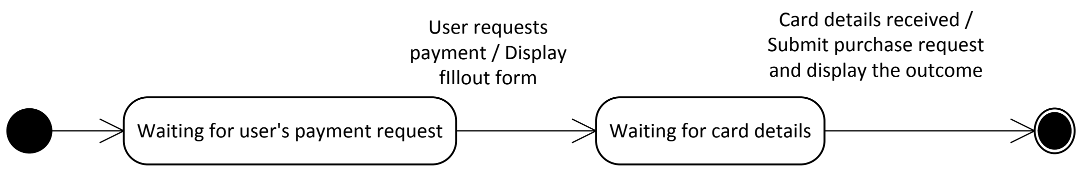

User requests payment / Display fIllout form
Card details received / Submit purchase request and display the outcome
Waiting for user's payment request
Waiting for card details

Figure 27: State machine diagram of the "server to server" payment process

Subsequently, the *cloud application* moves to the next state where it waits for the card details of the client. Once the user submits the form, the application receives the card details. Thereafter, 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.

The server to sever payment process is easier to implement than the transparent redirect. However, in this case the application handles electronic card details and therefore it is required to be partially *PCI* compliant. This adds an extra cost for the application development. In order to avoid this cost, the payment providers which adopts this process provides the developers with software libraries which encrypt and securely handle the card details ensuring this way *PCI* compliance.

6.2.1.3 *Hosted Payment Pages*

The hosted payment process is depicted in Figure 28. The following steps are applied in order to complete the payment process.

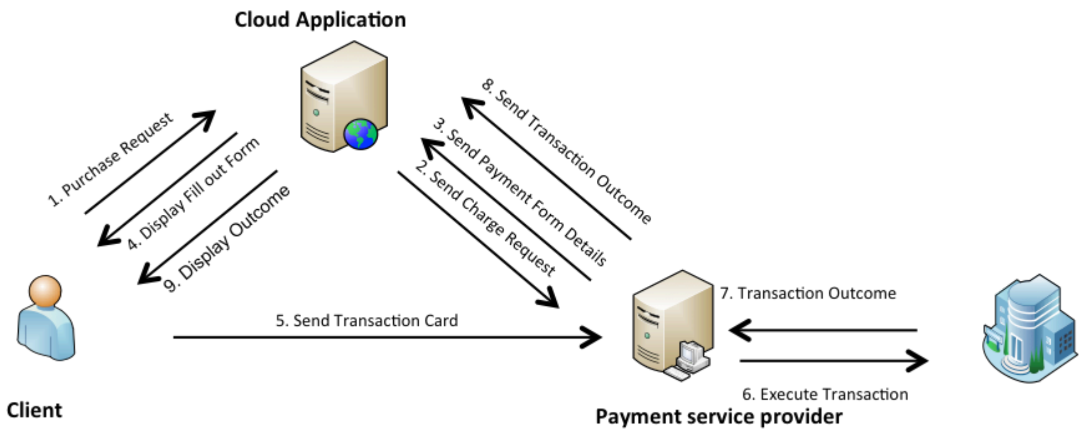


Figure 28: "Hosted payment pages" payment process

1. The client requests a new purchase from the application (e.g. an e-shop).
2. The application requests a new charge for the amount of the purchase.
3. The payment provider responds with the redirect URL for the payment form with the predefined amount.
4. The application redirects the user to the payment form hosted by the payment provider.
5. The client fills in the card details and submits the form. The card details are received by the payment provider.
6. The payment provider executes the transaction.
7. The bank or the payment processor responds to the payment provider with the outcome of the transaction.
8. The application displays the outcome to the client.

Figure 29 shows the state machine diagram of the *cloud application* throughout the transaction. Similar to the previous two processes, 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* requests from the payment provider a new charge with the predefined amount. The provider responds with the *redirect URL* or an *ID*, which guide the client to the payment form hosted by the payment provider. Thereafter, the *cloud application* moves to the second sate where it waits for the transaction code from the provider. Once it receives it, it sends a request to the provider confirming the transaction. The provider responds with the outcome of the transaction.

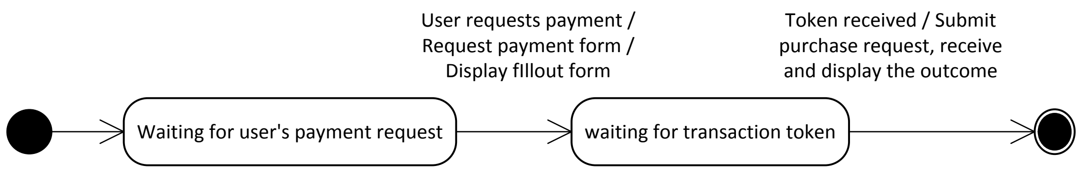

User requests payment / Request payment form / Display fIllout form
Token received / Submit purchase request, receive and display the outcome
Waiting for user's payment request
waiting for transaction token

Figure 29: State machine diagram of the "hosted payment page" payment process

The hosted payment pages payment process differs from the previous methods in the sense that the payment form is hosted by the payment provider and not by the application. This way the client is ensured that the card is charged exactly the agreed amount.

6.2.1.4 *Reference Payment Service Workflow*

After having studied the payment service providers, clustered them according to the payment process and built the respective state charts we can conclude that the providers can be grouped in one state machine diagram which includes two states (Figure 30):

1. Initial State: Waiting for user`s payment request. Transition Event: Payment request arrives. Action: Calculates and displays the fill out form
2. Second State: Waiting for Transaction details. The transaction details can either be the card details or the transaction token. *Transition Event*: Transaction details are received. *Action*: Application handles the purchase transaction, which involves requesting a new charge, and displaying the outcome of the transaction to the user.

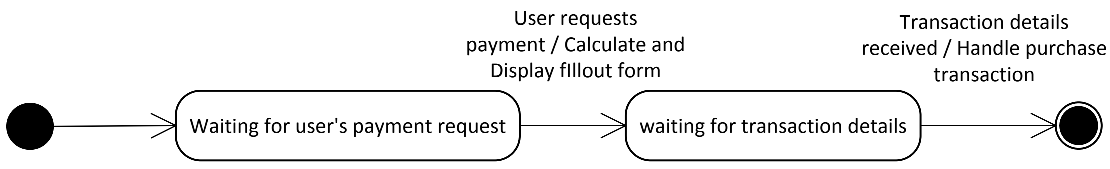

User requests payment / Calculate and Display fIllout form
Transaction details received / Handle purchase transaction
Waiting for user's payment request
waiting for transaction details

Figure 30: State machine diagram of the abstract payment process

6.2.2 Build the Platform Service Connector

The next task of the *Platform Service Modelling* phase is the construction of the *PSC*. As mentioned in 5.4.1, the *PSC* contains the abstract model of the workflow of the platform service. In order to enable the construction of the *PSC*, the *Reference Meta-Model* depicted in Figure 31 is defined.

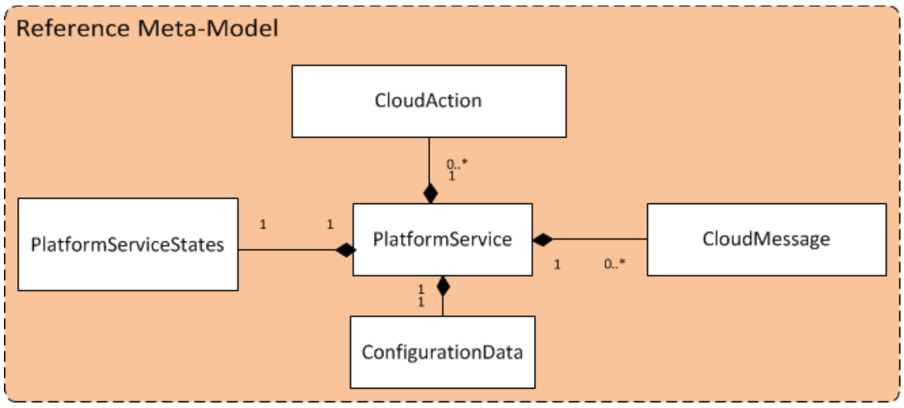


Figure 31: Reference Meta-Model

6.2.2.1 *The Reference Meta-Model*

The concepts of the meta-model have been defined based upon the need of the application to communicate with the *platform basic services* and coordinate the execution of the workflow for the completion of the operations. There are two basic operations, which the *cloud application* should be capable to perform in order to integrate a *platform basic service*. Firstly, it needs to be able to receive the requests which are initiated either by the user of the application or by the *platform basic service*. Secondly, the *cloud application* should be capable of sending requests to the service provider using the web *API* of the latter.

Following the requirements that the *Reference Meta-Model* should meet, the following concepts are defined:

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 the appropriate 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 from the client of the application or from the service provider, as shown in Figure 32. A separate *CloudAction* is defined to handle each incoming request and subsequently signals the transition to the next state. *CloudActions* leverage the Servlet programming model [210] in order to receive the requests and respond to the callers.

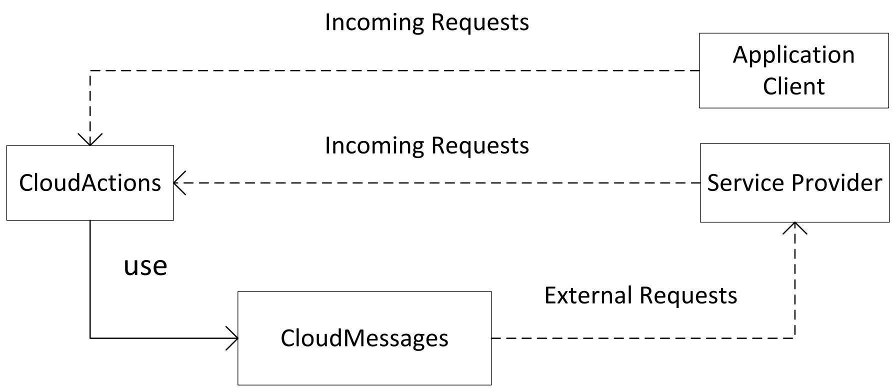


Figure 32: Role of the CloudActions and the CloudMessages

CloudMessage: *CloudMessages* can be used to perform requests from the *cloud application* towards the service provider using the web *API* of the latter (Figure 32). *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. A different *CloudMessage* is defined to implement each one of the operations offered by the service provider via the web *API*. For example in the case of the e-mail service, a *CloudMessage* is defined to send an e-mail using the web *API* of the service provider and including the required fields: *recipient*, *sender*, *title* and *body*.

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. The use of an *XML* file instead of a Java Enumeration type [211] to encode the states promotes the flexibility of the framework when the latter is required to change the states or the actions involved in an operation at run-time. An excerpt of a *Platform ServiceState* file is shown in Listing 2.

ConfigurationData*:* Certain configuration settings are required by each *platform service* provider. That information is captured in the *ConfigurationData*. Example of settings which needs to be defined are the clients' credentials required to perform web requests, authentication tokens and the redirect URL parameter which is often requested by the service provider in order to perform requests to the *cloud application*. An excerpt of the *ConfigurationData* is depicted in Listing 5.

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 [212] [213]. 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*.

6.2.2.2 *The Platform Service Connector (PSC)*

The *Reference Meta-Model* is used to construct the *PSC*. The *PSC* essentially consists of *CloudActions* and *CloudMessages,* which are used for the communication of the application with the *platform basic services,* and the *PlatformServiceStates* file which describes the sequence of execution of the *CloudActions*.

The *PSC* is created based upon the state machine diagram which is defined during the **Task 1a** of Figure 21 and is an abstract representation of the workflow of the examined *platform basic service* providers. Two rules are applicable during the construction of the *PSC*. The rules are based on the definition of the *CloudActions* and the *CloudMessages* mentioned earlier in 6.2, 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 33 shows the *Cloud Payment Service Connector*. It is constructed based on the state machine diagram defined in Figure 30 and using the concepts of the *Reference Meta-Model*. It consists of the following blocks:

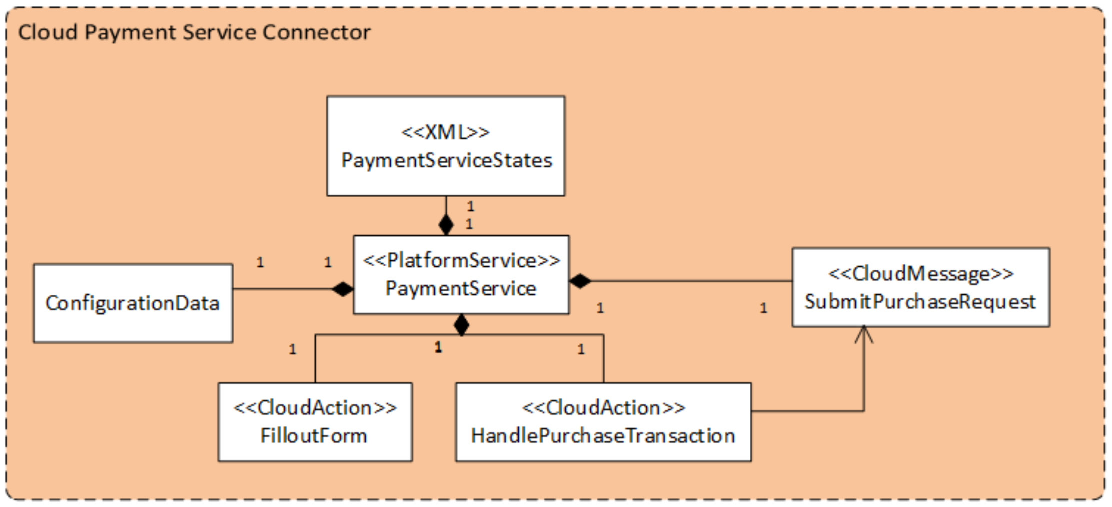

Cloud Payment Service Connector
<<XML>> PaymentServiceStates
<<PlatformService>> PaymentService
ConfigurationData
<<CloudMessage>> SubmitPurchaseRequest
<<CloudAction>> FilloutForm
<<CloudAction>> HandlePurchaseTransaction
1
1
1
1
1
1
1
1
1

Figure 33: Cloud Payment Service Connector

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. The communication is realised using the servlet technology.

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.

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.

ConfigurationData: The *ConfigurationData* contains the service settings required to complete the purchase operation. Particularly, the following pieces of information are listed: the *redirectUrl*, the *username,* and the *password*.

PaymentServiceStates: In the *PaymentServiceStates* file the states and the corresponding actions involved in the transaction are defined. The framework uses the file in order to guide the execution of the actions. A part of the description file is shown in Listing 2.

Listing 2: Payment service states description file

```
<StateMachine>
    <State name="WaitingPaymentRequest"
        action="org.paymentserviceframework.FilloutFormAction"
        nextState="WaitingTransactionDetails"/>
    <State name="WaitingTransactionDetails"
        action="org.paymentserviceframework.SendTransactionAction"
        nextState="Finish" />
</StateMachine>
```

The state description file essential corresponds to a *Finite State Machine* (*FSM*). An *FSM* defines the states of a system, the transitions between the states as well as the events which trigger a transition, and the produced output. In this research work an *FSM* represents the interaction of the *cloud application* with the *platform basic service* provider. Such an *FSM* is represented by the state diagram shown in Figure 30. Typically, a state transition table is required to define the transition from one state to the next one. However, the *platform basic services*, which have been examined, expose only a single transition from each state to the next one. Thus, a simplified version of a state transition table has been defined (Listing 2). In future, in case multiple transitions are possible from each state, the state description file can be extended to accommodate those transitions.

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.

6.3 Vendor Implementation Phase

During the *Vendor Implementation Phase* the provider specific connectors are constructed.

6.3.1 Build Provider Connectors

After having defined the *PSC*, the specific implementation and settings of each concrete provider needs to be infused (**Task 3a** of Figure 21). For each *CloudAction* and *CloudMessage* defined in the *PSC*, the respective provider specific blocks should be defined forming the *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 34. It contains the following blocks:

SpreedlyFilloutForm: This is a type of *CloudAction* implementing the *FilloutForm*.

SpreedlyHandlePurchaseTransaction. This is a type of *CloudAction* implementing the *HandlePurchaseTransaction.*

SpreedlySubmitPurchaseRequest: This is a type of *CloudAction* implementing the SubmitPurchaseRequest.

ConifgurationData: This file contains the specific configuration settings, which are required by the service provider. Therefore the file needs to be updated accordingly in order to match the specific provider. For example in the case of *Spreedly* on top of the generic payment service settings such as the *redirect URL* and the credentials, the *GatewayToken* is also expected.

Should the provider's implementation accurately match the model, the provider specific *CloudActions* and *CloudMessages* can reuse the functionality of the generic model. In case where the provider's implementation diverges from the generic model the model's functionality can be overridden.

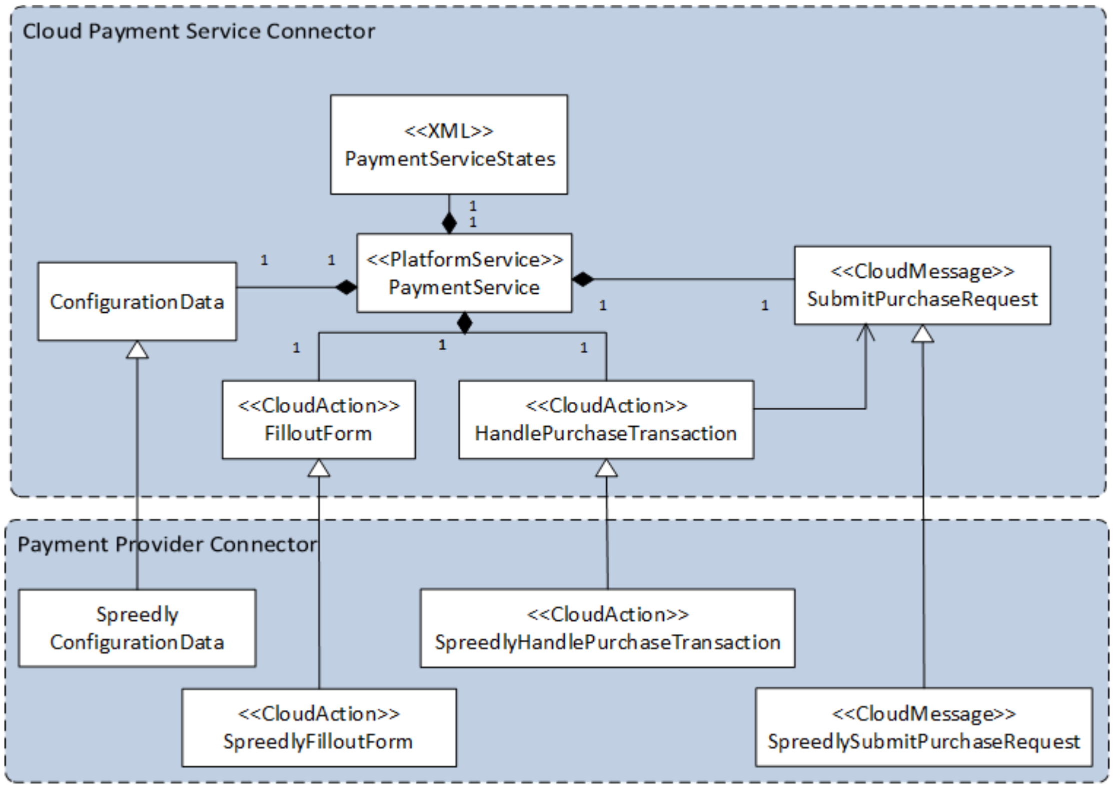


Figure 34: Cloud Payment Service Provider Connector

6.4 Summary

This Section concludes the description of the *Modelling Stage* of the workflow of the *platform services*. At this point, the workflow of the *platform service* has been described and an abstract model, namely the *PSC*, has been constructed. For each service provider which is supported by *SCADeF,* a specific vendor implementation has been implemented and inserted in the *PC* component. Both the *PSC* and the *PC* have been constructed with the use of the concepts defined in the *Reference Meta-Model*.

The work presented in this Chapter supports the Requirement 1 (R1), listed in Section 5.3, regarding the capability of the framework to support the workflow modelling of the *platform basic services* and the concrete providers. This task is performed by the *administrator* of the framework, which is responsible for enriching the framework with additional services and providers. Therefore, the Requirement 7 (R7), regarding the distinct role of the *administrator* is also fulfilled. Moreover, the ability of the framework to accommodate additional services and providers contributes to the Requirement 5 (R5), which refers to the generic and flexible nature of the framework.

The next Chapter describes the components and the mechanism available for executing the workflow, which was captured during the *Modelling Stage*. The execution mechanism allows application developers to use the operations defined in the supported *platform basic services.*

Chapter 7

Execution Stage of the Platform Service Workflow

Chapter 6 showed how the *PSC* and the *PC* containing the reference and the provider specific workflows respectively can be constructed. The next step is to allow the *SCADeF* framework to control the execution of the workflow when a specific operation is invoked by the application. This way the software developers do not need to coordinate the execution of the workflow and be aware of the specific sequence of steps required by each provider. To this end the aim of this Chapter is to describe how the framework can handle automatically the execution of the workflow.

Particularly, Section 7.1 describes the main component, which is responsible for the execution of the workflow, namely the *Platform Service Execution Controller (PSEC)*. During the *Execution Phase* the *PSC* and the *PC,* are managed by the *PSEC.*

Then, Section 7.2 illustrates the sequence of events which take place during the execution of the workflow. For that reason the payment service workflow defined in Section 6.2.1.4 is used.

Section 7.3 concludes the Chapter by mentioning major design patterns, which have been adopted throughout the design of *SCADeF,* such as the *Front Controller* and the *Factory* pattern.

7.1 The Platform Service Execution Controller (PSEC)

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 35.

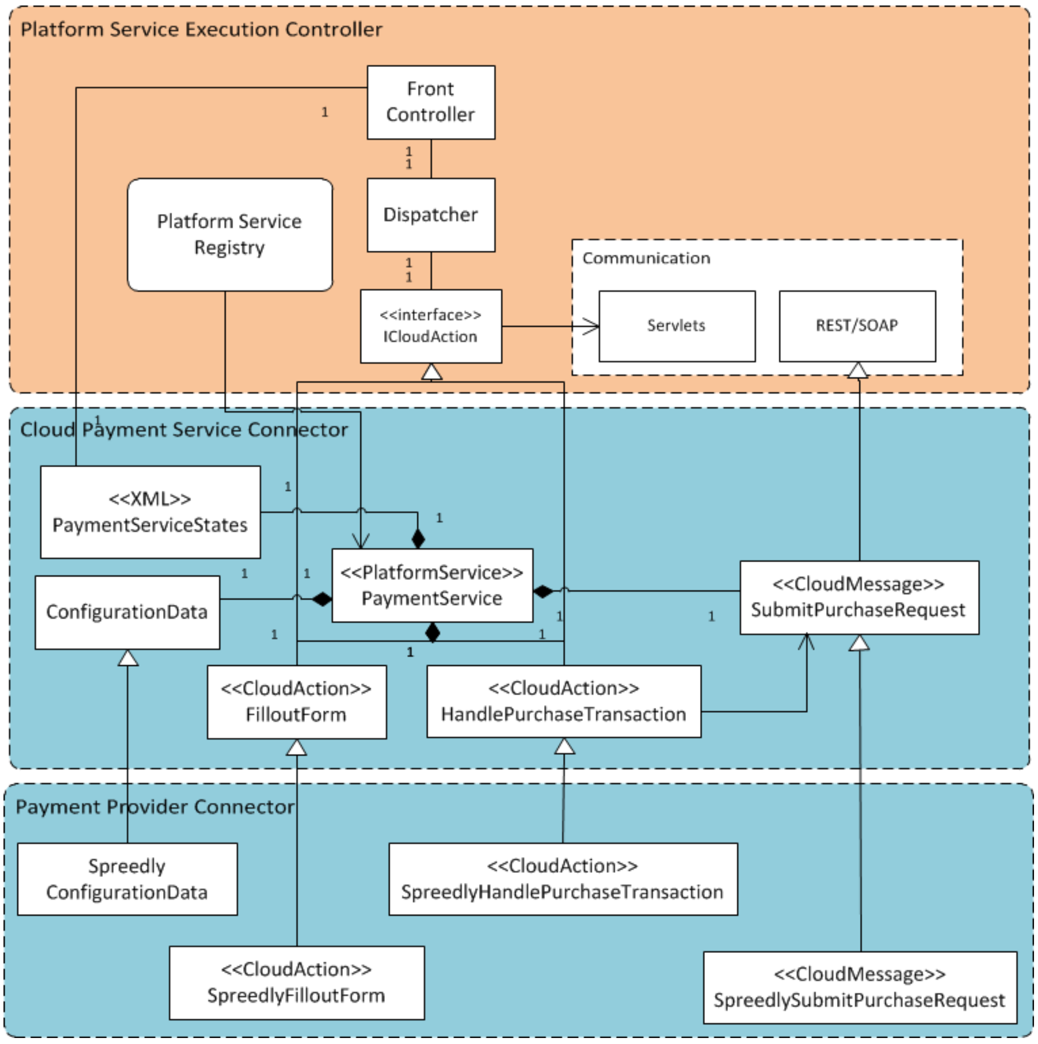

Platform Service Execution Controller
Front Controller
Dispatcher
Platform Service Registry
<<interface>> ICloudAction
Communication
Servlets
REST/SOAP
Cloud Payment Service Connector
<<XML>> PaymentServiceStates
ConfigurationData
<<PlatformService>> PaymentService
<<CloudMessage>> SubmitPurchaseRequest
<<CloudAction>> FilloutForm
<<CloudAction>> HandlePurchaseTransaction
Payment Provider Connector
Spreedly ConfigurationData
<<CloudAction>> SpreedlyHandlePurchaseTransaction
<<CloudAction>> SpreedlyFilloutForm
<<CloudMessage>> SpreedlySubmitPurchaseRequest

Figure 35: Platform Service Execution Controller

Front Controller: The *Front Controller* [210] serves as the entry point to the framework. It receives the incoming requests by the application user and the service provider.

Dispatcher: The *Dispatcher* [210] 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 *ICloudAction,* which is explained below. As stated in Section 6.2, the requests are handled by the *CloudActions*. Therefore the *Dispatcher* forwards the request to the appropriate *CloudAction*. In order to do so the *Dispatcher* gains access to the *platform service* states description file and based on the current state it triggers the corresponding action.

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 [214]. *ICloudAction* defines a method called *execute(),* which all *CloudActions* need to implement.

Communication patterns: The framework supports two types of communication pattern. The first one makes use of the *Java Servlets* and in particular the *HTTP* Servlet Request and Response objects [210]. These objects 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 enable the *CloudMessages* to perform external requests to the service providers.

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*. Its purpose is to provide the software engineer with a mechanism for deploying and releasing services.

7.2 PSEC sequence of execution

In order to illustrate how the *PSEC* enables the execution of the workflow, the example of the payment service is described. Figure 36 shows the execution flow of the payment service. Particularly, it depicts the transition from the first to the second state as depicted in Figure 30.

1. The client submits a purchase request to the *cloud application*.
2. The *Front Controller* receives the request. As mentioned above, it is the entry point to the framework. All incoming requests are received by this

component. Subsequently, the *Front Controller* forwards the request to the *Dispatcher*.

3. The *Dispatcher* needs to instantiate the appropriate *CloudAction* to handle the request. In order to do that it reads the *PaymentServiceStates* description, which lists the states and the respective actions to be called. The *ActionFactory* is used to instantiate all *CloudActions*.
4. Once the *Dispatcher* obtains an instance of the concrete *CloudAction* (*FilloutFormAction*), it calls the *execute()* method. As mentioned earlier in Section 7.1, all *CloudActions* implement the interface *ICloudAction*.
5. The Payment Service Provider sends the transaction token to the *CloudApplication*.
6. The *Front Controller* receives the request and forwards it to the Dispatcher.
7. The *Dispatcher* calls the *ActionFactory* and receives the concrete Action, *HanldePurchaseRequest CloudAction.*
8. The *HandlePurchaseRequest CloudAction* needs to submit a charge request to the payment provider. As stated in Section 6.2.2.1, *CloudMessages* are responsible for submitting web requests. Therefore, it obtains an instance of the *SubmitPurchaseRequest CloudMessage* using the *CloudMessageFactory.*
9. The *SubmitPurchaseRequest CloudMessage* sends a web request to the service provider. Then it receives the response and forwards it back to the *CloudAction*. The *CloudAction* evaluates the response and replies accordingly to the client who initiated the request.

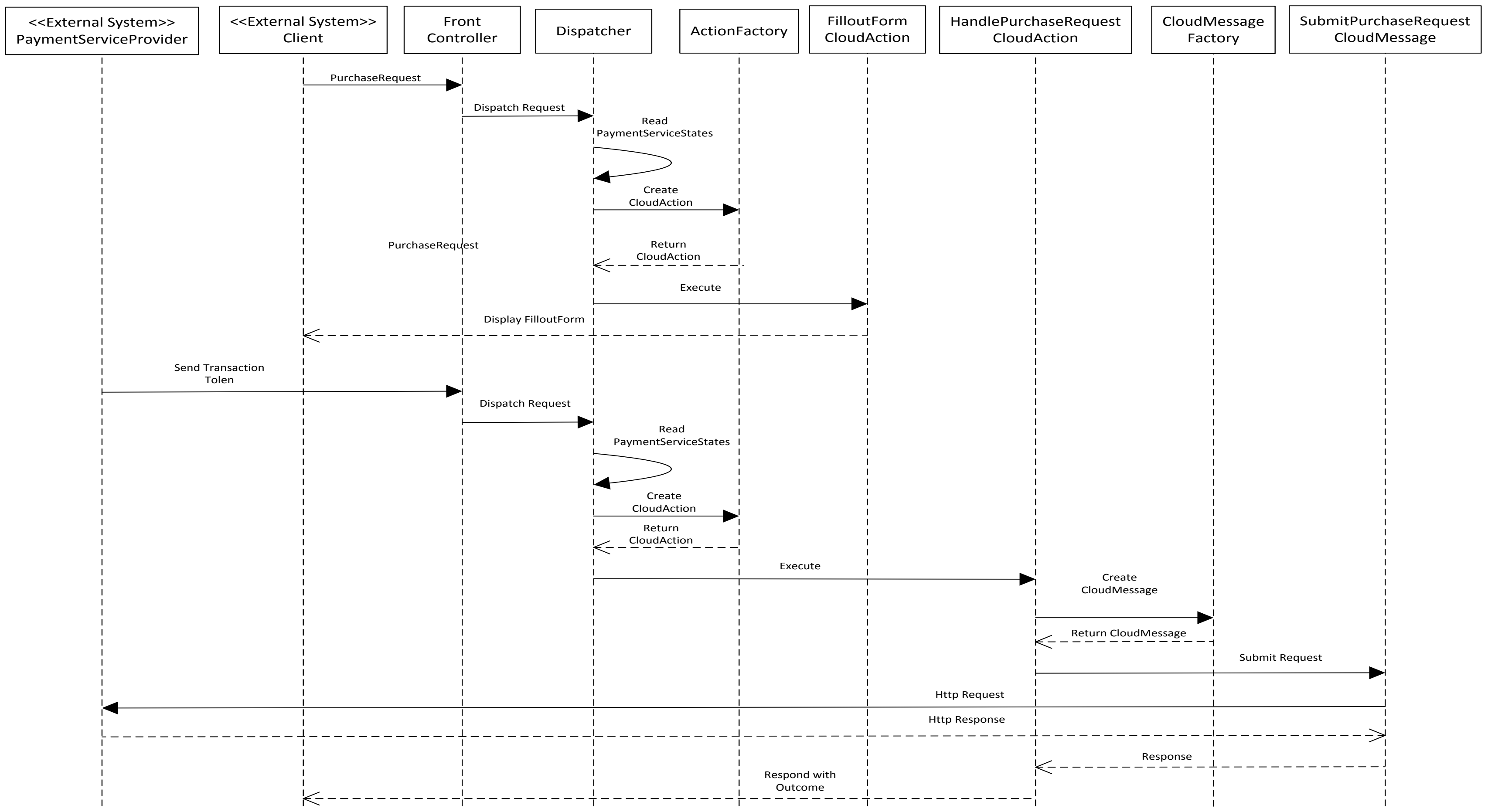


Figure 36: Payment service execution flow

7.3 Design patterns used in the *SCADeF* framework

Throughout the design of the development framework, design patterns have been adopted [162]. They describe certain classes and their interrelationships in order to address a general design problem. The use of such patterns promotes composability, maintainability and portability, features that should accompany the development of modern software. Particularly, the following software design patterns have been adopted:

1. **Factory:** This pattern, depicted in Figure 37, is used to enable the creation of concrete objects without the client knowing which exact object is instantiated. This way the client is decoupled from the process of object creation and thus new types of objects can be added without the need to change the client`s code. The factory pattern is used to instantiate *CloudActions*. The *Dispatcher* is the client who requests the concrete objects. New *CloudActions* can be added without the need to change the *Dispatcher*`s code.

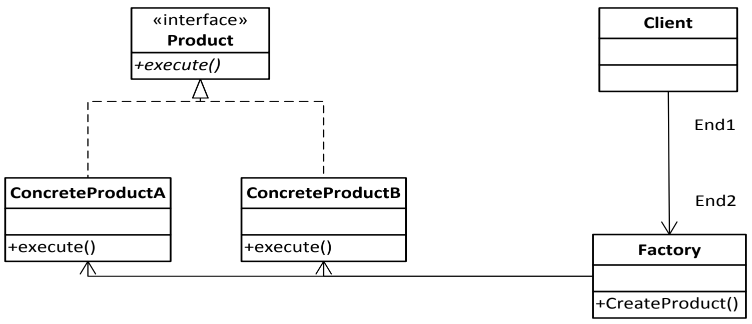


Figure 37: Factory Pattern

2. **Front Controller:** The *Front Controller* pattern, shown in Figure 38,, is used in web-based applications and constitutes the entry point to the application. The *Front Controller* component receives all incoming requests and forwards them to the *Dispatcher*. The latter determines how the requests are handled. The controller may perform initial tasks applicable to all incoming requests such as authentication and authorization. In the development framework, the *Front Controller* receives the requests either from the client of the application or from the

service provider and forwards them to the *Dispatcher* who decides on which action should handle the request.

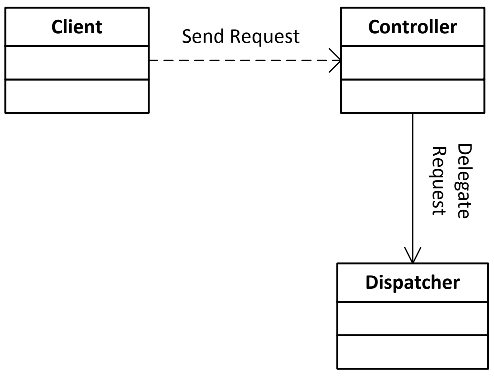


Figure 38: Front Controller

3. **Template Method:** The *Template Method* pattern, in Figure 39,, defines a sequence of steps to execute a task. It allows subclasses to alter the way certain steps are executed without changing the order. In the development framework, this pattern is used to construct the internal behavior of the *CloudActions*.

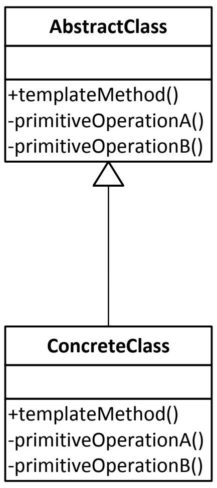


Figure 39: Template Method

The *CloudActions* in the *PSC* define the steps required to handle the request. However, the way these steps are executed may vary across the service providers. Therefore, the *CloudActions* defined in the *PC* may override the ones in the *PSC*. For example, the *HandlePurchaseRequest CloudAction*, defines two steps: (i) submit charge request to the service provider and (ii) display transaction outcome. While, these steps are applicable across the supported payment providers, the way they are implemented differs. Thus the *PC*s are constructed to capture the differences.

7.4 Summary

This chapter focused on the *Execution Stage* of the workflow of the *platform basic service*. During this stage, the *Platform Service Execution Controller* (*PSEC*) automates the execution of the workflow and thus the *consumers* of the framework can use the various *platform basic services* without being concerned with the specific workflow of the concrete providers. The work presented in this chapter supports the Requirement 2 (R2), regarding the automatic execution of the workflow by the development framework.

The next Chapter focuses on the *Modelling Stage* of the *Platform Service API Description* part. Specifically, it describes the definition of the *Reference API* and the subsequent mapping of the provider specific web *API* to the reference one.

Chapter 8

Modelling Stage of the Platform Service API

Section 5.1 stated the main variability points that may arise across the different categories of *platform basic services* and the corresponding providers which the proposed development framework addresses. Those are: a) the differences in the workflow during the execution of the operations of the various service providers, b) the differences in the web *API* published by the providers, and c) the variations in the configuration and authentication settings required by each provider.

Chapter 6 described the methodology, which is followed in order to address the differences in the workflows. Particularly, it involved the definition of the reference workflow and the subsequent mapping of the provider specific workflow to the reference one. Chapter 7 discussed the way the execution of the workflow can be automated by the *SCADeF* framework.

An analogous process is followed in this chapter in order to alleviate the differences among the web *APIs* offered by the various service providers. Addressing the heterogeneities among the *APIs* will further promote the wide exploitation of the *platform basic services*. It will also contribute towards enabling the software engineers to choose seamlessly the optimal service providers, given each time certain requirements such as the cost and the quality of the offered service. The configuration settings, which is the third variability defined in Section 5.1, is also required during the construction of the web *APIs* and thus contributes to the heterogeneity raised among the providers. Therefore this chapter additionally describes the way the various configuration settings are handled by *SCADeF*.

The structure of the Chapter is as follows:

Sections 8.1, 8.2 and 8.3 introduce the reader to the heterogeneities which may arise among the web *APIs* of the various providers and gives a high level overview of the proposed solution. The example of the e-mail service is used throughout this Chapter to demonstrate how the proposed solution can abstract the differences in the web *APIs* of the e-mail service providers.

Then, Sections 8.4 and 8.5 describe the *Modelling Stage* of the *API* description, which is divided into the *Platform Service Modelling Phase* and the *Vendor Implementation Phase* respectively, as shown in Figure 18 of Section 5.4. Particularly, Section 8.4 defines the methodology for creating the *Reference API* exposed to the developers while Section 8.5 focuses on capturing the provider specific *API* and mapping it into the reference one. In order to complete the construction of the *API* several configuration settings may be required depending on the provider. Section 8.6 describes the way they are captured and handled by the framework.

8.1 API variability example

Table 5 and Table 6 show two examples of *API* variability, which may be encountered across certain service providers.

Table 5: API variability in the "SendE-mail" operation of the e-mailing service

***SendGrid* (*Heroku*)**	from	to	subject	text
Amazon Simple E-mail Service	Source	Destination. ToAddresses	Message. Subject	Message. Body.Text
***Postmark* (*Heroku*)**	From	To	Subject	TextBody

The first one lists the parameters expected by three e-mailing service providers, namely the *SendGrid* [35], the *Amazon Simple E-mail Service* [200] and the *Postmark* [37] for the operation of sending an e-mail. The differences in the parameters in the *API* of the three service providers are illustrated.

Likewise, the second table displays the parameters expected by two payment service providers, namely *Spreedly* and *Stripe,* for the operation of charging a payment card. There is a significant heterogeneity in the API offered by the providers.

Table 6: API variability in the "chargeCard" operation of the cloud payment service

***Spreedly* (*Heroku*)**	amount	payment_method_token	currency_code
Stripe (Amazon)	amount	card	currency

Both examples demonstrate the differences in the parameters across the *APIs* of several providers offering the same *platform basic service*. This variation results in changes in the code when different providers need to be deployed. Therefore a mechanism is required to undertake the task of describing the provider specific *APIs* and hiding the peculiarities of each provider from the software engineers.

8.2 High-level overview of the API abstraction mechanism

Figure 40 shows an overview of the abstraction mechanism. The developer initiates the development of the application using a popular development environment such as *Eclipse* [215] and a programming language such as *Java*. When the application requires a *platform basic service* that is supported by the framework, the *API* description of the service is inserted in the framework. Consequently, the service description is parsed and the source code for the particular service is generated. The abstraction mechanism consists of two main parts, the *API* service description and the generation of the source code. The first one involves the definition of a *Reference API* which is exposed to the users of the framework and the subsequent mapping of the provider specific *API* to the reference one. The second part includes the abstract *platform service* models, which contain the template code common to all the services and the code generator. The latter receives the abstract models and the *Service Description File*, which was produced by the first part and subsequently, generates the source code, which is included in the main software application. The whole process, as it is explained in the next Sections, remains transparent to the users

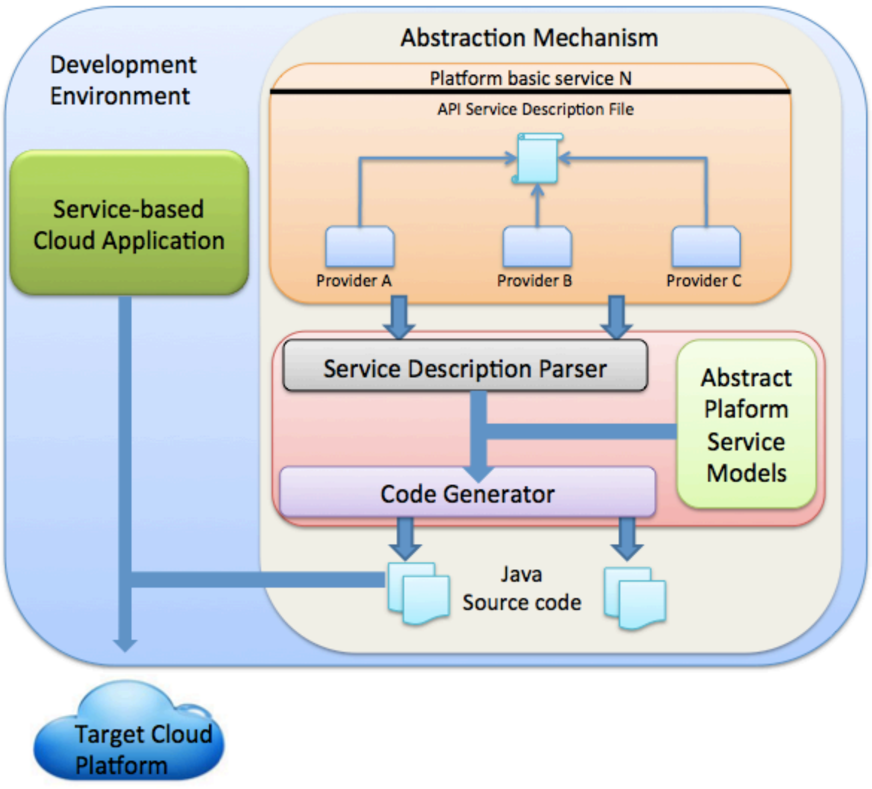


Figure 40: Overview of the API service description mechanism

As mentioned in 5.4 the process of adding a *platform service* and service provider to the framework is completed in three phases: a) the *Platform Service Modelling Phase*, b) The *Vendor Implementation Phase* and c) the *Execution Phase*.

In this chapter the first and second phase are examined, namely the ones including in the *Modelling Stage* of the *Platform Service API Description.*

8.3 The e-mail service example

In order to illustrate how the *SCADeF* framework can facilitate the *API* abstraction, the example of the e-mail *platform service* will be followed. This service has been chosen as an example, since it continuously gains attention and has the tendency to become an essential part of the majority of the *service-based cloud applications*, according to Gartner, the leading information technology research company [216]. Furthermore, e-mail services are provided by all the major *cloud application* platforms such as *Heroku, OpenShift, Engine Yard,* and *AWS*.

The e-mail *platform service* (Figure 41) enables software developers to use mailing functionality within their service-based *cloud application* without the need to set up and maintain their own e-mailing servers. Instead service providers who offer this service expose an *API,* which can be used in order to perform mailing operations such as: send an e-mail, create mailing lists, retrieve sent e-mails etc.

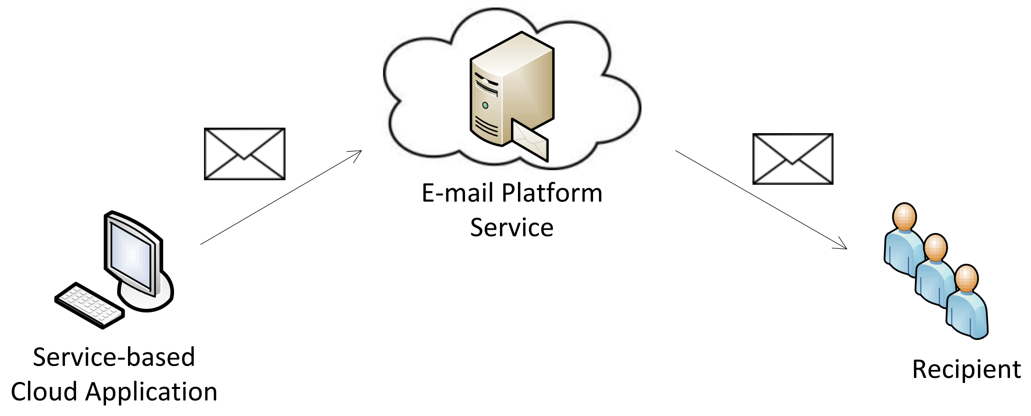


Figure 41: E-mail Platform Service

8.4 Platform Service Modelling Phase

Similar to the *Platform Service Workflow Description* part, where it is shown how the operation flow of the *platform basic service* providers can be abstracted, the goal of this phase is to demonstrate how a *Reference API*, which abstracts the respective web *API* of the concrete service providers, can be defined. Figure 42 shows the mapping of the provider specific *APIs* to the *Reference API*. The software engineers can use the *Reference API* and gain access to the providers who are supported by it. This Section describes the definition of the *Reference API* for a given *platform basic service*.

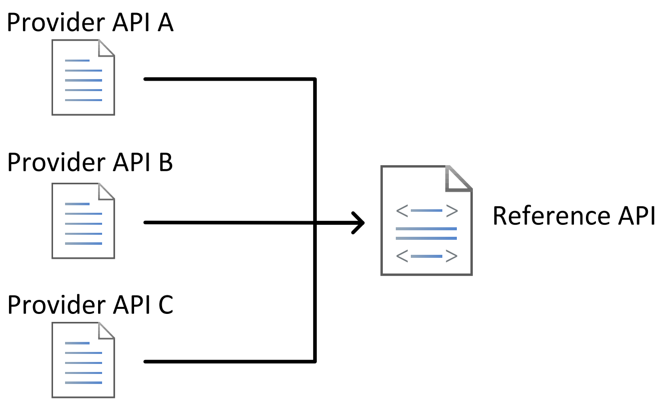


Figure 42: Mapping of the provider specific API to the Reference API

The *Reference API* is defined based on the following steps, which are also shown in Figure 43..

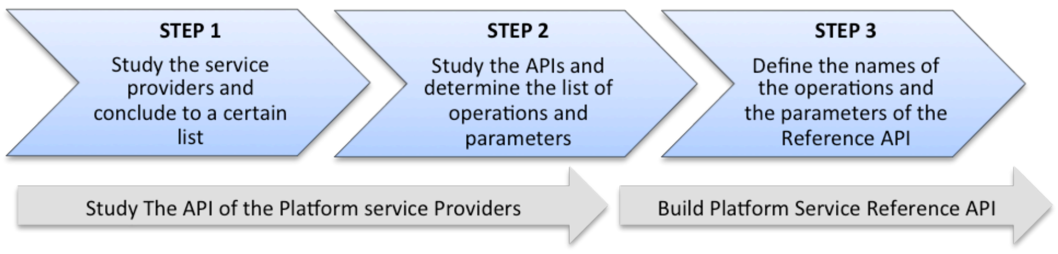

STEP 1
Study the service providers and conclude to a certain list
STEP 2
Study the APIs and determine the list of operations and parameters
STEP 3
Define the names of the operations and the parameters of the Reference API
Study The API of the Platform service Providers
Build Platform Service Reference API

Figure 43: Steps for the definition of the Reference API

1. Study the service providers implementing the particular *platform basic service* and conclude to a certain set to be included in the *Reference API*. A large number of available *platform basic service* providers needs to be examined in order to obtain an insight of the *platform service* and the offered functionality. Then, we need to decide on the concrete list of the providers to be considered for the *Reference API*. The decision is primarily based on whether the service provider is supported by a *cloud application* platform such as *Heroku* and *Engine yard*. Furthermore, the provider needs to support the majority of the common operations and also publish a *RESTful API*.
2. Study the *APIs* of the selected providers. After the examination of the selected providers, the common operations and the parameters, to be included in the *Reference API*, are determined. These operations and

parameters should be supported by all or the majority of the selected providers.

3. Define the parameters and the operations for the *Reference API.* The final step includes the definition of the naming of the operations and the parameters of the *Reference API* that is exposed to the users of the framework. The names are chosen so that they are self-descriptive and as close as possible to the original ones.

Step 1 and **Step 2** are included in the **Task 1b** of Figure 21 of Section 5.4.3, whereas the **Step 3** is included in the **Task 2b.**

8.4.1 Analysis of the API of the platform basic service providers

8.4.1.1 *The e-mail service providers*

E-mail service providers were examined and analysed in order to derive a common set of operations offered by the majority of providers, as well as the expected parameters for each operation. The concrete service providers, which are considered, are primarily those offered via major *cloud application* platforms such as *Heroku*, *Engine Yard* and *OpenShift*. These are the following: *Mailgun*, *SendGrid*, *Postmark*, *Mailjet* [217].

Mailgun is a transactional e-mail service provider which enables developers to send and receive e-mails via its *RESTful API.* It is offered via major application platforms such as: *Heroku, Rackspace, Engineyard, CloudControl* [218] and *Appfog* [96].

SendGrid has been found in 2009 and has become the industry`s leading cloud-based e-mail delivery service. The company offers both transactional and marketing e-mail delivery. *SendGrid* is offered by *Heroku, Rackspace* [219], *Engine Yard*, *OpenShift* and *Cloudbees* [220].

MailJet is an e-mail delivery platform for transactional and marketing e-mails. The company was founded in 2010. *MailJet* is primarily offered via Content Management Systems such as *WordPress, Joomla* and *Drupal.* It has been selected as a candidate for the *Reference API*, due to its growing popularity in the domain of e-mail services and also due to the fact that it provides its functionality via a *RESTful API*.

Similar to the previous providers, *Postmark* offers capabilities to the users for sending, receiving, and manipulating e-mails via a *RESTful API. Postmark* is available via *Heroku* and *Engine Yard* application platforms.

8.4.1.2 *The e-mail service operations to be examined*

Table 7 depicts the operations on which we focus and the operations that each provider supports. As mentioned at the beginning of this Section the example serves the purpose of illustrating how the abstraction mechanism can be employed in practice. Therefore, only a common subset of the operations offered by each of the e-mail service providers is included. Whereas this chapter analyses what is common in the *APIs* offered by different *platform service* providers, the following Chapter 9 will also discuss how to handle operations, which are offered only by specific providers.

Table 7: List of mailing operations supported by the service providers

Providers / **Operations**	*Mailgun*	*SendGrid*	*Postmark*	*Mailjet*
Send E-mail	✓	✓	✓	✓
Create Mailing List	✓	✓	✗	✓
Search	✓	✗	✓	✓
Bounce	✓	✓	✓	✓

1. **Send E-mail**: This is the basic operation of the e-mail service enabling an application to send e-mails.
2. **Create Mailing Lists**: Most of the providers allow the creation of mailing lists to enable bulk send of e-mails.
3. **Search**: This operation enables the software engineers to retrieve sent e-mails based on certain criteria.

4. **Bounce**: E-mails, which have not been delivered due to invalid address can be retrieved using the bounce operation.

Next, for each operation, we list the parameters expected by each provider respectively. For each operation a table is provided followed by the explanation of the parameters. The explanation is provided only for the first provider of the table and is also valid for the parameters of the rest of the providers listed in the same row.

8.4.1.3 *Send e-mail operation parameters*

Table 8: Send e-mail operation parameters

Mailgun	***SendGrid***	***Postmark***	***Mailjet***
to	to	To	to
cc	cc	Cc	cc
bcc	bcc	Bcc	bcc
text	text	TextBody	text
from	from	From	from
subject	subject	Subject	subject

1. **to:** It denotes the recipient of the e-mail.
2. **cc:** It denotes the recipients who are included in the “carbon copy” list.
3. **bcc:** It denotes the recipients who are included in the “blind carbon copy” list.
4. **text:** This parameter includes the body of the e-mail.
5. **from:** It denotes the sender of the e-mail.
6. **subject:** As the name implies, the parameter contains the subject of the e-mail.

8.4.1.4 *Bounce operation parameters*

Table 9: Bounce operation parameters

Mailgun	***SendGrid***	***Postmark***	***Mailjet***
limit	limit	count	limit
skip	offset	offset	skip

1. **limit:** It denotes the maximum number of bounced e-mails to be listed.
2. **skip:** It denotes the number of bounced e-mails to be skipped.

8.4.1.5 *Search operation parameters*

Table 10: Search operation parameters

Mailgun	***SendGrid***	***Postmark***	***Mailjet***
limit	✗	count	limit
recipient	✗	recipient	to_email
from	✗	fromemail	from
tag	✗	tag	from_domain

1. **limit:** It denotes the maximum number of e-mails to be returned.
2. **recipient:** Search criterion based on the recipient.
3. **from:** Search criterion based on the sender of the e-mails.
4. **tag:** Search criterion based on the tagging of the e-mails.

8.4.1.6 *Create mailing list operation parameters*

Table 11: Create mailing list operation parameters

Mailgun	***SendGrid***	***Postmark***	***Mailjet***
name	List	✗	Name

1. **name:** It denotes the name of the mailing list.

8.4.2 Build Platform Service Reference API

The last step in the process of creating the *Reference API*, as shown in Figure 43, is to define the reference parameters for each operation, which are exposed to the developers. The *administrator* of the framework determines the parameters' naming. The main purpose of the reference parameters is to abstract the provider specific ones.

Table 12 shows the reference parameters for each of the operations, which are included in the *Reference API*.

Table 12: Reference API

Send E-mail	Bounce	Search	Create Mailing List
to	limit	limit	listName
cc	offset	recipient	
bcc		from	
text		limit	
from		tag	
subject			

By nature, abstraction can only accommodate the common functionality of the providers under consideration. Therefore, inevitably certain provider specific functionality is left out. Sections 9.3.1 and 9.3.2 discusses how the framework can handle the additional functionality.

8.4.2.1 *The Service Description File*

The information, which is included in the *Reference API*, namely the operations and the parameters of *platform basic service* are captured and represented in an *XML* file. An excerpt of the file, which is known as *Service Description File,* is shown in the Listing 3.

The *XML* file includes the name of the service, namely e-mail service, the supported operations, as those defined earlier in the Section and the parameters included in each operation, which are captured in the *key* attribute. Next, the service providers' information will be added in the *Service Description File*.

Listing 3: Reference API captured in the Service Description File

```
<c:services>
     <c:service name="E-mailService">
          <c:operation name="bounce">
               <c:parameters>
                    <c:parameter>
                         <c:key>limit</c:key>
                    </c:parameter>
                    <c:parameter>
                         <c:key>offset</c:key>
                    </c:parameter>
               </c:parameters>
          </c:operation>
          <c:operation name="search">
               <c:parameters>
                    <c:parameter>
                         <c:key>limit</c:key>
                    </c:parameter>
                    <c:parameter>
                         <c:key>recipient</c:key>
                    </c:parameter>
               </c:parameters>
          </c:operation>
          <c:operation name="createMailingList">
               <c:parameters>
                    <c:parameter>
                         <c:key>listName</c:key>
                    </c:parameter>
               </c:parameters>
          </c:operation>
           <c:operation name="sendMail">
               <c:parameters>
                    <c:parameter>
                         <c:key>to</c:key>
                    </c:parameter>
                    <c:parameter>
                         <c:key>cc</c:key>
                    </c:parameter>
                    <c:parameter>
                         <c:key>bcc</c:key>
                    </c:parameter>
                    <c:parameter>
                         <c:key>from</c:key>
                    </c:parameter>
                    <c:parameter>
                         <c:key>text</c:key>
                    </c:parameter>
                    <c:parameter>
                         <c:key>subject</c:key>
                    </c:parameter>
               </c:parameters>
          </c:operation>
     <c:service>

</c:services>
```

8.5 Vendor Implementation Phase

After having defined the *Reference API*, the next task (**task 3b** of Figure 21) is to build the provider specific *API*.

8.5.1 Build the Provider-Specific API

The construction of the provider-specific *API* entails mapping the *Reference API* onto the provider`s *API*. The mapping is required by the *API Client Generator* in order to generate the client adapter as it is described in Section 9.1.1.

The mapping of the *APIs* is represented in *XML* template files. There are certain reasons why *XML* files were selected to capture the *API*. First *XML* provides a simple way to read and encode information. There are multiple library parsers available, which enable the creation and the manipulation of *XML* files. Furthermore, *XML* provides extensibility and thus allows new tags to be added when required. Being a *W3C* [221] standard, makes the language an industrial standard and therefore promotes portability and interoperability. It is not the first time that *XML* is used to describe the service related information. Major web service description languages such as *WSDL* [69] , *WADL* [222], and *SA-WSDL* [223] are based on *XML*. An alternative solution, adopted by related work such as *jClouds* and *mOSAIC*, is to hardcode the information related with the web *API* in the source code. However, such an approach impacts adversely on the maintainability and the extensibility of the approach. Every update, which occurs in the *API*, needs to be propagated manually to the source code.

8.5.1.1 *The Service Description File including the vendor' API*

Listing 4 shows an excerpt from the *Service Description File* including the *MailJet* mailing provider. In the excerpt we observe the mapping of the provider specific *API* to the reference one for the operations described earlier in the Section. The mapping is represented as a key-value pair. For example, for the operation "bounce" there are two pairs of parameters. The first one is *key: offset, value: skip* where the key represents the *Reference API* and the value represents the provider`s *API*. The second one is *key: limit, value: limit,* where it coincides that the key and the value are the same. Likewise, the mapping of the rest of the parameters and the operations are completed.

Listing 4: Mapping of provider specific API to the *Reference API*

```
<c:services>
     <c:service name="E-mailService">
          <c:providers>
               <c:provider  name="Mailjet"
                           baseUrl="https://api.mailjet.com/v3"
                           userName="testuser"
                           password="pass1">
                    <c:operation  name="bounce"
                                 endpoint="/reportEmailbounce"
                                 method="GET">
                         <c:parameters>
                              <c:parameter>
                                   <c:key>limit</c:key>
                                   <c:value>limit</c:value>
                              </c:parameter>
                              <c:parameter>
                                   <c:key>offset</c:key>
                                   <c:value>skip</c:value>
                              </c:parameter>
                         </c:parameters>
                    </c:operation>
                    <c:operation  name="search"
                                 endpoint="/reportEmailsent"
                                 method="GET">
                         <c:parameters>
                              <c:parameter>
                                   <c:key>limit</c:key>
                                   <c:value>limit</c:value>
                              </c:parameter>
                              <c:parameter>
                                   <c:key>recipient</c:key>
                                   <c:value>to_email</c:value>
                              </c:parameter>
                         </c:parameters>
                    </c:operation>
                    <c:operation  name="createMailingList"
                                 endpoint="/messageList"
                                 method="GET">
                         <c:parameters>
                              <c:parameter>
                                   <c:key>listName</c:key>
                                   <c:value>Name</c:value>
                              </c:parameter>
                         </c:parameters>
                    </c:operation>
               </c:provider>
          </c:providers>
     </c:service>
</c:services>
```

Apart from the mapping of the parameters, the *Service Description File* includes information about the configuration settings required by the specific service providers. Next Section discusses the various types of the configuration settings and the way they are handled by *SCADeF*.

8.6 Configuration Settings

The variability in the configuration settings is the third variability point when dealing with multiple *platform basic services* and service providers, as mentioned in Section 5.1. The fact that a *service-based cloud application* may depend on several *platform basic services* results in a significant number of configuration settings, which need to be handled by the software engineers. These settings spread across different service providers and operations. Therefore, the process of maintaining the required settings, and distributing them where needed, may become strenuous and error prone.

This Section attempts to identify and classify the various configuration settings into certain categories. For example there are settings, which are required by all operations of a service provider whereas there may be others applicable only for specific operations. An analogy can be drawn with *class variables* versus *method variables,* where the former are valid for all the methods of the class, whereas the scope of the latter is only the body of the method for which they are defined [211].

Subsequently, a mechanism is described for handling the settings according to the category to which they belong and thus offloading this task from the users of the framework.

8.6.1 Classification of the configuration settings

The configuration settings are classified based on two factors. The first one determines whether the setting is expected by the service provider as a parameter in the web request or is needed in order to construct the request. The second factor examines whether the settings are applicable for all the operations of a service provider or only to a particular operation.

Based on the above two factors the configuration settings can be classified in two categories each one divided in two sub-categories (Figure 44).

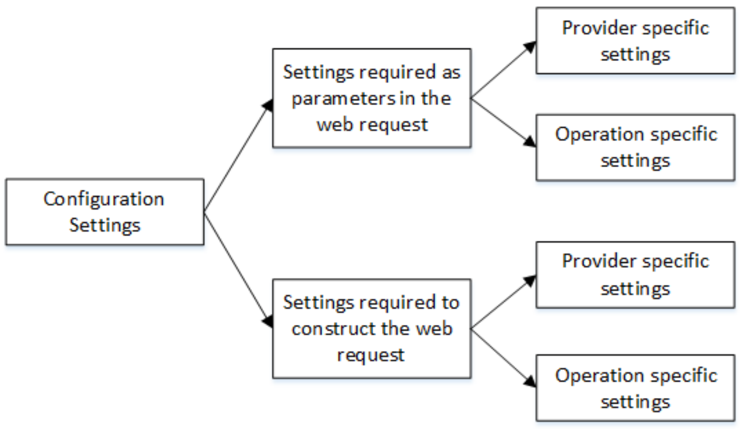

Configuration Settings
Settings required as parameters in the web request
Settings required to construct the web request
Provider specific settings
Operation specific settings
Provider specific settings
Operation specific settings

Figure 44: Configuration settings

1. **Settings required as parameters in the web request:** The first category of settings includes the ones who are required by the service provider and needs to be included as parameters in the web request. Example of such settings is the *gateway_token* required by *Spreedly* and the *client_id* required by the *Google Authentication* service. Settings in this category can further be classified in two sub-categories.
   a. **Provider specific settings**: Settings in this category are common for all the operations of a service provider and need to be present during the operation invocation. The *client_id* in the *Google Authentication* service is such an example, as shown later in the next Section, since it is required as parameter across all the operations during the authentication process.
   b. **Operation specific settings**: Settings in this category are applicable only to specific operations. Thus they need to be included only in the invocation of the specific operations. Such an example is the *gateway_token* required by the *Spreedly* payment service during the operation of charging a card and is used to denote the particular payment gateway to be deployed to execute the transaction.
2. **Settings required to construct the web request:** Contrary to the previous category, settings which are classified in this category are needed in order to construct the *HTTP* request to invoke a service provider's operation. Such examples are the *endpoint* and the credentials required during the web request. The complete list of the settings and further information

about the *HTTP* requests is provided during the description of the *Execution Phase* in Section 9.1.2 of the next Chapter.

a. **Provider specific settings:** Examples of settings, which belong in this category, are the *base_URL* and the user's credentials. The first declares the URL of the service provider against which *API* requests are made. The credentials are used in each request in order for the service provider to be able to authenticate the users.
b. **Operation specific settings:** Example of setting, which belongs in this category, is the *endpoint* that denotes the specific operations to be invoked.

Independently from the category the configuration settings are required to be present when a web request is made to the service provider. For that reason they are included in the *Service Description File*, which contains the information for the construction of the web clients as mentioned in 8.5. The reasoning behind the classification performed in this Section is that the code generator handles them separately during the generation of the web clients as it is described in Section 9.1.2 of the next Chapter.

8.6.2 Service Description File including the configuration settings

A real-world example where the various categories of configuration settings are included is shown in Listing 5. It represents part of the *Service Description File* for the *Google Authentication* service.

The settings that are used in order to construct the *HTTP* requests are encoded as *XML attributes.* As described earlier in Section 8.6.1 there are the *provider specific* and the *operation specific settings*. In this example the *provider specific setting* is encoded in the *XML element <serviceProvider>* and are the *base_URL* and the *name.* The *operation specific settings* are encoded in the each operation separately, inside the *<operation> XML element.* These are the *endpoint* the *(HTTP) method*[1]*,* and the *name.*

[1] The *endpoint* and the *HTTP method are part of the HTTP request and are further described in Section 9.1 of the next Chapter.*

Listing 5: Configuration settings captured in the Service Description File

```
<c:serviceProvider  name="GoogleAuthentication"
                    baseUrl="https://accounts.google.com/o/oauth2">
    <c:ProviderParameterSettings>
        <c:parameters>
            <c:parameter>
                <c:key>client_id</c:key>
                <c:value>433112534981</c:value>
            </c:parameter>
            <c:parameter>
                <c:key>redirect_uri</c:key>
                <c:value>http://localhost:8090/oauth2callback</c:value>
            </c:parameter>
        <c:parameters>
    </c:ProviderParameterSettings>
    <c:operations>
        <c:operation name="requestCode" endpoint="/auth" method="POST">
            <c:OperationParameterSettings>
                <c:parameters>
                    <c:parameter>
                        <c:key>reponse_type</c:key>
                        <c:value>token</c:value>
                    </c:parameter>
                    <c:parameter>
                        <c:key>scope</c:key>
                        <c:value>force</c:value>
                    </c:parameter>
                <c:parameters>
            </c:OperationParameterSettings>
        </c:operation>
        <c:operation name="requestToken" endpoint="/token" method="POST">
            <c:OperationParameterSettings>
                <c:parameters>
                    <c:parameter>
                        <c:key>client_secret</c:key>
                        <c:value>D9yv8uq1gVF30z17dWr6ffQEF</c:value>
                    </c:parameter>
                    <c:parameter>
                        <c:key>grant_type</c:key>
                        <c:value>authorization_code</c:value>
                    </c:parameter>
                <c:parameters>
            </c:OperationParameterSettings>
        </c:operation>
    </c:operations>
</c:serviceProvider>
```

The settings, which are required as parameters in the web requests, are encoded as key-value *XML elements.* Particularly, the *provider specific settings* are nested in the *<ProviderParameterSettings>*. These are the *client_id* and the *redirect_uri* and are used in both operations of the service. The *operation specific settings* are nested inside the *<OperationParameterSettings> XML element.* In the case of the *requestCode* operation these settings are the *response type* and the *force.* In the *requestToken* operation the *operation specific settings* are the *client_secret* and the *grant_type*[1].

Using the method described in this Section, the various configuration settings are encoded in the *Service Description File.* Thus, they can be handled by the framework and specifically by the code generator as it is described in Section 9.1.2 of the next Chapter.

8.7 Summary

This Chapter focused on the mechanism via which the *SCADeF* framework aims to address the variability in the APIs that the various service providers publish. It includes the definition of the *Reference API* that is exposed to the application developers and constitutes a common description of the operations and the parameters offered by the providers. Subsequently, during the *Vendor Implementation Phase*, the providers' specific *APIs* are mapped to the reference one.

In addition, a classification scheme of the configuration settings has been proposed. According to that the settings which are required for the construction of the *HTTP* request are distinguished from the ones used as parameters in the request. Moreover, depending on whether the settings are valid for a single operation or are used across all the operations of the provider, they are classified as *operation specific* or *provider specific* respectively.

The information related both to the *API* and the configuration settings are captured in the *Service Description File*, defined in Section 8.4.2.1. This paves the way for the automatic generation of the source code of the client adapters, which is described in the next chapter.

The work presented in this chapter supports the Requirement 3 (R3), listed in Section 5.3, regarding the alleviation of the *API* variability across the various

[1] The detailed description of each of the parameters used in the *Google authentication* service can be found in the following URL: *https://developers.google.com/identity/protocols/OAuth2WebServer*

service providers. It additionally fulfils the Requirement 8 (R8), which dictates the management of the configuration and the authentication tokens required by the providers. Furthermore, the methodology described in this Chapter, regarding the abstraction of the provider specific *APIs*, aims to enable the *administrator* to enrich the framework with additional *platform basic services* and providers. Therefore, this Chapter supports the Requirement 7 (R7) related to the distinct role of the *administrator* as well as the Requirement 5 (R5) associated to the generic and expandable nature of the framework.

Chapter 9

Execution Stage of the Platform Service API

Chapter 8 described the methodology through which the *Reference API* can be defined and the provider specific *API* can be mapped to the reference one. At the same time the configuration settings required for each concrete provider are also captured. The information is stored in the *Service Description File.* This information can be exploited in order to generate automatically the source code required to invoke the operations of the service providers.

The aim of this Chapter is to demonstrate the *Execution Phase* during which the code generation takes place. The outcome of the generation is a set of interfaces with the operations offered by the *platform basic service* and the respective implementation of the specific providers supported by *SCADeF*. The software developers access only the service interfaces while the concrete implementation remains transparent. This contributes to the initial aim of this research work, which is to "hide" the providers' implementation from the software developers.

The structure of the Chapter is as follows:

Section 9.1 describes the *API Client Generator* component which is responsible for the code generation. Specifically, certain code generation techniques are mentioned such as the visitor-based and the template-based. Subsequently, the Section discusses the process adopted by this research work and states the concrete input and output components of the code generator. The way the configuration settings are handled by the code generator is also described. A hybrid approach is followed which means that code is generated both at design and at run-time.

Section 9.2 compares the hybrid approach with alternative design methods such as a complete run-time approach. Finally, Section 9.3 concludes the Section by mentioning certain limitations of the proposed solution and how these can be addressed.

9.1 Execution Phase

The web *API* usually follows the principles of the *REST* architectural style. *REST* is an architectural style to develop web applications. Contrary to the more complex protocol *SOAP, REST* relies on simple *HTTP* request- response mechanism.

The basic parts when forming an *HTTP* request are:

1. **Request URI:** The *URI,* uniquely identifies the resource at which the request is targeted.
2. **Request Method:** There are four dominant *HTTP* methods used in the *REST* web *API* and identify the action to be performed: a) *GET* is used to retrieve resources, b) *POST* enables the creation of new resources, c) *PUT* is often used for updating operations and d) *DELETE* enables the deletion of resources.
3. **Request Parameters:** Parameters can be included in the request in two ways. In the case of the *GET* method, the parameters most often are appended in the *URI.* On the contrary, *POST* requests incorporate the parameters in their body.
4. **Credentials:** The credentials are used to authenticate the sender of the request and are usually in the form of: *username:password*.
5. **Request Header:** The header specifies the meta-data of the request such as the media type of the body, the date and the authentication credentials for *HTTP* authentication.

The construction of the web client, namely the *HTTP* requests, may be an error-prone and time-consuming process for the developers. Thus the proposed client generator undertakes the task of generating the source code, which implements the requests to the *platform basic service* providers. At the same time the interfaces, which abstract the various concrete implementations, are also generated.

9.1.1 API Client Generator

The *API Client Generator* essentially consists of a code generator, which enables the automatic generation of the source code required to invoke the *APIs* of the concrete service providers. As mentioned in Section 4.2.5, there are two major code generation approaches, namely the *Visitor-based* and the *Template-based.* The former uses a visitor mechanism to scan through the structure of the input model and accordingly it produces the output code in a text stream. This approach is more suitable when there is a significant variation among the input and output models and thus no template can be defined to accommodate the generated code.

By contrast, the latter approach is recommended when large part of the output model is common for all the input models and only specific pieces of information vary. In this case pre-defined templates of the output code are used and during the transformations only the missing information is filled based on the input models as was depicted in Listing 1.

This research work adopts a template-based approach. The task of the code generator is to produce the web clients. Therefore, code templates are constructed to keep the common source code of the web client and only the information pertaining to the individual service provider is filled each time.

9.1.1.1 *Code Generation Process*

The process of the code generation is depicted in Figure 45. The code generator accepts as input the following:

- **The Service Description File**: As mentioned in Section 8.5, this file contains the services which are used by the application, the concrete providers which are supported, as well as the mapping between the *Reference API* and the provider specific *API.*

- **The Template files**: These files contain the source code, which is common among the generated classes, also known as boilerplate code.

The code generator reads the *Template Files* and fills in the missing information regarding the services and the concrete providers as those obtained from the *Service Description File*. Subsequently, the following Java classes are generated (Example of generated code can be seen in Appendix D):

- 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.*
- 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.*

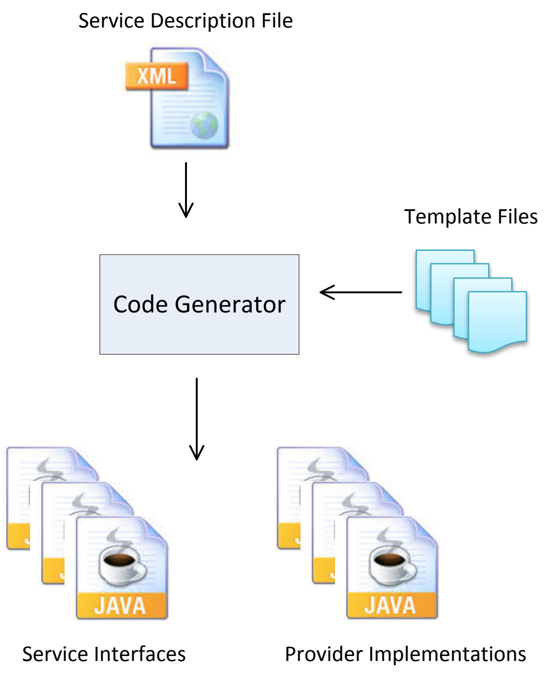


Figure 45: Code generation process

9.1.1.2 *Components involved in the code generation process*

Figure 46 describes the components involved in the code generation process. Those with stripes are components provided by the *SCADeF* framework, whereas the ones with blue color are generated by the code generator.

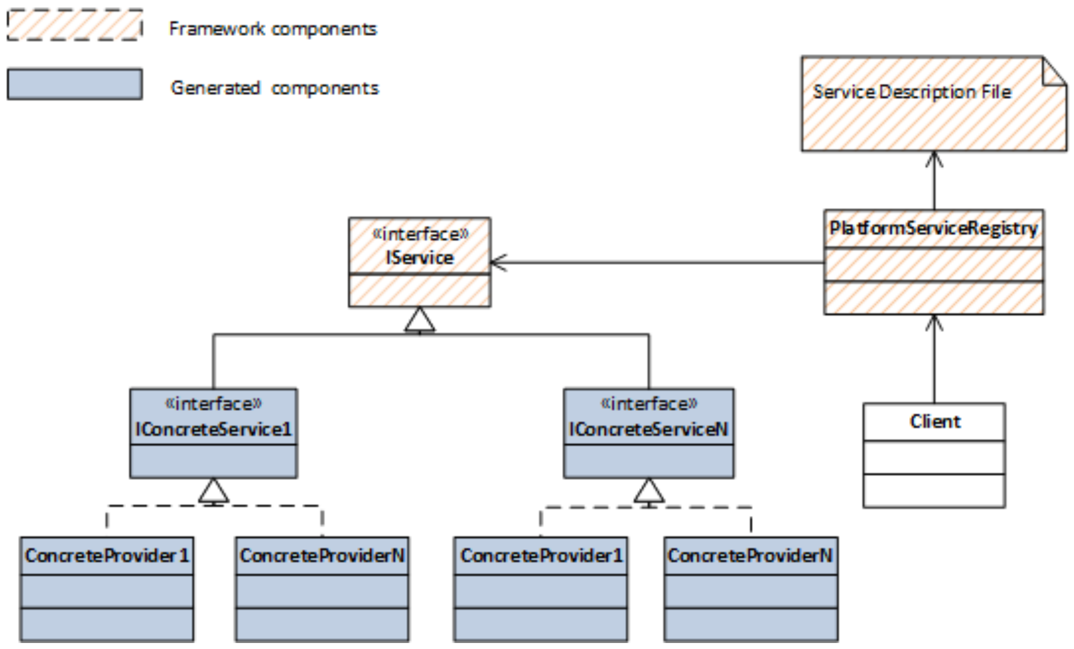


Figure 46: Components involved in the code generation process

Specifically, the components involved are:

1. **Service Description File:** This contains information about the generated services and the concrete providers.
2. **Platform Service Registry:** This reads the *Service Description File* and generates accordingly, the service interfaces, and the concrete implementation for each provider described in the file.
3. **IService:** This is an interface to all the services supported by the framework. It is "known" to the *Platform Service Registry* at design time.
4. **IConcreteService:** This is an interface to each concrete category of service providers. There is a separate interface for the mailing service, the payment service and the authentication service. The *ConcreteService* interfaces are generated by the *Platform Service Registry* based on the *Service Description File*.

5. **ConcreteProvider:** The *ConcreteProviders* contain the specific implementation of each specific service provider, which is described in the *Service Description File*. They essentially contain the actual web client implementation, which is generated by the *Platform Service Registry*.

9.1.1.3 *Accessing the output files of the code generation process*

Once the code generation process is completed, the software developers (here referred to as the *Client*) can make use of the generated services and providers. Figure 47 shows the sequence followed in order for the *Client* to gain access to the services and the concrete provider.

1. Initially, the *Platform Service Registry* reads the *Service Description File, which* has been edited by the user of the framework at design time. Consequently, the service interfaces and the concrete implementations are generated. It also keeps track of the concrete providers selected to implement each service.
2. When the *Client* requires a specific service, it requests it from the *Platform Service Registry*. The latter determines the concrete provider, which implements the service and returns an instance to the *Client*.

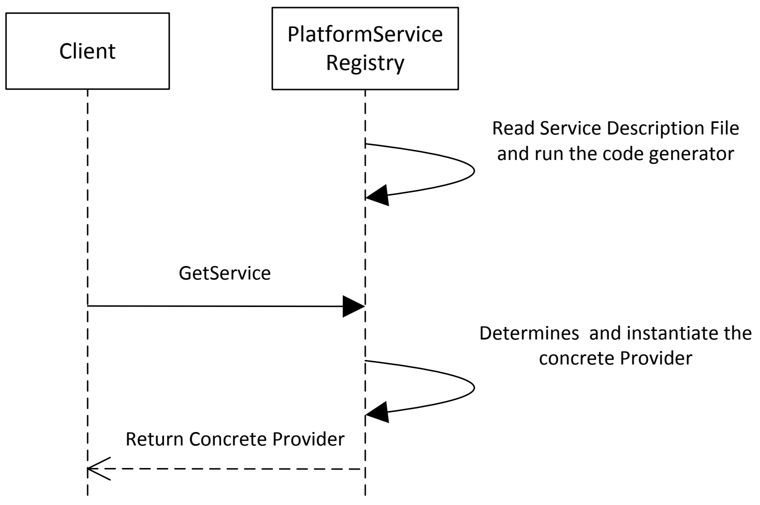

Client
PlatformService Registry
Read Service Description File and run the code generator
GetService
Determines and instantiate the concrete Provider
Return Concrete Provider

Figure 47: Code generation sequence diagram

9.1.2 Code generation of the configuration settings

The previous Section discussed the code generation of the *Reference API*, which is exposed to the software developers as well as the web client, which is required for the invocation of the operations of the service providers. As mentioned in Section 8.6 during the invocation of the operation several configuration settings are required.

The configuration settings where categorised based on whether they are used for the construction of the web request or as parameters during the submission of the web request to the provider. This Section discusses how the code generator handles them depending on the category in which they belong. Listing 6 depicts part of the configuration settings for the *Google Authentication* service.

Listing 6: Part of the *Google Authentication* configuration settings

```
<c:serviceProvider  name="GoogleAuthentication"
                    baseUrl="https://accounts.google.com/o/oauth2">
    <c:ProviderParameterSettings>
        <c:parameters>
            <c:parameter>
                <c:key>client_id</c:key>
                <c:value>433112534981</c:value>
            </c:parameter>
            <c:parameter>
                <c:key>redirect_uri</c:key>
                <c:value>http://localhost:8090/oauth2callback</c:value>
            </c:parameter>
        <c:parameters>
    </c:ProviderParameterSettings>
    <c:operations>
        <c:operation name="requestCode" endpoint="/auth" method="POST">
            <c:OperationParameterSettings>
                <c:parameters>
                    <c:parameter>
                        <c:key>reponse_type</c:key>
                        <c:value>token</c:value>
                    </c:parameter>
                    <c:parameter>
                        <c:key>scope</c:key>
                        <c:value>force</c:value>
                    </c:parameter>
                <c:parameters>
            </c:OperationParameterSettings>
        </c:operation>
    </c:operations>
</c:serviceProvider>
```

The *XML attributes baseURL, endpoint* and *method* are required for the construction of the *HTTP* request. The code generator is aware of the context of these *attributes* and accesses their values in order to fill in the *Template File*, which contains the boilerplate code for all the web requests. By contrast, the *XML elements client_id, redirect_uri, response_type* and *scope*, which are nested in the *XML* element *<parameter>,* are used as parameters in the web request. In this case the code generator does not need to know the context of this *elements.* It only appends them in the list of the parameters during the invocation of the operation, together with their corresponding values.

The separation of the settings into *provider and operation specific* follows the concept of *class inheritance* [211]. Inheritance contributes to code reuse by placing the common behaviour among the classes, such as class variables and methods, to a super class. Likewise, the settings which are common to all operations, are placed in a higher level of hierarchy, namely the *provider specific settings Section*. Thus the user of the editor saves time and effort by defining them only in one place in the *Service Description File*, and also the code generator accesses them only once.

9.2 Alternative design approach

This Chapter described the way the software developers can use the services supported by *SCADeF* and the construction process of the web clients for the invocation of the concrete service providers. The intention of the author is twofold: firstly, to standardise and automate the process of using a service and invoking the concrete provider; and secondly, to keep a familiar programming style for the users.

Therefore a hybrid solution has been adopted, where the construction process takes part partially at design-time and partially at run-time. The service interfaces and the concrete providers who are described in the *Service Description File* are generated at design-time. Thus the software developers can use the interfaces in order to gain access to the specific services while at the same time the concrete providers are not yet determined. Then at run-time when a service invocation occurs, the respective provider is determined and instantiated, based on the consumer's selection which is captured in the *Service Description File.* Additional providers can be supported at run-time. They are included in the

Service Description File and the *Platform Service Registry* undertakes the generation of the respective web clients. The whole process is transparent to the user of the framework. Consequently, new service providers can be selected at run-time, which further promotes the substitutability of the software.

The proposed solution best meets the requirements listed in 5.3. However, this is not the only possible solution. An alternative design suggests that both the service and the provider selection take place at run-time and no code is generated at design-time. This implies that there are no separate interfaces for each service as well as no implementation for each concrete provider. Instead there is a generic web client mechanism, which is configured each time accordingly to serve a request for a particular service and a specific provider.

Contrary to the proposed solution, the benefit of the alternative approach is that at run-time additional providers as well as categories of services can be added. Since there are no service interfaces generated at design-time a new service along with its operations can be described in the *Service Description File* and subsequently the web client mechanism constructs the concrete service invocation upon request.

However, the alternative approach presents a number of drawbacks. The fact that both the service and the provider are configured at run-time implies that the user should provide the following pieces of information: 1) the name of the service, 2) the name of the operation, and 3) concrete parameters. These pieces of information should match accurately the ones specified in the *Service Description File*. This fact adds a burden for the user and makes the process error prone.

Furthermore, the programming style of this approach deviates from the traditional one where the developers have at hand the interfaces with the provided operations and the expected parameters. Several *IDEs*, such as *Eclipse, Visual Studio* and *Netbeans* offer convenient auto-completion features based on the classes available at design-time. Such features are not applicable for the alternative approach.

Moreover, the construction of the web requests at real-time may impose a performance overhead. This overhead may be a deterrent factor for time-critical applications.

A second look at the benefit that this approach offers may reveal that it might not be that useful for the users. The ability to invoke a new category of service at run-time, such as the payment service or the e-mailing service, implies that the application should be capable of "understanding" the new service and efficiently communicate with it. However, such a case would require additional changes in the business logic of the application, which could only be performed at design time. Therefore the benefit of adding a new service at run-time is raised.

An intermediate approach between the proposed one and the alternative which was presented in this Section is to generate the service interfaces at design-time whereas let the web request to be constructed on the fly at run-time. The service interfaces, which contain the operations supported by each service, are generated automatically by the code generator. Thus the software developers can make use of them while developing the *cloud application.* By contrast, the concrete implementation of the service providers is not generated automatically as suggested by the presented approach in Section 9.1.1. Instead an *engine* undertakes the task of constructing the web request on the fly by fetching the necessary information from the *Service Description File* once an operation from the service operation is invoked.

This approach entails two limitations. Since the construction of the request takes place at run-time a performance overhead is imposed which may not be negligible in time critical applications. Moreover, in case the developer needs to interfere to the provider implementation in order to change it or enhance it, the framework needs to be bypassed and a separate implementation needs to be created. By contrast, the proposed approach involves the generation of the provider implementation. In this case the software developer can have direct access to the implementation and modify it accordingly.

Keeping the previous remarks in mind, the decisions for the final design of the *SCADeF* framework, which was described in this chapter, were based on the requirements for maintaining a balance between the flexibility of the framework and the provision of a user-friendly tool, which adopts a familiar programming style for the developers.

9.3 Limitations of the current approach

This Section discusses certain design concerns encountered throughout the construction of the *API* abstraction mechanism described in Chapter 8 and Chapter 9. They are essentially centered on the issue of the software *API* abstraction and how to achieve a balance between an efficient abstraction without compromising the functionality and the peculiarities of the service providers. Similar issues have been encountered by related approaches dealing with cloud service *API* abstraction, such as *jClouds*, *mOSAIC*, *openTOSCA* as we discuss later in this Section.

9.3.1 Design Issue 1: Missing parameters

Not all the *platform service* providers offer exactly the same set of parameters for a given operation. Additional, optional parameters may be available by specific providers offering extra functionality. Table 13 lists the supplementary parameters available by the e-mailing service providers described in this chapter for the operation "send email".

Table 13: List of additional parameters offered by the e-mailing service providers for the operation "send email"

Mailgun	***SendGrid***	***Postmark***	***Mailjet***
o:deliverytime	Tag	date	Mj-prio
o:testmode	ReplyTo	ccname	Mj-trackclick
o:campaign	TrackOpens	bccname	
	Headers	fromname	

The parameters are self-explanatory and thus no further description is provided. For further details on the functionality of the parameters the reader may look up the *API* of the service providers which is described on their website. The link to the providers' website is provided in the references Section.

There are two possible solutions to address the design issue. The first one involves direct access of the user of the framework to the native web client for the specific provider. Therefore, when the users require a specific parameter, which is not available by the *Reference API*, they can bypass it and instead access the low level *API* for the specific provider.

SCADeF directly accommodates this solution. The additional parameters are included in the *Service Description File* and the code generator produces the native operations with the additional parameters. Listing 7 shows the additional parameters, which are included in the "send email" operation of the *Postmark* e-mail service provider within the element *extra.*

Listing 7: Additional parameters included in the *Service Description File*

```
<c:services>
     <c:service name="E-mailService">
          <c:providers>
               <c:provider  name="PostMark"
                           baseUrl="htps://api.postmarkapp.com"
                           serName="testuser"
                           password="pass1">
                    <c:operation  name="sendEmail"
                                 endpoint="/email"
                                 method="POST">
                         <c:parameters>
                              <c:parameter>
                                   <c:key>to</c:key>
                                   <c:value>to</c:value>
                              </c:parameter>
                              <c:parameter>
                                   <c:key>cc</c:key>
                                   <c:value>cc</c:value>
                              </c:parameter>
                              <c:parameter>
                                   <c:key>bcc</c:key>
                                   <c:value>Bcc</c:value>
                              </c:parameter>
                              <c:parameter>
                                   <c:key>from</c:key>
                                   <c:value>From</c:value>
                              </c:parameter>
                              <c:parameter>
                                   <c:key>text</c:key>
                                   <c:value>TextBody</c:value>
                              </c:parameter>
                              <c:parameter>
                                   <c:key>subject</c:key>
                                   <c:value>Subject</c:value>
                              </c:parameter>
                         </c:parameters>
                         <c:extra>
                              <c:parameter>date</c:parameter>
                              <c:parameter>ccname</c:parameter>
                              <c:parameter>bccname</c:parameter>
                              <c:parameter>fromname</c:parameter>
                         </c:extra>
                    </c:operation>
               </c:provider>
          </c:providers>
     </c:service>

</c:services>
```

A similar solution is also offered by *jClouds*, the *mOSAIC* and the *openTOSCA*. They provide access to the native drivers, when the users need a specific functionality of the service providers. The benefit of this first solution is that it tackles the issue in a simple and straightforward manner. However, it breaks the encapsulation and cancels the abstraction mechanism.

A second solution proposes a callback mechanism through which the extra parameters are passed from the user (client) to the web client for constructing the web request. Figure 48 describes the mechanism.

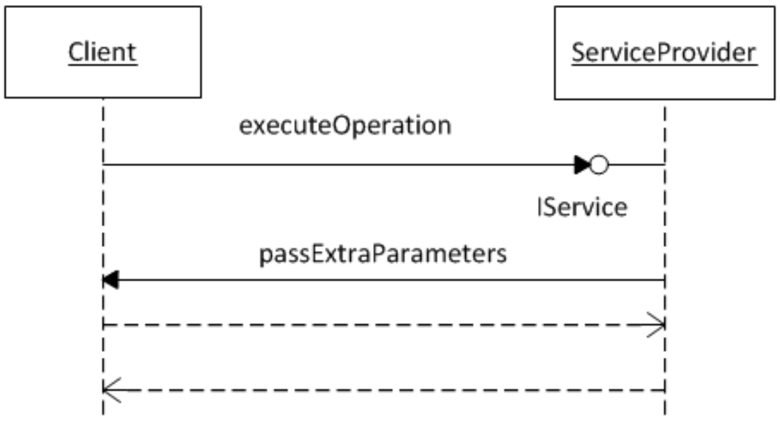


Figure 48: Callback mechanism for passing additional parameters

1. Initially, the client invokes an operation of the Service Provider through the *IService* interface, which describes the *Reference API*. It includes the parameters as those listed in the *Reference API*.
2. The web client of the particular service provider executes a callback function asking the client for any additional parameters for the given function.
3. The client provides the additional parameters, if any. Subsequently, the web client constructs and executes the web request.

This second solution is better aligned with the abstraction framework, since it makes use of the *Reference API*. In this sense it is more elegant than the first solution. However, with respect to the complexity involved, it requires that users become familiar with the callback mechanism and how they can make use of it.

Therefore, regarding the simplicity and the ease of use, the first solution proves to be more effective.

9.3.2 Design Issue 2: Missing operations

Not all the *platform service* providers offer the same set of operations. There may be providers who offer additional operations and thus more functionality. This issue is similar to the one discussed above. In order to cope with this, the first solution above is adopted, namely the users gain access to the native web client which implements the additional operations. The operations are described in the *Service Description File* and the code generator produces the supplementary operations.

9.3.3 Design Issue 3: Handling the response of a service operation

The current implementation of the *API* abstraction mechanism is able to abstract and unify the specific *APIs* exposed by the service providers as described throughout the Chapter 8 and Chapter 9. However, at this stage it does not act upon the response received by the providers. This means that it is up to the developer to handle the returned response of an operation.

Translating the outcome message from the provider means that the framework makes the application aware of the context of the message content. This allows the application to use the message for performing a business action or invoking another web service. However, this touches upon the field of service orchestration where a business process can interact with internal or external web services [224]. There are established and mature tools such as *Business Process Execution Language (BPEL)* and the *WSDL* explaining how orchestration can be achieved. By contrast the focus of the *SCADeF* framework is to abstract the differences in the web *API* of the various *platform basic service* providers rather than orchestrating the execution and invocation of the various services.

However, *SCADeF* can be extended in order to accommodate the response of the providers. The same methodology, as described in Chapter 8 and Chapter 9, can be adopted. According to that a reference response, which is exposed to the software developers, is defined. Subsequently, the various providers' responses are captured and mapped to the reference one. Then at run-time when a response is received, it is handled by a mechanism according to the pre-defined mapping.

9.4 Summary

This chapter described the *Execution Phase* of the *Platform Service API Description* part. This phase involved the components and the mechanism of the *SCADeF* framework, which enables the automatic generation of the client adapters. Specifically, the outcome of the code generation process is a set of service interfaces, which contains the operations offered by the *platform basic service,* described in Section 9.1.1.1 and 9.1.1.2 as well as the *Reference API*, defined in Section 8.4.2. Additionally, the web clients required to invoke each concrete service provider are also generated. The process of the code generation and the subsequent invocation of the concrete provider API remain transparent to the user. The latter interacts only with the service interfaces.

The work presented in this chapter fulfills the Requirement 4 (R4) listed in Section 5.1, regarding the automatic generation of the client adapters. Additionally, the capability of the real-time generation of the web clients supports the Requirement 6 (R6), which is related to the substitutability of the service providers. R6 paves the way towards enabling the application developers to choose seamlessly the service provider at real-time based on certain criteria such as the price, the quality of service, and the geographical region. The features described both by the R4 and the R6 enable the application developers to leverage the framework in order to seamlessly use the various *platform basic service providers.* Therefore the Requirement 7 (R7) regarding the distinct role of the application developer is also fulfilled.

The chapter also discussed alternative design approaches and certain limitations that the proposed development framework entails. These are the handling of the potential mismatch of the parameters and the operations between the concrete providers as well as the handling of the providers' response. In the first case, the software developers can gain direct access to the provider specific functionality.

In the latter case, a reference response can be defined and exposed to the developers. Subsequently, the providers’ response is mapped to the reference one in the same method as the providers’ *API* is mapped to the *Reference API*.

PART C

Conclusion and Future Work

Chapter 10

Conclusion – Future Work

This final chapter brings together the findings of the thesis, as well as putting forward new ideas for further work. At this point Section 10.1 presents a summary of the research work. Then Section 10.2 examines, in retrospective, the fulfillment of the contributions as those were set in the Introduction. Finally, Section 10.3 discusses future work, which could be motivated by this thesis.

10.1 Summary of the thesis

The research reported in this thesis focused on enabling the developers to leverage *platform basic services,* offered via the *cloud applications platforms,* in order to create *service-based cloud applications.* As discussed in Section 3.5, there are multiple benefits associated with the use of *platform basic services,* such as the rapid application development, the provision of a variety of ready to be used functionality and the integration possibilities using a lightweight HTTP-based API. However, as analysed in Chapter 5, there are various *platform basic service* providers offering their own custom implementation. In order for the applications to fully exploit the various *platform basic service* providers, they should be able to seamlessly choose the ones, which each time better serves the requirements at hand such as the quality of service, the pricing, and the security and privacy.

Towards this direction, the main question that this research work focused on was: how to enable the *service-based cloud applications* to integrate various *platform basic services* without being bound to the concrete implementation exposed by the concrete providers. In other words this work had to explore the degree to which the specific functionality exposed by the various *platform basic service* providers could be abstracted and become transparent to the software developers seeking to consume these services.

The experimentation with various *platform basic services* such as the payment service, the e-mail service, the image processing service, the authentication service and their respective providers showed that the providers tended to converge on a similar set of functionality and operations. Similar to traditional cloud resources, such as compute and storage services, where there are several abstraction frameworks, such as the *jClouds* and the *LibCloud* and standardisation approaches such as the *OCCI* and the *CDMI* as shown in Chapter 4, the growing popularity of the *platform basic service* leads to an overlapping set of offered operations as shown in Chapter 8. Thus it was feasible in this research work to formulate a methodology for defining the reference implementation that is exposed to the software developers and thereby abstract the specific providers' implementation.

The next step was to break down the specific providers` implementation and identify the concrete variability issues, which arise and needed to be addressed during the integration of the application with the various *platform basic service* providers. As stated in Section 5.1, these were:

- ➢ The differences in the workflow, which is required to complete an operation, across the various providers.
- ➢ The differences in the web *API* exposed by the providers.
- ➢ The management of the configuration settings and the authentication tokens required by each provider.

Chapters 5-9 focused on the proposition of the methodology, which alleviates the above mentioned variability issues and thereby enables the developers to use seamlessly the various *platform basic services* provided by the *cloud application platforms.* In support of the methodology the *SCADeF* framework was designed, comprising specific tools and components, to put into practice the proposed methodology using real examples of *platform basic services* having multiple variations and providers.

Specifically, throughout the thesis the following *platform basic services* and their respective providers have been implemented and integrated into the development framework:

1. The payment service. The concrete providers that were demonstrated were: a) *Spreedly*, b) *Stripe*, c) *Viva payments,* and d) *Braintree*.
2. The E-mail service. The concrete providers implementing the e-mail service were: a) *SendGrid*, b) *Mailgun*, c) *Postmark,* and d) *Mailjet.*
3. The Authentication service. The concrete authentication providers were: a) the *Google* authentication service and b) the *Facebook* authentication service.

In conclusion, the proposed methodology and the *SCADeF* framework, which are summarised in the following Section 10.2, have demonstrated the feasibility of the creation of *service-based cloud applications,* as a collection of *platform basic services,* independent of the concrete providers' implementation and thereby contribute towards the vision of empowering developers to produce code in a faster and better manner.

10.2 Fulfillment of the Contributions

This Section examines the contributions of the research thesis, against the goals that were defined in Section 1.5.4 and explains to what degree they have been fulfilled throughout the thesis.

C1: Clarification of the notions of *cloud applications platforms* and *platform basic services* and a subsequent exemplification of how these notions could be leveraged to accelerate the cloud-based development process and lead to the creation of *service-based cloud applications*.

As stated in Chapter 2 and explicitly analysed in Chapter 3, the field of *cloud platforms* is characterised by heterogeneity among the platform offerings and different styles of application development. As discussed in Section 1.1, The heterogeneous *PaaS* interfaces and technologies lead to a confusion on the consumers' side which in turn may result in a slower adoption rate of *PaaS* market compared to the *IaaS* and *SaaS*. To this end Chapter 3 presented a survey of available commercial *cloud platforms* and aimed to distinguish among the various categories.

Specifically, the survey revealed that the platforms could be classified into three categories according to the development style, which they adopt. The first one contains offerings which support widely used and standardised technologies such as programming languages, databases and web servers. The second group offers additional functionality via *platform basic services*. Thus the development time is reduced since the applications are not created from the ground up but they can rather be synthesised from a number of *platform basic services*. The third category adopts a different development style, which is based on an online graphical framework, which can be customised by the users according to their needs.

As stated in Section 3.4, this research work focused on the platforms of the second category because they adopt a traditional programming style and additionally offer custom functionality which can speed up the application development. These platforms were referred to as *cloud application platforms.*

The clustering of the survey and the clarification of the notion of the *cloud application platform* aims to provide a common understanding of the variants of the cloud platforms and underpin any future work carried out by researchers in this area.

In addition, this thesis contributed to the clarification of the concept of the *platform basic service,* which was hitherto not explicitly defined in the relevant literature. Specifically, Chapter 3 described the *platform basic service* as an autonomous and independently deployed unit of functionality, which is offered usually by *ISVs,* via the *cloud application platforms,* and through an HTTP-based protocol.

As highlighted in Chapters 3 and 4 the concepts of the *cloud application platforms* and *platform basic services* may constitute a major paradigm shift in the development of applications in the context of cloud computing and cloud platforms in particular. Applications will no longer have to be constructed from the ground up. Rather they can be synthesized from a number of *platform basic services.* leading to the creation of the so called *service-based cloud applications.* Among others, major benefits are that applications can be built in significantly less time, and rely on well tested functionality offered by third party providers.

At the same time a new software service ecosystem, envisaged for the cloud, is being brought about. In this ecosystem the *cloud application platform* acts as the common medium which brings all the interested parties together. *ISVs* are the parties who create and offer the *platform basic services* via the platform. They can either deploy the services in the platform or they can merely use the platform as a mean to offer their API and increase their popularity. Finally, *service-based cloud applications* utilise the offered *platform basic services* and are deployed on the *cloud application platforms.* In turn, these applications can be offered via the platform in the form of *platform basic services.*

C2: The formulation of a methodology, which enables the development of *service-based cloud applications* independent of the concrete *platform basic service* providers.

As discussed in Section 3.5, there is a growing popularity of the *cloud application platforms* and the *platform basic services.* Specifically, *Heroku* offers currently almost 150 services, while in the recent years additional platforms have launched their own marketplaces such as *Openshift* and *EngineYard.* This fact provides significant possibilities to the application developers who can exploit the various *platform basic services* and choose each time the concrete providers that better serves the requirement at hand (such as quality of service, security and pricing).

However, in order for the service-based cloud applications to completely leverage the various *platform basic service providers,* they need to be able to choose seamlessly each time the concrete provider. This has been one of the original aims of this thesis as discussed in Section 1.5, namely the cross-platform development of *service-based cloud applications*. This means that *cloud applications* are developed independently of the concrete implementation of the *platform basic service* providers offered by the target *cloud application platforms*.

This aim has been addressed by the methodology exposed throughout the Chapters 5-9. According to this discourse, the *platform basic service* providers are studied in order to conclude to a certain group, exposing a common set of functionality. Subsequently, the abstraction of the providers' implementation takes place, consisting of two parts:

i. *Platform Service Workflow Description*. Certain *platform basic services,* such as the Payment service, require more than one step to complete an operation such as the "purchase request". Thus, this part describes the way the steps involved in an operation, as these are implemented by the various providers, can be captured.
ii. *Platform Service API Description.* This part involves the definition of a *Reference API*, to which the software developers have access, the description of the providers' specific API, and the subsequent mapping of the providers' API to the reference one.

For each of the two parts three phases are defined:

Phase 1. *Platform Service Modeling Phase:* During this phase the abstract functionality offered by the *platform basic service* is defined, including the reference workflow and the *Reference API* that is exposed to the developers.

Phase 2. *Vendor Implementation Phase:* During this phase the specific workflow and API exposed by each service provider, are captured and mapped to the respective reference ones defined in the first phase.

Phase 3. *Execution Phase.* During this phase the workflow, which has been defined in the previous two phases, is executed in order to complete the operation requested by the application. Moreover, the web API client required to invoke the specific operations of the concrete provider is generated. The whole process remains transparent to the application and the software developer.

This set of parts and phases, which have been proposed, enables the development of *service-based cloud applications,* agnostic to the concrete *platform basic* *service* providers. For each phase specific tasks have been defined as outlined in Section 5.4.3. In addition a set of tools have been constructed such as the *SDE* and the *API* client generator in order to assist in the implementation of the tasks. Thus, the combination of the defined parts, phases and tools constitutes the proposed methodology.

Furthermore, a side contribution of the methodology is the fact that it enables partially the portability of the *cloud applications* across the various *cloud applications platforms (CAPs).* The portability involves the *platform basic services* and is achieved by allowing an application to be ported across the various *CAPs* without the need to reengineer the integration with the *platform basic service* providers.

C3: The design of the *SCADeF* framework to support the above methodology.

One of the main challenges and contributions of this thesis was to propose a methodology, which sets the research underpinnings for enabling the creation of *service-based cloud applications,* independent of the concrete implementation of the *platform basic service* providers. However, this research work proceeded even further to demonstrate how the methodology could be instantiated in practice.

As a result an additional contribution, presented in this thesis, was the construction of a development framework, which supports and implements the proposed methodology for the development of platform agnostic *service-based cloud applications.* The *SCADeF* framework consists of the following tools and components, which are used throughout the process of abstracting a *platform basic service provider,* as it was described throughout the Chapters 5-9:

1. *The Reference Meta-Model.* The *Reference Meta-Model*, described in Section 6.2.2.1, is used during the P*latform Service Modeling Phase* of the *Workflow Description* part and contains two major concepts, the *CloudAction* and the *CloudMessage.* The first is used to handle incoming requests to the framework either by the application or by the service provider. The latter contains the provider specific *API* and is used

internally by the *CloudAction* in order to invoke specific operations of the service providers.

2. *The Service API description editor.* This editor is used during the *Platform Service Modelling Phase* of the *API Description* part in order to define the *Reference API*, the provider specific API and the subsequent mapping of the latter to the former one.
3. *The API Client Generator.* This component, described in Section 9.1.1, is used during the *Execution Phase* of the *API Description* part and is responsible for generating the web *API* client required to invoke the operations of the concrete service providers.
4. *The Platform Service Execution Controller.* This component, described in Section 7.1, is used during the *Execution Phase* of the *Workflow Description* part and handles the execution of the workflow required to complete the operations. Thus the whole process remains transparent to the software developers.

C4: The construction of a toolset to enable the operation of the *SCADeF* framework by software developers.

Following the construction of the *SCADeF* framework, a toolset was built in order to help developers leverage the capabilities of the development framework. The toolset which is described in Appendix B comprises a graphical service description editor and has been implemented in the form of *Eclipse* plug-in. It can be used from both users who have been defined in the Requirement 7 in Section 5.3.7 namely, the *administrator* and the *consumer.*

As shown in Appendix B, the *administrator,* using the graphical editor, is able to add new *platform basic services* and providers to *SCADeF*. For each provider the supported operations are added. Then for each operation, the provider specific *API* is matched to the *Reference API* as it was proposed in Section 8.4. At the same time the configuration variables, discussed in Section 8.6, are inserted.

The *consumer* can browse through the available services and select the ones required in the application, as seen in Appendix B. All the information regarding the *APIs*, the configuration settings and the consumer's choice, is translated automatically into the *Service Description File,* defined in Sections 8.4.2.1, 8.5.1 and 8.6.2, which is required by the *API Client Generator* in order to generate the client for each provider and the respective service interfaces as described in Section 9.1.1.

C5: Manifestation of how the micro-service architectural style could be applied in the field of cloud computing with the use of *cloud application platforms* and *platform basic services.*

As stated in Section 4.2.7.2, the *micro-service* architectural style proposes the development of applications based on a collection of micro-services. The latter were defined as services, which run and are deployed independently of the rest of the application. The fact that each *micro-service* is independent of the rest of the application improves the scalability of the whole system. Each *micro-service* can be scaled separately without affecting the overall system. At the same time the application becomes more resilient since any failures can be isolated. In addition, the *micro-services* can be replaced, when required, since they are loosely coupled with the rest of the application (They usually communicate via a *HTTP*-based *API*). As reported in Section 4.2.7.2, the multiple benefits of the *micro-services*, contribute to the increasing popularity of this novel architectural style and major software enterprises, such as *Oracle* and *IBM,* embrace and promote this style of application development.

Towards contributing to the further promotion and adoption of the *micro-services* this thesis demonstrated how *platform basic services,* a core concept of this research work, have the potential to put into practice the *micro-service* architectural style in the field of cloud computing and *cloud platforms* in particular. As demonstrated in Section 4.2.7.2, the *platform basic services* share the same characteristics with the *micro-services,* namely they are deployed independently, they offer a concrete set of functionality and become available via an *HTTP*-based *API.* On top of that, as discussed in Section 3.5, there is a large number of third-party *platform basic service* providers, offered via the *cloud application platform,* that can be exploited during the application development. Thus by leveraging the methodology and the *SCADeF* framework proposed throughout the Chapters 5-9, the *platform basic services* can be exploited for the creation of *service-based cloud applications* in accordance with the proposition of using the *micro-services* to create software applications.

10.3 Future Work

10.3.1 Incorporating billing and recommendation capabilities into the *SCADeF* framework

As dictated by the aims that were set for this research work, the prototype implementation of the development framework has shown how the specific implementation of the various *platform basic service* providers can remain transparent to the software developers. However, the framework can be extended with additional features and functionalities.

Specifically, a billing mechanism can be implemented in order to provide the software developers with precise information about the cost of the *service-based cloud applications.* As mentioned in Chapter 2 and 3, the *cloud applications* use cloud resources such as web servers, databases and *platform basic services* and are charged on a subscription plan. Therefore, the total cost of the application varies according to the consumed resources. A billing mechanism could monitor the consumption of the resources or collaborate with existing monitoring mechanisms and provide either at run-time the current cost of the application or at design-time in advance an estimation of the cost based on the resources and the concrete service providers selected by the software developers.

Furthermore, the growing number of *platform basic service* providers, as stated in Chapter 3, may create an extra workload for software developers, who have to find the optimal choice. To this end a recommendation system can provide proposals about the most appropriate provider, given certain criteria such as the price, the offered quality, the security and privacy of the service. The use of ontologies may contribute to the formalisation and homogenisation of the *platform basic service* descriptions offered by the various *CAPs.* Then, the services descriptions may be retrieved by the recommendation system, which can analyse them and based on reasoning techniques it can classify them according to specific characteristic, such as the price and the *SLAs* and thereby propose the optimal choice to the users.

10.3.2 Construction of a complete development environment to support the creation of *service-based cloud applications*.

Improvements can be made to the prototype implementation of the *SCADeF* framework in order to extend its functionality. Specifically, the graphical editor can be enriched with additional features such as the definition of the *CloudActions* and their operations. Once the *CloudActions* are defined in the editor, the boilerplate code could be generated automatically.

The current version of the framework supports the creation of *service-based cloud applications* using the *Java* programming language. However, the framework could be extended to support additional programming languages such as Python and PHP.

The long-term vision of the author is the construction of a complete *IDE* which supports developers throughout the development phase of a *service-based cloud application* via extended functionality such as the billing and recommendation mechanism mentioned in Section 10.3.1.

10.3.3 Enhance the functionality of the *SCADeF* framework with additional platform basic services and providers

In order to prove that the proposed methodology is able to support the development of *service-based cloud applications* using *platform basic services*, the prototype implementation of the development framework included three categories of *platform basic services* and ten respective providers as shown throughout the Chapters 5-9 and also stated in Section 10.1.

Future tasks, could involve enriching the framework with additional *platform basic services* and providers such as the image and video processing service and the SMS service.

10.3.4 Measure the performance of the *SCADeF* framework

The proposed methodology and the development framework, which was implemented, demonstrated how the *service-based cloud applications* could be created leveraging *platform basic services* independent of the concrete providers' implementation. However, since time-critical applications were out of the scope of this thesis, the potential performance overhead of the framework has not been measured.

Therefore, in order to extend the eligibility of the framework for time-critical applications the performance overhead could be examined. However, the expected overhead is anticipated to be relatively small since, as stated in Section 9.1, the source code of the target service provider is directly invoked without the execution of any intermediate transformations.

10.3.5 Investigating the use of ontologies as enablers for the homogenisation of the service description of the platform basic service providers

As discussed in Section 8.5.1, XML files are used by the framework in order to capture the *API* of the *platform basic service providers.* XML is widely used for encoding information and there are multiple parsers available. However, the use of ontologies could constitute an alternative approach for capturing the providers' functionality. According to Gruber [225], ontologies are formal explicit knowledge over a shared conceptualisation that is standardized or commonly accepted by certain group of people. In the context of this research thesis, they can be leveraged in order to describe the *APIs* of the various service providers. Towards this direction, initial work has been published by the author [226].

In particular, the characteristic of ontologies that could be exploited is that ontologies can describe unambiguously the providers' *API* and thus avoid any semantic conflicts. Additionally, ontologies can inherit concepts from other ontologies and can be reused if necessary. Therefore, they do not need to be constructed from the ground up but they can rather be based on an established one such as the *Linked USDL (Universal Service Description Language)* [227] and the *Minimal Service Model (MSM)* [228].

Furthermore, ontologies are commonly accepted and shared descriptions of a domain. As such they can increase the consensus for a common description of a service. To this end, future work could investigate whether the use of ontologies might have the potential to extend the scope of this thesis and rather than abstracting the providers' specific API, they could contribute to the homogenisation of the various APIs and to their convergence towards a common API. For example, an ontology could be created and published for each category of *platform basic services* describing the operations and the *Reference API.* Then, the concrete providers could adhere to this ontology and create their service offering accordingly. However, such an approach would require the contribution of a well-established and recognisable institution, which could undertake the task of creating and disseminating the ontologies.

10.3.6 Extending the scope of the *SCADeF* framework beyond the *cloud application platforms*

Throughout the period this research has been carried out, additional work has been under consideration, which had to be left out of the scope of this thesis. Furthermore, additional thoughts and opportunities for further exploration have appeared. To this end, this Section states future work, which could be motivated by the research topic presented in this thesis.

As discussed in Section 10.2 one of the contributions of this research work was the clustering of the *cloud platforms* into three categories*,* on the basis of the software development style they adopt, and the subsequent clarification of the *cloud application platforms.* Platforms in the first category offer a low-level of vendor lock-in at the expense of a higher application development time. By contrast, platforms in the third category allow for rapid application development using graphical environment at the expense of a high level lock-in.

The clustering was dictated by the early finding that the issue of cross-platform development application couldn't be addressed at the whole spectrum of the *cloud platforms.* Rather the research efforts had to narrow down to a specific subset of platforms exposing similar characteristics. As reported in Chapter 4, this thesis chose to focus on the *platform basic services* and the way the concrete implementation of those services can be abstracted away from the software developers. Towards this direction, the methodology and the *SCADeF* framework proposed throughout the Chapters 5-9, demonstrated how the *service-based cloud applications* can remain agnostic to the concrete implementations of the *platform basic services*, which are offered via the *cloud application platforms.*

Future work can focus on the way the applicability of the framework can be extended to also include *cloud platforms* from the two other categories, as those defined in Section 3.3. Specifically, platforms in the first category can leverage the *SCADeF* framework in order to increase the functionality they offer to their users and at the same time maintain the low level of vendor lock-in. In particular, users will be able to use the *platform basic services,* which are available via the *cloud application platforms,* through the framework without being bound to the concrete implementations. In this case, the framework can reside in the platform and can be provided to the applications as a service via the use of libraries. Thus, the framework will facilitate the provision of *platform basic services* in the applications deployed in the platforms of the first category.

10.4 List of publications by the author

This Section presents a list of publications, which were produced as an outcome of this research work, and their relations with the thesis' chapters.

No.	Publication	Chapters
1.	F. Gonidis, I. Paraskakis, and D. Kourtesis, "Addressing the Challenge of Application Portability in Cloud Platforms," in *the 7th South-East European Doctoral Student Conference*, Thessaloniki, 2012, pp. 565–576.	Chapters 4
2.	F. Gonidis, I. Paraskakis, and D. Kourtesis, "Cloud application portability. An initial view," in *the 6th Balkan Conference in Informatics*, Thessaloniki, 2013, pp. 275-282.	Chapters 3,4
3.	F. Gonidis, I. Paraskakis and A. J. H. Simons, "Existing approaches for cross platform development and deployment of cloud applications," in *the 8th South-East European Doctoral Student Conference*, Thessaloniki, 2013, pp. 270-274.	Chapter 4

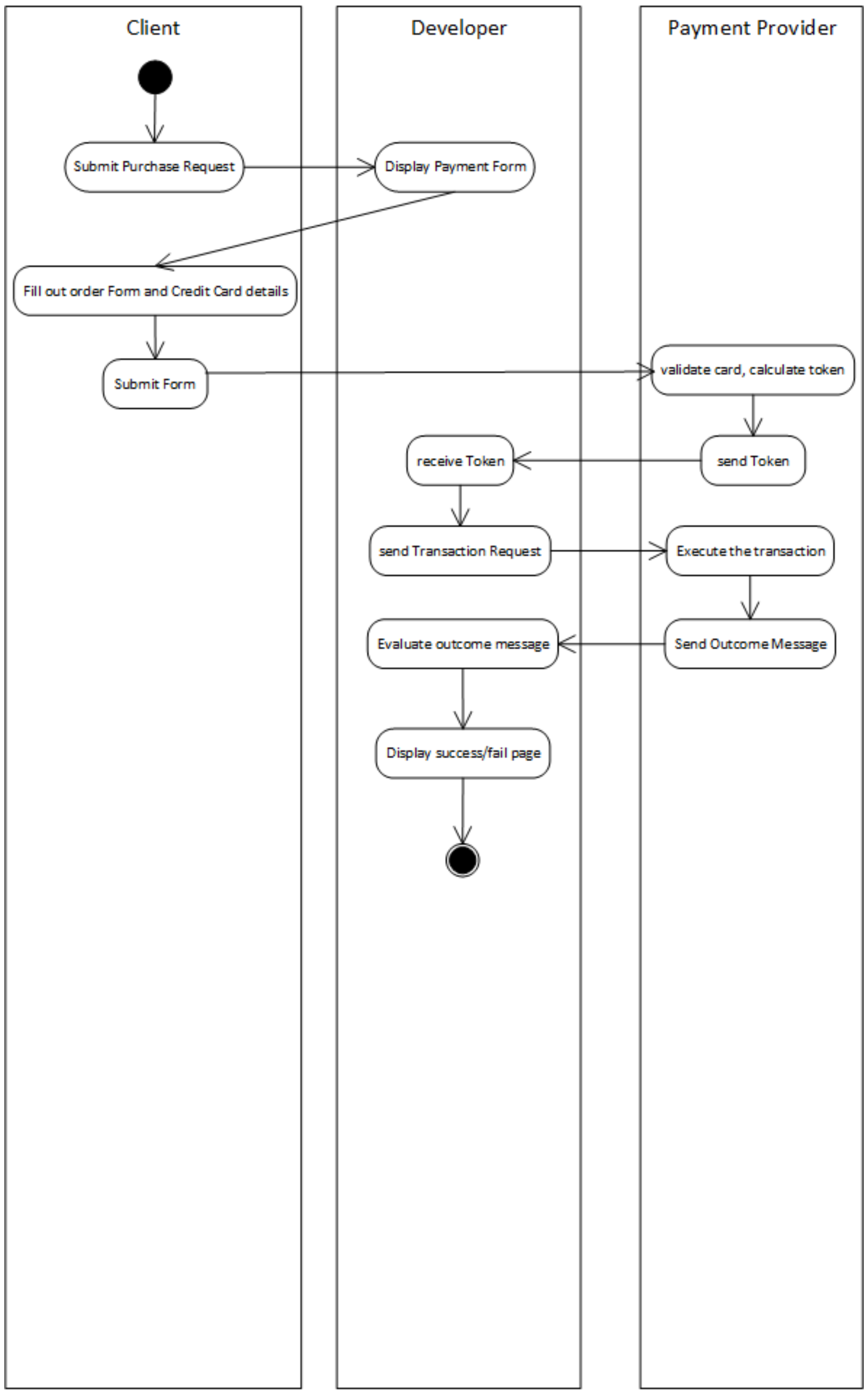
4.	F. Gonidis, I. Paraskakis and A.J.H Simons, "On the role of ontologies in the design of service-based cloud applications," in *the 2nd Workshop on Dependability and Interoperability in Heterogeneous Clouds*. Porto, 2014, pp. 1-12.	Chapters 8,9
5.	F. Gonidis, I. Paraskakis and A.J.H Simons, "A development framework enabling the design of service-based cloud applications," in *the 2nd International Workshop on Cloud Service Brokerage*. Manchester, 2004, pp. 139-152.	Chapters 6,7
6.	F. Gonidis, I. Paraskakis and A.J.H Simons, "Leveraging platform basic services in cloud application platforms for the development of cloud applications," in *6th International Conference on Cloud Computing Technology and Science,* Singapore, 2014, pp.751-754.	Chapters 5-9
7.	F. Gonidis, I. Paraskakis and A.J.H Simons, "Rapid development of service-based cloud applications: The case of cloud application platforms," *International Journal of Systems and Service-Oriented Engineering (IJSSOE),* vol.5, no. 4, 2015, pp.1-25.	Chapters 5-9

APPENDIX: A

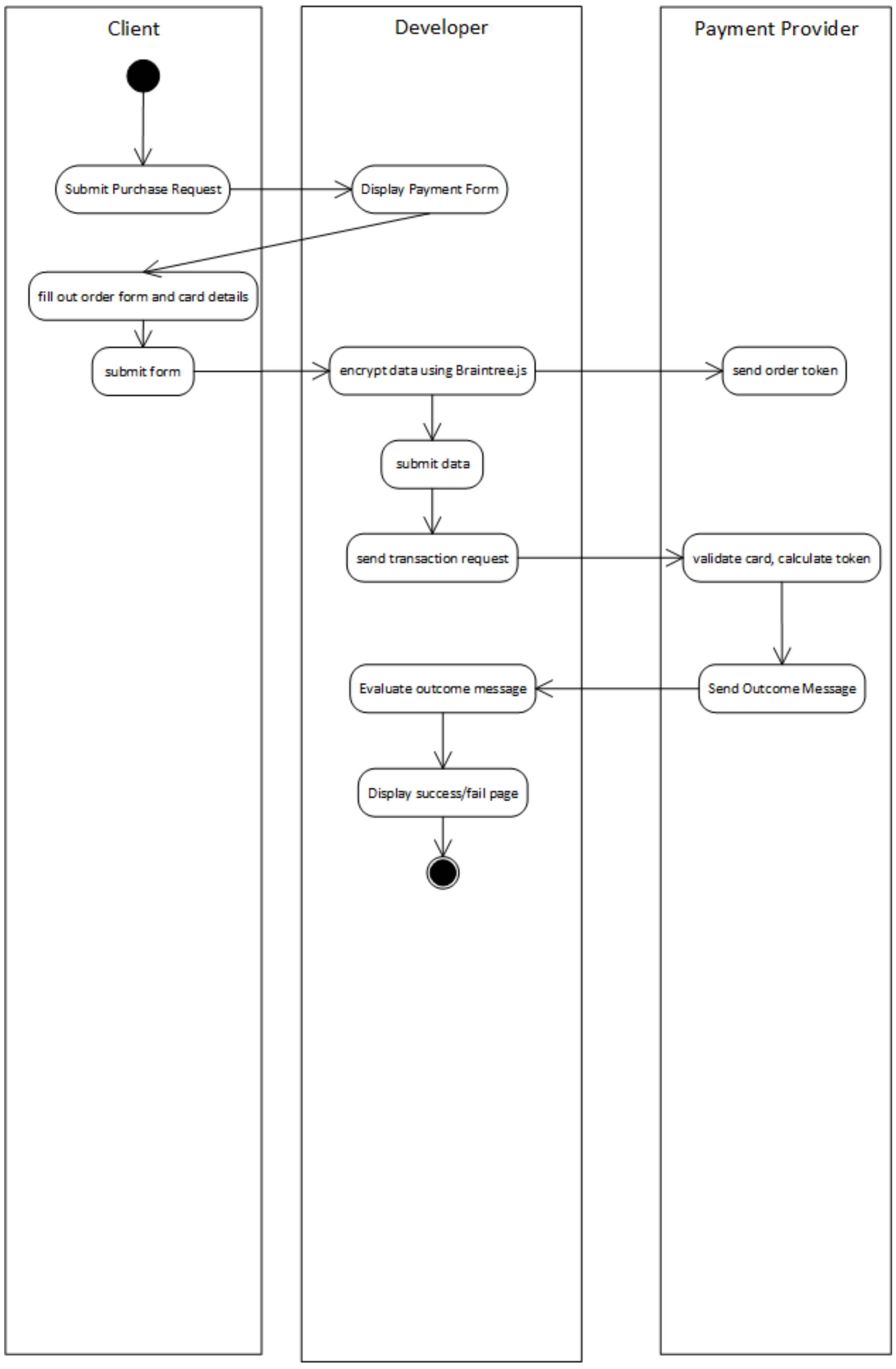
Payment Service Providers

Appendix A lists the activity diagrams of the payment service providers, tested during the definition of the payment service reference workflow.

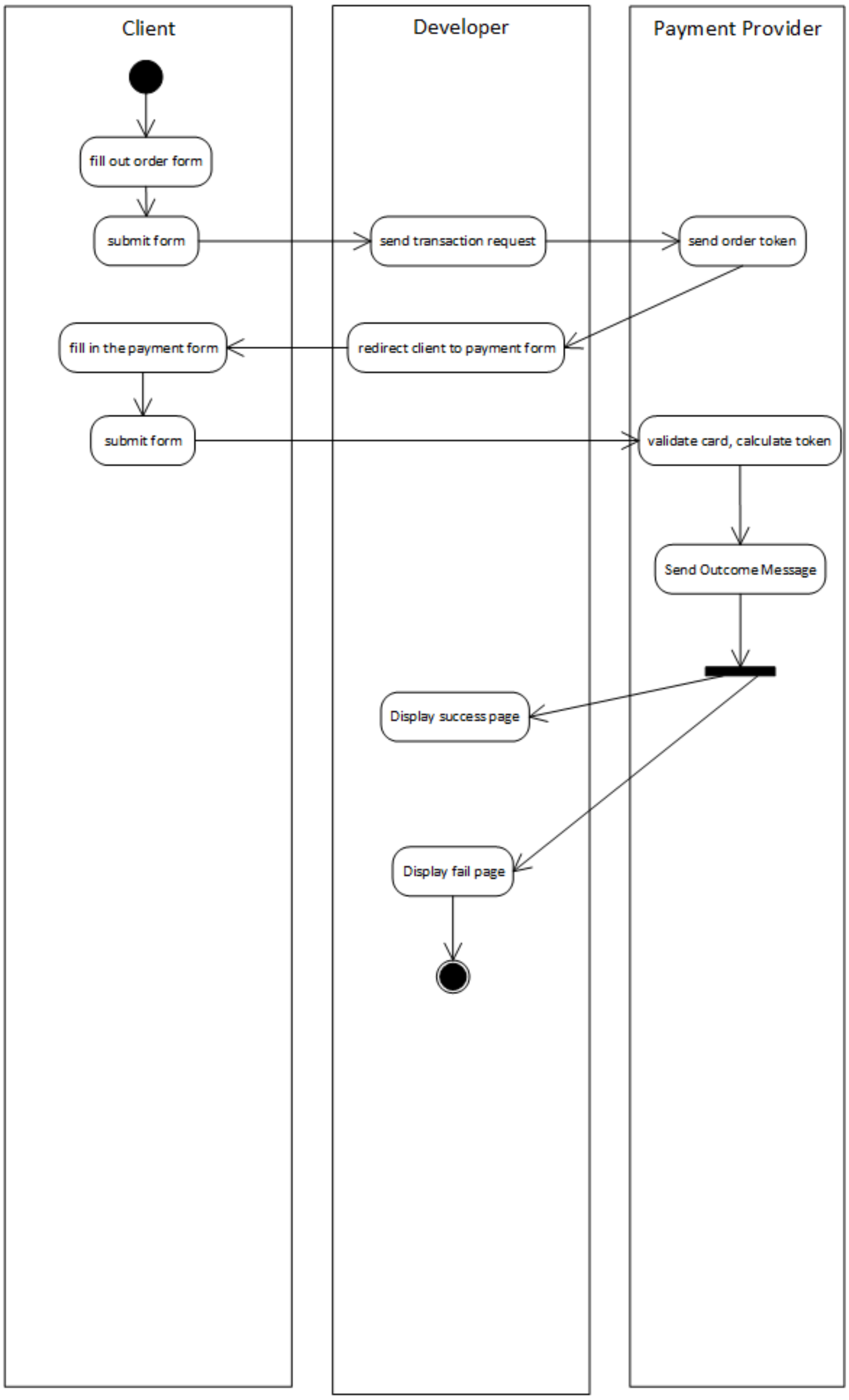
A.1 Spreedly


Client
Developer
Payment Provider
Submit Purchase Request
Display Payment Form
Fill out order Form and Credit Card details
Submit Form
validate card, calculate token
send Token
receive Token
send Transaction Request
Execute the transaction
Send Outcome Message
Evaluate outcome message
Display success/fail page

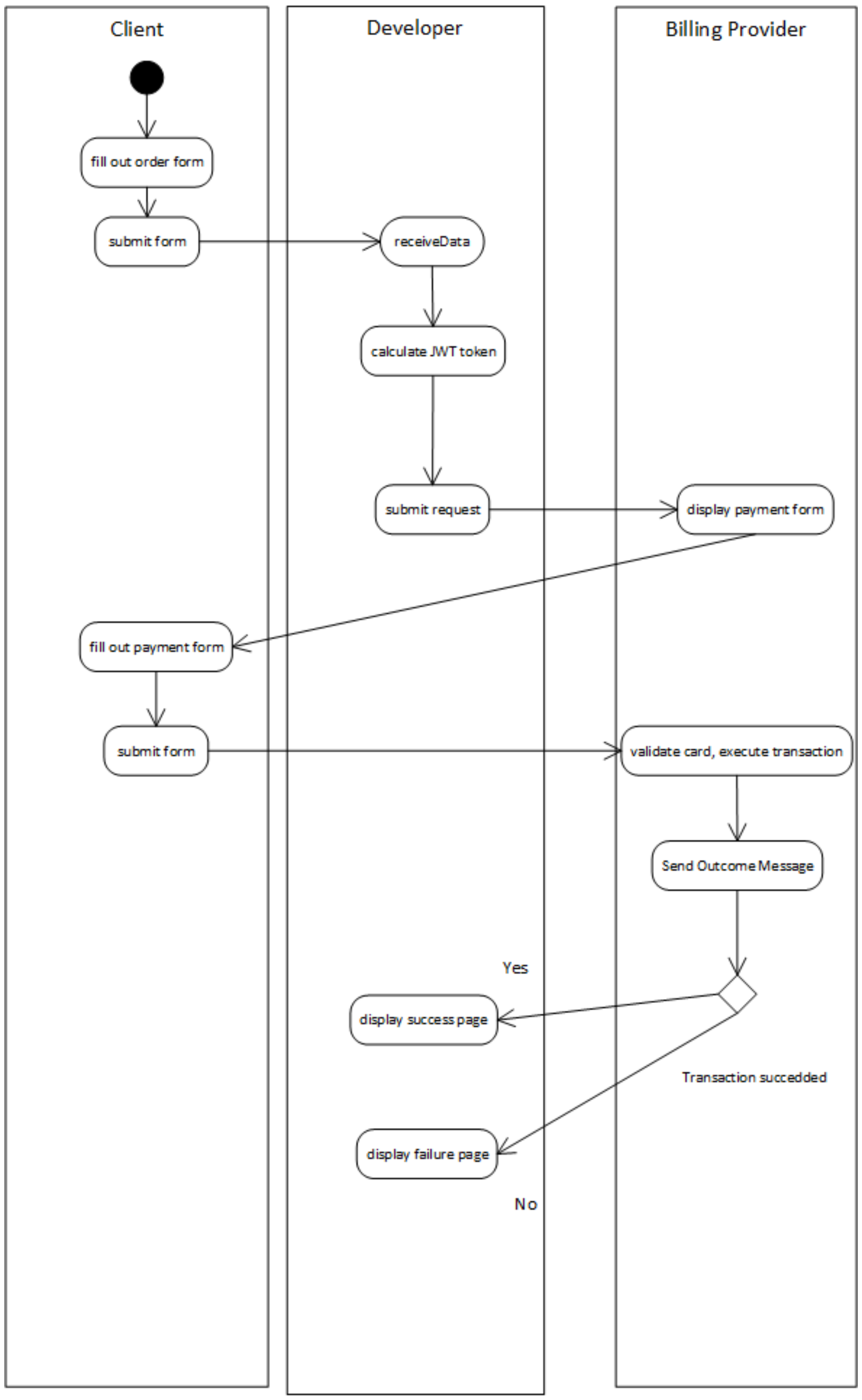
A.2 Braintree



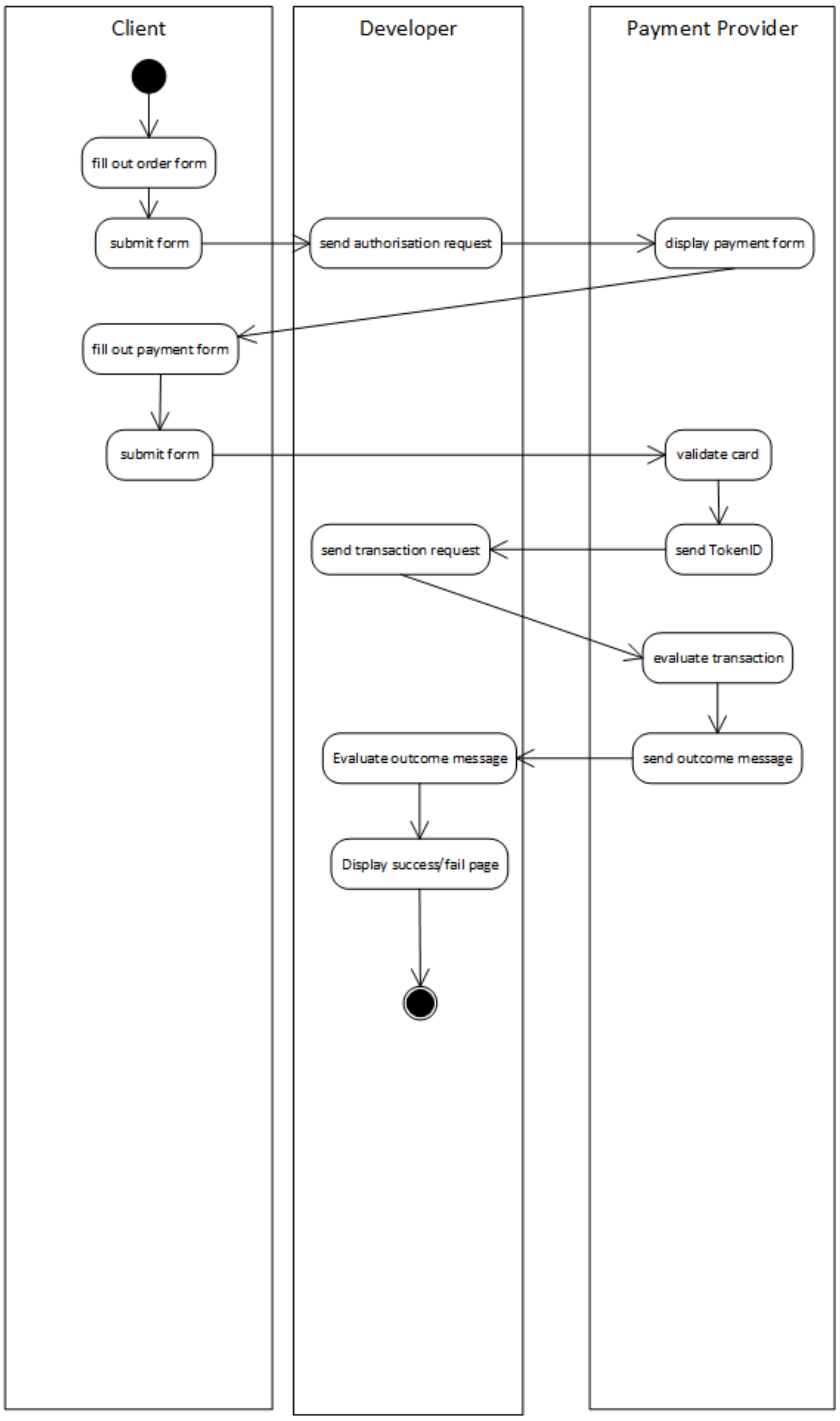
A.3 Viva Payment



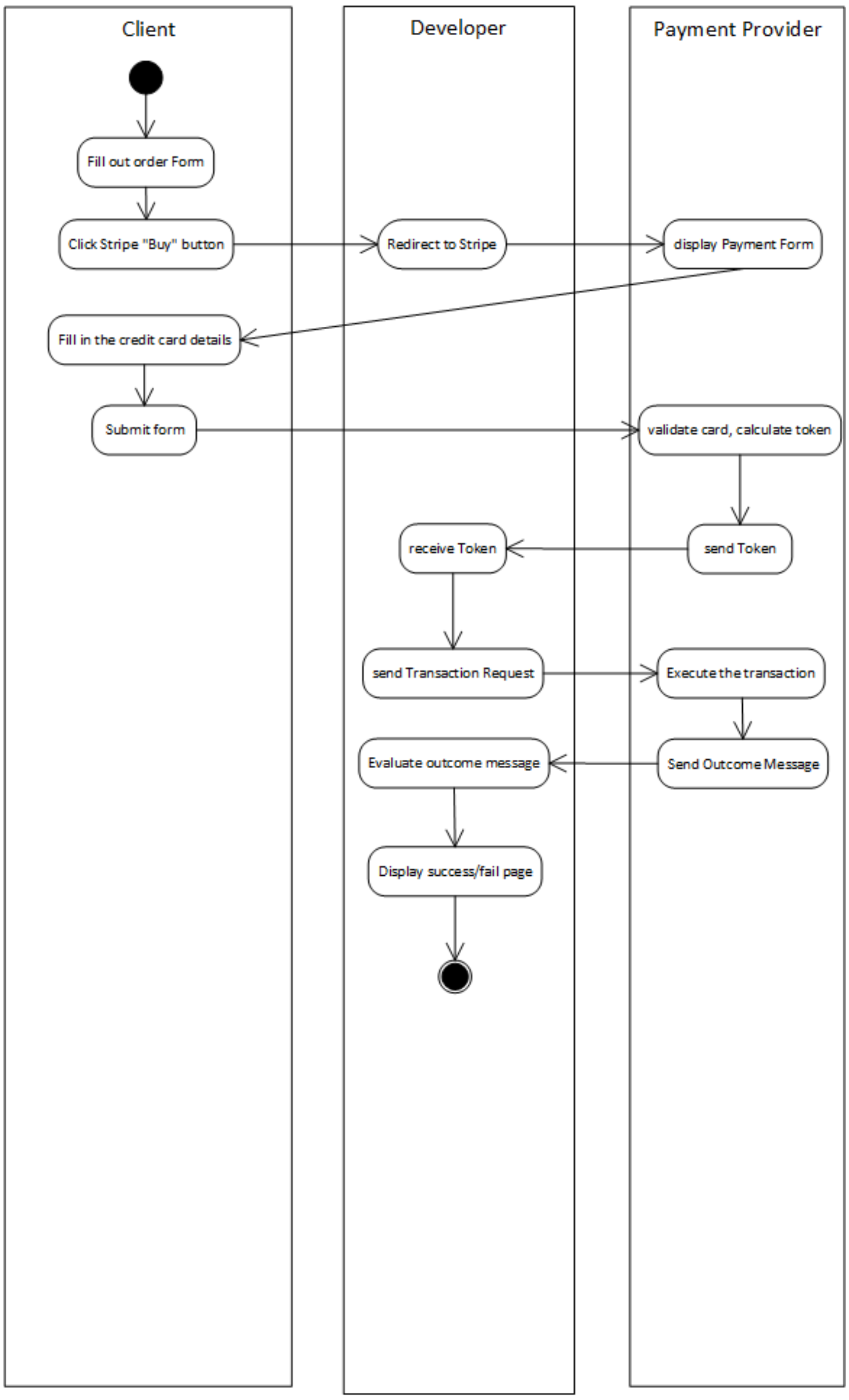
A.4 Google Wallet



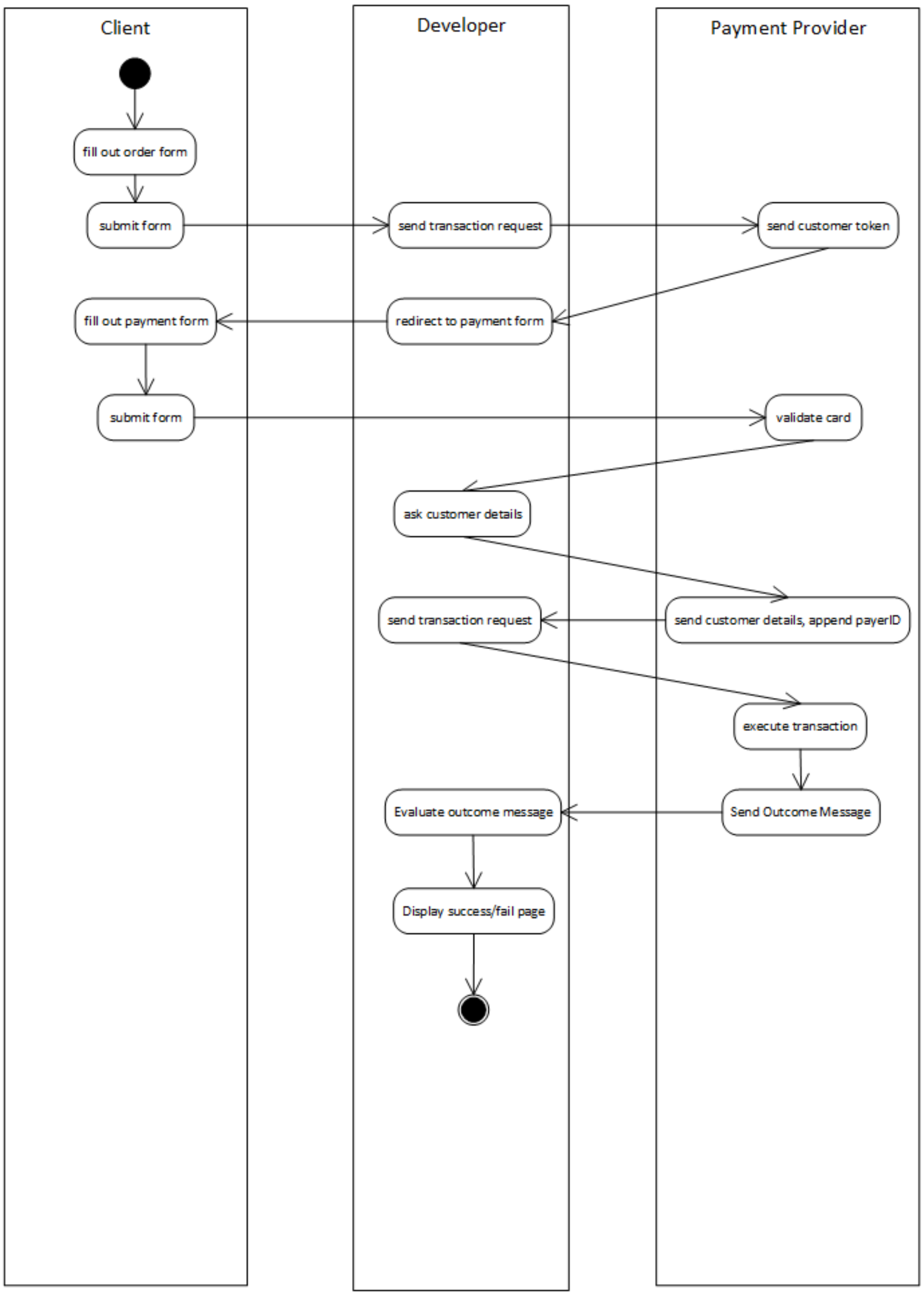
A.5 Amazon Flexible Payments



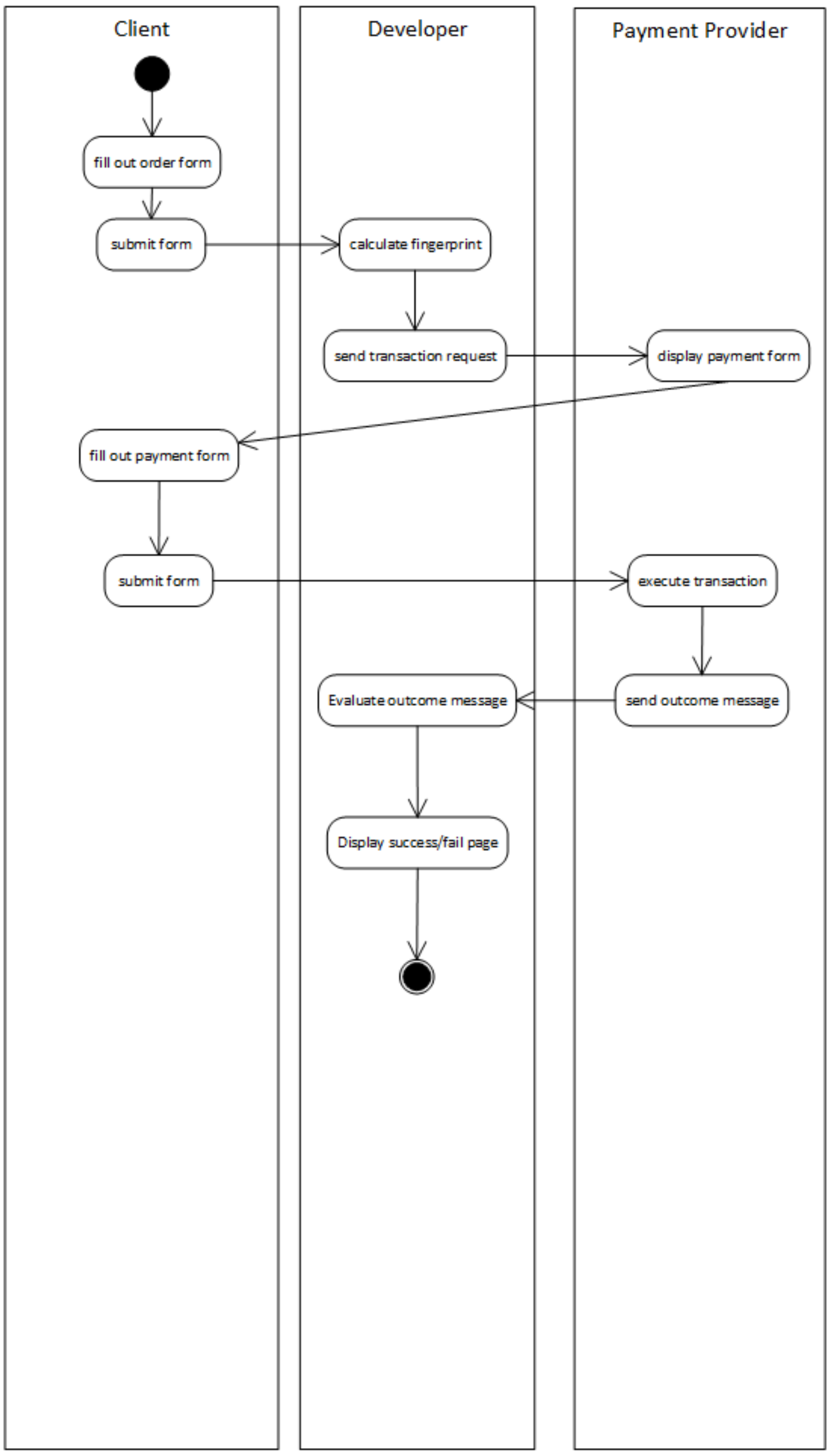
A.6 Stripe


Client
Developer
Payment Provider
Fill out order Form
Click Stripe "Buy" button
Redirect to Stripe
display Payment Form
Fill in the credit card details
Submit form
validate card, calculate token
send Token
receive Token
send Transaction Request
Execute the transaction
Send Outcome Message
Evaluate outcome message
Display success/fail page

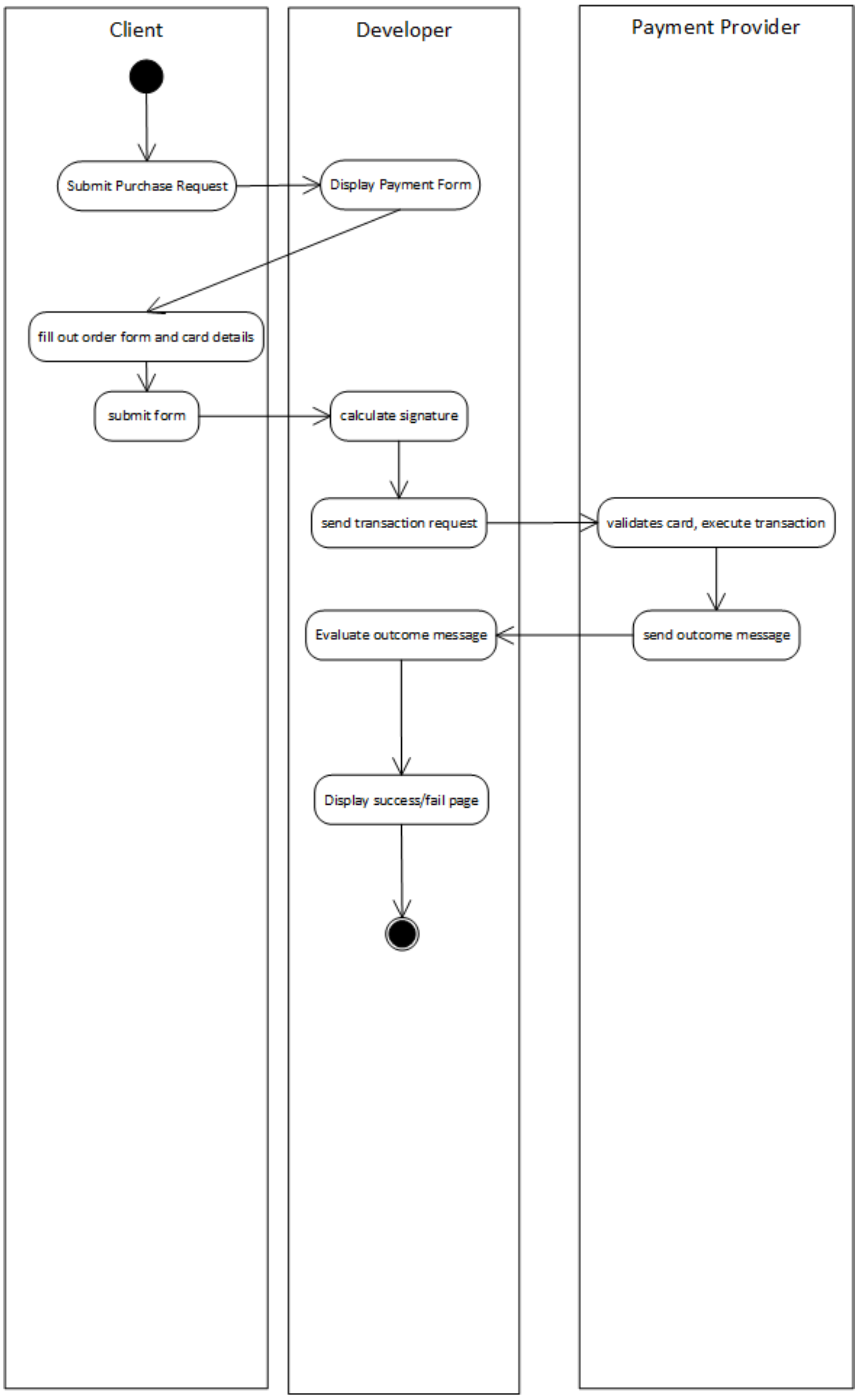
A.7 PayPal Express Checkout



A.8 AuthoriseNetSIM



A.9 Chargify


Client
Developer
Payment Provider
Submit Purchase Request
Display Payment Form
fill out order form and card details
submit form
calculate signature
send transaction request
validates card, execute transaction
send outcome message
Evaluate outcome message
Display success/fail page

APPENDIX: B

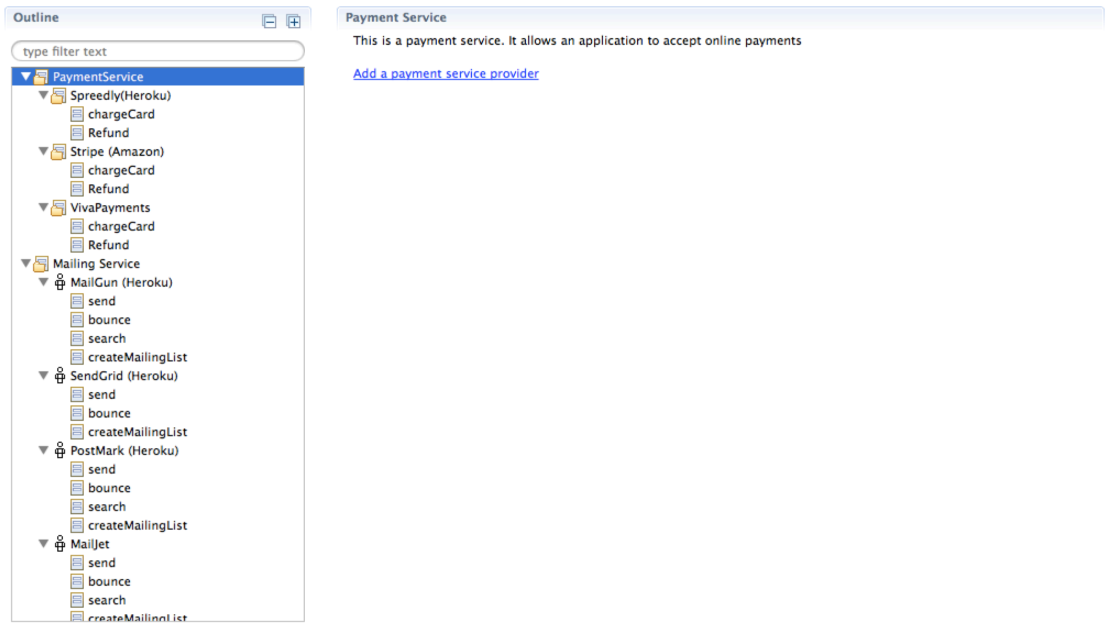
Service Description Editor

The *Service Description Editor (SDE)* is a prototype tool, which enables the software developers to leverage the *SCADeF* framework, which is proposed in this research work. Specifically it facilitates two actions:

i. It allows the administration of the framework to add new *platform basic services* and concrete providers.
ii. It allows the consumers of the framework to select the concrete service providers they want to integrate with the *service-based cloud applications.*

B.1 Add new platform basic service provider

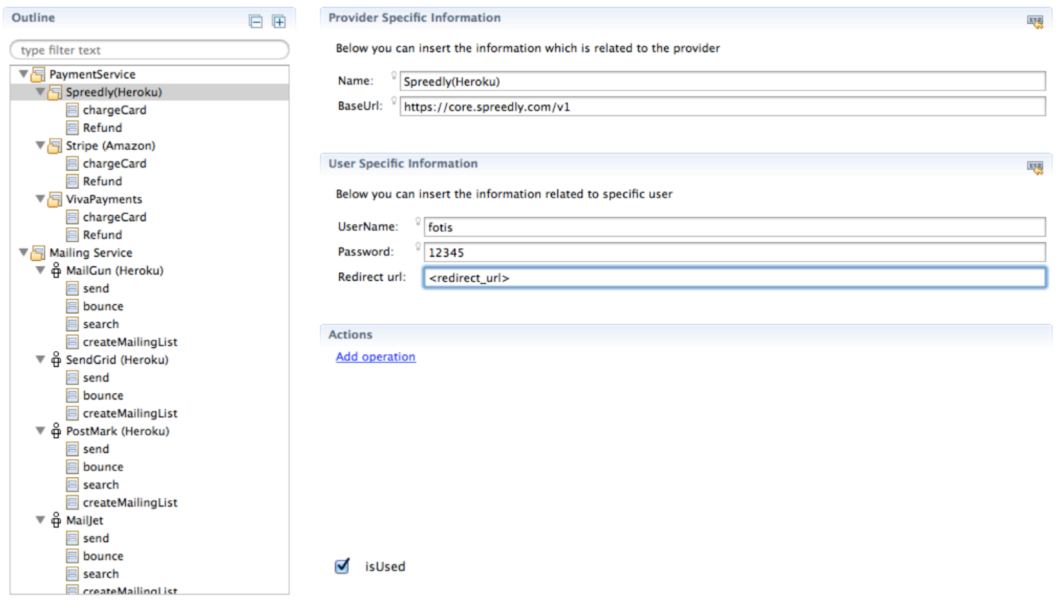
Figure 49 shows the main window of the *SDE.* The *platform basic services* and the respective providers, supported by the framework are listed in the left side of the window such as the payment and the mailing service. The administrator of the framework has the option to add a new payment service provider by clicking the "Add a payment service provider" option.



This is a payment service. It allows an application to accept online payments

Figure 49: Add new platform basic service provider

Next, a new view appears requesting the information required by the specific provider. Figure 50, illustrates the example of the *Spreedly* payment service offered by the *Heroku* platform.



Below you can insert the information which is related to the provider

Name: Spreedly(Heroku)
BaseUrl: https://core.spreedly.com/v1

Below you can insert the information related to specific user

UserName: fotis
Password: 12345
Redirect url: <redirect_url>

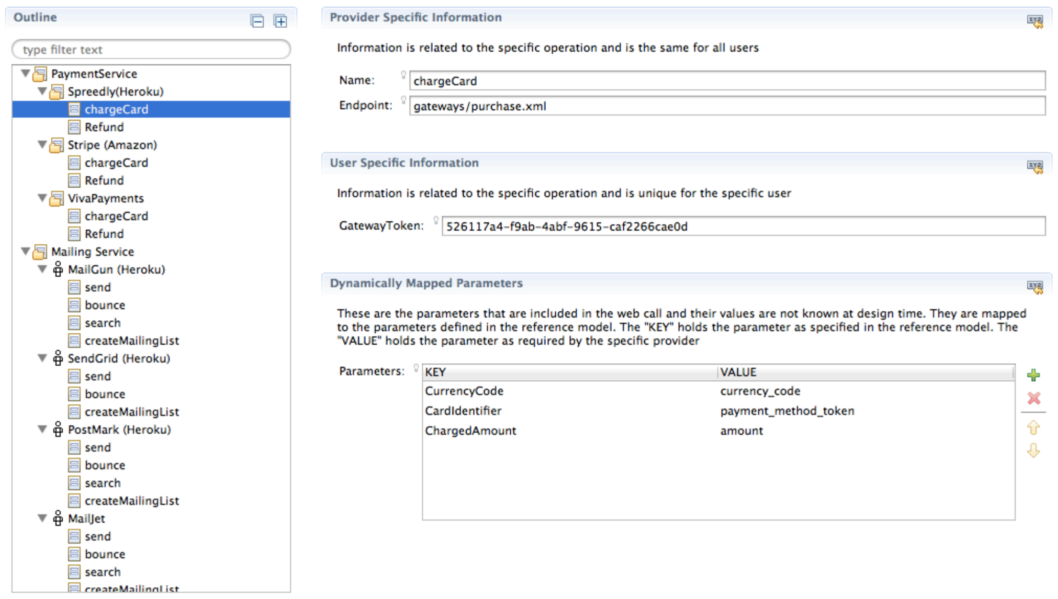
[✓] isUsed

Figure 50: Add Information related to the new service provider

Specifically, the provider (*Name, BaseUrl*) and the user (*UserName, Password* and *Redirect url*) specific information should be filled in. Next, the "Add operation" option allows the definition of the operations supported by the provider. The *isUsed* option, which appears at the bottom of the window, enables the software developers to integrate the specific provider with the *service-based cloud application*.

B.2 Add operations to the new service provider

Figure 51, depicts how an operation can be defined for a service provider. Specfically, the *chargeCard* operation is defined for the *Spreedly* payment provider. The provider (Name, Endpoint) and the user (GatewayToken) specific information are filled in. In the lower part of the window, the “Dynamically Mapped Parameters” are requesting. The “KEY” contains the *Reference API* parameters for the specific service, while the “VALUE” column contains the respective parameters of the specific provider.



Provider Specific Information

Information is related to the specific operation and is the same for all users

Name: chargeCard

Endpoint: gateways/purchase.xml

User Specific Information

Information is related to the specific operation and is unique for the specific user

GatewayToken: 526117a4-f9ab-4abf-9615-caf2266cae0d

Dynamically Mapped Parameters

These are the parameters that are included in the web call and their values are not known at design time. They are mapped to the parameters defined in the reference model. The "KEY" holds the parameter as specified in the reference model. The "VALUE" holds the parameter as required by the specific provider

Parameters:

KEY	VALUE
CurrencyCode	currency_code
CardIdentifier	payment_method_token
ChargedAmount	amount

Figure 51: Add information related to the operations of the service provider

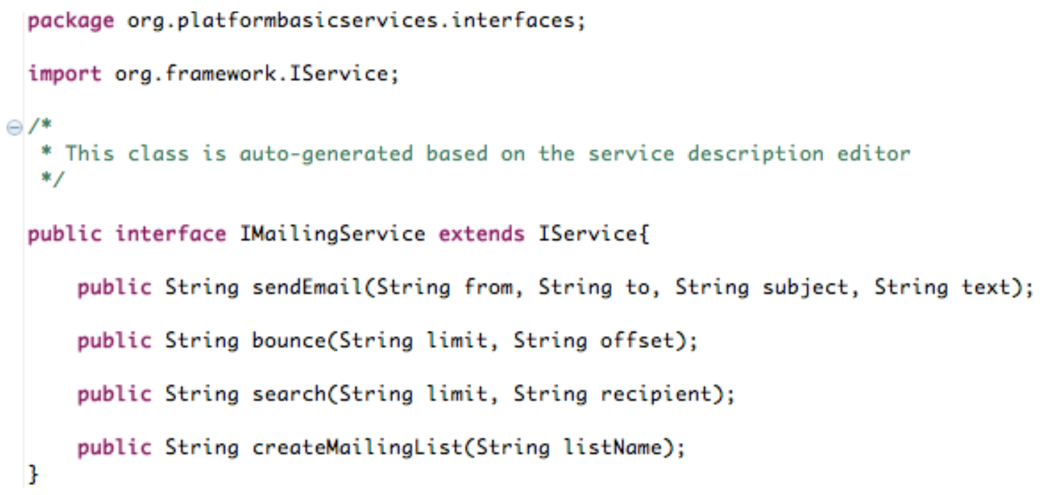
APPENDIX: B

Service Description Editor

APPENDIX: C

Example of Auto-generated source code

Figure 56 and Figure 53 depict the source code, which is generated, for the e-mail service interface and the Postmark implementation respectively.

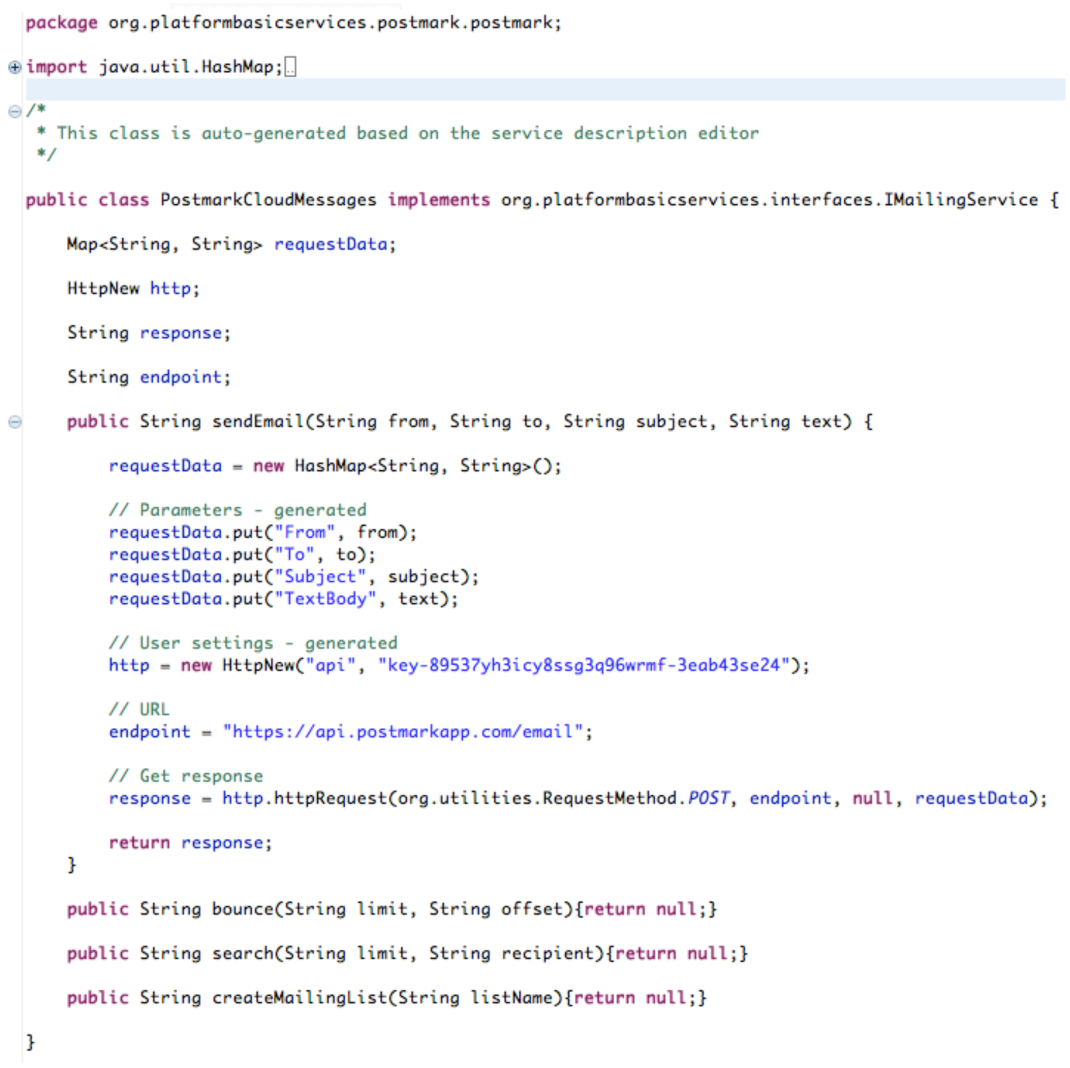


```
package org.platformbasicservices.interfaces;

import org.framework.IService;

/*
 * This class is auto-generated based on the service description editor
 */

public interface IMailingService extends IService{

    public String sendEmail(String from, String to, String subject, String text);

    public String bounce(String limit, String offset);

    public String search(String limit, String recipient);

    public String createMailingList(String listName);
}
```

Figure 52: E-mail service interface

APPENDIX: C

Example of Auto-generated source code



```
package org.platformbasicservices.postmark.postmark;

import java.util.HashMap;

/*
 * This class is auto-generated based on the service description editor
 */

public class PostmarkCloudMessages implements org.platformbasicservices.interfaces.IMailingService {

    Map<String, String> requestData;

    HttpNew http;

    String response;

    String endpoint;

    public String sendEmail(String from, String to, String subject, String text) {

        requestData = new HashMap<String, String>();

        // Parameters - generated
        requestData.put("From", from);
        requestData.put("To", to);
        requestData.put("Subject", subject);
        requestData.put("TextBody", text);

        // User settings - generated
        http = new HttpNew("api", "key-89537yh3icy8ssg3q96wrmf-3eab43se24");

        // URL
        endpoint = "https://api.postmarkapp.com/email";

        // Get response
        response = http.httpRequest(org.utilities.RequestMethod.POST, endpoint, null, requestData);

        return response;
    }

    public String bounce(String limit, String offset){return null;}

    public String search(String limit, String recipient){return null;}

    public String createMailingList(String listName){return null;}

}
```

Figure 53: Postmark implementation

List of acronyms

1. **AEB** – Amazon Elastic Beanstalk
2. **API** – Application Programming Interface
3. **AWS** – Amazon Web Services
4. **CAP** – Cloud Application Platform
5. **DSL** – Domain Specific Language
6. **GAE** – Google App Engine
7. **HTTP** - Hypertext Transfer Protocol
8. **IaaS** – Infrastructure as a Service
9. **IDE** – Integrated Development Environment
10. **ISV** – Independent Software Vendors
11. **PaaS** – Platform as a Service
12. **PC** – Provider Connector
13. **PSC** – Platform Service Connector
14. **PSEC** – Platform Service Execution Controller
15. **REST** – Representational State Transfer Protocol
16. **SaaS** – Software as a Service
17. **SCADeF** – Service-based Cloud Application Framework
18. **SDE** – Service Description Editor
19. **SLA** – Service Level Agreement
20. **SOAP** – Simple Object Access Protocol
21. **SOC** – Service-oriented Computing
22. **TOSCA** – Topology and Specification for Cloud Application
23. **XML** - Extensible Mark-up Language

References

[1] G. Pallis, "Cloud Computing: The new frontier of internet computing," *IEEE Internet Comput.*, vol. 14, no. 5, pp. 70–73, Sep. 2010.

[2] T. Mikkonen and A. Taivalsaari, "Web applications: spaghetti code for the 21st century," Sun microsystems Inc., Mountain View, CA, 2007.

[3] E. W. Dijkstra, "Programming: From Craft to Scientific Discipline," in the International Computing Symposium, Liege, 1977.

[4] N. Wirth, "The Development of Procedural Programming Languages Personal Contributions and Perspectives," in Modular Programming Languages, W. Weck and J. Gutknecht, Eds. Springer Berlin Heidelberg, 2000, pp. 1–10.

[5] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Commun ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[6] W. Pree, "Component-based software development-a new paradigm in software engineering?," in the *4th International Computer Science Conference*, Hong Kong, 1997, pp. 523–524.

[7] O. Nierstrasz, S. Gibbs, and D. Tsichritzis, "Component-Oriented Software Development," *Commun. ACM*, vol. 35, no. 9, pp. 160–165, 1992.

[8] M. Mcllroy, "Mass Produced Software Components," *Proc. NATO Conf. Software Eng.*, pp.88-98, Jan. 1969.

[9] A. Repenning, A. Ioannidou, M. Payton, W. Ye, and J. Roschelle, "Using components for rapid distributed software development," *IEEE Softw.*, vol. 18, no. 2, pp. 38–45, Mar. 2001.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. (1999). Hypertext Transfer Protocol -- HTTP/1.1 [Online]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[11] Tim Bray, Jean Paoli, C. M. Sperber-McQueen, Eve Maler, Francois Yergeau, and John Cowan, "Extensible Markup Language (XML)," W3C, 2006.

[12] M. Bichier and K.-J. Lin, "Service-oriented computing," *Computer*, vol. 39, no. 3, pp. 99–101, Mar. 2006.

[13] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia, "A view of cloud computing," *Comm.ACM,* vol. 53, no. 4, Apr. 2009.

[14] Amazon Web Services. (2013). [Online]. Available: https://aws. amazon. com

[15] Google Cloud. (2015). [Online]. Available: https://cloud.google.com

[16] F. Gonidis, I. Paraskakis, and D. Kourtesis, “Cloud application portability. An initial view,” in *6th Balkan Conference in Informatics*, Thessaloniki, 2013, pp. 275-282.

[17] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: state-of-the-art and research challenges,” *J. Internet Serv. Appl.*, vol. 1, no. 1, pp. 7–18, May 2010.

[18] P. Mell and T. Grance, “The *NIST* definition of cloud computing,” *Natl. Inst. Stand. Technol.*, Gaithersburg, 800-145, MD, 2011.

[19] J. Staten, L. E. Nelson, D. Bartoletti, L. Herbert, W. Martorelli, and H. Baltazar, “Predictions 2015: The days of fighting the cloud are over,” Forrester Research Inc., Cambridge, MA, 2014.

[20] D. M. Smith, D. C. Plummer, and D. W. Cearley, “The What, Why and When of Cloud Computing,” Gartner Inc., Stamfodrd, CT, G00168582, 2009.

[21] Gartner Inc. (2015). [Online]. Available: http://www.gartner.com/

[22] (2014). *Gartner Identifies the Top 10 Strategic Technology Trends for 2015* [Online]. Available: http://www.gartner.com/newsroom/id/3143521

[23] International Data Corporation, IDC. (2015). [Online]. Available: https://www.idc.com

[24] L. Carvalho, M. Fleming, A. Hilwa, R. P. Mahowald, and B.McGrath, “Worldwide competitive public cloud platform as a service 2014–2018 Forecast and 2013 Vendor Shares,” IDC, Framingham, MA, 243315, 2014.

[25] S. M. M. Tahaghoghi and H. E. Williams, *Learning MySQL. Get a handle on your data*. Sebastopol, CA: O’Reilly Media, 2006.

[26] Apache Tomcat. (2015). [Online]. Available: http://tomcat.apache.org

[27] K. Schaefer, J. Cochran, S. Forsyth, D. Glendenning, and B. Perkins, *Professional Microsoft IIS 8*. Hoboken, N.J.: Wiley / Wrox, 2012.

[28] Heroku. (2015). [Online]. Available: https://www.heroku.com

[29] Openshift. (2015). [Online]. Available: https://www.openshift.com

[30] Engine Yard. (2015). [Online]. Available: https://www.engineyard.com

[31] R. T. Fielding, “Architectural styles and the design of network-based software architectures”. Ph.D. dissertation, Univ. of California, Irvine, CA, 2005.

[32] D. Kourtesis, K. Bratanis, D. Bibikas, and I. Paraskakis, “Software Co-development in the era of cloud application platforms and ecosystems: The case of CAST,” in *Collaborative Networks in the Internet of Services*, vol. 380, L. M. Camarinha-Matos, L. Xu, and H. Afsarmanesh, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 196–204.

[33] S. Newman, *Building microservices*. Sebastopol, CA: O’Reilly Media, 2015.

[34] Heroku Add-ons. (2015). [Online]. Available: https://elements.heroku.com/ addons

[35] Sendgrid. (2015). [Online]. Available: https://sendgrid.com

[36] Mailgun. (2015). [Online]. Available: https://www.mailgun.com

[37] Postmark. (2015). [Online]. Available: https://postmarkapp.com

[38] Spreedly. (2015). [Online]. Available: https://spreedly.com/

[39] Stripe. (2015). [Online]. Available: https://stripe.com/

[40] E. Jendrock, I. Evans, D. Gollapudi, K. Haase, and C. Srivathsa, *The Java EE 6 tutorial: Basic concepts*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2010.

[41] Gigaom research. (2015). [Online]. Available: http://research.gigaom.com

[42] D. S. Linthicum. (2014). *Why PaaS growth is disproportional to other sectors* [Online]. Available: http://research.gigaom.com/2014/10/why-paas-growth-is-disproportional-to-other-sectors/

[43] J. KinCaid. (2009). *Coghead Grinds To A Halt, Heads To The Deadpool* [Online]. Available: http://techcrunch.com/2009/02/18/coghead-grinds-to-a-halt-heads-to-the-deadpool/.

[44] Sun Microsystems. (2015). [Online]. Available: http://www.oracle.com/sun

[45] N. Langley. (2002). *Write once, run anywhere Computer* [Online]. Available: http://www.computerweekly.com/feature/Write-once-run-anywhere

[46] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[47] G. A. Lewis, "The Role of Standards in Cloud Computing Interoperability," Carnegie Mellon, Pittsburgh, PA, CMU/SEI-2012-TN-012, Oct. 2012.

[48] F. Gonidis, I. Paraskakis, and D. Kourtesis, "Addressing the Challenge of Application Portability in Cloud Platforms," in *7th South-East European Doctoral Student Conference*, Thessaloniki, 2012, pp. 565–576.

[49] "Cloud Data Management Interface (CDMI)," Storage Networking Industry Association (SNIA), 1.1.0, Aug. 2014.

[50] IBM. (2015). [Online]. Available: http://www.ibm.com

[51] Hewlett Packard. (2015). [Online]. Available: http://www.hp.com

[52] R. Cattell, *Jdbc database access with Java: A tutorial annotated reference*. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 1997.

[53] Salesforce. (2015). *10 great cloud applications and services for SMEs* [Online]. Available: http://www.salesforce.com/uk/socialsuccess/cloud-computing/10-great-cloud-applications-services-smes.jsp

[54] Oracle. (2012) *Cloud application. Driving enterprise-grade cloud applications: The benefits of cloud without compromise* [Online]. Available: http://www.oracle.com/us/c-central/cio-executive-insights/ess-oracle-cloud-1731443.pdf

[55] S. L. Garfinkel and H. Abelson, *Architects of the information society: Thirty-Five Years of the Laboratory for Computer Science at MIT*. Cambridge, MA: The MIT Press, 1999.

[56] Hewlett-Packard. (2011). *Five myths of cloud computing* [Online]. Available: http://www.hp.com/hpinfo/newsroom/press_kits/2011/HPDiscover2011/DISCOVER_5_Myths_of_Cloud_Computing.pdf

[57] M. A Vouk, "Cloud computing–issues, research and implementations," *CIT J. Comput. Inf. Technol.*, vol. 16, no. 4, pp. 235–246, 2008.

[58] D. Zissis and D. Lekkas, "Addressing cloud computing security issues," *Future Gener Comput Syst*, vol. 28, no. 3, pp. 583–592, Mar. 2012.

[59] A. Sinha, "Client-server computing," *Commun ACM*, vol. 35, no. 7, pp. 77–98, Jul. 1992.

[60] I. Chengalur-Smith and P. Duchessi, "The initiation and adoption of client–server technology in organizations," *Inf. Manage.*, vol. 35, no. 2, pp. 77–88, Feb. 1999.

[61] T. G. Lewis, "Where is client/server software headed?," *Computer*, vol. 28, no. 4, pp. 49–55, Apr. 1995.

[62] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environment Workshop,* Austin, TX, 2008, pp. 1–10.

[63] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *Int J High Perform Comput Appl*, vol. 15, no. 3, pp. 200–222, Aug. 2001.

[64] I. Foster and C. Kesselman, Eds., *The grid: Blueprint for a new computing infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[65] S. Marston, Z. Li, S. Bandyopadhyay, and A. Ghalsasi, "Cloud computing - The business perspective," in *44th Hawaii International Conference on System Sciences*, Kauai, HI, 2011, pp. 1–11.

[66] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: A research roadmap," *Int. J. Coop. Inf. Syst.*, vol. 17, no. 02, pp. 223–255, Jun. 2008.

[67] Y. Wei and M. B. Blake, "Service-Oriented Computing and Cloud Computing: Challenges and Opportunities," *IEEE Internet Comput.*, vol. 14, no. 6, pp. 72–75, 2010.

[68] M. N. Huhns and M. P. Singh, "Service-oriented computing: key concepts and principles," *IEEE Internet Comput.*, vol. 9, no. 1, pp. 75–81, Jan. 2005.

[69] Erik Christensen, Greg Meredith, Francisco Curbera, and Sanjiva Weerawarana, "Web Services Description Language (WSDL)," World Wide Web (W3C), wsdl-20010315, Mar. 2001.

[70] D. Box, D. Ehnebuske, G. Kakivaya et al. (2010). *Simple Object Access Protocol (SOAP) 1.1.* [Online]. Available: http://www.w3.org/TR/2000/NOTE-SOAP-20000508

[71] D. S. Linthicum, *Cloud computing and SOA convergence in your enterprise: A step-by-step guide*. Boston, MA: Addison-Wesley Professional, 2009.

[72] Amazon Elastic Compute Cloud. (2015). [Online]. Available: https://aws.amazon.com/ec2/

[73] A. Mohamed (2009). *A history of cloud computing* [Online]. Available: http://www.computerweekly.com/feature/A-history-of-cloud-computing

[74] GAE. (2015). [Online]. Available: https://appengine. google.com/

[75] Windows Azure. (2013). [Online]. Available: https://azure.microsoft.com

[76] Google Compute Engine. (2015). [Online]. Available: https://cloud.google. com/compute/

[77] Amazon Simple Storage Service. (2015). [Online]. Available: https://aws.amazon.com/s3/

[78] Storage Documentation | Windows Azure. (2015). [Online] Available: http://www.windowsazure.com/en-us/documentation/services/storage/

[79] Zoho Creator. (2015). [Online]. Available: https://www.zoho.com/creator/

[80] SalesForce. (2015). [Online]. Available: http://www.salesforce.com

[81] Y. V. Natis, E. Knipp, R. Valdes, M. Pezzini, D. Sholler, D. W. Cearley, D. M. Smith, and J. Thompson, "Who's who in application platforms for cloud computing: The enterprise generalists," Gartner Inc, Stamfodrd, CT, G00170223, 2009.

[82] M. Carlson, M. Chapman, A. Heneveld, S. Hinkelman, D. Johnston-Watt, A. Karmarkar, T. Kunze, A. Malhotra, J. Mischkinsky, A. Otto, V. Pandey, G. Pilz, Z. Song, and P. Yendluri, "Cloud application management for platforms," OASIS, 2012.

[83] Lee Badger, Tim Grance, Robert Patt-Corner, and Jeff Voas, "*NIST* cloud computing synopsis and recommendations," National Institute of Standards and Technology, MD 20899 - 8930, 2012.

[84] R. Johnson, J. Holler, A. Arendsen, T. Risberg, and C. Sampaleanu, *Professional Java development with the Spring framework*. Hoboken, N.J.: Wiley / Wrox, 2005.

[85] H. Lam and T. L. Thai, *.NET framework essentials, 2nd edition*. Sebastopol, CA: O'Reilly Media, 2002.

[86] B. Momjian, *PostgreSQL: Introduction and concepts*. Indianapolis, IND: Addison-Wesley, 2001.

[87] Y. V. Natis, B. J. Lheureux, M. Pezzini, D. W. Cearley, E. Knipp, and D. C. Plummer, “PaaS road map: A continent emerging,” Gartner Inc, Stamford, CT, G00209751, 2011.

[88] Amazon Elastic Beanstalk. (2015). [Online]. Available: https://aws.amazon. com/elasticbeanstalk/

[89] D. Zeginis, F. D’Andria, S. Bocconi, J. Gorronogoitia Cruz, O. Collell Martin, P. Gouvas, G. Ledakis, and K. A. Tarabanis, “A user-centric multi-PaaS application management solution for hybrid multi-Cloud scenarios,” *Scalable Comput. Pract. Exp.*, vol. 14, no. 1, Apr. 2013.

[90] App Cloud. (2015). [Online]. Available: http://www.salesforce.com/ platform/ solutions/automate-business-processes/

[91] G. Breiter and M. Behrendt, “Life cycle and characteristics of services in the world of cloud computing,” *IBM Journal of Research and Development,* vol. 53, no. 3, pp. 527-534, Jul 2009.

[92] Instagram. (2015). [Online]. Available: https://instagram.com

[93] R. Branson, P. Canahuati, and N. Shortway. (2014). *Migrating From AWS to FB* [Online]. Available: http://instagram-engineering.tumblr.com/post/ 89992572022/migrating-aws-fb

[94] S. Kolb and G. Wirtz, “Towards application portability in platform as a service,” in *8th International Symposium on Service Oriented System Engineering*, Oxford, 2014, pp. 218–229.

[95] Nawsher Khan, A. Noraziah, Elrasheed I. Ismail, Mustafa Mat Deris, and Tutut Herawan, “Cloud computing: Analysis of various platforms,” *Int. J. E-Entrep. Innov.*, vol. 3, no. 2, p. 51-59, 2012.

[96] M. Pastaki Rad, A. Sajedi Badashian, G. Meydanipour, M. Ashurzad Delcheh, M. Alipour, and H. Afzali, “A survey of cloud platforms and their future,” in *International Conference on Computational Science and its Applications,* Suwon, 2009, pp. 788–796.

[97] S. Ried and J. R. Rymer, “The Forrester wave^{TM}: Platform-as-a-service for vendor strategy professionals, Q2 2011,” Forrester, 2011.

[98] Saugatuck Technology Inc. (2010). *Development in the cloud: A Framework for PaaS and ISV flexibility* [Online]. Available: http:// www.galeos.cz/ uploads/Soubory/DatasheetyProgress/saugatuck_progress_cloud_dev_framework.pdf

[99] R. Cattell, “Scalable SQL and NoSQL data stores,” *SIGMOD Rec*, vol. 39, no. 4, pp. 12–27, Dec. 2010.

[100] Rapidcloud. (2015). [Online]. Available: https://rapidcloud.io

[101] Shelly Cloud. (2015). [Online]. Available: https://shellycloud.com

[102] AppFog. (2015). [Online]. Available: https://www.appfog.com

[103] IBM Bluemix. (2015). [Online]. Available: http://www.ibm.com/cloud-computing/bluemix/

[104] Play. (2015). [Online]. Available: https://www.playframework.com

[105] Grape. (2015). [Online]. Available: https://www.ruby-toolbox.com/projects/ grape

[106] Rack. (2015). [Online]. Available: http://rack.github.io

[107] V. Viswanathan, “Rapid web application development: A Ruby on Rails tutorial,” *IEEE Softw.*, vol. 25, no. 6, pp. 98–106, Nov. 2008.

[108] A. Harris and K. Haase, *Sinatra: Up and running. Ruby for the web, simply*. Sebastopol, CA: O’Reilly Media, 2011.

[109] Redhat. (2015). [Online]. Available: http://www.redhat.com

[110] D. Golding, *Beginning CakePHP*. New York, NY: Apress, 2008.

[111] JBoss Application Server. (2015). [Online]. Available: http://jbossas.jboss. org

[112] zend Server. (2015). [Online]. Available: http://www.zend.com/en/products/ zend_server

[113] A. A. Donovan, B. W. Kernighan, *The Go programming language.* Boston, MA: Addison-Wesley, 2015.

[114] webapp2. (2015). [Online]. Available: https://webapp-improved.appspot. com

[115] A. Holovaty and J. Kaplan-Moss, *The Definitive Guide to Django: Web Development Done Right (Expert’s Voice in Web Development)*. Berkeley, CA: Apress, 2009.

[116] J. K. VanDyk, *Pro Drupal development*. New York, NY: Apress, 2007.

[117] Cloud Foundry. (2015). [Online] . Available: https://www.cloudfoundry.org

[118] K. Banker, *MongoDB in action*. Greenwich, CT, USA: Manning Publications Co., 2011.

[119] J. L. Carlson, *Redis in action.* Greenwich, CT, USA: Manning Publications Co., 2013.

[120] B. Fitzpatrick, “Distributed caching with memcached,” *Linux J*, vol. 2004, no. 124, p. 5–, Aug. 2004.

[121] Amazon DynamoDB. (2015). [Online]. Available: http://aws.amazon.com/ dynamodb/

[122] S. Ramanathan, S. Goel, and S. Alagumalai, “Comparison of cloud database: Amazon’s SimpleDB and Google’s Bigtable,” in *International Conference on Recent Trends in Information Systems*, Kolkata, 2011, pp. 165–168.

[123] MC Brown, *Getting Started with Couchbase Server*. Sebastopol, CA: O’Reilly Media, 2012.

[124] Cloud SQL. (2015). [Online]. Available: https://cloud.google.com/sql/

[125] Cloud Datastore. (2015). [Online]. Available: https://cloud.google.com/ datastore/docs/concepts/overview

[126] E. Redmond. (2015). *A little Riak book* [Online]. Available: http://littleriakbook.com

[127] Google Cloud Storage. (2015). [Online]. Available: https://cloud.google.com/storage/

[128] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in GitHub: Transparency and collaboration in an open software repository," in *ACM Conference on Computer Supported Cooperative Work*, New York, NY, 2012, pp. 1277–1286.

[129] Bitbucket. (2015). [Online]. Available: https://bitbucket.org

[130] Cloudinary. (2015). [Online]. Available: http://cloudinary.com

[131] Heroku add-on services. (2015). [Online]. Available: https://elements.heroku.com

[132] Engineyard add-ons. (2015). [Online]. Available: https://addons.engineyard. com

[133] Openshift marketplace. (2015). [Online]. Available: https://marketplace.openshift.com

[134] "Open virtualization format specification," Distributed Management Task Force, DSP0243, 2013.

[135] T. Metsch and A. Edmonds, "Open cloud computing interface - infrastructure," Open Grid Forum, GFD-P-R.184, 2011.

[136] A. Chaddha, "Cloud interoperability and standardisation," *SETLabs Brief.*, vol. 7, no. 7, pp. 19–26, 2009.

[137] Cloud Computing Interoperability Forum (CCIF). (2012). [Online]. https://groups.google.com/forum/#!forum/cloudforum

[138] D. Doug and P. Gilbert, "Cloud Infrastructure Management Interface (CIMI) Primer," Distributed Management Task Force (DMTF), DSP2027, 2012.

[139] Distributed Management Task Force (DMTF). (2015). [Online]. Available: https://www.dmtf.org

[140] "Cloud Data Management Interface (CDMI^{TM})," Storage Networking Industry Association (SNIA), Version 1.0.2, 2012.

[141] OASIS - Advancing open standards for the information society. (2015). [Online]. Available: https://www.oasis-open.org

[142] OASIS Cloud Application Management for Platforms. (2015). [Online] Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev =camp

[143] G. Katsaros, M. Menzel, A. Lenk, J. Rake-Revelant, R. Skipp, and J. Eberhardt, "Cloud Application Portability with TOSCA, Chef and Openstack," in *IEEE International Conference on Cloud Engineering*, Boston, MA, 2014, pp. 295–302.

[144] J. Durand, A. Otto, G. Pilz, and T. Rutt, "Cloud application management for platforms," OASIS, camp-spec-v1.1-cs01, 2014.

[145] K. Bakshi and M. Skilton, "Cloud computing portability and interoperability," The Open Group, Reading, UK, G135.

[146] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical review of vendor lock-in and its impact on adoption of cloud computing," in *International Conference on Information Society*, London, 2014, pp. 92–97.

[147] M. Singhal, S. Chandrasekhar, T. Ge, R. Sandhu, R. Krishnan, G.-J. Ahn, and E. Bertino, "Collaboration in multicloud computing environments: Framework and security issues," *Computer*, vol. 46, no. 2, pp. 76–84, Feb. 2013.

[148] D. Petcu, "Consuming resources and services from multiple clouds," *J. Grid Comput.*, vol. 12, no. 2, pp. 321–345, Jun. 2014.

[149] M. D. Hogan, F. Liu, A. W. Sokol, and T. Jin, "*NIST* cloud computing standards roadmap," *NIST*, SP500-291-v1.0, 2011.

[150] E. M. Maximilien, A. Ranabahu, R. Engehausen, and L. C. Anderson, "Toward cloud-agnostic middlewares," in *24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, New York, NY, 2009, pp. 619–626.

[151] D. Baur, S. Wesner, J. Domaschka, "Towards a Model-based execution-ware for deploying multi-cloud applications," in *2nd International Workshop on Cloud Service Brokerage*, Manchester, 2014, pp. 124-138.

[152] S. Subashini and V. Kavitha, "Review: A survey on security issues in service delivery models of cloud computing," *J Netw Comput Appl*, vol. 34, no. 1, pp. 1–11, Jan. 2011.

[153] D. Petcu and A. V. Vasilakos, "Portability in clouds: approaches and research opportunities," *Scalable Comput. Pract. Exp.*, vol. 15, no. 3, pp. 251-270, Dec. 2014.

[154] J. Guillén, J. Miranda, J. M. Murillo, and C. Canal, "A Service-oriented framework for developing cross cloud migratable software," *J Syst Softw*, vol. 86, no. 9, pp. 2294–2308, Sep. 2013.

[155] N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg, "Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems," in *Sixth International Conference on Cloud Computing*, Santa Clara, CA, 2013, pp. 887–894.

[156] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *Computer*, vol. 37, no. 7, pp. 56–64, Jul. 2004.

[157] J. Mooney, "Bringing portability to the software process," Dept. of Statistics and Comp. Sci., West Virginia Univ., Morgantown, WV, 1997.

[158] S. Comella-Dorda, K. Wallnau, R. C. Seacord, and J. Robert, "A survey of black-box modernization approaches for information systems," in *International Conference on Software Maintenance,* San Jose, CA, 2000, pp. 173–183.

[159] D. V. Silakov and A. V. Khoroshilov, “Ensuring portability of software,” *Program. Comput. Softw.*, vol. 37, no. 1, pp. 41–47, Jan. 2011.

[160] K. Geiger, *Inside ODBC*. Redmond, WA: Microsoft Press, 1995.

[161] J Richter, *Applied Microsoft .NET framework programming*. Redmond, Washington: Microsoft Press, 2002.

[162] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston:MA: Addison-Wesley Professional Computing Series, 1995.

[163] jclouds. (2015). [Online]. Available: http://www.jclouds.org

[164] Apache Libcloud. (2015). [Online]. Available: https://libcloud.apache.org

[165] fog. (2015). [Online]. Available: http://fog.io

[166] pckgcloud. (2015). [Online]. https://github.com/pkgcloud/pkgcloud

[167] elibcloud. (2015). [Online]. Available: https://github.com/esl/elibcloud

[168] K. Geihs, “Middleware challenges ahead,” *Computer*, vol. 34, no. 6, pp. 24–31, Jun. 2001.

[169] P. A. Bernstein, “Transaction processing monitors,” *Commun ACM*, vol. 33, no. 11, pp. 75–86, Nov. 1990.

[170] Y. V. Natis, D. W. McCoy, B. Gassman, J. Sinur, J. Thompson, M. Pezzini, L. F. Kenney, T. Friedman, M. R. Gilbert, G. Phifer, W. R. Schulte, and B. J. Lheureux, “Who’s who in middleware, 1Q04,” Gartner, Stamford, CT, G00120308, 2004.

[171] D. Alur, D. Malks, J. Crupi, G. Booch, and M. Fowler, *Core J2EE patterns (core design series): Best practices and design strategies*. Mountain View, CA: Sun Microsystems, Inc., 2003.

[172] M. E. Taylor, “WebSphere MQ Primer: An introduction to messaging and webSphere MQ,” IBM, REDP-0021-01, 2012.

[173] C. Zhang and H. A. Jacobsen, “Quantifying aspects in middleware platforms,” in *the 2Nd International Conference on Aspect-oriented Software Development*, New York, NY, 2003, pp. 130–139.

[174] D. Petcu, B. Martino, S. Venticinque, M. Rak, T. Máhr, G. Lopez, F. Brito, R. Cossu, M. Stopar, S. Šperka, and V. Stankovski, “Experiences in building a mOSAIC of clouds,” *J. Cloud Comput. Adv. Syst. Appl.*, vol. 2, no. 1, p. 12, Dec. 2013.

[175] D. Petcu, C. Craciun, M. Neagul, I. Lazcanotegui, and M. Rak, “Building an interoperability API for Sky computing,” in *International Conference on High Performance Computing and Simulation*, Istanbul, 2011, pp. 405–411.

[176] K. Jeffery, G. Horn, and L. Schubert, “A vision for better cloud applications,” in *International Workshop on Multi-cloud Applications and Federated Clouds*, New York, NY, 2013, pp. 7–12.

[177] K. Kritikos, J. Domaschka, and A. Rossini, "SRL: A scalability rule language for multi-cloud environments," in *6th International Conference on Cloud Computing Technology and Science*, Singapore, 2014, pp. 1–9.

[178] D. Ardagna, E. Di Nitto, P. Mohagheghi, S. Mosser, C. Ballagny, F. D'Andria, G. Casale, P. Matthews, C.-S. Nechifor, D. Petcu, A. Gericke, and C. Sheridan, "MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds," in *ICSE Workshop on Modeling in Software Engineering*, Zurich, 2012, pp. 50–56.

[179] MODAClouds Multi-Cloud DevOps Toolbox. (2015). [Online]. Available: http://multiclouddevops.com/technologies.html

[180] R. Zabolotnyi, P. Leitner, W. Hummer, and S. Dustdar, "JCloudScale: Closing the gap between IaaS and PaaS," *ACM T. Internet Techn.*, vol. 15, no. 3, pp. 1-20, Sep. 2015.

[181] J. Gullien, J. Miranda, J. M. Murillo, C. Canal, "Developing migratable multicloud applications based on MDE and adaptation techniques," in *Second Nordic Symposium on Cloud Computing & Internet Technologies,* Oslo, 2013, pp. 30-37.

[182] Apache Maven Project. (2015). [Online]. Available: https://maven.apache. org

[183] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery – A modeling tool for TOSCA-based cloud applications," in *11th International Conference on Service Oriented Computing*, Berlin, 2013, pp. 700–704.

[184] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA – A runtime for TOSCA-based cloud applications," in *11th International Conference on Service Oriented Computing*, Berlin, 2013, pp. 692–695.

[185] Bran Selic, "MDE basics with a UML focus," presented at the *12th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Model-Driven Engineering*, Bertinoro, 2012.

[186] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Syst. J.*, vol. 45, no. 3, pp. 621 –645, Jul. 2006.

[187] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, Mar. 2006.

[188] Paul Boocock. (2012). "Jamda Model Compiler Framework," *The JAMDA project* [Online]. Available: http://jamda.sourceforge.net/docs/index.html.

[189] "XPand," *Eclipse Modeling M2T - Home*. (2012). [Online]. Available: http://www.eclipse.org/modeling/m2t/?project=xpand.

[190] M. Hamdaqa, T. Livogiannis, and L. Tahvildari, "A reference model for developing cloud applications," in 1st *International Conference on Cloud Computing and Services Science,* Noordwijkerhout, pp. 98–103, 2011.

[191] P. Mohagheghi, A. J. Berre, A. Henry, F. Barbier, A. Sadovykh, “REuse and Migration of legacy applications to interoperable cloud services,” in *Third European Conference ServiceWave,* Ghent, 2010, pp. 195-196.

[192] P. Mohagheghi and T. Sæther, “Software engineering challenges for migration to the service cloud paradigm: Ongoing work in the REMICS Project,” in *IEEE World Congress on Services*, Washington, DC, 2011, pp. 507–514.

[193] G. Baryannis, P. Garefalakis, K. Kritikos, K. Magoutis, A. Papaioannou, D. Plexousakis, and C. Zeginis, “Lifecycle management of service-based applications on multi-clouds: A research roadmap,” in *International Workshop on Multi-cloud Applications and Federated Clouds*, New York, NY, 2013, pp. 13–20.

[194] A. H. Ranabahu, E. M. Maximilien, A. P. Sheth, and K. Thirunarayan, “A domain specific language for enterprise grade cloud-mobile hybrid applications,” in *SPLASH Workshops,* New York, NY, 2011, pp. 77–84.

[195] Robert Eckstein, *Java SE application design with MVC.* Sun Microsystems Inc, 2007.

[196] “GlassFish Server. (2015). [Online]. Available: https://glassfish.Java.net

[197] D. Namiot and M. Sneps-Sneppe, “On Micro-services architecture,” *International Journal of Open Information Technologies,* vol. 2, no. 9, 2014, pp. 24-27.

[198] Bob Rhubart, “Microservices and SOA,” *Oracle Mag.*, Mar. 2015.

[199] S. Pautasso, O. Zimmerman andF. Leymann, “Restful web services vs. "big"' web services making the right architectural decision.” In 17th International Conference on World Wide Web, Beijing, 2008, pp. 805-814.

[200] Amazon Simple E-mail Service. (2015). [Online]. Available: https:// aws.amazon.com/ses/

[201] Google Mail. (2015). [Online]. Available: https://cloud.google.com/ appengine/docs/Java/mail/

[202] S. Kent, “Model driven engineering,” in *the Third International Conference on Integrated Formal Methods,* Turku, 2002, pp. 286–298.

[203] N. Li, C. Pedrinaci, M. Maleshkova, J. Kopecky, and J. Domingue, “OmniVoke: A framework for automating the invocation of web APIs,” in *the Fifth IEEE International Conference on Semantic Computing*, Palo Alto, CA, 2011, pp. 39–46.

[204] F. Gonidis, I. Paraskakis and A.J.H Simons, “Leveraging platform basic services in cloud application platforms for the development of cloud applications,” in *the Sixth International Conference on Cloud Computing Technology and Science,* Singapore, 2014, pp. 751-754.

[205] F. Gonidis, I. Paraskakis and A.J.H Simons, “A development framework enabling the design of service-based cloud applications,” in *the Sceond*

International Workshop on Cloud Service Brokerage. Manchester, 2004, pp. 139-152.

[206] R. T. Fielding and R. N. Taylor, “Principled design of the modern web architecture,” *ACM Trans Internet Technol*, vol. 2, no. 2, pp. 115–150, May 2002.

[207] F. Gonidis, I. Paraskakis and A.J.H Simons, “Rapid development of service-based cloud applications: The case of cloud application platforms,” *International Journal of Systems and Service-Oriented Engineering (IJSSOE),* vol.5, no. 4, 2015, pp.1-25.

[208] Google Wallet. (2014). [Online]. Available: http://www.google.gr/wallet/

[209] “Payment Card Industry (PCI) Data Security Standard,” PCI Security Standards Council, Version 2.0, 2010.

[210] Jason Hunter and William Crawford, *Java servlet programming*. Sebastopol, CA:O’Reilly Media, 2001.

[211] Ken Arnold, James Gosling, and David Holmes, *The Java programming language*. Boston, MA:Addison Wesley Professional, 2005.

[212] W. L. Hürsch and C. V. Lopes, “Separation of concerns,” Northeastern University, Boston, MA, NU-CCS-95-03, 1995.

[213] J. S. Poulin, “Measuring software reusability,” in *Third International Conference on Software Reuse: Advances in Software Reusability,* Rio De Janeiro, 1994, pp. 126–138.

[214] I. R. Forman, N. Forman, D. J. V. Ibm, I. R. Forman, and N. Forman, *Java Reflection in Action*. Greenwich, CT, USA: Manning Publications Co., 2004.

[215] S. Shavor, J. D’Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy, *The Java developer’s guide to Eclipse*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[216] T. Austin, “The cloud email and collaboration services market, 2011 Update,” Gartner, Stamford, CT, G00215500, Aug. 2011.

[217] MailJet. (2015). [Online]. Available: https://www.mailjet.com

[218] CloudControl. (2015). [Online]. Available: https://www.cloudcontrol.com

[219] Rackspace. (2015). [Online]. Available: http://www.rackspace.com

[220] CloudBees. (2015). [Online]. Available: https://www.cloudbees.com

[221] World Wide Web Consortium. (2015). [Online]. Available: http://www.w3. org

[222] Marc Hadley, “Web Application Description Language (WADL),” World Wide Web (W3C), 20090831, 2009.

[223] Joel Farrell and Holger Lausen, “Semantic annotations for WSDL and XML Schema (SAWSDL),” World Wide Web (W3C), 20070828, 2007.

[224] C. Peltz, "Web services orchestration and choreography," *Computer*, vol. 36, no. 10, pp. 46–52, Oct. 2003.

[225] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowl Acquis*, vol. 5, no. 2, pp. 199–220, Jun. 1993.

[226] F. Gonidis, I. Paraskakis and A.J.H Simons, "On the role of ontologies in the design of service-based cloud applications," in *Second Workshop on Dependability and Interoperability in Heterogeneous Clouds*. Porto, 2014, pp. 1-12.

[227] C. Pedrinaci, J. Cardoso, and T. Leidig, "Linked USDL: A vocabulary for web-scale service trading," in *11th International Conference Extended Semantic Web Conference*, Crete, 2014, pp. 68–82.

[228] M. Maleshkova, C. Pedrinaci, N. Li, J. Kopecky, and J. Domingue, "Lightweight semantics for automating the invocation of Web APIs," in *International Conference on Service-Oriented Computing and Applications*, Irvine, CA, 2011, pp. 1–4.