Handling Cloud PaaS Platforms Heterogeneity

by

Eman Hossny Abdelghony

A thesis submitted in partial fulfillment for the degree of

Doctor of Philosophy in Computer Science

under supervision of

Prof. Fatma A. Omara
Prof. Hesham A. Hassan
Dr. Sherif A. Khatab

DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF COMPUTERS AND INFORMATION
CAIRO UNIVERSITY
EGYPT

2017
Declaration of Authorship

I, Eman Hossny, declare that this thesis titled, ‘Handling Cloud PaaS Platforms Heterogeneity’ and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.

- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.

- Where I have consulted the published work of others, this is always clearly attributed.

- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

- I have acknowledged all main sources of help.

Signed:

Date:
Acknowledgements

At first, I thank Allah who provided me with all the countless success reasons and the fruitful environment by which this work is fulfilled. I am also so grateful to my supervisors, Prof. Fatma Omara, Prof. Hesham Hassan, and Dr. Sherif Khatab, for providing continuous support, guidance, mentoring, and technical advice over this thesis. In addition to their continuous review of the thesis and its related research articles. Finally, I thank my family for their incredible patience and support throughout the lifetime of this thesis.

Part of this work was done in the ITEA2 EASI-CLOUDS project (http://easi-clouds.eu). The Egyptian consortium in the project was funded by ITIDA ITAC from 10/2012 to 1/2014.
Abstract

The providers of Platform as a Service (PaaS) have their own specific APIs (i.e., heterogeneous). These specific APIs make developers locked inside a specific platform and they can’t able to port their applications easily among different platforms. As a result, vendor lock-in problem has taken place. One solution for this problem is to use generic APIs with specific adapters for different PaaS platforms to implement portable applications. However, any update in a PaaS specific-API makes its corresponding adapter to become unusable, which causes what is called API dynamic adaptation problem. Therefore, the work in this thesis provides three main contributions to solve these problems, (1) extend the Compatible One Application and Platform Service (COAPS) generic deployment-API, (2) propose service-based generic-APIs (Std-PaaS APIs) and (3) propose a Semantic-based Generation of Generic-API adapters (STAGER) framework. According to the first contribution, the COAPS generic deployment-API has been extended to deploy applications on Google App Engine (GAE) platform, besides CloudFoundry (CF) and OpenShift (OS). Regarding the second contribution, the proposed service-based generic-APIs (called Std-PaaS APIs) can be used by cloud developers to implement generic applications. In addition, the proposed Std-PaaS APIs can include a set of generic APIs for each PaaS service (e.g., blob storage, datastore, messaging, etc.). Currently, the Std-PaaS APIs include two generic APIs; blob storage and NoSQL datastore services. With respect to the third contribution, the proposed STAGER framework is used to semi-automatically generate the adapters for any semantically annotated generic APIs. STAGER framework has been evaluated using the Std-PaaS APIs by generating their adapters for two PaaS platforms; Google App Engine (GAE) and Windows Azure. Although there are some overheads for semantically annotating the PaaS APIs, the evaluation results prove the feasibility of STAGER framework and promote the usage of the generated adapters for implementing portable cloud applications. The main feature of our code generation approach is that it is based on SPARQL queries and can map each generic method into one or more specific methods. Therefore, it is considered more flexible than other existing code generation approaches.
Contents

Declaration of Authorship ................................. i
Acknowledgements ........................................ ii
Abstract ................................................... iii
List of Figures ............................................. vii
List of Tables .............................................. ix
Abbreviations .............................................. x

1 Introduction .............................................. 1
  1.1 Motivation ........................................... 2
  1.2 Problem Definition ................................. 3
  1.3 Thesis Contributions ............................... 5
  1.4 Thesis Outline ..................................... 6
  1.5 Summary ............................................ 7

2 Background .............................................. 8
  2.1 Cloud Computing .................................... 8
      2.1.1 Cloud Service Models ......................... 9
      2.1.2 Cloud Deployment Models .................... 10
  2.2 PaaS Service Model ................................ 10
      2.2.1 PaaS Application Provisioning .............. 11
      2.2.2 Cloud Blob Storage Service (CBSS) .......... 12
          2.2.2.1 Google Cloud Storage (GCS) APIs .... 12
          2.2.2.2 Azure Blob Storage APIs ............. 12
      2.2.3 NoSQL Datastore Service .................... 13
          2.2.3.1 GAE NoSQL datastore APIs ............ 14
          2.2.3.2 Azure NoSQL datastore APIs .......... 15
  2.2.4 Compatible One Application Platform Service (COAPS) API .. 16
      2.2.4.1 COAPS API Overview ...................... 16
      2.2.4.2 PaaS Application Description Model (PADM) of COAPS 19
      2.2.4.3 Deployment Scenario using COAPS API .... 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3</td>
<td>Semantic Ontology</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>2.3.1 Ontology Specification Languages</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>2.3.2 SPARQL Query Language</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>2.3.3 Source Code Representation Ontology (SCRO)</td>
<td>24</td>
</tr>
<tr>
<td>2.4</td>
<td>JAVA Reflection Mechanism and API Design</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>2.4.1 JAVA reflection mechanism</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>2.4.2 API Design</td>
<td>25</td>
</tr>
<tr>
<td>2.5</td>
<td>Summary</td>
<td>26</td>
</tr>
<tr>
<td>3</td>
<td>Literature Review</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>3.1 Standardization Efforts</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>3.2 Research Projects</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>3.3 Generic APIs</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>3.4 Discussion</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>3.5 Summary</td>
<td>41</td>
</tr>
<tr>
<td>4</td>
<td>The Proposed Generic Cloud APIs</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>4.1 COAPS Deployment API</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>4.1.1 Motivation To Extend COAPS API</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>4.1.2 Extending COAPS API To Support GAE</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>4.1.2.1 GAE PaaS Platform Overview</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>4.1.2.2 The Extended COAPS Architecture</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>4.1.2.3 Road Map To Extend COAPS API For Supporting GAE</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>4.1.3 COAPS Summary</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>4.2 Std-PaaS Implementation APIs</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>4.2.1 Std-PaaS APIs Architecture</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>4.2.2 The Proposed CBSS Generic-API</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>4.2.2.1 Studying Specific CBSS API</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>4.2.2.2 Comparison of CBSS API of GAE and Azure Platforms</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>4.2.2.3 CBSS Generic-API</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>4.2.3 The Proposed NSDS Generic-API</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>4.2.3.1 Studying Specific NSDS API</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>4.2.3.2 Comparison of NSDS API of GAE and Azure Platforms</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>4.2.3.3 NSDS Generic-API</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>4.3 Summary</td>
<td>59</td>
</tr>
<tr>
<td>5</td>
<td>The Proposed Semantic-Based Generation of Generic-API Adapters for Portable Cloud Applications (STAGER) Framework</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>5.1 STAGER Overview</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>5.2 STAGER Architecture</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>5.3 Motivation and Assumptions</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>5.4 Components of STAGER Framework</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>5.4.1 STAGER Ontology</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>5.4.1.1 SCRO*</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>5.4.1.2 Domain Specific Ontology</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>5.4.1.3 Std-PaaS APIs Ontology</td>
<td>69</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Relationships among the proposed contributions ............... 6

2.1 Cloud Computing Architecture ............................... 9
2.2 COAPS API Architecture .................................... 17
2.3 UML diagram for PADM of COAPS API ...................... 19
2.4 PaaS Application Manifest Sample .......................... 21
2.5 A snapshot of SCRO Ontology ............................... 25

4.1 GAE Application Structure ................................. 44
4.2 COAPS new Architecture with GAE ......................... 45
4.3 GAE module is added to COAPS API ....................... 46
4.4 The Credentials file of GAE-PaaS API ..................... 47
4.5 Std-PaaS APIs Architecture ............................... 49

5.1 The architecture of STAGER framework ....................... 62
5.2 The process follow of STAGER framework .................... 63
5.3 The STAGER Ontology Structure ............................. 67
5.4 The New Properties of SCRO+ Ontology ..................... 69
5.5 Domain Specific Ontologies for Blob Storage and NoSQL Datastore 70
5.6 A snapshot of the Std-PaaS APIs ontology .................... 70
5.7 Semi-automatic Adapter Validation and Generation (SAVG) Architecture 74
5.8 The initial version of the SPARQL query .................... 75
5.9 A SPARQL query to retrieve a list of generic methods ........ 76
5.10 A visual representation for the automatic SPARQL query generation with depth two .................... 79

6.1 App Converter Class Diagram ............................... 83
6.2 App Converter Component .................................. 84
6.3 The class diagram of the main classes in GAE Adapter ........ 84
6.4 Sequence diagram for deploying an application using COAPS API with GAE extension .................. 93
6.5 COAPS API Web Client .................................... 93
6.6 Create Environment ........................................ 94
6.7 Create Application ......................................... 95
6.8 Deploy Application Response ............................... 95
6.9 A sample application deployed on GAE using COAPS API + GAE extension 96
6.10 A sample application deployed on CF Platform ............. 96
6.11 The path detection for the createBlob generic method for Azure adapter 98
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.12</td>
<td>The semantic annotations of the createBlob generic method</td>
<td>99</td>
</tr>
<tr>
<td>6.13</td>
<td>Parse six libraries using the SCOP component</td>
<td>101</td>
</tr>
<tr>
<td>6.14</td>
<td>The size of generated ontologies vs. Avg. parsing time</td>
<td>101</td>
</tr>
<tr>
<td>6.15</td>
<td>The Avg. code generation time of CBSS generic-APIs for both GAE and Azure</td>
<td>102</td>
</tr>
<tr>
<td>6.16</td>
<td>Execution Time of CBSS Generic-API vs. CBSS Native-API of GAE</td>
<td>104</td>
</tr>
<tr>
<td>6.17</td>
<td>Execution Time of NSDS Generic-API vs. NSDS Native-API of GAE</td>
<td>104</td>
</tr>
</tbody>
</table>
# List of Tables

2.1 Google Cloud Storage (GCS) APIs ............................... 13  
2.2 Azure Blob Storage APIs ................................... 14  
2.3 Google Cloud Datastore APIs .................................. 15  
2.4 Azure Table Storage APIs ..................................... 16  
2.5 The Generic Methods of COAPS API ............................. 18  
3.1 Research Efforts for Vendor Lock-in Problem ...................... 40  
4.1 Appengine-web XML file ................................. 44  
4.2 Comparison between the supported PaaS platforms by COAPS API . 48  
4.3 Generic CBSS API vs. Specific CBSS API ...................... 53  
4.4 Google Cloud Datastore (GCD) vs. Azure Table Storage ................ 57  
6.1 The required Utilities API of CBSS with its Annotations .......... 88  
6.2 The required Utilities API of NSDS with its Annotations .......... 89  
6.3 Example for Adding Semantic Annotations to some PaaS specific-APIs 90  
6.4 Semantic annotations of the CBSS generic-API .................. 91  
6.5 Semantic annotations of the NSDS generic-API .................. 92  
6.6 The size of the static utilities API vs. the size of the generated adapters 105
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCORDS</td>
<td>Advanced Capabilities for CompatibleOne Resource Description System</td>
</tr>
<tr>
<td>ACL</td>
<td>Access Control List</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BLOB</td>
<td>Binary Large OBjects</td>
</tr>
<tr>
<td>CORDS</td>
<td>CompatibleOne Resource Description System</td>
</tr>
<tr>
<td>CAPEX</td>
<td>CAPital EXpenses</td>
</tr>
<tr>
<td>CBSS</td>
<td>Cloud Blob Storage Service</td>
</tr>
<tr>
<td>CF</td>
<td>Cloud Foundry</td>
</tr>
<tr>
<td>COAPS</td>
<td>Compatible One Application and Platform Service</td>
</tr>
<tr>
<td>CDMI</td>
<td>Cloud Data Management Interface</td>
</tr>
<tr>
<td>DAML</td>
<td>DARPA Agent Markup Language</td>
</tr>
<tr>
<td>GAE</td>
<td>Google App Engine</td>
</tr>
<tr>
<td>GCD</td>
<td>Google Cloud Datastore</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JSP</td>
<td>JavaServer Pages</td>
</tr>
<tr>
<td>JAR</td>
<td>Java ARchive</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>NoSQL</td>
<td>Not only Structured Query Language</td>
</tr>
<tr>
<td>OASIS</td>
<td>Organization for the Advancement of Structured Information Standards</td>
</tr>
<tr>
<td>OCCI</td>
<td>Open Cloud Computing Interface</td>
</tr>
<tr>
<td>OPEX</td>
<td>OPerating EXpenses</td>
</tr>
<tr>
<td>OS</td>
<td>OpenShift</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform as a Service</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>PADM</td>
<td>PaaS Application Description Model</td>
</tr>
<tr>
<td>PBS</td>
<td>Platform Basic Services</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>RDFS</td>
<td>Resource Description Framework Schema</td>
</tr>
<tr>
<td>S3</td>
<td>Simple Storage Service</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>SAVG</td>
<td>Semi-automatic Adapter Validation Generation</td>
</tr>
<tr>
<td>SCOP</td>
<td>Source Code Ontology Population</td>
</tr>
<tr>
<td>SCRO</td>
<td>Source Code Representation Ontology</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SPARQL</td>
<td>Simple Protocol And RDF Query Language</td>
</tr>
<tr>
<td>STAGER</td>
<td>SemanTic-based GenERation of Generic-API Adapters</td>
</tr>
<tr>
<td>Std-PaaS APIs</td>
<td>Standard Platform as a Service Application Programming Interface</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WAR</td>
<td>Web Application Archive</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The National Institute of Standards and Technology (NIST) has defined cloud computing as [1]:

“A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction”.

Therefore, the motivation behind the increasing adoption of cloud computing is the ease of usage whereby everything is provided as a service. Cloud computing provides a pool of virtualized resources that are shared by end users and can be accessed on demand. Moreover, cloud computing can be scaled up and down based on the user needs [2–4]. On the other hand, the cloud supports computing as a utility that is similar to the currently available utilities (e.g., electricity, water, and gas) [5]. Therefore, users can pay only for their actual usage excluding the cost of installation and maintenance of cloud infrastructure. Cloud computing provides three main service models, namely; Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [2, 3]. These service models will be described later in Chapter 2.

Although cloud computing has many advantages, it presents some challenges. These challenges include privacy, security, reliability, availability, scalability, portability, interoperability, cooling and energy consumption [4, 6, 7]. Security is considered the most challenge in cloud computing, because the users’ data and applications are publicly

1
available on clouds. In addition, cloud data centers require huge amount of energy and cooling, so, the energy consumption is considered a challenging issue. On the other hand, the lack of standard or common API among heterogeneous clouds makes the users locked in a specific cloud (i.e., vendor lock-in) which leads to interoperability and portability problems. The vendor lock-in problem refers to the inability for migrating virtual machines, data, or applications among different cloud. The work in this thesis focuses on the PaaS vendor lock-in problem.

1.1 Motivation

PaaS (also called Cloudware [8]) is one of the main service models that are provided by cloud computing. Lawton [9] said that “PaaS creates an online virtual environment for application development”. So, it provides the cloud developers with online Integrated Development Environment (IDE) to develop, test, deploy, and manage their applications remotely through PaaS clouds. Cloud developers can implement their applications by interacting with the PaaS platforms through their Web browsers or through an IDE, such as Eclipse [10].

Therefore, the adoption of PaaS clouds was increased due to several reasons that can be summarized as follows [4, 10, 11]:

- PaaS platforms remove the burden of the installation and maintenance of operating system, IDE, and data storage from cloud developers to PaaS providers.
- They increase the productivity of cloud developers by allowing them to concentrate on implementing their applications, then deploying and managing them on PaaS clouds.
- Cloud developers can pay only for their actual usage without purchasing software with licences.
- PaaS platforms are considered a good starting point for small companies without requiring up-front investment.
- PaaS platforms provide multi-tenant application development which allows cloud developers to cooperate with each other and share applications source code.
Each PaaS platform provides cloud developers with two categories of APIs, namely: implementation API and management API. A PaaS implementation API, also called service-based API, provides cloud developers with access and control of the Platform Basic Services (PBS)\(^1\), such as network, message queueing, and data storage, which they typically use to develop cloud applications; whereas the management API, also called deployment API, helps developers to deploy and manage (such as scaling, billing, and logging) their cloud applications [13]. According to the work in this thesis, a generic deployment API has been extended and a generic implementation API has been proposed to overcome the vendor lock-in problem.

### 1.2 Problem Definition

Although the PaaS cloud has many advantages, it suffers from heterogeneity problem that is coming from two main shortcomings:

1. **Lack of standards**: since the cloud is in its early stage, there is no standards or common API for cloud computing, specially for different PaaS platforms [14].

2. **PaaS proprietary API**: each PaaS provider has a set of specific and heterogeneous APIs with different programming languages for each platform basic service.

The PaaS heterogeneity problem makes cloud developers to be locked in a specific PaaS platform and they cannot easily migrate their applications or data to another PaaS platform (i.e., vendor lock-in). The vendor lock-in problem makes cloud developers worried about adopting PaaS clouds. This problem distracts cloud developers because of many issues, such as PaaS providers can increase the resources price, remove security polices, or change service level by increasing the resources downtime [15, 16]. Although, this problem may lead to violation in the Service Level Agreement (SLA), it is hard for cloud developers to migrate their applications to another PaaS platform because of the following two issues [17]:

- The lock-in may occur on **application level** and this may require re-factoring the applications from the beginning.

\(^1\)PBS is a software that provides a specific functionality in order to develop service-based cloud applications [12].
Chapter 1. Introduction

• The lock-in may occur on data level where the data cannot be transferred between different PaaS providers because each cloud provider has its own data format.

The vendor lock-in problem has a tight relation with data or application portability and interoperability in PaaS platforms. So, it is required to differentiate between PaaS portability and interoperability. The Cloud Computing Use Case discussion group [18] has defined the portability as “the ability to run components or systems written for one environment in another environment” and the interoperability as “the ability to write code that works with more than one cloud provider simultaneously, regardless of the differences between the providers”. In addition, Petcu [19] has specified a set of synonyms for cloud interoperability, such as “avoid vendor lock-in”, “develop your application one, deploy anywhere”, etc. Based on these definitions, we can summarize that PaaS portability refers to the ability to move data or applications among different PaaS platforms with no or little modification; while PaaS interoperability refers to the ability of developing generic applications to be deployed on different PaaS platforms.

Although Kolb and Wirtz [20] claimed that “portability between PaaS is a difficult task”, as well as, it is considered “the second most important obstacle hindering increasing cloud adoption”, many techniques have been proposed to overcome the vendor lock-in problem and allow application portability and interoperability [11, 15, 20–23]. One of these techniques is to implement applications using generic APIs and develop an adapter for each specific-PaaS platform. These generic APIs provide cloud developers with a set of generic methods, such that each adapter should map the generic methods into their corresponding specific-API calls. However, updating a specific API (e.g., renaming methods or changing input parameters) may require re-factoring or even re-implementation of its corresponding adapter; otherwise software applications that use the adapter will not run correctly with the new API. According to the work in this thesis, the term API dynamic adaptation is used to name this problem.

Therefore, the work in this thesis overcomes the vendor lock-in problem and supports application portability among heterogeneous PaaS platforms. It proposes a semantic-based framework, called STAGER, which overcomes the API dynamic adaptation problem by semi-automatically generating adapters for generic APIs.
1.3 Thesis Contributions

According to the work in this thesis, the main achievements are:

- Proposes STAGER (Semantic-Based Generation of Generic-API Adapters for Portable Cloud Applications) as a framework that overcomes the API dynamic adaptation problem by semi-automatically generating the adapters for generic APIs [24].

- Proposes service-based generic-APIs (called Std-PaaS APIs) that will be used by STAGER to generate their adapters. Std-PaaS APIs include a set of generic APIs for each platform basic service (e.g., database, blob storage, NoSQL datastore, messaging, email, authentication, and authorization). Currently, Std-PaaS APIs provide generic APIs for two main services; Cloud Blob Storage Service (CBSS) and NoSQL Datastore Service (NSDS) [13]. However, it is designed in an extendable way to support other services.

- Extends a generic deployment-API for PaaS platforms (called COAPS [21]). The COAPS API is extended to allow developers to deploy their applications on GAE, besides CF and OS [25, 26].

Figure 1.1 depicts the relations between our contributions that will be clarified as follows:

1. The Std-PaaS APIs are entered, as an input, to STAGER framework.

2. STAGER generates the specific adapters (i.e., output from STAGER) of the given generic APIs for target PaaS platforms.

3. The cloud developer uses the Std-PaaS APIs to implement generic JAVA applications.

4. The cloud developer configures the implemented application to use a specific adapter.

5. The cloud developer uses the COAPS API to deploy his/her generic application on multiple heterogeneous PaaS platforms.
1.4 Thesis Outline

The thesis proceeds as follows. Chapter 2 provides a general background over cloud computing with more elaboration of its service and deployment models. In addition, it provides background over the main technologies that are used in this thesis, such as PaaS deployment API (e.g., COAPS API), PaaS implementation API (e.g., blob storage APIs and NoSQL datastore APIs), ontology, and semantic modeling.

Chapter 3 provides literature review over some of the available solutions for the vendor lock-in and cloud portability problems. Many efforts are done to address these problems by building standards or open source to allow the cloud portability. Therefore, this chapter explores the state of the art for the currently available standardization efforts, research projects, and generic APIs to solve the vendor lock-in problem. Finally, this chapter ends with a discussion that specifies the limitations of the current related work.

Chapter 4 presents the proposed generic PaaS-APIs, which include deployment API and implementation API. In the generic deployment API, we have extended the COAPS API to support GAE, besides OS and CF. In the generic implementation API, we have proposed Std-PaaS APIs, which can include generic APIs for a set of platform basic services. Currently, Std-PaaS APIs include two main services: cloud blob storage and NoSQL datastore.
In this chapter, we have provided an introduction about cloud computing, its service models, and its main challenges. Moreover, we have presented our motivation that focuses on PaaS service model. In addition, we have elaborated the problem definition and the main contributions. Finally, the thesis structure has been presented.
Chapter 2

Background

In this chapter, a general overview about most of the used concepts in the thesis are presented. More specially, Section 2.1 provides an introduction about the main cloud computing service and deployment models. Section 2.2 elaborates the PaaS cloud model and how it can be used to provision a cloud application. Subsequently, this Section presents the heterogeneous APIs of platform basic services, such as blob storage and NoSQL datastore for GAE and Azure PaaS platforms. These platform basic services will be used in our case studies. Section 2.3 presents an introduction about ontology and semantic modeling using SPARQL query language that will be used to generate the source codes of generic-API adapters. Finally, Section 2.4 presents the JAVA reflection mechanism, which will be used to parse the specific APIs of platform basic services. Furthermore, it provides quick notes for API design that will be used for designing service-based generic-API.

2.1 Cloud Computing

Cloud Computing is not a new concept rather it is a new and attractive model for computing, which is based mainly on the virtualization technology. It’s main objectives are providing any Information Technology (IT) resources, such as storage, network, and processing power, as a service across the internet and allowing users to pay for their actual usage by applying pay-as-you-go billing model. By this way, cloud computing
allows users to use their Operating Expenses (OPEX) rather than their Capital Expenses (CAPEX) [2, 27]. Figure 2.1 illustrates the cloud computing architecture that is composed of three service models and four deployment models. These models will be elaborated in the following Sub-section.

![Cloud Computing Architecture](image)

**Figure 2.1: Cloud Computing Architecture [28]**

### 2.1.1 Cloud Service Models

Three main service models are provided by cloud computing, namely; Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). Users need only to interact with a certain service model based on their needs. The main features of these service models are [2, 3]:

**SaaS**, which is the highest service model, provides customers with stand-alone applications whereby they can acquire these applications on-demand and only pay-per-use. Google Docs, Gmail, Dropbox, and Facebook are considered examples of SaaS.

**PaaS**, which is the middle service model, provides cloud developers with a specific platform to allow them to implement, build, deploy, and manage their applications remotely over clouds. A PaaS platform consists of operating systems and application frameworks. Google App Engine (GAE)\(^1\) and Microsoft Windows Azure\(^2\) are examples of commercial PaaS platforms. Other examples of open source PaaS solutions are CloudFoundry\(^3\),

\(^1\)https://appengine.google.com/
\(^2\)https://azure.microsoft.com/
\(^3\)https://pivotal.io/platform/
Chapter 2. Background

Heroku\textsuperscript{4}, and OpenShift\textsuperscript{5}. The PaaS service model is the main focus in this research and it will be elaborated deeply in Section 2.2.

\textbf{IaaS}, which is the lowest service model, provides system managers with virtual machines. These virtual machines contain storage, CPU, and memory. IaaS service model uses the virtualization technology to partition the physical computing resources among virtual machines. Examples of IaaS are Amazon EC2\textsuperscript{6} and OpenStack\textsuperscript{7}.

2.1.2 Cloud Deployment Models

There are four deployment models for cloud computing, namely; private cloud, public cloud, hybrid cloud, and community cloud \cite{2}. In the private cloud, the cloud resources are specified for only one organization. In the public cloud, the cloud resources can be used by anyone over the internet. The hybrid cloud is a combination between private and public clouds. It allows applications to run locally then burst (i.e., deploy) to cloud to meet peak demands. Therefore, the hybrid cloud is called “cloud bursting” \cite{3}. In the community cloud, the cloud resources are shared by a group of organizations.

2.2 PaaS Service Model

The PaaS service model is one of the main service models that are provided by cloud computing. PaaS model has been defined by NIST as \cite{1}:

\begin{quote}
\textit{The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.}
\end{quote}

Therefore, the motivation behind increasing the adoption of PaaS service model is to simplify the usage where cloud developers only need a web browser to access PaaS clouds

\footnotesize{\textsuperscript{4}https://www.heroku.com/}
\footnotesize{\textsuperscript{5}https://www.openshift.com/}
\footnotesize{\textsuperscript{6}https://aws.amazon.com/ec2/}
\footnotesize{\textsuperscript{7}https://www.openstack.org/}
and to deploy and manage their applications. On the other hand, PaaS clouds shift the installation and maintenance of software to PaaS providers. So, cloud developers can concentrate on the application development process and this will lead to increase their productivity. For more details about the available PaaS platforms with a comparison among their features, see [29, 30].

2.2.1 PaaS Application Provisioning

Three phases are required to provision cloud applications; development, deployment, and management [31]. The development phase means implementing cloud applications and testing them. The deployment phase means uploading cloud applications to a specific PaaS platform to be ready for execution. The management phase means starting the deployed cloud applications and executing management operations, such as billing, logging, scaling, and monitoring. Based on these phases, each PaaS platform provides cloud developers with two different and specific APIs, one for developing their applications (i.e., PaaS implementation API) and the other for deploying and managing their applications (i.e., PaaS management API) [13].

A PaaS implementation API (i.e., service-based API) provides cloud developers with API to access and control the platform basic services, such as network, message queueing, and data storage. This implementation API is typically used to develop cloud applications. For example, GAE provides a set of specific implementation APIs for storing large files on cloud, sending emails, and manipulating images [32]. Since most of the software applications require database or file storage, we have explored cloud blob storage and NoSQL datastore as examples of PaaS Implementation APIs in Sub-section 2.2.2 and Sub-section 2.2.3 respectively.

A PaaS management API (i.e., deployment API) helps cloud developers to deploy and manage their applications remotely over PaaS clouds. For example, a specific PaaS management API includes APIs for deploying, logging, billing, scaling, and monitoring applications. An example for PaaS deployment API is presented in Sub-section 2.2.4.
2.2.2 Cloud Blob Storage Service (CBSS)

The CBSS is considered an example of a PaaS implementation API. It helps cloud developers to store large files remotely in the cloud. These large files are called BLOBs (Binary Large OBjects) or objects and they can be of any type and up to terabytes in size. These blobs are stored inside containers (or buckets) [33]. Each PaaS provider supports developers with a proprietary API for blob storage; for example, GAE provides Google Cloud Storage (GCS) API [34] and Azure provides the Azure Blob Storage API [35] that will be elaborated in the following Sub-sections.

2.2.2.1 Google Cloud Storage (GCS) APIs

GCS APIs allow users to store and retrieve large files at any time. These files are called objects and they must be stored in buckets that must belong to a project. Buckets cannot be nested and can be used to organize files and control their access. Also, buckets cannot be shared between projects. A project can contain several buckets [34]. Table 2.1 provides a summary of GCS APIs and the role of each API. These APIs are implemented in JAVA and are elaborated as follows:

- **GcsService** is an interface that used to create and access files in GCS.
- **GcsFilename** is a class that creates a GCSFile with a given bucket and a file name.
- **GcsFileOptions** is a class that is used to store options (e.g., Mime Type and Access Control List (ACL)) for creating Google storage files.
- **GcsOutputChannel** is an interface that provides output channel to write data to GCS.

2.2.2.2 Azure Blob Storage APIs

Azure Blob Storage APIs allow users to store and retrieve large files up to a terabyte (TB). These files are called blobs, which must be stored in containers that must belong to a storage account. Azure supports two types of blobs; block blob and page blob.
Table 2.1: Google Cloud Storage (GCS) APIs

<table>
<thead>
<tr>
<th>A specific GAE API</th>
<th>Its role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create instance of GcsService</td>
<td>This instance is used to create a blob.</td>
</tr>
<tr>
<td>Create instance of GcsFilename</td>
<td>This instance is used by GCSService instance to create a file given a bucket name and a file name.</td>
</tr>
<tr>
<td>Create instance of GcsFileOptions</td>
<td>It is used to set file permissions and content type.</td>
</tr>
<tr>
<td>Create instance of GcsOutputChannel</td>
<td>It is used to open an output channel to a file to allow writing on it. It is done using the created GcsFileName instance and the created GcsFileOptions instance. It will create the file if it does not exist; otherwise, it will replace it with the new one.</td>
</tr>
<tr>
<td>Create instance of OutputStream</td>
<td>This output stream is used to write data to a file. This is done using Channels.newOutputStream Java method which takes the created GcsOutputChannel instance as a given input.</td>
</tr>
</tbody>
</table>

The block blob has a maximum size of up to 200 GB; whereas the page blob has a maximum size of up to one TB. A storage account can contain several containers [35]. Table 2.2 provides a summary of Azure blob storage APIs and the role of each API. Azure blob storage provides several main APIs that are specified as the follows:

- **CloudStorageAccount** is a class that creates a Windows Azure Storage account based on an account name and a key.

- **CloudBlobClient** is a class that creates a storage client, which can be used to execute requests against the Azure blob service.

- **CloudBlobContainer** is a class that creates a container to store a blob.

- **BlobContainerPermissions** is a class that updates a container permission.

- **CloudBlockBlob** is a class that represents a blob, which can be uploaded as a set of blocks.

### 2.2.3 NoSQL Datastore Service

The NoSQL datastore service is considered another example of a PaaS implementation API. It is a new technology for non-relational database. NoSQL stands for Not only Structured Query Language. Its main objective is to store huge amount of data in a
Chapter 2. Background

Table 2.2: Azure Blob Storage APIs

<table>
<thead>
<tr>
<th>A specific Azure API</th>
<th>Its role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create instance of CloudStorageAccount</td>
<td>This is done using CloudStorageAccount.parse method which takes, as a given input, a connection string that is containing an account name and a key.</td>
</tr>
<tr>
<td>Create instance of CloudBlobClient</td>
<td>This is done by calling the createCloudBlobClient method of the created CloudStorageAccount.</td>
</tr>
<tr>
<td>Create instance of CloudBlobContainer</td>
<td>This is done by calling the getContainerReference method of the created CloudBlobClient.</td>
</tr>
<tr>
<td>Create instance of BlobContainerPermissions</td>
<td>This is used to change a container permission to be publically accessible. This step is optional, whereby the container has a default permission “private access”.</td>
</tr>
<tr>
<td>Create an instance of CloudBlockBlob</td>
<td>This is done by calling getBlockBlobReference method of the created CloudBlobContainer. This instance is used to upload a blob as a set of blocks.</td>
</tr>
<tr>
<td>Call the CloudBlockBlob.upload method</td>
<td>The upload method takes the FileInputStream of a given file and its length. It is used to upload data from a given input stream to the created CloudBlockBlob.</td>
</tr>
</tbody>
</table>

scalable way [36]. Therefore, it is convenient for cloud computing applications, such as facebook and twitter.

The NoSQL datastore stores the data as objects (called entities) such that each entity has a unique key and has a set of properties which are key-value pairs. In addition, each entity should belong to a table (or kind). An entity is similar to a row in a relational database. The main feature in the NoSQL datastore is that it is schemaless (i.e, it is not required that the entities of the same kind should have the same set of properties). Each PaaS provider supports developers with a proprietary API for NoSQL datastore service. For example, GAE provides Google Cloud Datastore (GCD) API [37] and Azure provides the Azure Table Storage API [38] which are elaborated in the following Sub-sections.

2.2.3.1 GAE NoSQL datastore APIs

GAE provides Google Cloud Datastore (GCD) APIs for NoSQL datastore service. These APIs store data as objects, where the category of an object is called kind. The kind is similar to a table in relational database. GAE stores the data of an object in an entity. Each entity has a unique key and a set of properties [37]. Table 2.3 provides a summary of GCD APIs and the role of each API. These APIs are implemented in JAVA and elaborated as follows:
Chapter 2. Background

- **DatastoreService** is an interface that is used to store, retrieve, or delete entities in GCD.

- **Entity** is a class that represents the main unit of data storage.

- **KeyFactory** is a class that is used to generate a unique key based on a given kind and a key string.

- **Key** is a class that represents the primary key of an entity.

### Table 2.3: Google Cloud Datastore APIs

<table>
<thead>
<tr>
<th>A specific GAE API</th>
<th>Its role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create instance of <strong>DatastoreService</strong></td>
<td>This instance is used to store, retrieve, or delete an entity.</td>
</tr>
<tr>
<td>Create instance of <strong>Entity</strong>.</td>
<td>This instance is used to create an entity given a kind and a unique key.</td>
</tr>
<tr>
<td>Create instance of <strong>KeyFactory</strong>.</td>
<td>It is used to create key based on a given kind and a unique key string</td>
</tr>
<tr>
<td>Create instance of <strong>Key</strong></td>
<td>It is used to store the primary key of an entity</td>
</tr>
</tbody>
</table>

#### 2.2.3.2 Azure NoSQL datastore APIs

Azure provides Azure table storage APIs for NoSQL datastore service. These APIs store large amounts of non-relational data as entities. Similar to Azure blob storage APIs, Azure table storage APIs require a storage account with an account name and a key. Each storage account can include multiple tables and each table can include a set of entities. Each entity includes a set of properties as key-value pairs. The maximum number of properties per entity can reach to 252 property and the size of an entity can reach to one MB. An entity is identified by a partition key and a row key, which is a unique identifier for an entity. The partition key has been used to quickly retrieve entities [38].

Table 2.4 provides a summary of Azure table storage APIs and the role of each API. These APIs are implemented in JAVA and are elaborated as follows [38]:
• *CloudTableClient* is a class that creates a storage client, which can be used to execute requests against the Azure table storage service.

• *CloudTable* is a class that represents a non-relational table.

• *TableOperation* is a class that represents operations for table entities, such as insert, update, retrieve, and delete entities.

<table>
<thead>
<tr>
<th>A specific Azure API</th>
<th>Its role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create instance of <em>CloudTableClient</em>.</td>
<td>This is done by calling the <em>CloudStorageAccount.createCloudTableClient</em> method.</td>
</tr>
<tr>
<td>Create instance of <em>CloudTable</em>.</td>
<td>This is done by calling the <em>CloudTableClient.getTableReference</em> method. This table can be created if it is not exist using <em>createIfNotExists</em> method. Also, it can be used to execute operations on a table using the <em>execute</em> method.</td>
</tr>
<tr>
<td>Create instance of <em>TableOperation</em></td>
<td>This is used to create an operation to be executed on table entities. This is done using some methods such as <em>insertOrReplace</em>, <em>delete</em>, or <em>retrieve</em>.</td>
</tr>
</tbody>
</table>

### 2.2.4 Compatible One Application Platform Service (COAPS) API

This Sub-section provides an introduction about COAPS API including its architecture and its PaaS Application Description Model (PADM). In addition, it presents the deployment scenario of a PaaS application using COAPS API.

#### 2.2.4.1 COAPS API Overview

COAPS API is a generic PaaS Deployment API with a specific adapter for each PaaS platform [21, 39]. It is represented as RESTful JAVA open source under Apache 2.0 license [40]. COAPS API deploys and manages applications on heterogeneous PaaS platforms. Thereby, there is no need for developers to learn the different deployment APIs of all targeted heterogeneous PaaS platforms. They need only to learn COAPS API to deploy their applications on any PaaS platform as long as that PaaS platform is supported by COAPS API.
COAPS API has been tested and used in two international research projects, EASICLOUDS\textsuperscript{8} and CompatibleOne\textsuperscript{9} \cite{[21]}. EASI-CLOUDS provides an open-source cloud infrastructure with interoperability, portability, reliability, and security. CompatibleOne is an open source cloud broker \cite{[41]}. It provides middleware layer to allow heterogeneous cloud resources (e.g., Infrastructure, Platform, Application) to interoperate with each other. It describes cloud resources using an object-oriented model called CompatibleOne Resource Description System (CORDS). It provides an execution platform called Advanced Capabilities for CORDS (ACCORDS). The authors of CompatibleOne project use COAPS API as a middleware layer between ACCORDS platform and PaaS providers.

The COAPS API architecture is illustrated in Figure 2.2. It clarifies the two PaaS platforms that are supported by COAPS, namely; CloudFoundry (CF) and Openshift (OS), with their specific implementations (i.e., specific adapters). In addition, it clarifies that COAPS API can support other PaaS solutions. Further more, Figure 2.2 specifies the generic methods of COAPS API that can be used by cloud developers. A brief description about these generic methods is provided in Table 2.5.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{COAPS_API_Architecture.png}
\caption{COAPS API Architecture \cite{[39]}}
\end{figure}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Generic Method & Description \\
\hline
Create the environment & Create the environment \\
\hline
Update the environment & Update the environment \\
\hline
Create the application & Create the application \\
\hline
Update the application & Update the application \\
\hline
Deploy the application & Deploy the application \\
\hline
Start the application & Start the application \\
\hline
Stop the application & Stop the application \\
\hline
Un-deploy the application & Un-deploy the application \\
\hline
Delete the application & Delete the application \\
\hline
\end{tabular}
\caption{Generic Methods of COAPS API}
\end{table}

\textsuperscript{8}https://itea3.org/project/easi-clouds.html
\textsuperscript{9}http://www.sucreproject.eu/content/compatibleone-open-cloud-broker
Table 2.5: The Generic Methods of COAPS API

<table>
<thead>
<tr>
<th>Operation</th>
<th>REST Method</th>
<th>REST URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Environment</td>
<td>POST</td>
<td>/environment</td>
<td>Creates a new environment given the env manifest. It returns an XML</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>environment descriptor with a unique envID.</td>
</tr>
<tr>
<td>Describe Environment</td>
<td>GET</td>
<td>/environment/{envId}</td>
<td>Returns an XML environment descriptor for the given appID.</td>
</tr>
</tbody>
</table>

**Application Management Methods**

<table>
<thead>
<tr>
<th>Operation</th>
<th>REST Method</th>
<th>REST URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Application</td>
<td>POST</td>
<td>/app</td>
<td>Creates a new application given the app manifest. It returns an XML</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>application descriptor with a unique appId.</td>
</tr>
<tr>
<td>Deploy Application</td>
<td>POST</td>
<td>/app/{appId}/action/</td>
<td>Deploy an app given its appId on an env given its envID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>deploy/env/{envId}</td>
<td></td>
</tr>
<tr>
<td>Start Application</td>
<td>POST</td>
<td>/app/{appId}/start</td>
<td>Start an app given its appId</td>
</tr>
<tr>
<td>Stop Application</td>
<td>POST</td>
<td>/app/{appId}/stop</td>
<td>Stop an app given its appId</td>
</tr>
<tr>
<td>Destroy Application</td>
<td>DELETE</td>
<td>/app/{appId}</td>
<td>Delete an app given its appId</td>
</tr>
<tr>
<td>Describe Application</td>
<td>GET</td>
<td>/app/{appId}</td>
<td>Returns an XML application descriptor for the given appID</td>
</tr>
</tbody>
</table>

COAPS API provides two main resources: **application** and **environment**. The Application resource represents any software program that can be deployed on a PaaS platform. The environment resource refers to the required configurations that are needed by an application to be deployed, such as runtime (e.g., PHP, Ruby, or Java), framework (e.g., tomcat or spring), and services (e.g., database) [21, 39].

In addition, COAPS API provides two generic interfaces, **RESTApplicationManager** and **RESTEnvironmentManager**, to handle the operations of the application and environment resources respectively. The RESTApplicationManager interface provides
prototypes to manage the application (e.g., create, deploy, start, stop, and delete application). The RESTEnvironmentManager interface provides prototypes to manage the environment (e.g., create and delete environment). These prototypes of both RESTApplicationManager and RESTEnvironmentManager are presented in Table 2.5 and it can be re-implemented by each specific PaaS platform to be used as an adapter for COAPS API.

2.2.4.2 PaaS Application Description Model (PADM) of COAPS

COAPS API requires PaaS applications to be described through a PaaS Application Description Model (PADM), which is based on Open Cloud Computing Interface (OCCI) standards. PADM represents an application manifest that is used to specify the PaaS application properties, its required software components (such as runtime, framework, and services), and its source archive independently of any PaaS platform. A UML diagram for the main components that constitute PADM of COAPS API is illustrated in Figure 2.3. These components are elaborated as follows [26]:

Figure 2.3: UML diagram for PADM of COAPS API

10http://occi-wg.org/
• **PaaS application**: an application is identified by a unique name and is tied to an environment. Each application version defines a set of instances, which can be run after the application is deployed, and specifies the application deployable (i.e., source archive, its content-type, location, and multi-tenancy supportability). The content-type of an application deployable may be **artifact** or **file**. The artifact value refers to the application’s source packaging format, which may be bundled (e.g., WAR, JAR, EAR, ZIP) or extracted as a local folder. The file value refers to a configuration file to upload and execute in the target folder once the application is deployed (e.g., an XML file or a script). Deployable entities can be a set of artifacts and configuration files [21, 39]. Currently, COAPS supports only the artifacts format (personal communication with Yangui, June 12, 2013).

• **PaaS environment**: a PaaS environment is identified by a unique name and should define an environment template. An environment template specifies the memory that is needed by deployed application instances and defines a set of environment nodes and their types (e.g., container such as tomcat or database such as MySQL). These environment nodes represent the software components that are required by a deployed application [21, 39]. For example, when developers need to scale up or down the memory of their application instances, a new environment should be created (or updating an existed one) with the new memory size and then deploy the application again. A sample of a PaaS application manifest is depicted in Figure 2.4.

### 2.2.4.3 Deployment Scenario using COAPS API

After a PaaS application manifest is created for an application, COAPS API is used to deploy that application. The required scenario to deploy an application using COAPS API is as follows [26]:

• A developer selects one of the supported PaaS platforms (e.g., CF or OS) that will be used to host an application.

• Select “Create” environment method, which takes a PaaS environment manifest as input. The create environment method returns an XML file of the newly created
environment with a unique envID. This created environment will be used later to deploy applications on the selected PaaS platform.

- Select “Create” application method, which takes a PaaS application manifest as input and returns an XML file of the newly created application with a unique appID.

- Select “Deploy” application method, which takes an appID and an envID as inputs. It calls the API of the targeted PaaS platform to deploy the given application.

- Select “Start” application method, which takes an appID as an input. It calls the API of the targeted PaaS platform to start the given application; after the start method is executed, the application will be ready to be executed by end users.

```
<paas_application_manifest name="ServletSampleApplicationManifest">
  <description>This manifest describes a Sample Servlet.</description>
  <paas_application name="ServletSampleApplication" environment="JavaWebEnv">
    <description>Sample Servlet application description.</description>
    <paas_application_version name="version1.0" label="1.0">
      <paas_application_deployable name="SampleServlet.war" content_type="artifact"
        location="C:\deployable" multitenancy_level="SharedInstance"/>
      <paas_application_version_instance name="Instance1" initial_state="1"
        default_instance="true"/>
    </paas_application_version>
  </paas_application>
  <paas_environment name="JavaWebEnv" template="TomcatEnvTemp">
    <paas_environment_template name="TomcatEnvTemp" memory="65">
      <description>Tomcat Server Environment Template</description>
      <paas_environment_node content_type="container" name="tomcat" version="" provider="CF"/>
    </paas_environment_template>
  </paas_environment>
</paas_application_manifest>
```

**Figure 2.4: PaaS Application Manifest Sample [26]**
2.3 Semantic Ontology

Gruber [42] has defined an ontology as “explicit specification of a conceptualization”. By this way, it is used to describe the knowledge of a specific domain as a set of concepts and a set of relations between these concepts. The concepts are described as classes and the relations are described as properties of these classes [43]. Therefore, the main objectives of a semantic ontology are [44, 45]:

- Sharing knowledge by defining common vocabularies of a specific domain to allow domain experts to share them.
- Overcoming heterogeneity by hiding syntactical differences of different PaaS APIs [46].
- Allowing reusability by designing the ontology in a modular base, through inheritance, to increase its reusability. (i.e., the upper ontology describe more general knowledge and the lower ontology describe more specific knowledge).
- Specifying domain assumptions clearly.
- Analyzing a domain knowledge.

One of the main contributions of this thesis is to introduce the semantic ontology to define a set of shared vocabularies among different PaaS APIs. So, the domain of our proposed ontology is heterogeneous APIs of the platform basic services. The proposed ontology is created by representing all Object Oriented concepts that are defined in each PaaS specific-API as ontological instances. The main structure of the proposed ontology is based on Source Code Representation Ontology (SCRO) [47]. The SCRO ontology will be elaborated later in this Section.

2.3.1 Ontology Specification Languages

A semantic ontology can be defined using a set of description languages, such as RDF (Resource Description Framework), RDFS (Resource Description Framework Schema),
DAML (DARPA Agent Markup Language), DAML+OIL, OWL (Web Ontology Language), and OWL-S (OWL-Semantic). A brief overview about RDF and OWL is presented as follows.

**RDF** (Resource Description Framework) is an XML-based language [48]. It describes resources on the web and relations between these resources. The information described by RDF are not designed to be displayed to end users. However, it is designed to be *machine-readable*. This information is structured as triples. Each triple presents a statement and consists of three parts, namely; subject, predicate, and object. Each part in a triple is identified by a unique URI. The main disadvantage of RDF is that it cannot define any domain specific knowledge. Therefore, RDFS (RDF Schema) is used to define domain specific classes and relations [49]. Thus, RDFS is helpful for defining semantic annotations of a specific domain.

**OWL** (Web Ontology Language) specifies and shares ontologies [50]. It is built on top of RDF. However, it is a stronger language than RDF because it provides more features to represent semantics of a specific domain. In addition, the main advantage of OWL is that it can infer new knowledge from an existing ontology. OWL language is composed of three sub-languages, OWL Lite, OWL DL (OWL Description Logic) and OWL Full. **OWL Lite** provides the simplest OWL language. So, it is used for representing simple constraints. **OWL DL** provides support for Description Logic. **OWL Full** provides more expressiveness (i.e., anything can be represented because it has no constraints on expressiveness). An OWL ontology is composed of three main components, namely; individuals (i.e., instances or objects), properties (i.e., slots or relations between instances), and classes [43].

### 2.3.2 SPARQL Query Language

Simple Protocol and RDF Query Language (SPARQL) is used to execute queries against OWL ontologies [45]. It is very helpful in inferring new knowledge from an existing ontology. SPARQL is similar to SQL but it uses RDF triples. For example, List 2.1 clarifies a sample of SPARQL query to retrieve all common services that belong to heterogeneous PaaS providers.
Chapter 2. Background

2.3.3 Source Code Representation Ontology (SCRO)

SCRO is implemented using OWL-DL language [47, 51]. It represents a source code as an ontology. SCRO defines a set of concepts and a set of relationships that are used to semantically describe Object Oriented programs. For example, it defines ClassType concept to represent different Object Oriented classes, such as abstract or static classes, and Method concept to represent different Object Oriented methods, such as static or constructor methods. In addition, it defines two relations to represent the data types of the input(s) and output parameters of an Object Oriented method, namely; hasInputType and hasOutputType respectively. Figure 2.5 illustrates a snapshot of SCRO ontology with its object properties. In this thesis, a specially annotated ontology, called STAGER ontology, has been designed based on SCRO ontology.

2.4 JAVA Reflection Mechanism and API Design

In this Section, a brief introduction about the JAVA reflection mechanism is presented. Followed by a background about API design and the characteristics of a good API design.
2.4.1 JAVA reflection mechanism

JAVA reflection mechanism is used to parse an Object Oriented class or interface to identify its methods and fields [52]. The reflection mechanism can also parse each Object Oriented method in order to identify its type (e.g., static or instance method) and identify its in/out parameters with their data types. In addition, it can identify the exceptions that can be thrown by a method.

In this thesis, the JAVA reflection mechanism is used to parse a set of PaaS specific APIs and a set of generic APIs. These parsed data are inserted later into our STAGER ontology.

2.4.2 API Design

Application Programming Interface (API) is a set of interfaces that are represented as a library or package to be used by developers to help them in building their applications [53, 54]. Each API has a specific purpose, such as API for storage, authentication, or
database. APIs should be bug free. They are considered the main way to divide the systems into components with specified interfaces and this will lead to loosely coupled components. One advantage of loosely coupled components is to allow system scalability. APIs should be written once and read many times by different developers. Therefore, API design is an important process. The characteristics of a good API design are:

- Easy to learn and expand.
- Readable API in order to get readable code.
- Specify the input, output, precondition, and postcondition of each method.
- Consistent API such that it can be remembered.

There are several instructions that can be followed to design a good API, such as identifying the requirements and use cases of an API, outline the API, design the first draft, review the API, and get a feedback based on a user study. For more details about these instructions, see [54].

### 2.5 Summary

In this chapter, a background about the most used concepts in the thesis has been introduced. More especially, we have presented the main cloud computing service models with more details about the PaaS cloud model and how the PaaS model can be used to provision applications on the cloud. In addition, some examples of the heterogeneous APIs of platform basic services have been elaborated. Moreover, we have provided an introduction about ontology and semantic modeling using SPARQL query language. Finally, we have presented a background about the JAVA reflection mechanism and brief notes for API design.
Chapter 3

Literature Review

The vendor lock-in problem is considered one of the most important challenges which prevents cloud adoption. Therefore, many research studies have been done to overcome this problem and support cloud portability. Petcu and Vasilakos [55] have categorized these efforts into four types based on the methodology that is used to solve the vendor lock-in problem; model driven engineering (MDE), standards, open/generic APIs, and semantics. On the other hand, Gonidis et al. [56] have identified four main characteristics that may hamper cloud portability as follows:

- **Programming languages and frameworks**: each PaaS platform provides a set of heterogeneous programming languages and frameworks.

- **Data storage**: each PaaS platform provides a set of heterogeneous types of data storage, which can include blob storage, relational databases, NoSQL datastores, and message queues.

- **Platform specific services**: each PaaS platform provides a set of specific services, such as sms, email, and monitoring services, with specific APIs. Cloud developers can use these services through their specific APIs.

- **Platform specific configuration files**: some PaaS platforms require a specific configuration file, which includes information about how to deploy and execute the application. For example, GAE requires `appengine-web.xml` file, which includes the application ID that is used in the deployment process.
In addition, da Silva et al. [57] have categorized the cloud portability into four types: virtual machine (VM) portability, IaaS application portability, PaaS application portability, and data portability. The work in this thesis focuses on applications portability over PaaS clouds. So, we will classify the current related work into three categories; standardization efforts, research projects, and generic APIs. Section 3.1 elaborates the current standardization efforts to overcome the vendor lock-in problem. In addition, the current research projects to overcome this problem are presented in Section 3.2. Section 3.3 elaborates the currently available generic APIs to solve the lock-in problem. Finally, a brief discussion is introduced in Section 3.4.

3.1 Standardization Efforts

In this Section, we will explore a set of standardization efforts that has been addressed to overcome the cloud vendor lock-in problem, such as Cloud Application Management for Platforms (CAMP), Topology and Orchestration Specification for Cloud Applications (TOSCA), Open Cloud Computing Interface (OCCI), Cloud Data Management Interface (CDMI), and Open Virtualization Format (OVF).

CAMP: OASIS has introduced Cloud Application Management for Platforms (CAMP) standard, which provides standard management API to allow deploying and managing (i.e., starting, stopping, monitoring, billing, etc.) applications over different PaaS platforms [58]. CAMP API is a RESTful API that is based on JSON. Therefore, it is independent on PaaS platforms and programming languages. Each application has a set of requirements and each PaaS platform has a set of capabilities. So, CAMP matches the application requirements against the PaaS capabilities and generates deployment plans. Meanwhile, CAMP represents the deployed application as assembly template. Finally, CAMP combines the assemblies and the deployment plans into Platform Deployment Package (PDP), which can be used to migrate an application from a PaaS platform to another. Therefore, CAMP has two main advantages:

- It can be implemented as a plug-in inside the application development environment.
Chapter 3. Literature Review

- It helps cloud developers to port their applications among different PaaS platforms using a standard management API.

However, the main drawbacks of CAMP are: (1) if the ported application is implemented using a PaaS proprietary API (e.g., GAE datastore), then it cannot be ported to another PaaS platform that uses another proprietary API (e.g., Azure datastore) and (2) PDP deployment of CAMP is only supported by a very small number of new PaaS platforms, such as Solum\textsuperscript{1} and Brooklyn\textsuperscript{2}.

**TOSCA:** OASIS has provided Topology and Orchestration Specification for Cloud Applications (TOSCA) standard, which provides an XML-based modeling language to automatically deploy and manage cloud applications [59]. In TOSCA, cloud applications can be specified by a topology and a set of management plans. A topology is used to describe the application components and their relationships, whereas management plans describe the management operations that need to be executed to deploy and manage a cloud application. The management plans are defined using standard workflow languages, such as BPEL\textsuperscript{3} and BPMN\textsuperscript{4}. Finally, TOSCA combines the application topology and management plans into a Cloud Service Archive (CSAR) that is ready to be deployed on a PaaS platform. TOSCA standard is better than CAMP standard because it supports the re-usability of the application components. However, the main drawback of TOSCA is that most of the currently available PaaS platforms could not support the CSAR deployment except a very small number of PaaS platforms, such as OpenTOSCA\textsuperscript{5} ecosystem.

**OCCI:** Open Grid Forum (OGF) has introduced Open Cloud Computing Interface (OCCI) standard, which provides standard RESTful APIs for managing clouds [60]. OCCI is started by creating a remote management API for IaaS model then it is evolved to manage the other two models (PaaS and SaaS). There are many IaaS platforms that are based on OCCI standard, such as jclouds, OpenStack, BigGrid, and OpenNebula. Moreover, COAPS API, which is a generic PaaS API for deploying and managing cloud applications on multiple heterogeneous PaaS platforms, is based on OCCI standard [21].

\textsuperscript{1}https://wiki.openstack.org/wiki/Solum
\textsuperscript{2}https://wiki.apache.org/incubator/BrooklynProposal
\textsuperscript{3}Business Process Execution Language
\textsuperscript{4}Business Process Model and Notation
\textsuperscript{5}http://www.opentosca.org/
Currently, the COAPS API supports two PaaS platforms; Cloud Foundry (CF) and Openshift (OS). In a previous work, we have implemented a new adapter in the COAPS API to deploy cloud applications on GAE, besides CF and OS [25, 26]. The COAPS API is similar to IBM Altocumulus project [61] which provides users with a uniform interface to allow them to deploy their applications on multiple PaaS platforms such as IBM HiPODS, Amazon EC2, and GAE.

**CDMI**: SNIA organization has provided Cloud Data Management Interface (CDMI) as an international standard to manage cloud storage [62]. CDMI is similar to Amazon S3 (Simple Storage Service). However, Amazon S3 is a proprietary solution. CDMI provides a standard interface to help cloud developers to execute CRUD (i.e., create, read, update, and remove) operations on data elements of IaaS cloud storage. Moreover, CDMI provides another standard interface to manage IaaS cloud storage. Examples for these management operations include backup, access control list, manage containers, billing, etc. By this way, CDMI includes two interfaces: data path (which is used to store and retrieve data) and control path (which is used to manage data). In addition, these interfaces are based on REST technology and they provide a set of standard methods for three data elements; blobs, containers, and message queues. However, CDMI did not support table storage and SQL databases. Livenson and Laure [63] have implemented an open source prototype that is based on CDMI. Their prototype, called CDMI-proxy, provides generic APIs with specific adapters to access two heterogeneous cloud storage; blobs and message queues. However, their specific adapters are implemented manually.

**OVF**: DMTF organization has proposed an open standard for virtual machines called Open Virtualization Format\(^6\) (OVF). OVF provides a standard format for the heterogeneous virtual machines (VMs) in order to allow porting them among heterogeneous IaaS platforms in an easy way. On the other hand, porting applications over different IaaS clouds can be done by provisioning a VM on the target IaaS cloud with the target framework (i.e., language and packages), then deploying applications on this VM. In addition, it is possible to build the application inside a container (e.g., Docker\(^7\)) and then migrate the container, which contains the application with its dependencies, to other IaaS clouds [64].

\(^6\)http://www.dmtf.org/standards/ovf
\(^7\)https://www.docker.com
3.2 Research Projects

This Section introduces a set of currently available research projects that have been done to address the cloud vendor lock-in problem, such as Cloud4SOA, PaaSport, mOSAIC, and MODAClouds.

**Cloud4SOA:** Zeginis et al. [11] have presented the Cloud4SOA project, which aims to provide a harmonized API for managing (i.e., deploying, monitoring, migrating) applications across multiple PaaS platforms. Their API is based on semantic and it has a set of specific adapters for some PaaS platforms, such as AWS Beanstalk, Heroku, OS, and CF. However, Cloud4SOA has some overhead because its structure is composed of multiple layers. Furthermore, it requires that the applications are implemented based on SOA.

**PaaSport:** Bassiliades et al. [65] have provided semantically-based cloud-broker which aims to deploy, migrate, and manage cloud applications over heterogeneous PaaS platforms. In addition, PaaSport provides a unified API with specific adapters to support PaaS portability. This unified API is based on the CAMP standard and Cloud4SOA. However, its adapters are manually implemented which will lead to API dynamic adaptation problem. The proposed STAGER framework is similar to the PaaSport in that the PaaSport requires the PaaS providers to semantically annotate their PaaS platforms through an OWL ontology. Moreover, the PaaSport provides an ontology-based recommendation algorithm that selects the most suitable PaaS platform based on the application requirements, which may be functional or non-functional requirements.

**mOSAIC:** Petcu et al. [66] have provided the mOSAIC (Open Source API and platform for Multiple Clouds) project, which supports application portability and interoperability among multiple clouds. mOSAIC provides cloud developers with an open source cloud API with a set of adapters. This API is a generic API, which aims to deploy applications across different IaaS clouds. However, the main drawback of this API is that it was designed based on event driven that is a complex programming style. In addition, this API focuses on deploying applications across IaaS clouds. Currently, mOSAIC proposes adapters for a set of IaaS clouds, such as Amazon EC2, OpenNebula, and Eucalyptus. For more info about the mOSAIC project see [67].
As a part of mOSAIC project, Cretella and Martino [68] have presented an approach to semantically annotate cloud APIs in order to allow application portability. The application portability can be satisfied by understanding the functions of cloud APIs and then mapping these functions to its corresponding APIs of another provider. They have proposed an approach to semantically analyze and annotate cloud APIs in order to understand their functions. Same as our STAGER framework, the authors of mOSAIC have used the reflection mechanism to analyze the different cloud APIs and convert them into an ontology. Furthermore, they have proposed an automatic API alignment technique in order to map a given specific API to a given generic API, which called neutral API. However, the output of this alignment technique must be manually validated. In addition, they did not provide a case study to validate their proposed solutions.

Furthermore, Cretella and Di Martino [46] have proposed a semantic engine, as a component of mOSAIC project, to support cloud application portability. Their semantic engine helps cloud developers to semantically detect the required cloud APIs and resources that are suitable to develop their cloud applications. This semantic engine provides cloud developers with a list of functions related to cloud and a list of patterns related to application design to help them for designing their applications with the required functions. However, this semantic engine is just a prototype and the semantic annotation of cloud APIs has been manually done. Same as our STAGER framework, they have used the Jena library and JAVA programming language to implement their semantic engine.

Moreover, Di Martino et al. [69] have proposed a semantic-based common model that can be used to represent cloud applications through design patterns. Their common model supports cloud portability and interoperability. They have extended an ontology language for design patterns, called ODOL, to include semantic annotation using OWL-S ontology. Cloud developers can use this extended ontology to design their cloud applications independently of any cloud platform. In order to support application portability, they have proposed an automatic mapping methodology to transform a designed cloud pattern of an application into its specific implementation of a target cloud platform.

**MODAClouds** has provided an approach that is based on model driven engineering (MDE), instead of providing standards or generic APIs, to solve the vendor lock-in problem [70, 71]. The main objective of the MDE is to provide *abstraction* and *automation*
The abstraction is satisfied by making the application development to be independent of the target PaaS platform, whereas the automation is satisfied by automatically transforming an abstracted application into a specific application to be deployed on the target PaaS platform. MODAClouds designs and executes cloud applications over heterogeneous clouds (IaaS, PaaS, or SaaS). It provides a modeling language, called MODACloudML, to help developers to describe cloud applications. Moreover, it provides an IDE which helps the developers to design their applications independent on any PaaS platform. This IDE can semi-automatically map the designed application into code to be ready for execution.

da Silva et al. [57] have proposed a MDE approach to solve the PaaS application portability problem. They have created a meta model to represent the concepts of cloud platform in higher level of abstraction. Based on this meta model, they have proposed a domain specific language (DSL) that helps cloud developers to specify their applications in an abstract way, called Platform Independent Model (PIM). Based on their proposed DSL and PIM, they have defined a set of transformations, called Platform Specific Model (PSM), which can be used to generate the code of the abstracted applications to be deployed on different platforms. They have evaluated their approach on two PaaS platform, GAE and Azure.

Ranabahu et al. [72] have proposed an approach based on abstraction driven in order to support cloud application portability. Their approach is very similar to the MDE, where it helps cloud developers to specify their applications through an abstract language (i.e., DSL). Then, their abstracted applications can be converted, through a code generation engine, into specific applications to be deployed on target PaaS platforms. However, to deploy an abstracted application on a set of PaaS platforms, it is required to generate a specific application for each one of the target PaaS platforms, which may be considered time consuming. Moreover, the abstraction cannot present all specific features that are existing in the target PaaS platform. By this way, a generated application needs to be manually customized to support specialized features.
3.3 Generic APIs

In this Section, we will explore a set of currently available generic APIs that overcomes the cloud vendor lock-in problem and help cloud developers to implement cloud applications independently over specific PaaS platforms. These generic APIs will be introduced in chronological order as follows.

Maximilien et al. [73] have proposed a prototype of a middleware that is independent on cloud platforms in order to overcome the cloud vendor lock-in problem. Their middleware provides REST-based generic APIs with specific adapters. These generic APIs provide methods to execute the CRUD operations of cloud resources. They have implemented the adapters of their generic APIs for three cloud platforms; GAE, AWS, and IBM private cloud.

Loutas et al. [74] have proposed a semantic-based architecture, called RASIC, to help developers to design and deploy their cloud applications over heterogeneous IaaS platforms. Their architecture is based on semantics and SOA. The semantic annotations are used to annotate cloud applications and resources, whereas SOA is used to help developers to design their applications as web services. Their architecture provides common APIs with static adapters for four heterogeneous IaaS resources, namely; network, storage, instance, and image.

Hill and Humphrey [75] have proposed an abstraction layer, called CSAL, to hide the heterogeneity of different cloud storage services. In addition, CSAL helps cloud developers to port their storage-based applications over heterogeneous cloud platforms. Currently, CSAL provides REST common APIs for three cloud storage services; tables, blobs, and queues. The authors have evaluated their work on two cloud platforms; Windows Azure and Amazon AWS (SimpleDB, S3, and SQS). The main advantage of the CSAL is that it provides a unified namespace to access each one of the supported services (e.g., the unified namespace allows accessing a container using its name without caring about its location). However, the adapters of their common APIs are manually implemented. So, whenever the specific storage APIs of the supported cloud platforms are changed, their adapters will be unusable.
Silva et al. [33] have proposed Service Delivery Cloud Platform (SDCP), which provides service-based common APIs in order to support interoperability among different PaaS platforms. In addition, it provides cloud developers with a toolkit to help them implementing their applications. On the other hand, cloud developers can use this toolkit to implement a new plug-in (i.e., adapter) for a specific PaaS provider. However, this adapter is implemented manually. So, whenever a PaaS specific-API is changed, its corresponding adapter will be unusable. Currently, SDCP provides common APIs for three services: blob storage, columnar database, and notification. An encryption method is provided by the common API of their storage service in order to encrypt data before uploading it to clouds.

Kolb and Wirtz [20] have proposed a common model for different PaaS platforms to solve the application portability problem. In their model, they have defined the structure of PaaS platform to be composed of three layers, namely: Infrastructure, Platform, and Management. They converted their common model into a taxonomy which specifies a set of common properties among different PaaS platforms. Each specific PaaS provider specifies his profile which contains specific values for the taxonomy properties. An application can be ported by matching its profile, which specifies the application requirements, against the profiles of the different PaaS platforms. The matching process will output a list of PaaS platforms that are suitable for deploying that application. However, the main drawback of their model is that they ignore the low-level implementation details while solving the application portability problem. In other words, their model only selects one or more of suitable PaaS platforms to deploy an application and still a cloud developer needs to use a specific PaaS API to implement his application. While, STAGER framework provides cloud developers with a service-based generic-APIs to help them in implementing portable cloud applications.

Cunha et al. [76] have proposed the PaaS Manager framework to overcome the vendor lock-in problem and support the cloud applications portability. The PaaS Manager provides a common RESTful API for deploying and managing cloud applications among different PaaS platforms. Their common API provides a set of specific adapters, which have been manually implemented, for the supported PaaS platforms, such as CF, Heroku, and CloudBees.
Rafique et al. [23] and Walraven et al. [77] have proposed a policy-driven middleware, called PaaSHopper, which develops and deploys cloud applications on hybrid clouds. The PaaSHopper provides two main layers; an abstraction layer to allow interoperability and portability of cloud applications over heterogeneous PaaS platforms and a policy-driven execution layer to allow deployment and execution of cloud applications. It provides a uniform API, which is Java-based, for three services, namely; blob storage, data storage, and asynchronous tasks. Their work is very close to our proposed STAGER. Their uniform API can be semantically annotated and then imported into the STAGER framework to semi-automatically generate the adapters for the supported PaaS platforms. However, their uniform API is not an open source. On the other hand, the adapters of their uniform API have been manually implemented. Therefore, when a new PaaS platform is added or when an existed PaaS provider changes his API, then a new adapter need to be (re)implemented (i.e., the adapters are suffering from the API dynamic adaptation problem). Furthermore, the PaaSHopper is in its early stage because it only provides a prototype.

Alomari et al. [15, 22] have proposed the CDPort (Cloud Data Portability) framework to solve the data portability problem among different cloud databases. According to CDPort, a standard API and a common data model to hide the heterogeneity of different cloud data storage services have been proposed. It provides an adapter for each one of the supported databases. This framework has been tested by implementing a portable JAVA application for supporting NoSQL databases. One of the limitations of this portable application is that it requires an update in its source code to support a specific adapter (e.g., when they need to work with Amazon SimpleDB, they need to change some part of their application source-codes to use the SimpleDBAdapter). Another problem in the CDPort framework is that the adapters source-codes are manually implemented. Thus, when a specific database API is modified, the adapters cannot be used and their source codes need to be manually updated.

Kolb and Röck [78], Röck and Kolb [79] have proposed a unified interface, called Nucleus, for deploying and managing cloud applications over different PaaS platforms. Their interface provides language and platform independent API, which is based on REST and JSON. In addition, their unified API includes a set of specific adapters for each one of the supported PaaS platforms, such as CF, OS, and Heroku. However, when a PaaS
provider changes his API, his corresponding adapter need to be manually updated (i.e.,
their adapters suffer from the API dynamic adaptation problem).

Brogi et al. [80] have proposed SeaClouds, which provides common RESTful APIs with
a set of specific adapters for deploying and managing applications over PaaS and
IaaS platforms. In SeaClouds, the components of the application are specified through
TOSCA standard. In addition, TOSCA and CAMP standards are used to manage appli-
cations at runtime. Although SeaClouds is an open source, it is a language dependent
because it is based on JAVA.

Yasrab and Gu [81] have proposed Multi-Cloud PaaS Architecture (MCPA), which pro-
vides a platform and unified APIs to manage and deploy service-based cloud appli-
cations over heterogeneous PaaS platforms. The MCPA is adopted from SeaClouds.
Moreover, it combines three open source tools, namely; Opscode-Chef, mOSAIC, and
P-TOSCA to support PaaS portability and interoperability. Opscode-Chef is used to
manage cloud applications; whereas mOSAIC is used to discover and monitor cloud
applications; and P-TOSCA, which is TOSCA extension, is used to orchestrate and
describe cloud applications.

Pallavi and Babu [82] have provided generic RESTful APIs, called Open PaaS Database
API (ODBAPI), to overcome the heterogeneity of different structured and unstructured
databases. Their generic APIs provide generic operations to insert, remove, and update
records on heterogeneous databases. In addition, the authors have implemented a set
of static adapters, called virtual data stores, to map their generic APIs into their cor-
responding specific one. Moreover, they have provided a manifest in order to discover
the heterogeneous databases and allow applications to be automatically deployed. They
have evaluated their work using two case studies; MySQL and MongoDB.

Kumar M and Jose [83] have proposed a Generic Cloud Framework (GCF) to help devel-
opers to develop and deploy generic applications over heterogeneous IaaS platforms.
The GCF provides a strategy to implement generic applications that are domain specific,
such as applications for medical systems, applications for agriculture systems, etc. Their
framework includes a set of generic APIs with a set of static adapters, called wrapper.
However, their framework is in its early stage; it is just a prototype and it has been
evaluated using only one case study for the agriculture domain. Moreover, they did not specify the IaaS services that the framework already hides their heterogeneity.

Application portability over IaaS clouds can be done by implementing the applications using open generic APIs, such as Jcloud, Libcloud, Fog, and Deltacloud. Jclouds\(^8\) provides JAVA-based generic APIs as an open source for three services; compute, storage, and load balancer. Fog\(^9\) is similar to Jclouds, but it is based on Ruby. Libcloud\(^10\) provides python-based unified API for compute and DNS services. Deltacloud\(^11\) provides RESTful generic APIs using Ruby language, for compute and storage services, with a set of drivers to manage a set of different IaaS clouds. Thus, these generic APIs are service and language dependent. However, the adapters of all these generic APIs have been manually implemented. So, they may suffer from API dynamic adaptation problem whenever the specific APIs are changed.

In this thesis, we propose an integrated framework, called STAGER, that uses two methodologies; semantics and generic APIs; to overcome the PaaS vendor lock-in problem and support application portability. The proposed generic APIs hide the heterogeneity of different cloud services and help cloud developers to implement cloud applications that are agnostic to PaaS platforms. Moreover, the proposed generic APIs provide a set of specific adapters to allow the mapping between the generic APIs and their corresponding specific APIs. On the other hand, the semantics are used to semi-automatically generate the specific adapters.

### 3.4 Discussion

Table 3.1 summarizes the current research studies to overcome the cloud vendor lock-in problem. Each study is identified with respect to four features:

- **Standard**: it means that a research effort is proposed by an organization for standards.

---

\(^8\)http://jclouds.apache.org/
\(^9\)http://fog.io/
\(^10\)https://libcloud.apache.org/
\(^11\)https://deltacloud.apache.org/
• **Generic Management API**: it means that a research effort provides a generic management API that can be used in the deployment and management phases of a cloud application provisioning. These phases are elaborated in Sub-section 2.2.1.

• **Generic Implementation API**: it means that a research effort provides a generic implementation API that can be used in the implementation phase of a cloud application provisioning.

• **Cloud Model**: it specifies the cloud model that can be handled by a research effort.

• **API Dynamic Adaptation Problem**: it refers if the adapters of generic APIs, which are provided by a research effort, suffer from the API dynamic adaptation problem.

It should be noted that most of the currently related work focuses on providing standards or generic APIs in order to overcome the cloud vendor lock-in problem and support application portability. These generic APIs may focus on management operations for deploying and managing cloud applications among different cloud platforms. On the other hand, these generic APIs may focus on service operations for implementing cloud applications independently of cloud platforms. However, the adapters of the currently available generic APIs are manually implemented. So, they are suffering from the API dynamic adaptation problem. In addition, the current standardization efforts (e.g., CAMP and TOSCA) are in its early stage and they are not supported by most of the leading PaaS platforms.

In this thesis, the STAGER framework has been proposed as a step towards overcoming the API dynamic adaptation problem. STAGER provides generic APIs, which can include management operations or service operations, with semi-automatically generated adapters. In order to evaluate the STAGER framework, we cannot use any of the currently available generic APIs because they are not open source. Thus, we have proposed service-based generic-APIs for blob storage and NoSQL datastore as case studies; and STAGER has been used to generate the adapters for two of the leading PaaS platforms; GAE and Windows Azure. The proposed generic APIs can be used by cloud developers to implement applications that are independent on heterogeneous PaaS platforms. By
this way, the proposed STAGER framework differs from the currently available research efforts by the following:

- It overcomes the API dynamic adaptation problem.
- It can generate the adapters for any semantically annotated generic API.

<table>
<thead>
<tr>
<th>Research Effort</th>
<th>Standard</th>
<th>Generic Manag. API</th>
<th>Generic Implem. API</th>
<th>Cloud Model</th>
<th>API Synchronization Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAMP (2014)</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
<td>PaaS</td>
</tr>
<tr>
<td>TOSCA (2014)</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COAPS (2013, 2016)</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cloud4SOA (2013)</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nucleus (2016)</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PaaS Manager (2014)</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDPort (2014, 2015)</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
<td>PaaS</td>
</tr>
<tr>
<td>PaaSShopper (2014, 2015)</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDCP (2013)</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ODBAPI (2016)</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDMI-proxy (2011)</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SeaClouds (2016)</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td></td>
<td>IaaS</td>
</tr>
<tr>
<td>MCPA (2016)</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td></td>
<td>PaaS</td>
</tr>
<tr>
<td>mOSAIC</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td></td>
<td>SaaS</td>
</tr>
<tr>
<td>CSAL (2010)</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCCI</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jclouds</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
<td>IaaS</td>
</tr>
<tr>
<td>Fog</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
<td>PaaS</td>
</tr>
<tr>
<td>Deltacloud</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
<td>SaaS</td>
</tr>
<tr>
<td>RASIC (2010)</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDMI</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.5 Summary

In this Chapter, we have classified the related work into three categories; standardization efforts, research projects, and generic APIs. These categories are elaborated in details. Finally, a brief discussion is introduced to summerize the existing research studies to solve the vendor lock-in problem.
Chapter 4

The Proposed Generic Cloud APIs

In this chapter, the details of the proposed PaaS generic-APIs, which include deployment API and implementation API, will be explained. In the first Section, the motivation to extend COAPS generic deployment API will be presented, then our extension to this API will be elaborated. In the second Section, the proposed Std-PaaS implementation APIs will be presented, which include generic implementation APIs for both cloud blob storage and NoSQL datastore services.

4.1 COAPS Deployment API

Recall that the COAPS API is a generic PaaS Deployment-API and it is represented as RESTful JAVA open source. Its main objective is to deploy and manage applications on heterogeneous PaaS platforms. Currently, the COAPS API supports only two PaaS platforms; CloudFoundry (CF) and Openshift (OS). In order to allow the COAPS API to support a new PaaS platform, one needs to add a specific adapter in COAPS API for this new PaaS platform [21, 39].

4.1.1 Motivation To Extend COAPS API

There are three main motivations to extend COAPS API as follows[21]:
1. COAPS API is a generic PaaS deployment-API and provided as a RESTful JAVA open source. Thus, cloud developers can use COAPS API to deploy one or more applications on heterogeneous PaaS platforms.

2. COAPS API is already tested and used in two international research projects, namely; CompatibleOne and EASI-CLOUDS.

3. The PADM, which is elaborated in Chapter 2, of COAPS is based on OCCI standards.

### 4.1.2 Extending COAPS API To Support GAE

In this Section, an overview about GAE PaaS platform is provided. Furthermore, the COAPS architecture with GAE is presented. Finally, the road map that is followed for extending COAPS API is illustrated.

#### 4.1.2.1 GAE PaaS Platform Overview

GAE is a PaaS platform provided by Google. It helps developers to build and deploy their web applications on Google cloud. It supports multiple programming languages, such as JAVA, Python, Go, and PHP. Currently, GAE does not support deployment of Web Application Archive (WAR) files [84]. Therefore, it is not possible to deploy regular JAVA web applications on GAE. GAE requires applications to be implemented either using the GAE plugin for Eclipse or manually using the same directory structure supported by GAE. After a GAE application is created, it includes a War directory, which will be used later to deploy that application. The required directory structure for deploying applications to GAE is clarified in Figure 4.1. It contains two main directories:

- **Src** contains JAVA source files. It is assumed that JAVA language is used for building applications to be deployed on GAE.

- **War** contains JSP files, besides the following directories:

  1. **WEB-INF** contains `appengine-web.xml` file, which is used for deploying the application on GAE platform, and `web.xml`, which is used to identify the available servlets and their URL mappings, among other things.
2. *Lib* contains all required JAR files.

3. *Classes* contains compiled classes.

![GAE Directory Structure](image)

**Figure 4.1: GAE Application Structure**

The *appengine-web* XML file is used to configure the application. It contains the application ID and the version number that are used through the deployment process. A very simple *appengine-web* XML file is presented in Table 4.1.

**Table 4.1: Appengine-web XML file**

```xml
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  <application>gaeadapter</application>
  <version>1</version>
  <threadsafe>true</threadsafe>
</appengine-web-app>
```

4.1.2.2 The Extended COAPS Architecture

COAPS deployment-API has been extended to include deploying applications on GAE PaaS platform, besides CF and OS. Whereas both CF and OS PaaS platforms use the
same application packaging (i.e., a WAR archive), while deploying the same application on GAE requires application repackaging (i.e., restructuring the application to be in the required GAE application-structure). Figure 4.2 illustrates COAPS architecture with GAE. Mainly, two components are added; App Converter and GAE Adapter.

![Figure 4.2: COAPS new Architecture with GAE](image)

The App Converter component overcomes the problem of deploying JAVA web application into GAE [26]. It converts a JAVA web application (WAR archive) into GAE application. For simplicity, it is assumed that the given JAVA web application uses any external service (e.g., a database) through a RESTful API. The usage scenario of the App Converter is as follows:

- It receives (from a cloud consumer) a manifest that specifies the location of an application WAR archive, GAE application ID, and version number that are used to create appengine-web.xml configuration file.
- It extracts the application WAR archive and maps the extracted folders to the directory structure that is required by GAE (see Figure 4.1).

The GAE Adapter deploys an application on GAE platform using the generated WAR directory to be executed on GAE cloud [25]. Both CF and OS platforms provide developers with client APIs to deploy and manage (i.e., start, stop, and delete) their
applications. These applications are provided as WAR archives. On the contrary, GAE does not provide a client API to deploy and manage applications. The developers can deploy their applications to GAE either by using GAE plugin for Eclipse or using Appcfg command line tool [85]. Since GAE plugin for Eclipse cannot be called as a standalone API and it requires the Eclipse Integrated Development Environment (IDE), the Appcfg command line tool is used to implement the deployment methods inside GAE-PaaS API.

4.1.2.3 Road Map To Extend COAPS API For Supporting GAE

The generalization of COAPS API has been proved after extending it to include deploying applications on GAE, besides CF and OS. Thereby, the following points elaborate the road map that is followed to extend COAPS API [25, 26]:

1. A new dummy module is added to the COAPS-Core to include the GoogleAppEngine-api (called GAE-PaaS API) as it is depicted in Figure 4.3. The COAPS-Core includes the PaaS platforms that are supported by COAPS.

2. COAPS API provides a PaaS application-manifest schema that can be used to generate the manifest classes. This schema is used by GAE-PaaS API to generate the PaaS application-manifest classes, which will be used later to process an application manifest that is given by a developer to deploy an application.

3. GAE-PaaS API includes the deployment operations of GAE platform through COAPS API. However, deploying applications on GAE require a Gmail account. Therefore, an account information is defined in the credentials file inside GAE-PaaS API. So, a developer can update this file to include his account information. A snapshot of the credentials file is depicted in Figure 4.4.
4. The *App Converter* component is defined inside GAE-PaaS API to convert JAVA web applications into GAE applications.

5. GAE-PaaS API creates *EnvironmentManagerResource* class, which implements the *RestEnvironmentManager* interface as explained in Chapter 2. The *EnvironmentManagerResource* class provides a specific implementation for GAE to the methods (create, delete, and describe) environments.

6. GAE-PaaS API creates *ApplicationManagerResource* class, which implements the *RestApplicationManager* interface as explained in Chapter 2. The *ApplicationManagerResource* class provides a specific implementation for GAE to the methods (create, deploy, and start) applications.

7. Google provides *Appcfg* command line tool to be used by developers to deploy their applications on GAE platform [85]. Therefore, *Appcfg* tool has been used in the *ApplicationManagerResource* class to deploy applications on GAE platform.

8. Many other classes are added to GAE-PaaS API to provide utilities to be used by both *ApplicationManagerResource* and *EnvironmentManagerResource* (e.g., some of these utilities are preserving a pool of the created applications and the created environments, as well as, generating links for all the REST methods that are defined in the *ApplicationManagerResource* and *EnvironmentManagerResource*, etc.).

### 4.1.3 COAPS Summary

The main advantage of COAPS is that it allows developers to use a generic API (i.e., COAPS API) to deploy one or more applications on heterogeneous PaaS platforms (i.e., CF, OS, and GAE). Thereby, there is no need for developers to learn the different
deployment APIs of all targeted heterogeneous PaaS platforms. However, COAPS API has two main drawbacks:

- It is limited to only deploy and manage applications on heterogeneous PaaS platforms. It did not provide consideration about how to monitor running applications and manage their resources (e.g., scale the resources up/down through the run time, migrate the applications in case of fault, etc.) [21].

- The specific adapters (i.e., GAE adapter, CF adapter, and OS adapter) of COAPS API are suffering from the API dynamic adaptation problem (i.e., updates in a specific API, may require re-factoring or even re-implementation of its corresponding adapter).

In general, to deploy an application to GAE, the application must be implemented by GAE plugin. However, this problem has been overcome by creating App Converter component to convert a regular JAVA web application into a GAE application. It is worth to mention that GAE-PaaS API does not provide methods to stop, restart, and delete the deployed applications. This is because GAE platform is proprietary and does not provide such API. However, GAE allows developers to manage their deployed applications online using a dashboard. Table 4.2 summarises a comparison between the currently supported PaaS platforms by COAPS API.

**Table 4.2: Comparison between the supported PaaS platforms by COAPS API**

<table>
<thead>
<tr>
<th>Feature</th>
<th>CF-PaaS API/OS-PaaS API</th>
<th>GAE-PaaS API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployment API</td>
<td>org.cloudfoundry.client.lib and com.openshift.client respectively</td>
<td>No API, only appcfg tool is provided</td>
</tr>
<tr>
<td>Management API</td>
<td>org.cloudfoundry.client.lib and com.openshift.client respectively</td>
<td>No API, only the online dashboard of the application is used in the management</td>
</tr>
<tr>
<td>Deployable Type</td>
<td>WAR archive</td>
<td>WAR directory</td>
</tr>
</tbody>
</table>

### 4.2 Std-PaaS Implementation APIs

In this Section, the proposed standard PaaS implementation-API (called Std-PaaS APsI) will be presented. Std-Paas APIs are designed to include generic APIs for different
services. Currently, Std-PaaS APIs provide generic APIs for two main services; **Cloud Blob Storage Service (CBSS)** and **NoSQL Datastore Service (NSDS)**. We have designed the generic APIs for these two services because of most of software applications require file storage and/or data storage services. The architecture of the Std-PaaS APIs and the proposed generic APIs of both CBSS and NSDS will be elaborated in the following Sub-sections.

### 4.2.1 Std-PaaS APIs Architecture

Std-PaaS APIs are proposed to overcome the vendor lock-in problem by providing cloud developers with a set of standard implementation-APIs for each service to help them for implementing their cloud applications in a generic fashion. Std-PaaS APIs work as a middleware layer to communicate between heterogeneous PaaS platforms and cloud developers. Std-PaaS APIs architecture is illustrated in Figure 4.5. This architecture is composed of three main layers; **Std-PaaS APIs** (the bottom layer), **adapters** (the middle layer), and **PaaS specific-APIs** (the top layer).

![Std-PaaS APIs Architecture](image)

*Figure 4.5: Std-PaaS APIs Architecture*
Chapter 4. The Proposed Generic Cloud APIs

The **Std-PaaS APIs** layer provides cloud developers with a set of generic methods to allow them to develop generic applications. The **adapters** layer provides an adapter for each PaaS specific-platform to map the generic methods of the Std-PaaS APIs into their corresponding PaaS specific-API methods. The **PaaS specific-APIs** layer provides PaaS specific-API for each PaaS platform.

There are two objectives have been considered for designing Std-PaaS APIs. The first one is to hide heterogeneity of different PaaS platforms; while the second one is to design generic APIs with a high level of technicality. Std-PaaS APIs are designed in an extendable way to support many generic APIs for different services (e.g., generic APIs for database, for blob storage, for authentication and authorization, etc). In this thesis, generic APIs for CBSS and NSDS are proposed. In addition, Std-PaaS APIs can be extended to include another generic APIs for other PaaS services.

The main advantages of Std-PaaS APIs with respect to cloud developers are as follows:

- Cloud developers do not need to waste more time in learning the specific APIs of each PaaS platform. They need only to learn the proposed Std-PaaS APIs.
- Whenever the PaaS specific-APIs are changed, the adapters of Std-PaaS APIs can be semi-automatically generated through STAGER framework.
- Cloud developers can use Std-PaaS APIs to implement generic applications (i.e, write once). Finally, the implemented generic applications can be deployed on multiple PaaS platforms with no change in the source code (i.e., deploy many).

However, cloud developers need only to configure the class path of the generic applications to use a specific adapter. Assume, for example, a developer needs to implement an application to be deployed on Windows Azure platform and GAE platform. Then, he/she can use Std-PaaS APIs to implement this application. Finally, to deploy this application on Windows Azure, a developer needs to configure the application class-path to use the generated adapter of Windows Azure; while to deploy the same application on GAE, he/she needs to configure the application class-path to use the generated adapter of GAE.
4.2.2 The Proposed CBSS Generic-API

As a part of Std-PaaS APIs, a generic implementation API for CBSS is proposed [13]. This generic API helps cloud developers to store large files (i.e., blobs) on a set of heterogeneous PaaS clouds. The blobs are stored in buckets or containers. Each PaaS platform provides a specific API for CBSS (e.g., GAE provides Google Cloud Storage (GCS) API [34] and Microsoft Azure provides Azure blob storage API [35]).

To design the generic API for CBSS, the following methodology is adopted, which is described in more details in the next Sub-sections:

1. We analyze the vendor-specific blob storage API for some PaaS platforms (e.g., GAE and Azure).
2. We compare the specific APIs and identify a set of semantically common features among them, as well as, a set of unique features for each API.
3. Finally, we design the generic API based on the common features.

4.2.2.1 Studying Specific CBSS API

The specific CBSS API of both GAE and Azure were studied and tested. GCS allows users to store and retrieve large files (called objects) at any time [34]. These files must be stored in buckets that must belong to a GAE project. A summary of GCS API is presented in Table 2.1.

Microsoft Azure provides Azure Blob Storage as CBSS API. Azure Blob Storage allows users to store and retrieve large files (called blobs) up to a terabyte [35]. These files must be stored in containers that must belong to a storage account. A summary of Azure blob storage API is presented in Table 2.2.

4.2.2.2 Comparison of CBSS API of GAE and Azure Platforms

A comparison between GAE and Azure CBSS APIs was done. This comparison is based on identifying a set of semantically common features between both GAE and Azure,
as well as, identifying a set of unique features for each PaaS platform. Typical usage
scenarios of both GAE and Azure CBSS APIs involve the following five semantic steps:

- **Configuration** is used to get the required parameters from a user to execute a
  specific CBS API.

- **Create Container** is used to create a container.

- **Create Blob** is used to create a blob inside a container.

- **Upload Blob** is used to upload a given input stream into the created blob.

- **Access Blob** is used to access the uploaded blob.

These common steps hide many implementation-specific details, such as:

1. A stored file in GAE is called an object, whereas in Azure it is called a blob. Therefore, an object and a blob are semantically equivalent.

2. Azure stores blobs in containers, whereas GAE stores them in buckets. Therefore, containers and buckets are semantically equivalent.

3. GAE requires to use an existing container, whereas Azure creates a container if it does not exist.

4. Azure requires a storage account (with an account name and a key) to access its blob storage service, whereas GAE requires a Gmail account and a project-ID to access its blob storage service.

5. Azure provides a method to update container permissions, whereas GAE does not support this option.

6. GAE requires the file content-type of an uploaded file, whereas Azure does not require the file content-type.
4.2.2.3 CBSS Generic-API

A preliminary version of cloud blob storage service (CBSS) generic-APIs is proposed to include five generic methods, namely; configuration, createContainer, createBlob, uploadBlob, and accessBlob [13]. Table 4.3 summarizes how the preliminary version of CBSS generic-API is implemented in each specific PaaS platform.

Table 4.3: Generic CBSS API vs. Specific CBSS API

<table>
<thead>
<tr>
<th>Generic CBSS API</th>
<th>Azure Blob Storage [35]</th>
<th>Google Cloud Storage (GCS) [34]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td>The configuration parameters include the Azure storage account with an account name and a key, a container name, which will be created if it does not exist, and an input stream for a file to be uploaded.</td>
<td>The configuration parameters include a container name, which must be created a priori; otherwise an exception is raised and an input stream for a file to be uploaded.</td>
</tr>
<tr>
<td>Create Container</td>
<td>Create the following instances: a cloud storage account, a cloud blob client, and a cloud blob container.</td>
<td>The container must be created from Google developer console before the application starts. This is because no API is available for creating a container. So, no container establishment is needed from inside an application.</td>
</tr>
<tr>
<td>Create Blob</td>
<td>Create an instance of CloudBlockBlob using the previously created container.</td>
<td>Create the following instances: GcsFilename to specify a bucket and a file name, GcsFileOptions to specify a file content type and its Access Control List (ACL), and GcsService to use the previously created instances to create or replace the given file; the return value is a GcsOutputChannel instance.</td>
</tr>
</tbody>
</table>
| Upload Blob      | - Call the upload method of the previously created CloudBlockBlob instance and give it an input stream of a given file. A URI of the uploaded file is returned, which follows the template: http://<your_Azure_storage_service>/.blob.core.windows.net/<container_name>/<file_name>
  - For example: http://myteststore3.blob.core.windows.net/mycontainer/capture.png | - No direct method to do the upload. So, we need to copy data from a source to a destination using a buffer for example. A URI of the uploaded file is returned, which follows the template: http://storage.googleapis.com/<bucket_name>/<path_to_object>
  - For example: http://storage.googleapis.com/my_bucket_test/Capture.PNG |
| Access Blob      | Using the previously returned URI. | Using the previously returned URI. |
The preliminary version of CBSS generic-API is enhanced and extended to include the following generic methods [86]:

- **Authentication**: aims to authenticate a current user and make sure that this user is authorized to execute blob operations. It has the prototype as `com.std.paas.api.authenticate(java.lang.String)`. It takes, as an input, an XML manifest string, which contains information about the account name and the key of a current user. It returns, as an output, a storage account object.

- **Create Blob**: aims to create a virtual blob inside a container. A virtual blob refers to a blob that is not physically available inside a data center. It has the prototype as `com.std.paas.api.createBlob(java.lang.String, java.lang.Object, java.lang.String, java.lang.String, com.std.paas.api.staticcode.HelperFunctionalities.AccessRights)`. It returns, as an output, a virtual new blob and it has several input parameters as follows:
  - The first parameter is a string to represent a blob content-type.
  - The second parameter is a storage account object.
  - The third parameter is a string for the container name that is used to store a blob in it.
  - The forth parameter is a string for a blob name that will be stored.
  - The final parameter is an object to represent the blob access right that may be public access or private access.

- **Upload Blob**: aims to upload an input stream on an existed virtual blob. By this method, a virtual blob is converted into a physical blob, which is physically stored inside a data center. It has the prototype as `com.std.paas.api.uploadBlob(java.lang.Object, java.io.InputStream)`.

- **Download Blob**: aims to download a blob by storing its content in an output stream. It has the prototype as `com.std.paas.api.downloadBlob(java.lang.Object, java.io.OutputStream)`. It takes two input parameters as a virtual blob object and an input stream object, respectively. It returns, as an output, a boolean variable to represent the upload status.
java.lang.String, java.lang.String, java.io.OutputStream, long). It returns, as an output, a boolean variable to represent the download status and it has several input parameters as follows:

- The first parameter is a storage account object.
- The second parameter is a string for the container name, which will be used to download a blob from it.
- The third parameter is a string for a blob name that will be downloaded.
- The forth parameter is an output stream object to download a blob in it.
- The final parameter is an integer variable to represent the start position, inside a blob, to start downloading from it.

- **Delete Blob**: aims to delete a physical blob. It has the prototype as com.std.paas.api.deleteBlob(java.lang.Object, java.lang.String, java.lang.String). It returns, as an output, a boolean variable to represent the delete status and it takes three input parameters as follows:

  - The first parameter is a storage account object.
  - The second parameter is a string for the container name that will be used to delete a blob from it.
  - The final parameter is a string for a blob name that will be deleted.

- **List Blobs**: aims to list all blobs that are available in a specific container. It has the prototype as com.std.paas.api.listBlobs(java.lang.Object, java.lang.String). It requires two input parameters as a storage account object and a string for the container name, which is needed to list its blobs, respectively. It returns, as an output, an object which contains the blobs list.

### 4.2.3 The Proposed NSDS Generic-API

As a part of Std-PaaS APIs, a generic implementation API for NSDS is proposed. This generic API helps cloud developers to handle NoSQL datastore on a set of heterogeneous PaaS clouds. The NoSQL datastore stores the data as objects (called entities) such that each entity has a unique key and a set of properties which are key-value pairs.
Furthermore, each entity must belong to a table (or kind). The NoSQL datastore has two main features, namely; non-relational database and schemaless. Each PaaS platform provides a specific API for NSDS (e.g., GAE provides Google Cloud Datastore (GCD) API [37] and Azure provides the Azure Table Storage API [38]). To design the generic API for NSDS, we adopt the same methodology that is followed to design the CBSS generic-API. This methodology is elaborated in the following Sub-sections.

4.2.3.1 Studying Specific NSDS API

The specific NSDS API of both GAE and Azure were studied and tested. GCD API allows users to store data as objects (i.e., entities). These entities must be stored in kinds (i.e., tables) that must belong to a GAE project [37]. An entity is identified by a kind and a unique key. Furthermore, an entity can include a set of properties as key-value pairs. See Table 2.3, which provides a summary of GCD API.

Microsoft Azure provides Azure table storage as NSDS API. Azure table storage allows users to store large amounts of non-relational data as entities [38]. These entities must be stored in tables that must belong to a storage account. An entity is identified by a table, a partition key, and a unique row key. A partition key can be used to categorize the rows of a table, such that the rows with the same partition key can be queried faster than that with different partition keys. See Table 2.4, which provides a summary of Azure table storage API.

4.2.3.2 Comparison of NSDS API of GAE and Azure Platforms

A comparison between the NSDS APIs of both GAE and Azure platforms has been done. This comparison is based on identifying a set of semantically common features between both GAE and Azure, as well as, identifying a set of unique features for each PaaS platform. Table 4.4 summaries the features of both GCD and Azure table storage. Some examples of semantically common features between the NSDS APIs of both GAE and Azure platforms are as follows:

1. A datastore row in GAE is called an object, whereas in Azure it is called an entity. Therefore, an object and an entity are semantically equivalent.
2. Azure stores entities in tables, whereas GAE stores them in kinds. Therefore, tables and kinds are semantically equivalent.

Some examples of the unique features between the NSDS APIs of both GAE and Azure are as follows:

1. Azure requires a storage account (with an account name and a key) to access its NSDS, whereas GAE requires a Gmail account and a GAE project to access its NSDS.

2. A datastore row in GAE is identified by a unique key, whereas Azure identifies a datastore row by a partition key and a unique row key. By this way, multiple rows in Azure table can have the same partition key.

<table>
<thead>
<tr>
<th>Features</th>
<th>Google Cloud Datastore (GCD)</th>
<th>Azure Table Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Requires Gmail account and GAE project.</td>
<td>Requires a storage account.</td>
</tr>
<tr>
<td>Table Representation</td>
<td>Called Kind.</td>
<td>Called CloudTable.</td>
</tr>
<tr>
<td>Entity unique-key</td>
<td>A unique key.</td>
<td>Composed from a partition key and a unique row key.</td>
</tr>
<tr>
<td>Entity is identified by</td>
<td>A kind and a key</td>
<td>A table name, a partition key and a row key</td>
</tr>
</tbody>
</table>

### 4.2.3.3 NSDS Generic-API

We have proposed a generic API for NoSQL datastore service (NSDS). This generic API helps cloud developers to create, retrieve, and delete datastore entities on a set of heterogeneous PaaS platforms. The generic API of the NSDS includes the following generic methods:

- **Authentication**: aims to authenticate a current user and make sure that this user is authorized to execute datastore operations. It has the prototype as `com.std.paas`. 
Chapter 4. The Proposed Generic Cloud APIs

api.authenticateDS(java.lang.String). It takes, as an input, an XML manifest string, which contains info about the account name and the key of a current user. It returns, as an output, a datastore account object.

- **Create Entity**: aims to create an entity and store its data inside a datastore table. It has the prototype as `com.std.paas.api.createEntity(java.lang.Object, java.lang.String, java.lang.String, java.lang.String, java.lang.String, java.util.HashMap)`. It returns, as an output, an object of the created entity. In addition, it has several input parameters as follows:
  
  - The first parameter is a datastore account object.
  - The second parameter is a string for the table name, which will be used to store an entity.
  - The third parameter is a string for a partition key that is used to categorize rows inside a table.
  - The forth parameter is a row key that is used to uniquely identify a row.
  - The final parameter is a hash map that contains the properties of an entity as key-value pairs.

- **Retrieve Entity**: aims to get the data of a datastore entity. It has the prototype as `com.std.paas.api.retrieveEntity(java.lang.Object, java.lang.String, java.lang.String, java.lang.String)`.

  It returns, as an output, an entity object which contains the table name, partition key, row key, and properties of the retrieved entity. In addition, it has several input parameters as follows:
  
  - The first parameter is a datastore account object.
  - The second parameter is a string for the table name, which will be used to retrieve an entity from it.
  - The third parameter is a string for a partition key of an entity.
  - The forth parameter is a row key of an entity to be retrieved.

- **Delete Entity**: aims to delete a datastore entity. It has the prototype as `com.std.paas.api.deleteEntity(java.lang.Object, java.lang.String, java.lang.String)`.

  It returns, as an output, a boolean variable to represent the delete status and it takes several input parameters as follows:
– The first parameter is a datastore account object.
– The second parameter is a string for the table name, which will be used to delete an entity from it.
– The third parameter is a string for a partition key of an entity.
– The forth parameter is a row key of an entity to be deleted.

4.3 Summary

In this chapter, we have presented the proposed generic APIs for PaaS platforms which include:

• **Deployment API**: we have extended the COAPS deployment API to include GAE platform, besides CF and OS.

• **Implementation API**: we have proposed generic APIs for two main cloud services, blob storage and NoSQL datastore. These generic APIs will be used by STAGER framework, in the next Chapter, to semi-automatically generate their adapters.
Chapter 5

The Proposed Semantic-Based Generation of Generic-API Adapters for Portable Cloud Applications (STAGER) Framework

In this Chapter, a framework for semantic-based generation of generic-API adapters for portable cloud applications, called STAGER, will be presented. First, an overview about STAGER framework is presented. Second, the architecture of STAGER framework and the process flow of its components are illustrated. Third, the STAGER motivation and assumptions about the PaaS proprietary API, programming language, semantic annotation, and utilities API are specified. Finally, the main components of STAGER framework are elaborated.

5.1 STAGER Overview

One approach for supporting application portability (i.e., overcoming PaaS vendor lock-in problem) is to implement applications using generic APIs and develop an adapter
for each specific PaaS platform. The generic API provides cloud developers with a set of generic methods for each service (e.g., database service, blob storage service, etc.). Each adapter translates these generic methods into their corresponding specific-API calls. However, updating a PaaS specific-API (e.g., renaming methods or changing input parameters) may require re-factoring or even re-implementation of its corresponding adapter; otherwise applications that use the adapter will not be executed correctly with the new API. We use the term API dynamic adaptation to name this problem [24]. In this thesis, STAGER framework has been proposed with two main objectives:

- Overcomes the PaaS vendor lock-in problem by providing service-based generic-APIs (called Std-PaaS APIs) with a set of specific adapters for different PaaS platforms. Currently, the Std-PaaS APIs include generic APIs for two main services, namely; blob storage and NoSQL datastore.

- Overcomes the API dynamic adaptation problem by semi-automatically generating the adapters for generic APIs.

In order to satisfy these objectives, STAGER framework has been integrated by combining two main methodologies as follows:

- **Generic APIs**: STAGER provides a generic implementation API, as a middleware layer, to communicate between cloud developers and a set of heterogeneous PaaS platforms.

- **Semantic**: STAGER uses the semantic annotations to semi-automatically generate the adapters for the proposed generic implementation-API (i.e., Std-PaaS APIs). This generic API, with their adapters, will be used by developers to implement generic cloud applications.

## 5.2 STAGER Architecture

The architecture of STAGER framework is illustrated in Figure 5.1, which specifies the main inputs and output of STAGER. This framework requires, as inputs, a generic API for each *platform basic service*, as well as, a specific API for each PaaS platform. A
generic API is composed from a set of generic methods. A specific API is also composed from a set of specific methods. The methods of the generic/specific API should be annotated to describe the conceptual meanings of the I/O parameters of each method. These annotations are based on a common domain ontology, called STAGER ontology. STAGER framework generates, as an output, the source code of a set of specific adapters for the given generic APIs.

![Diagram of STAGER framework]

**Figure 5.1:** The architecture of STAGER framework

The process flow of STAGER framework is illustrated in Figure 5.2. STAGER framework is composed of five main components, namely: Source Code Ontology Population (SCOP), STAGER ontology, Semi-automatic Adapter Validation and Generation (SAVG), Std-PaaS APIs, and Utilities API.
Chapter 5. The Proposed STAGER framework

The main inputs and outputs of STAGER components can be summarized as follows:

- **SCOP**: requires, as inputs, an annotated specific API for each PaaS platform, an annotated generic API for each service (e.g., Std-PaaS APIs), and an annotated utilities API, which represents a set of methods that are implemented manually. The SCOP parses the given APIs and inserts them into the STAGER ontology.

- **STAGER Ontology**: integrates Std-PaaS APIs, utilities API, as well as, the heterogeneous APIs for different PaaS platforms as an ontology. STAGER ontology has been entered as an input to SAVG component.

- **SAVG**: requires, as an input, STAGER ontology. It validates whether the desired PaaS platform supports all generic APIs that are used by cloud developers. If the validation fails, it will notify cloud developers with a list of generic methods.
Chapter 5. The Proposed STAGER framework

that cannot be generated. Otherwise, it can generate the source code of a specific adapter for the desired PaaS platform. The generated adapter maps the generic calls into their corresponding specific calls.

- **Std-PaaS APIs**: represents the proposed generic implementation-APIs for each service (called service-based generic-APIs), such as datastore or blob storage.

- **Utilities API**: represents a set of helper methods that cannot be generated automatically and are needed to satisfy a specific service.

5.3 Motivation and Assumptions

Due to the lack of generic APIs for developing cloud applications, as well as, the currently available generic APIs are suffering from the API dynamic adaptation problem, STAGER framework is proposed. This framework provides Std-PaaS APIs that include service-based generic-APIs for implementing portable cloud application. In addition, STAGER framework provides a component in order to semi-automatically generate the adapters for generic APIs.

STAGER framework has four main actors; **PaaS cloud provider**, **specific-API annotator**, **generic-API designer**, and **cloud developer**. The role of each actor is clarified through the scenario of using STAGER framework as follows:

1. **PaaS cloud provider**: provides a set of service-based specific-APIs.

2. **Specific-API annotator**: annotates the PaaS specific-APIs and passes them as an input to STAGER framework.

3. **Generic-API designer**: designs and annotates a set of service-based generic-APIs and passes them as an input to STAGER framework. Also, the generic-API designer designs and annotates the required utilities API and passes them as an input to STAGER framework.

4. The SCOP component parses all these APIs (specific, generic, and utilities) and inserts them into the STAGER ontology.
5. Based on the STAGER ontology, the SAVG component validates whether the desired PaaS platform supports all generic APIs. If the validation fails, it will notify cloud developers with a list of generic methods that cannot be generated. Otherwise, it generates the source codes of a set of specific adapters for the desired PaaS platforms.

6. **Cloud developer**: uses the generated adapters to implement generic applications that can be deployed on multiple heterogeneous PaaS platforms.

STAGER framework is designed in a customizable way. The customization of this framework can be done in two ways:

1. **By generating a new adapter for a new PaaS platform**: the specific-API annotator annotates the specific-API of the new PaaS and passes them as an input to STAGER framework, which can semi-automatically generate a new adapter for the desired PaaS platform.

2. **By adding a new generic-API**: the generic-API designer designs and annotate the prototypes of a set of new generic methods and their required utilities methods (if exist). Then, the new generic API are passed to STAGER framework to semi-automatically generate their adapters.

However, STAGER framework involves a set of assumptions that are specified as follows. We address the case that an application uses the proprietary API of a PaaS platform and this application needs to be ported to another PaaS platform. The proprietary API can be management API (such as deploying, scaling, and logging) or service API (such as blob, datastore, and messaging). STAGER framework can work for both management API and service API. Currently, the service API has been used in the evaluation, while the management API is postponed as future work.

A real set of PaaS standard-APIs should include generic APIs for each service (e.g., database, blob storage, NoSQL datastore, messaging, email, authentication, and authorization). Therefore, our target is not to abstract all available service interfaces using a single API, but to define a set of generic APIs for each service (called service-based generic-API). These generic APIs can be combined in the Std-PaaS APIs.
The proposed Std-PaaS APIs are JAVA-based. So, only JAVA applications can currently benefit from STAGER framework. In addition, STAGER framework works for generating adapters for only API wrappers (written in JAVA) because the implementation uses JAVA reflection mechanism to parse PaaS specific-APIs. However, STAGER framework can cope with other programming languages provided that they support a mechanism similar to JAVA reflection (e.g., .NET calls it reflection \[87\] and Ruby calls it introspection \[88\]). Although the world is heading to a RESTful API, most of the currently available PaaS specific-APIs are not REST (e.g., GAE provides REST API for only one service, which is Task Queues \[89\]). Therefore, the proposed Std-PaaS APIs are created as API wrapper.

For generating adapters for the Std-PaaS APIs, we need some codes, which need to be implemented manually. These manually implemented codes are combined as Utilities APIs, which may need to be extended whenever a new platform is introduced, as well as, a new set of methods is implemented. Most of the utilities API involves type conversions and loops.

STAGER framework assumes that the semantic annotations of all Object Oriented methods, which may be specific, generic or utility, are available as inputs. Currently, these semantic annotations are done manually and they include the semantic meaning of the input(s) and the output parameters of each method. By this way, STAGER framework can import any generic API and generate their specific adapters provided that this generic API is semantically annotated.

5.4 Components of STAGER Framework

This Section elaborates on the components of the proposed STAGER framework. The STAGER components can be categorized into three types; service independent (e.g., SCOP and SAVG), service dependent (e.g., Std-PaaS APIs and utilities API), and service partially-dependent components (e.g., the STAGER ontology).
5.4.1 STAGER Ontology

The STAGER ontology is a service partially-dependent component because it is composed of service independent and service dependent ontologies. It is designed based on SCRO ontology to store ontological instances of a set of specific PaaS APIs, generic APIs, and a set of utilities API. Figure 5.3 illustrates the structure of the STAGER ontology. The STAGER ontology includes three main ontologies, namely; SCRO+, domain specific, and Std-PaaS APIs ontologies. These ontologies have been elaborated in the next Sub-sections.

**Figure 5.3: The STAGER Ontology Structure**

5.4.1.1 SCRO+

SCRO+ is a service independent ontology. It is an enhanced version of SCRO ontology. Although SCRO ontology can be used to represent a PaaS specific-API as ontology, it lacks the way of specifying the semantic meaning of the I/O parameters of an Object Oriented method. For example, we need to represent a method inside SCRO ontology with the prototype `int meth1(java.lang.String,java.lang.String)`. This method requires two input parameters of type `java.lang.String` to represent a `userName` and a `password`, respectively. In addition, it returns an integer to represent a `connectionStatus`. Therefore, an ontological instance with the name `meth1(java.lang.String,java.lang.String)`
should be added to SCRO ontology under the method concept. Furthermore, two relationships should be defined for this instance as hasInputType with the value `java.lang.String` and hasOutputType with the value `int`. However, there are three extra problems in the definition of this ontological instance:

1. The method `meth1` requires two input parameters of type `java.lang.String`. However, only one relationship with the name `hasInputType` and the value `java.lang.String` can be defined. This is because it is illegal to define the same relationship with the same value more than one time.

2. SCRO does not provide relationships to define the semantic meaning of each input parameter of an Object Oriented method. Since these two input parameters of `meth1` have the same data type, we need to identify the semantic meaning of each one to differentiate between them.

3. SCRO does not provide relationships to define the semantic meaning of an output parameter of an Object Oriented method.

Therefore, SCRO\textsuperscript{+} has been introduced as an enhanced version of SCRO ontology to overcome these problems. SCRO\textsuperscript{+} defines three new properties as follows (see Figure 5.4):

1. `hasSemanticInput`: an object property which specifies the semantic annotation of an input parameter of a method. e.g., for `meth1`, two relationships need to be defined; `hasSemanticInput` with the value `username` and `hasSemanticInput` with the value `password`.

2. `hasSignatureSemantic`: a data property which specifies the required order for the semantic annotations for all input parameters of a method. e.g., for `meth1`, the order of its input parameters needs to be defined by defining the relationship `hasSignatureSemantic` with the value `[username, password]`.

3. `hasSemanticOutput`: an object property which specifies the semantic annotation of an output parameter of a method. e.g., for `meth1`, the relationship `hasSemanticOutput` with the value `connectionStatus` should be defined.
5.4.1.2 Domain Specific Ontology

The domain specific ontology is a service dependent ontology because it stores vocabularies that are needed to satisfy a specific service by different PaaS platforms. On the other hand, this ontology unifies the concepts used by different PaaS platforms to satisfy a specific service (e.g., GAE uses a bucket to store a blob; while Azure uses a container to store a blob. So, the bucket and the container concepts have the same semantic and should be unified). Therefore, it is required to create a domain specific ontology for each specific service (e.g., blob storage, NoSQL datastore, messaging, database, and authentication). In our case studies, two domain specific ontologies are defined for blob storage and NoSQL datastore (see Figure 5.5). These ontologies store all vocabularies that are needed to satisfy the blob storage service (e.g., blob, blob URI, and container) and the NoSQL datastore service (e.g., entity and datastore account).

5.4.1.3 Std-PaaS APIs Ontology

The Std-PaaS APIs ontology is a service dependent ontology because it semantically describes and shares a set of service-based generic-APIs among different PaaS platforms. Each generic API provides a set of generic methods which hides the heterogeneity of a
Figure 5.5: Domain Specific Ontologies for Blob Storage and NoSQL Datastore

Specific service. Some examples for these generic APIs are generic API for blob storage service, generic API for messaging service, generic API for database service, etc. Figure 5.6 illustrates a snapshot of the Std-PaaS APIs ontology with a set of services, such as messaging, authentication, and cloud blob storage. The right part of Figure 5.6 highlights the generic methods that are defined for CBSS.

Figure 5.6: A snapshot of the Std-PaaS APIs ontology
5.4.2 Source Code Ontology Population (SCOP)

The SCOP component is a service independent component and is considered one of the major components in STAGER because it automatically parses and analyzes a set of PaaS APIs, either specific or generic, and a set of utilities API. Furthermore, it inserts the analyzed data into the STAGER ontology. Therefore, the SCOP component requires, as inputs, one or more PaaS API(s) with their semantic annotations and it outputs an updated version of the STAGER ontology. The SCOP component has two main functions:

- **Automatic analysis of a PaaS API**: it uses the reflection mechanism [52] to parse a PaaS specific-API, which is represented as JAR library, in order to get more information about the library classes, the methods of each class, the I/O parameters of each method, etc.

- **Semantic Model of a PaaS API**: it inserts the data retrieved, from the automatic analysis function, into the STAGER ontology as a set of instances. Next, these newly inserted instances are semantically annotated by the specific-API annotator.

After the SCOP component finishes its functions, it will output an updated version of the STAGER ontology. This newly updated version includes the semantic models for all given PaaS APIs. Finally, the STAGER ontology with the SPARQL query language forms the main infrastructure to execute the SAVG component.

5.4.3 Std-PaaS APIs

The Std-PaaS APIs is a service dependent component because it provides a set of generic implementation-APIs for each service. These generic APIs work as a middleware layer to hide heterogeneity of different PaaS platforms and help cloud developers to implement generic applications. For more details about the Std-PaaS APIs, see Chapter 4.
The main advantages of the Std-PaaS APIs with respect to cloud developers are as follows:

1. Cloud developers do not need to waste more time learning the service-based specific-APIs for each PaaS platform. They only need to learn the proposed Std-PaaS APIs.

2. Whenever the PaaS specific-APIs are changed, STAGER framework can generate the adapters for the Std-PaaS APIs in a semi-automatic way.

3. Cloud developers can use the Std-PaaS APIs to implement generic applications (i.e., write once).

4. The implemented generic applications can be deployed on multiple PaaS platforms with no change in the source codes (i.e., deploy many). However, cloud developers only need to configure the class path of the generic applications to use a specific adapter.

5.4.4 Utilities API

The utilities API is a service dependent component because it stores a set of static codes (i.e., manually implemented codes) for each service. These codes are difficult to be automatically generated and are needed during the generation of the specific adapters. Most of the utilities API involves type conversions (e.g., convert from a generic access right into specific access right of a specific PaaS) and loops (e.g., loop to upload a file data as a set of blocks) and it should be small in size because it requires a manual update for its codes.

An example of utilities API, each PaaS provider has a specific API to represent access rights of a blob (e.g., the blob access rights is represented in GAE as String; while it is represented in Azure as Enum). So, to hide the heterogeneity of these access rights, a generic model for representing the blob access rights is designed. This generic model is defined in the utilities API and can be used by cloud developers to specify the access rights of a blob. Furthermore, a specific method, for each PaaS provider, is created to convert a given generic access right to its specific one. These specific methods are defined in the utilities API and are used in the adapter code generation process.
5.4.5 Semi-automatic Adapter Validation and Generation (SAVG)

The SAVG component is a service independent component and is considered one of the major components in STAGER because it overcomes the API dynamic adaptation problem [24]. The SAVG component is implemented independently from the generic APIs. So that it can generate the adapters for any semantically annotated generic API. It requires, as an input, the STAGE ontology. The SAVG component has two main functions:

- **Generic-API Validation**: it validates whether the desired PaaS platform supports all generic APIs that are used by cloud developers. If the validation fails, it will notify cloud developers with a list of generic methods which cannot be generated. Otherwise, it will execute the generic-API adapter generation.

- **Generic-API Adapter Generation**: after the generic-API validation succeeds, the SAVG component generates the source codes of a specific adapter for the desired PaaS platform. The specific adapter maps the generic methods, which are defined in the STAGE ontology, into their corresponding specific-API calls.

It should be noted that the SAVG component is a semi-automatic process because it depends on the utilities API and the semantic annotations of the PaaS APIs. Therefore, SAVG component requires that the inputs and outputs of the generic/specific methods are semantically annotated. The semantic annotations link the inputs and outputs to concepts in the STAGE ontology. For example, the STAGE ontology allows us to represent an input to a method as a file or an output of another method as a blob. These links are essential in the semantic search process. Finally, the SAVG component outputs a set of specific adapters for generic APIs. These adapters are linked to the application code developed by cloud developers who use the generic APIs to build their portable applications.

5.4.5.1 SAVG Architecture

The architecture of the SAVG component is illustrated in Figure 5.7. It presents the main inputs and output of the SAVG component. The SAVG requires, as inputs, a
specific PaaS provider name and the STAGER ontology. In addition, it outputs an adapter for the given specific PaaS provider.

The SAVG function (i.e., the semi-automatic adapter generation) is based mainly on searching the STAGER ontology. A generic API is composed of a set of generic methods. A specific API is also composed of a set of specific methods. Recall that the semantic annotation (to represent the concepts of the I/O parameters) of each generic method, as well as, each specific method are available and stored in the STAGER ontology.

Therefore, to generate the code of a generic API, a list of its generic methods and their semantic annotations are retrieved from the STAGER ontology by using the `genericMethodsList` method (elaborated in the next Sub-section). To generate the code of a generic method G, the SAVG component can automatically generate a SPARQL query, based on an initial one, to retrieve a sequence of specific methods, called `path`, that starts from the semantic concept of one of the G's input parameters and ends at the semantic concept of the G's output parameter. The path depth represents the number of specific methods that are returned after executing the generated SPARQL query.
The initial version of the SPARQL query \textit{(initialSPARQLQuery)} and its visual representation are depicted in Figure 5.8. Each generic method has an array, called \textit{semantic-Param}, to represent the semantic concepts of its input parameters and a variable, called \textit{semanticOutput}, to represent the semantic concept of its output parameter. The initial SPARQL query searches the STAGER ontology to find a path of depth one (i.e., path contains a single specific-API method) such that this specific method has the following:

1. A semantic input similar to one of the generic method’s semantic inputs, which is represented by the index \textit{startPoint} and is initialized by zero to point to the first input parameter.

2. A semantic output similar to the the generic method’s semantic output.

3. A provider name similar to the given specific-PaaS provider name.
It should be noted that this SPARQL query is a generic SPARQL query because it is independent of the function of a generic method (i.e., the same SPARQL query is used to generate the code of any generic method, e.g., a generic method for creating a blob, a generic method for deleting a datastore entity). However, this SPARQL query requires, as inputs, the semantic concepts of the I/O parameters of a generic method, as well as, a specific PaaS provider name for which a generic method will be implemented. Then, the initial SPARQL query is executed on the STAGER ontology. If no results are returned, a new SPARQL query is generated automatically using the automaticQueryGeneration method (elaborated in the next Sub-sections). However, if results are returned, this means that a path with depth one is detected, called detectionPath. A detection path represents a sequence of specific methods that links the semantic inputs and the semantic output of a generic method. Once a detection path is produced, the source code can be generated for this path using the generateCode4DetectionPath method (will be elaborated later).

5.4.5.2 The genericMethodsList method

This method retrieves, from the STAGER ontology, a list of generic methods for a specific service (e.g., CBSS or NSDS) by executing a SPARQL query on this ontology. Sample from this SPARQL query is illustrated in Figure 5.9. This SPARQL query provides all instances that are defined under the GenericAPI concept, as well as, the semantic annotations of the I/O parameters of each instance. The returned instances would represent the generic methods, which their source codes will be generated.

![Figure 5.9: A SPARQL query to retrieve a list of generic methods](image-url)
5.4.5.3 The automaticQueryGeneration Method

In a nutshell, the main objective of the automaticQueryGeneration method is to automatically generate a new SPARQL query based on a given SPARQL query by increasing the path depth of the given SPARQL query by one (specific-API method). If no results are returned from the newly generated query, the automaticQueryGeneration method continues generating another SPARQL query until either the generated SPARQL query succeeds (i.e., a path is detected) or the path depth reaches its maximum limit (maxDepth). Whenever a path depth reaches maxDepth, this means that it is not possible to link between the semantic of one of the generic method’s input parameters and the semantic of the generic method’s output parameter. In this case, the SPARQL query is initialized again to detect a path with depth one from another input parameter of the generic method. That is, it increases the startPoint index by one to link between the semantic of the next input parameter and the semantic output of the generic method. The details of the automaticQueryGeneration method are presented in Algorithm 1.

Algorithm 1: The algorithm of the automaticQueryGeneration method

Input:
- query ⊳ a previously generated SPARQL query
- providerName ⊳ the name of a specific PaaS provider

Output:
- newQuery ⊳ the newly generated SPARQL query

```java
1 part1 ← query.substring(0 : lastIndexOf('? '));
2 part2 ← query.substring(lastIndexOf('? '));
3 methodName ← part2.substring(0 : firstIndexOf(' space'));
4 count ← Integer.parseInt(lastMethodName.getLastChar());
5 par1+ = methodName + " scro:hasSemanticOutput ?out" + count + ":.";
6 count + + ;
7 newMethodName ← "?meth" + count ;
8 par1+ = newMethodName + " scro:hasSemanticInput ?out" +(count - 1) + ":.";
9 par1+ = newMethodName + " scro:hasProvider scro:" + providerName + ":.";
10 par2 = part2.replace(lastMethodName, methodName) ;
11 newQuery ← par1 + part2;
12 Return newQuery;
```
The input of the \textit{automaticQueryGeneration} method is a previously generated SPARQL query and a specific PaaS provider name. In the first execution of Algorithm 1, the \textit{automaticQueryGeneration} method takes the initial SPARQL query as input. The main function of this method is to generate a new SPARQL query by incrementing the path depth of the given SPARQL query to accommodate one more specific-API method. This new specific method has the following:

1. A semantic input similar to the semantic output of the last specific method in the given SPARQL query.

2. A semantic output similar to the semantic output of the generic method.

3. A provider name similar to the given specific-PaaS provider name.

A visual representation of a generated SPARQL query with depth two is depicted in Figure 5.10, which aims to find two methods such that:

- The first method has a semantic input similar to the semantic input of one of the generic method’s input parameters.

- The second method has a semantic input similar to the semantic output of the first method.

- The second method has a semantic output similar to the semantic output of the generic method.

- Both methods have the given specific-PaaS provider name.

Finally, the \textit{automaticQueryGeneration} method returns the newly generated query. Once the newly generated SPARQL query is executed on the STAGER ontology, if results are returned, this means that a path is detected (\textit{detectionPath}). However, if no results are returned from this query, the \textit{automaticQueryGeneration} method continues generating another SPARQL query until either the generated SPARQL query succeeds or the path depth reaches its maximum limit. Whenever a right path is detected, the \textit{generateCode4DetectionPath} method, which will be elaborated in the next Sub-section, is used to generate the code of the given generic method.
5.4.5.4 The \textit{generateCode4DetectionPath} Method

Whenever a generated SPARQL query is executed and results are returned, this means a path is detected. This detection path represents a sequence of specific methods which link between the semantic inputs and the semantic output of a generic method. Therefore, the \textit{generateCode4DetectionPath} method is used to generate the code of a generic method. The input of the \textit{generateCode4DetectionPath} method is a \textit{detectionPath}, which is an array including a set of specific-method names, and the name of the generic method for which the adapter code will be generated. The main objective of the \textit{generateCode4DetectionPath} method is to store the calling statement, of each specific method in the \textit{detectionPath}, in a buffer of the generated code. Finally, this buffer represents the output of the \textit{generateCode4DetectionPath} method.

Algorithm 2 clarifies the details of the \textit{generateCode4DetectionPath} method. As shown in this Algorithm, there is a loop on the \textit{detectionPath}, which generates a code for the calling statement of each specific method. This generated code is appended to a buffer. Lastly, a code for the return statement of the generic method is generated and appended to the buffer, which is the output of the \textit{generateCode4DetectionPath} method.
Chapter 5. The Proposed STAGER framework

Algorithm 2: The pseudocode of the generateCode4DetectionPath method

Input:
- detectionPath \(\triangleright\) a sequence of specific methods
- genericMethodName \(\triangleright\) the name of a generic method to generate its specific code

Output:
- generatedCode \(\triangleright\) a buffer stores the generated code of the given genericMethodName

1. \(buffer \leftarrow ""\);
2. for each method \(M\) in detectionPath ;
   3. \(tempCode \leftarrow callingStatement(M)\); \(\triangleright\) write a code to call \(M\)
   4. \(buffer + = tempCode;\)
   5. endfor ;
6. \(returnStmt \leftarrow returnStatement(genericMethodName)\);
   \(\triangleright\) write a code for the returned result from the genericMethodName
   7. \(buffer + = returnStmt;\)
8. Return \(buffer;\)

The overall algorithm of SAVG component is presented in Algorithm 3. It requires a specific PaaS provider name and the STAGER ontology. The algorithm starts by retrieving a list of generic methods that are defined in the STAGER ontology, then generating the adapter code for each generic method, and finally, the generated codes are returned.

5.5 Summary

In this Chapter, a framework for semantic-based generation of generic-API adapters for portable cloud applications, called STAGER, has been introduced. An overview about STAGER framework, its architecture, as well as, the process flow of its components have been elaborated. Furthermore, the STAGER assumptions about the PaaS specific-API, programming language, semantic annotation, and utilities API have been specified. Finally, the main components of STAGER framework have been elaborated.

Input:

- providerName ▷ the name of a specific PaaS provider
- Domain-specific ontology (STAGER ontology) ▷ represents the specially annotated ontology

Output:

- generatedAdapter ▷ a buffer stores the generated code for an adapter of the given providerName

```
genericMethodList ← genericMethodsList();
generatedAdapter ← "";
for each method M in genericMethodList;
    startPoint ← 0;
    ▷ it is used by the initialSPARQLQuery
    count ← Number of input parameters of M;
    genericMethodName ← getNameOf(M);
    ▷ get the Name of the method M
    while startPoint < count;
        query ← initialSPARQLQuery;
        detectionPath ← executeQuery(query);
        ▷ execute the given SPARQL query on STAGER ontology
        pathDepth ← 1;
        ▷ initial Path depth
        maxDepth ← 20;
        ▷ maximum Path depth
        while pathDepth < maxDepth and detectionPath = φ
            query = automaticQueryGeneration(query, providerName);
            detectionPath = executeQuery(query);
            pathDepth ++;
        end while;
        if detectionPath ≠ φ ▷ found a path
            buffer = generateCode4DetectionPath(detectionPath, genericMethodName);
            generatedAdapter += buffer;
        else ▷ try to relate between the next input parameter of a generic method and its output parameter
            startPoint ++;
            ▷ to get the next input parameter of the generic method M
        end if
    end while;
    if startPoint > count
        print("Failed in generating the code of the method:" + genericMethodName);
    end if
end for
if generatedAdapter ≠ ""
Return generatedAdapter;
end if
```
Chapter 6

Experimental Evaluation

In this Chapter, the details of the implementation of both the proposed generic APIs and STAGER framework will be presented in Section 6.1. Section 6.2 will present a case study to evaluate the extended COAPS API and two case studies for testing and evaluating STAGER framework. Finally, the evaluation results will be discussed in Section 6.3.

6.1 Implementation

In this Section, the implementation of the proposed generic APIs is presented. We focused on making both the generic PaaS implementation API and deployment API are independent and loosely coupled as possible. The main reason of this is to help any further enhancements on both APIs to be relatively easy and less complicated. In addition, the implementation details of the components of STAGER framework will be elaborated.

6.1.1 GAE-COAPS API Implementation

After extending the COAPS API, we can state that COAPS API provides three implementations, namely; CF-PaaS API, OS-PaaS API, and GAE-PaaS API to support the deployment on CF, OS, and GAE platforms, respectively. Similar to CF-PaaS API and OS-PaaS API, GAE-PaaS API is implemented using JAVA programming language and
is delivered as a RESTful web application (using Jersey JAX-RS library in the implementation). The implementation of GAE-PaaS API is divided into two main components; App Converter and GAE Adapter. The implementation of these components are elaborated in the following Paragraphs. Furthermore, GAE-PaaS API is now available as an open source through Git repository [90].

**App Converter Implementation**: it is implemented as a REST JAVA application in order to convert any JAVA web application into GAE application. The class diagram of this component is illustrated in Figure 6.1. App Converter component requires, as an input, a manifest XML file, which specifies the location of a given JAVA web application. In addition, this manifest specifies the GAE application ID that will be used later in the deployment process. App Converter uses Apache Ant\(^1\), which is a JAVA build tool, in order to repackage the given JAVA application to be in the same structure of GAE applications. Finally, App Converter outputs a new JAVA application that is ready to be deployed on GAE platform. A snapshot of the App-Converter I/O is depicted in Figure 6.2.

![Figure 6.1: App Converter Class Diagram](image)

**GAE Adapter Implementation**: it is implemented as a REST JAVA application by following the road map that is specified in Sub-section 4.1.2.3. Many classes are implemented to satisfy the required functions of the GAE Adapter. However, the two main classes are ApplicationManagerResource and EnvironmentManagerResource, which re-implement the COAPS generic methods that are specified in Table 2.5. A snapshot of the class diagram of the ApplicationManagerResource and EnvironmentManagerResource classes is illustrated in Figure 6.3.

\(^1\)http://ant.apache.org/
6.1.2 STAGER Implementation

In this Sub-section, the implementation of all components of STAGER framework will be discussed. This implementation is based on JAVA programming language. Recall that the components of STAGER framework is divided into three categories; service independent (e.g., SCOP and SAVG), service dependent (e.g., Std-PaaS APIs and utilities API), and service partially-dependent components (e.g., the STAGER ontology).
6.1.2.1 Implementation of Service Independent Components

In this Sub-section, the implementation of the service independent components of STAGER, namely; SCOP and SAVG, will be discussed.

**SCOP Component Implementation**

The SCOP component requires as inputs a PaaS API (either specific or generic) and its semantic annotation. The SCOP component provides two main functions, namely; automatic analysis of a PaaS API and semantic model of a PaaS API.

For the **automatic analysis**, a JAVA application is implemented which uses the reflection mechanism to automatically parse a given API. The parse operation gets more meta-data about each Object Oriented class from the given API. These meta-data include class name, its super type, its implemented interfaces (if exist), its defined methods with their categories (i.e., static, abstract, or instance method), their return types, and their exceptions that will be thrown (if exist).

For the **semantic model**, a JAVA application is implemented, called SourceCode2Ontology. This application uses the Jena JAVA API, which is a framework for developing semantic JAVA applications. The SourceCode2Ontology application stores, in the STAGER ontology, the previous parsed meta-data as ontological instances and their semantic annotation as relationships. By this way, a given API is converted to ontology.

**SAVG Component Implementation**

The implementation of the SAVG component is based mainly on the STAGER ontology. The SAVG component provides two main functions, namely; generic-API validation and generic-API adapter generation.

For the **Generic-API Validation**, a JAVA application, which uses Jena API and the SPARQL query language, is implemented to validate whether the target PaaS platform supports all generic APIs that are used by cloud developers. It requires, as inputs, the STAGER ontology and the provider name of a target PaaS to which a specific adapter is to be validated. The validation could only be failed in the case of there is no corresponding annotations of the target PaaS platforms. In this case, the SAVG
component will notify the developers with a list of generic methods that cannot be generated. Otherwise, it will execute the *Generic-API Adapter Generation*.

For the *Generic-API Adapter Generation*, a JAVA application, which uses Jena API and the SPARQL query language, is implemented to generate code for each generic method of a specific adapter. By this way, a specific adapter provides a specific implementation for a set of generic methods. To generate the code of a generic method, a set of SPARQL queries is automatically generated to detect a right path (i.e., a sequence of specific methods calls) which connects the semantic meaning of the input(s) and the output parameters of this generic method (See Algorithm 3 in Chapter 5).

### 6.1.2.2 Implementation of Service Dependent Components

In this Sub-section, the service dependent components of STAGER framework will be implemented by considering two services; CBSS and NSDS. Recall that we have designed the generic APIs for these two services because of most of software applications require file storage and/or data storage services. The service dependent components of STAGER include Std-PaaS APIs and utilities API. So, we will start by elaborating the implementation of the proposed generic APIs (i.e., Std-PaaS APIs). Then, the implementation of the utilities API for both CBSS and NSDS will be introduced.

**Std-PaaS APIs Implementation**

The Std-PaaS APIs are designed in an extendable way to include generic APIs for different PaaS services. Currently, they include generic APIs for two services; CBSS and NSDS. The prototypes of the proposed generic APIs for both CBSS and NSDS are designed based on studying the specific APIs of CBSS and NSDS for two PaaS platforms; GAE and Azure. We have selected these two PaaS platforms because they are considered two of the leading PaaS platforms [91]. In addition, both of them provide its own specific APIs for both CBSS and NSDS.

Currently, the Std-PaaS APIs help cloud developers to store large files remotely on a set of heterogeneous PaaS clouds, as well as, store different entities on heterogeneous cloud datastores. They are designed, based on JAVA programming language, to include a set
of generic methods. The generic API of CBSS includes the following generic methods that are elaborated in Sub-section 4.2.2.3:

- **authentication**: to authenticate a current user and make sure that he/she is authorized to execute blob operations.
- **createBlob**: to create a virtual blob inside a container.
- **uploadBlob**: to upload an input stream on an existed virtual blob.
- **downloadBlob**: to download a blob.
- **deleteBlob**: to delete a physical blob.
- **listBlobs**: to list all blobs that are available in a specific container.

Whereas the generic API of NSDS includes the following generic methods that are elaborated in Sub-section 4.2.3.3:

- **authentication**: to authenticate a current user and make sure that he/she is authorized to execute datastore operations.
- **createEntity**: to create an entity and store its data inside a datastore table.
- **retrieveEntity**: to get the data of a datastore entity.
- **deleteEntity**: to delete a datastore entity.

**Utilities API Implementation**

The utilities API stores a set of static source codes for each service which cannot be generated automatically and is needed during the generation of the specific adapters. Most of the utilities API involves type conversions and loops and it should be small in size because it requires a manual update of its codes. Since the utilities API is a service dependent component, it will be explored through two case studies for CBSS and NSDS.

1. **CBSS Utilities API Implementation**

A sample of the CBSS utilities methods with its semantic annotations is presented in Table 6.1. These methods are needed to implement the adapters for the CBSS generic API. For example, each PaaS platform provides a specific API for the blob access rights.
So, a generic model for the blob access rights is defined in the utilities API to hide heterogeneity of different access rights of blobs. In addition, a specific method (e.g., `getAccessRight4Azure` and `getAccessRights4GCS`) is defined for each PaaS platform to convert a generic access right into its corresponding one.

Furthermore, each PaaS platform provides a specific API to list blobs. Therefore, a generic blob is defined, as utilities API, to store the name and the URI of each blob in a blob list. In addition, a specific method (e.g., `listBlobs4Azure` and `listBlobs4GCS`) is defined for each PaaS platform to convert a specific blob list into a generic blob list.

### Table 6.1: The required Utilities API of CBSS with its Annotations

<table>
<thead>
<tr>
<th>Method Prototype</th>
<th>Semantic Input(s)</th>
<th>Semantic Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>copy(java.io.InputStream, java.io.OutputStream)</code></td>
<td>1. Input Stream object</td>
<td>Upload Status flag</td>
</tr>
<tr>
<td><code>getAccessRights4Azure(GenericAccessRights)</code></td>
<td>1. Access Rights object</td>
<td>Virtual Container Access Rights</td>
</tr>
<tr>
<td></td>
<td>2. Virtual Container Not Public object</td>
<td></td>
</tr>
<tr>
<td><code>getAccessRights4GCS(GenericAccessRights)</code></td>
<td>1. Access Rights object</td>
<td>Virtual Container Access Rights</td>
</tr>
<tr>
<td></td>
<td>2. Cloud File Builder Not Configured object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Container Name String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Container Name String</td>
<td></td>
</tr>
</tbody>
</table>

### 2. NSDS Utilities API Implementation

A sample of the NSDS utilities methods with their semantic annotations is presented in Table 6.2. These methods are needed to implement the adapters for the NSDS generic API. For example, each PaaS platform provides a specific API for representing a datastore entity. So, a generic entity is defined, as utilities API, to store all data of an entity, such as table name, row key, partition key, and properties.
In addition, a specific method (e.g., retrieveGenericEntity4Azure and retrieveGenericEntity4GAE) is defined for each PaaS platform to convert a specific datastore entity into a generic entity. Furthermore, a specific method (e.g., getDynamicEntity4Azure and getDynamicEntity4GAE) is defined for each PaaS platform to create a specific entity for a specific PaaS platform.

### Table 6.2: The required Utilities API of NSDS with its Annotations

<table>
<thead>
<tr>
<th>Method Prototype</th>
<th>Semantic Input</th>
<th>Semantic Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>retrieveGenericEntity4Azure (com.microsoft.azure.storage.table.TableResult, java.lang.String, java.lang.String, java.lang.String)</td>
<td>1. Table Result object 2. Table Name string 3. Partition key string 4. Row key string</td>
<td>Generic Entity object</td>
</tr>
<tr>
<td>retrieveGenericEntity4GAE (com.google.appengine.api.datastore.Entity)</td>
<td>1. GAE specific entity object</td>
<td>Generic Entity object</td>
</tr>
<tr>
<td>getDynamicEntity4GAE (java.lang.String, java.lang.String, java.util.HashMap)</td>
<td>1. Table name string 2. Row key string 3. Properties map</td>
<td>A specific entity for GAE</td>
</tr>
</tbody>
</table>

### 6.1.2.3 Implementation of Service Partially-Dependent Components

STAGER framework provides only one service partially-dependent component, which is the STAGER ontology. The STAGER ontology is implemented, as an OWL ontology, using protégé ontology editor. Recall that the STAGER ontology is composed from three main ontologies; SCRO+, domain-specific, and Std-PaaS APIs ontologies. These ontologies are categorized into service independent and service dependent ontologies. So, the implementation of the service dependent ontologies will be illustrated through the case studies of CBSS and NSDS.
Furthermore, the STAGER ontology includes instances from utilities API, PaaS specific-APIs, and generic APIs. Recall, Table 6.1 and Table 6.2 specify the semantic annotations for utilities API. Later on, the semantic annotations for PaaS specific-APIs will be presented in Table 6.3 and the semantic annotations for PaaS generic-APIs will be presented in Table 6.4.

**SCRO\(^+\) Ontology Design**

SCRO\(^+\) ontology is a service independent ontology. It is an enhanced version of SCRO ontology [47]. To get SCRO\(^+\) ontology, the prot\'\-g\'\`e editor is used to update SCRO ontology with two new object properties, \textit{hasSemanticInput} and \textit{hasSemanticOutput}, and one data property, \textit{hasSignatureSemantic}. These new properties help the specific-API annotator and the generic-API designer to specify the semantic annotations of the I/O parameters of specific, generic, or utilities API.

In the CBSS case study, the semantic annotations of both GCS API and Azure storage API are defined. Table 6.3 provides example to specify the semantic annotations of two instances; one from GCS API and another from Azure storage API. For each specific method, the semantic annotation of its input (\textit{Semantic Input}) and output (\textit{Semantic Output}) are defined.

**Table 6.3: Example for Adding Semantic Annotations to some PaaS specific-APIs**

<table>
<thead>
<tr>
<th>Specific Method Prototype</th>
<th>Semantic Input</th>
<th>Semantic Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{com.microsoft.azure.storage.CloudStorageAccount.parse(java.lang.String)}</td>
<td>\texttt{storageConnectionString}</td>
<td>StorageAccount</td>
</tr>
<tr>
<td>\texttt{com.google.appengine.tools.cloudstorage.GcsServiceFactory.createGcsService(com.google.appengine.tools.cloudstorage.RetryParams)}</td>
<td>\texttt{manifestString}</td>
<td>StorageAccount</td>
</tr>
</tbody>
</table>

**Domain-specific Ontology Design**

The domain-specific ontology is a service dependent ontology because it defines all vocabularies that are needed to satisfy a specific service. Therefore, it is required to create a domain-specific ontology for each service. In our case studies, two domain-specific ontologies are defined for CBSS and NSDS (see Figure 5.5).
Std-PaaS APIs Ontology Design

The Std-PaaS APIs ontology combines a set of service dependent ontologies. Each one of these service dependent ontologies stores the prototypes of a set of generic methods, as well as, their semantic annotations. These generic methods are used to satisfy a specific service (e.g., blob storage, datastore, and messaging). Recall that the semantic annotations of a generic method are used to describe the semantic meaning of the I/O parameters of this generic method. These annotations are defined by specifying the semantic of the input(s) and the output parameters of a generic method using the object properties `hasSemanticInput` and `hasSemanticOutput`, respectively. In our case studies, the prototypes of the CBSS generic-APIs and NSDS generic-APIs, with their annotations, are stored as a part of the Std-PaaS APIs ontology. The semantic annotations of the CBSS generic methods are presented in Table 6.4; whereas the semantic annotations of the NSDS generic methods are presented in Table 6.5.

**Table 6.4: Semantic annotations of the CBSS generic-API**

<table>
<thead>
<tr>
<th>Generic Method Prototype</th>
<th>Semantic Input(s)</th>
<th>Semantic Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Authentication</strong></td>
<td>1. Manifest String Storage Account Object</td>
<td></td>
</tr>
<tr>
<td><strong>Upload Blob</strong></td>
<td>1. Virtual Blob Object 2. Input Stream Object Upload Status Flag</td>
<td></td>
</tr>
<tr>
<td><strong>Delete Blob</strong></td>
<td>1. Storage Account Object 2. Container Name String 3. Blob Name String Delete Status Flag</td>
<td></td>
</tr>
<tr>
<td><strong>List Blobs</strong></td>
<td>1. Storage Account Object 2. Container Name String Blobs List Object</td>
<td></td>
</tr>
</tbody>
</table>
Table 6.5: Semantic annotations of the NSDS generic-API

<table>
<thead>
<tr>
<th>Generic Method Prototype</th>
<th>Semantic Input(s)</th>
<th>Semantic Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AuthenticateDS</strong></td>
<td>1. Manifest String</td>
<td>Datastore Account Object</td>
</tr>
<tr>
<td></td>
<td>1. Datastore Account Object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Table Name String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Partition Key String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. Row Key String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5. Properties Map</td>
<td></td>
</tr>
<tr>
<td><strong>Create Entity</strong></td>
<td>1. Datastore Account Object</td>
<td>New Entity Object</td>
</tr>
<tr>
<td></td>
<td>2. Table Name String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Partition Key String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. Row Key String</td>
<td></td>
</tr>
<tr>
<td><strong>Retrieve Entity</strong></td>
<td>1. Datastore Account Object</td>
<td>Retrieved Entity Object</td>
</tr>
<tr>
<td></td>
<td>2. Table Name String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Partition Key String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. Row Key String</td>
<td></td>
</tr>
<tr>
<td><strong>Delete Entity</strong></td>
<td>1. Datastore Account Object</td>
<td>Delete Status Flag</td>
</tr>
<tr>
<td></td>
<td>2. Table Name String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Partition Key String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. Row Key String</td>
<td></td>
</tr>
</tbody>
</table>

6.2 Case Studies

In this Section, a deployment case study is presented to evaluate the extended COAPS API. In addition, two case studies; CBSS and NSDS; are presented for testing STAGER framework by generating adapters for the generic APIs of these two case studies.

6.2.1 Deployment Case Study

Sellami et al. [21] have created a generic web client to use COAPS API to test the deployment and management of applications on CF and OS platforms. So, the same web client is used to test the deployment of applications on GAE platform. The sequence diagram to deploy an application using COAPS API with the proposed GAE extension is clarified in Figure 6.4. The sequence diagram follows the same steps described in Subsection 2.2.4.3 besides the interactions with the proposed App Converter component. A snapshot of the web client of COAPS API is depicted in Figure 6.5.
To deploy an application through the COAPS API, an application manifest needs to be identified based on the PADM of COAPS API. The application manifest that is clarified in Figure 2.4 is used in the testing process. It should remember that this application manifest is independent on the target PaaS platform (GAE in this case). Besides the application manifest, the source archive of the application is needed to start...
the deployment process. Now, the deployment process is ready to be started as follows:

1. The developer specifies the URL of the target PaaS platform (i.e., GAE-PaaS API) that will be used by COAPS API to deploy applications on that platform.

2. The developer selects, from the Action menu, *Create Environment* method and supplies the environment manifest (see Figure 6.6) in the Request Body section. An XML, which represents the newly created environment, with a unique *envID* is returned.

3. The developer selects, from the Action menu, *Create Application* method and supplies an application manifest that includes, among others, the path of an application source archive (see Figure 6.7). The *create application* method returns an XML, which represents the newly created application, with a unique *appID*.

4. The developer selects *Deploy Application* method and specifies the previously created *envID* and *appID*. The *deploy application* method returns the XML application descriptor, which includes a URL to run the deployed application (Figure 6.8).

5. Once an application is deployed, it is become ready to execute. The developer can use the previously returned URL to execute the application.

![Figure 6.6: Create Environment](image-url)
The case study application, which is used to test GAE-COAPS API, is a JAVA web application for Bot Trace Back (BTB) that has been implemented in the EASI-CLOUD international project [92]. A snapshot when deploying the BTB application on GAE using the GAE-COAPS API is depicted in Figure 6.9. Another snapshot for the same
application after deploying on CF using similar steps is illustrated in Figure 6.10. Un-
fortunately, GAE-PaaS API did not support methods to allow developers to manage
(restart, stop, and delete) their deployed applications. However, they can manage their
applications on-line from GAE dashboard.

Figure 6.9: A sample application deployed on GAE using COAPS API + GAE
extension

Figure 6.10: A sample application deployed on CF Platform
6.2.2 Adapter Generation Case Study

Since there is no open source generic implementation-APIs for PaaS platforms, two generic APIs for two main services, CBSS and NSDS, are proposed as a part of the Std-PaaS APIs. The specific adapters for these generic APIs are generated using the SAVG component for two PaaS platforms; GAE and Windows Azure. We have selected these two PaaS platforms because of two reasons; (1) GAE and Azure are considered two of the leading PaaS platforms [91] and (2) both of them provide its own specific APIs for both CBSS and NSDS. For example, if cloud A has provided searching APIs which are not available in cloud B. Then, STAGER framework cannot generate the adapter for cloud B.

The following steps are followed to generate the adapters of the proposed generic APIs for both GAE and Windows Azure platforms:

1. Both of the prototypes of the generic methods and their semantic annotations are inserted in the STAGER ontology.

2. The specific APIs of CBSS and NSDS for both GAE and Azure platforms are analyzed by SCOP component. Then, the analyzed data, as well as, their semantic annotations are inserted into the STAGER ontology.

3. The STAGER ontology is fed, as an input, to the SAVG component.

4. To generate a specific adapter for GAE, the SAVG is executed and provided with GAE as the provider name. At the end of this step, the GAE adapters for the CBSS generic-API, as well as, for the NSDS generic-API are generated.

5. To generate a specific adapter for Azure, the SAVG is executed again with Azure as the provider name. At the end of this step, Azure adapters for the CBSS generic-API, as well as, for the NSDS generic-API are generated.

Whenever a PaaS specific-API is updated, its corresponding adapter can be re-generated by executing the previous steps such that the updated version of the PaaS specific-API and its semantic annotations should be entered into the STAGER ontology. By this way, the API dynamic adaptation problem has been overcome to a large extent.
To clarify the STAGER feasibility, we will discuss the process of generating code of a generic method (e.g., createBlob) for a specific adapter (e.g., Azure adapter). Figure 6.11 shows a visual representation of a subset of Azure storage API with its semantic annotations. According to this Figure, each specific method is presented by oval and it has two input/output arcs. The input arc to each method represents its semantic input; whereas the output arc from each method represents its semantic output. In addition, a specific class is represented by circle.

![Figure 6.11: The path detection for the createBlob generic method for Azure adapter](image)

The semantic annotations of the createBlob generic method is clarified in Figure 6.12. This method requires several input parameters, which are specified in Table 6.4, with semantic annotations as storageAccount, containerName, blobName, etc. While it outputs an object with semantic annotations as newBlob. To generate the code of the
createBlob generic method for Azure adapter, a SPARQL query is automatically generated to search about a specific method that has a semantic input storageAccount and has a semantic output newBlob, as well as, it belongs to Azure provider. Unfortunately, as it appears in Figure 6.11, the method createCloudBlobClient has a semantic input storageAccount, but it does not have a semantic output newBlob. Therefore, the previous SPARQL query fails in its execution and is updated by automatically increasing its path depth by another method. Now, the new SPARQL query searches about two specific methods such that the first method has a semantic input storageAccount and has any semantic output X; and the second method has a semantic input X (i.e., same as the semantic output of the first method) and has a semantic output newBlob. In addition, the two methods are belonging to Azure provider. The search results found the first method createCloudBlobClient has a semantic input storageAccount and has a semantic output blobClient. However, the search results found that the second method getContainerReference has a semantic input blobClient and does not have a semantic output newBlob. Therefore, the process of automatically generating a SPARQL query will continue until it succeeds at a path depth equals seven, which is represented by bold arcs in Figure 6.11. In addition, the seven ovals present the specific methods that form the source code of the createBlob generic method.

After a right path is detected, the sequence of the specific methods in that path is called to form the generated source code of a specific adapter. Sample of the generated adapters for the CBSS generic-API and NSDS generic-API for Azure and GAE platforms are available on GitHub [93, 94]. These generated adapters can be used by cloud developers to implement portable applications with a blob storage or NoSQL datastore services.
6.3 Evaluation and Discussion

A quantitative evaluation is executed to specify the metrics that are used to measure the performance of STAGER framework and to identify the factors that affect the performance of this framework. The metrics which are used to measure the STAGER framework performance are: (1) the time consumed by each component (SCOP and SAVG); (2) the time consumed by the final application (execution time) using the generated adapter vs. the specific API; (3) the size of the utilities API; and (4) the manual effort that is needed to semantically annotate the different APIs (specific, generic, or utilities). Also, several factors are identified that may affect the system performance, such as the PaaS specific-APIs, the nature of the different PaaS APIs (e.g., blob storage API, datastore API, etc.), and Std-PaaS APIs definition.

6.3.1 Performance Metrics

A computer with Intel(R) Core(TM) i5-3210M CPU @ 2.50GHZ, 4.00 GB RAM, and 64-bit operating system is used to measure the performance of STAGER framework.

The SCOP component performance; the SCOP component is executed to parse a set of different libraries with different sizes and the average parsing time is computed after executing the parsing process of each library 20 times. Figure 6.13 presents the average parsing time (in seconds) for these libraries vs. number of parsed classes. According to this Figure, there is a linear relation between the average parsing time and the number of parsed classes. Note that the confidence interval does not appear in this Figure because it is very small and it is less than 0.389. Figure 6.14 shows that there is a linear relation between the size of the generated ontology (in kilobytes) and the average parsing time of these libraries.

The SAVG component performance; the SAVG component is tested by generating the adapters of the CBSS and NSDS generic-APIs for both Azure and GAE platforms. The average time for generating the code of each generic method is computed by executing the SAVG component 20 times for each method. Figure 6.15 clarifies the average code generation time (in milliseconds) of the CBSS generic-APIs for both GAE and
Azure adapters. It should be noted that the variance of the average code generation time is directly proportional with several points as follows:

1. **The number of failed trials to generate a path.** A trial starts with one input parameter and increases a path depth until it succeeds or hits the max depth then
starts a new trial from the next parameter. For example, in the method `createBlob` for Azure adapter, it fails in the first trial to generate a path which starts from `contentType` and ends with `newBlob`. So, the SAVG component starts a new trial to generate a path which starts from `storageAccount` and ends with `newBlob` and it succeeds in this trial.

2. The length of a successfully generated path. The generated path represents a set of specific methods that are needed to be called. Therefore, the generation time increases as the number of the specific methods in the generated path increases.

3. Total access to ontology when calling each specific method. For example, to call a specific method which is neither constructor nor static method, we need to create an instance of its parent to be used to call this method. In this case, we need to inquire the ontology about how to create an instance of type equals to the parent of the specific method and has the target provider. In addition, if one or more parameters of a specific method are not available, then we need to inquire the ontology about how to create an instance with the same semantic of a missed parameter. For example, one can create an instance of `RetryParams` class using the static method `getDefaultInstance`.

![SAVG Component](image_url)

**Figure 6.15:** The Avg. code generation time of CBSS generic-APIs for both GAE and Azure
Measuring the execution time using generic APIs vs. specific APIs; Two generic applications are implemented; one that uses the CBSS generic-APIs and the other uses the NSDS generic-APIs. The blob storage generic application helps users to upload files, of any type, remotely on the cloud. The NoSQL datastore generic application allows users to insert entities, of any type, with their properties as a datastore on the cloud. The main advantage of these generic applications is that they are implemented once and deployed on GAE and Windows Azure with no change in the source code. However, only the class path of these generic applications needs to be configured to use a corresponding adapter, when deploying these generic applications on a specific PaaS platform. These generic applications are deployed on both GAE and Azure platforms. In addition, four other specific applications are implemented as follows:

- Application uses GCS API.
- Application uses Azure blob storage API.
- Application uses GAE datastore API.
- Application uses Azure datastore API.

We will explore the results when using a specific adapter (e.g., GAE adapter) versus the results when using GAE specific-APIs for both CBSS and NSDS. We measured the execution time of all generic and specific methods. In particular, the JAVA timer (i.e., `System.nanoTime()`) is used to get the time before and after the calling of each method. Next, these times are subtracted and divided by 1000000 to determine the execution time of each method in milliseconds.

For the evaluation of the blob storage, we used one blob file of type `pdf` with size 1.51MB to be (uploaded, downloaded, deleted, etc.) on a specific bucket that already contains ten blob files. The execution time is measured when executing all the specific and the generic methods ten times using the same blob file and the same bucket. For the evaluation of the datastore, we used an entity with specific name, key, and three properties (e.g. name, id, and gpa) to be (created, retrieved, and deleted) from GAE datastore. The execution time is measured when executing all the specific and the generic methods ten times using the same entity.
Figure 6.16 and Figure 6.17 clarify that there are no statistically significant differences between the average time using both (generic and specific) APIs which means that there is no significant overhead when using the adapters. Also, Figure 6.16 and Figure 6.17 show large values for confidence interval this is because of the large variation in the execution time of each method. This large variation is because all these methods are executed on the cloud. Therefore, the execution time is directly proportional to the network bandwidth and the size of the input. For example, uploading large files to the cloud will consume more time than uploading small files. Also, creating entity that contains large number of properties will consume more time than creating entity with small number of properties.

![Figure 6.16: Execution Time of CBSS Generic-API vs. CBSS Native-API of GAE](image1.png)

![Figure 6.17: Execution Time of NSDS Generic-API vs. NSDS Native-API of GAE](image2.png)
Chapter 6. *Experimental Evaluation*

**Measuring the size of the utilities API:** since the utilities API represents a set of static codes which cannot be generated, it is considered an overhead when using STAGER framework. Therefore, we measured the size of the static Utilities codes (by counting no. of lines) and found it indeed significant (∼ 35% of total adapter code). However, most of the static codes involves type conversions and loops, which can be automated or at least hinted to the generic-API designer. The relaxation of this assumption is a subject of future work. The analysis of the Utilities code is discussed in Table 6.6.

**Table 6.6:** The size of the static utilities API vs. the size of the generated adapters

<table>
<thead>
<tr>
<th></th>
<th>No. of lines of code</th>
<th>Utilities API (static code)</th>
<th>Generated Adapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>- For Google Cloud Storage (GCS)</td>
<td>35 (37.6%)</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>- For Azure Blob Storage</td>
<td>48 (35.2%)</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>- For GAE Datastore</td>
<td>17 (36.1%)</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>- For Azure Datastore</td>
<td>31 (31.6%)</td>
<td>98</td>
<td></td>
</tr>
</tbody>
</table>

**The manual effort of the APIs semantic-annotation:** in order to use STAGER framework, different APIs (specific, generic, and utilities) need to be semantically annotated. The APIs semantic-annotations are done once, and then, they are used to generate the adapters for several generic methods. Whenever the specific APIs are changed (e.g., adding a new parameter to a specific method), the STAGER ontology is updated only with new changes and this update is done once. After that, the SAVG component is executed again to generate a new adapter. On the other hand, if the adapter is implemented manually, then we need to update the code in all the generic methods that call this specific method. Therefore, the needed updates of the semantic annotations are done once, whereas the number of updates in the manual adapters can be as many as the number of generic APIs that use the proprietary APIs.

Although the semantic annotations are manually done at this stage, still the needed effort to annotate the whole APIs can be in some cases less than the effort for manually implementing the adapters and updating it whenever the specific APIs are changed. For example, consider the case of multiple generic APIs (e.g., used by multiple software companies). Each generic API would entail the development of adapters to mediate between the generic-API calls and the specific-API calls. Moreover, each change in
the specific API would entail adjusting the adapters for all generic APIs. In this case, instead of adjusting multiple adapters, only the specific APIs need to be annotated once per change.

### 6.3.2 Factors Affecting Performance

Three main factors are defined which may affect the previously mentioned metrics; the PaaS specific-API, the nature of the different service-based PaaS-APIs (e.g., blob storage API, message API, etc.), and the definition of the Std-PaaS APIs.

For a **PaaS specific-API**, if a PaaS specific-API is modified or updated, then the updated version of the specific API should be entered, as an input, to the SCOP component and its semantic annotation should be added to the STAGER ontology. By increasing the number of classes in a PaaS specific-API, the time consumed by the SCOP component is increased.

For the **nature of the different service-based PaaS-APIs**, the implementation of SCOP and SAVG components are considered independent from any PaaS API. We already tested the SCOP component with six different APIs and it succeeded in parsing them and inserting the parsed data into the STAGER ontology. In addition, the SAVG component is tested with the CBSS generic-API and the NSDS generic-API. It succeeded in generating the adapters of these generic APIs for two of the leading PaaS platforms; GAE and Azure. Moreover, the SAVG component can generate the adapters for any semantically annotated generic APIs. In the future, we plan to use the SAVG component to generate adapters for other generic APIs.

For the **definition of the Std-PaaS APIs**, this definition is dependent on the semantic annotations of the different PaaS specific-APIS. For example, the `downloadBlob` generic method has semantic inputs `storageAccount` and `containerName` and has semantic output `downloadStatus`, which is a boolean variable. So, to implement the code for the `downloadBlob` generic method, the SAVG component must find a set of specific methods, which have semantic input `storageAccount` or `containerName` and have semantic output `downloadStatus`. Otherwise, it fails in generating the code of this generic method.
In addition, the parameters of each generic method in the Std-PaaS APIs are ordered such that the first parameter is used to detect the right path quickly. For example, the listBlobs generic method has two input parameters with semantics storageAccount and containerName respectively, and has one output parameter with semantic blobList. Its main objective is to find a set of specific methods which have semantic input storageAccount and have semantic output blobList and this is already defined in both GAE and Azure APIs. However, if the two input parameters of the listBlob method are swapped, its main objective will become finding a set of specific methods which have semantic input containerName and have semantic output blobList and this is not defined in the case of GAE API (i.e., there is no path that can link between a containerName and a blobList). So, the SAVG will fail in this trial and will make another trail with the second input parameter.

6.4 Summary

In this Chapter, the implementation details of both the proposed generic APIs (COAPS, CBSS, and NSDS) and STAGER framework have been elaborated. In addition, a case study to evaluate the extended COAPS API and another two case studies for testing and evaluating STAGER framework have been presented. Finally, the evaluation results have been discussed.
Chapter 7

Conclusion and Future Work

In this Chapter, the conclusion and remarks of the work in this thesis followed by some suggestions for future research will be discussed.

7.1 Conclusion

Throughout this thesis, we showed how the PaaS vendor lock-in problem hinders cloud adoption and prevents cloud developers from migrating their applications among heterogeneous clouds. In addition, the currently available solutions, which provide generic APIs to solve the vendor lock-in problem, are suffering from API dynamic adaptation problem. This is because these solutions provide static adapters for their generic APIs. Thus, updating the specific APIs may require re-factoring or even re-implementation of its corresponding adapter.

Therefore, Std-PaaS APIs have been proposed to overcome the PaaS vendor lock-in problem. These APIs can include a set of service-based generic-APIs. The Std-PaaS APIs work as a middleware to hide the heterogeneity of service APIs of different PaaS platforms. By this way, cloud developers can use the Std-PaaS APIs to implement applications once and deploy them on multiple heterogeneous PaaS platforms. As case studies, we have designed generic APIs for two services; Cloud Blob Storage Service (CBSS) and NoSQL Datastore Service (NSDS). We have selected these two services because of most of software applications require file storage and/or data storage. The
proposed generic APIs of both CBSS and NSDS are included as a part of the Std-PaaS APIs, which can be extended to include generic APIs for other services.

In addition, STAGER framework for semantic-based generation of generic-API adapters for Portable Cloud Applications has been proposed. STAGER overcomes the API dynamic adaptation problem by semi-automatically generating the adapters for any semantically annotated generic APIs. The adapter generation process is not dependent on PaaS in particular but can be applied to any semantically annotated generic APIs. By this way, the generic APIs can be imported from elsewhere provided that they are semantically annotated. However, to the best of our knowledge, we could not find PaaS generic implementation-APIs as an open source. So, the Std-PaaS APIs, which currently include generic APIs of both CBSS and NSDS, are used to test and evaluate STAGER framework. STAGER has succeeded to generate the adapters of the Std-PaaS APIs for two PaaS platforms; GAE and Azure. We have selected these two PaaS platforms because they are considered two of the leading PaaS platforms. Furthermore, both of them provide its own specific APIs for both CBSS and NSDS.

In order to evaluate the generated adapters, We have implemented two generic applications (one for CBSS and the other for NSDS) that use generated adapters and implemented another four applications that use the native APIs (GCS, Azure storage, GAE datastore, and Azure datastore). Then, the execution time is measured when executing all the specific and the generic methods ten times using the same data. We found that there is no significant overhead when using the adapters. Thus, the evaluation results prove the feasibility of STAGER and promote the usage of the generated adapters for implementing portable cloud applications.

It should be noted that STAGER has the following two limitations:

- **The adapter generation depends on the utilities API**, which is static codes that cannot be generated by STAGER. So, we have counted no. of lines of the static Utilities codes and found it indeed significant (\( \sim 35\% \) of total adapter code). However, most of the static codes involves type conversions and loops, which can be automated in future work.

- **The semantic annotations of the generic/specific APIs**. These annotations are not available and need to be executed by an annotator. However, the
APIs semantic-annotations are done once, and then, they are used to generate
the adapters for several generic methods. In addition, whenever the specific APIs
are changed (e.g., adding a new parameter to a specific method), the STAGER
ontology is updated only with new changes and this update is done once. After
that, STAGER framework is executed again to generate a new adapter. On the
other hand, if the adapter is implemented manually, then we need to update the
adapter source codes in all the generic methods that call this specific method.
Therefore, the needed updates of the semantic annotations are done once, whereas
the number of updates in the manual adapters can be as many as the number of
generic APIs that use the proprietary APIs.

By this way, STAGER is considered a step towards reducing the effort of manually im-
plementing the adapters and updating it whenever the PaaS specific-APIs are changed.

In addition, the COAPS generic deployment API has been extended to allow deployment
of applications on GAE, besides CF and OS. The COAPS API helps cloud developers
to deploy and manage their applications independently of any PaaS platform. So, it has
been extended to include the following:

- Application deployment on GAE PaaS platform, besides CF and OS.

- Application repackaging for GAE platform to allow the same application to be
deployed on multiple PaaS platforms (i.e., GAE, CF, and OS) with no change in
neither the application source codes nor the application configuration.

The extended COAPS-API has been evaluated using a JAVA web application for Bot
Trace Back (BTB), which has been implemented in the EASI-CLOUD international
project. By this case study, the Extended COAPS API has succeeded to deploy the
BTB application on both GAE and CF with no change of its source codes.

7.2 Future Work

The Std-PaaS APIs are service dependent and the policy of designing generic APIs of a
specific service depends on many factors, such as service popularity, available features,
the nature of applications and use cases, etc. STAGER framework provides the mechanism that uses such generic APIs once designed. As a future work, STAGER framework can be combined with a mechanism to identify the most requested services. In addition, STAGER can be combined with a recommendation mechanism to identify the optimal PaaS providers based on big data analysis.

In the future, the proposed Std-PaaS APIs can be updated to provide generic APIs for other services, besides CBSS and NSDS. In addition, the Std-PaaS APIs can be implemented as stateful and RESTful API, beside the stateless and API wrapper. Moreover, generating polyglot adapters for the Std-PaaS APIs (i.e., generating adapters with other programming languages, such as Python and PHP, besides JAVA). In addition, STAGER framework can be used to generate adapters for other PaaS platforms, besides GAE and Azure.

Moreover, overcoming the STAGER limitations by providing an automatic approach to semantically annotate the heterogeneous PaaS APIs and automatically generate the utilities API. Text mining techniques can be combined with this automatic approach in order to analyze the documentations of the heterogeneous PaaS APIs and extract the concepts that can be used to annotate these heterogeneous APIs.
References


References


References

114


References


References


Thesis Publications

Article 1:


Article 2:


Article 3:


Article 4:

Article 5:

Eman Hossny, Sherif Khattab, Fatma Omara, and Hesham Hassan. STAGER: A framework for semantic-based generic-API adapters generation for portable cloud applications. [Submitted for publication].
معالجة عدم تجانس بنية التشغيل
في الحوسبة السحابية

إعداد
إيمان حسنى عبدالغني

رسالة مقدمة كجزء من متطلبات الحصول على درجة الدكتوراه في علوم الحاسب

تحت إشراف
أ. د./ فاطمة عبدالستار عمارة
أ. د./ هشام أحمد حسن
د./ شريف عصام الدين خطاب

قسم علوم الحاسب
كلية الحاسبات والموارد
جامعة القاهرة
جمهورية مصر العربية
سبتمبر 2017
ملخص الرسالة

إن بنية التشغيل (PaaS) التي تقدم خدمات من خلال الحوسبة السحابية تعاني من مشكلة عدم تجانس واجهات برامج التطبيقات (APIs) مما يجعل مطورات التطبيقات مضطرين لبناء تطبيقاتهم باستخدام بنية تشغيل معينة والذي سيؤدي إلى عدم قدرة مطورات التطبيقات على نقل تطبيقاتهم بسهولة بين بنين التشغيل المختلفة. ونتيجة لذلك ظهرت مشكلة الارتباك بالطبع لبرامج التطبيقات إحدى الحلول لهذه المشكلة. حيث أن الواجهات العامة لبرمجة التطبيقات تقدم مجموعة من الخدمات الخاصة التي تساعد مطورات التطبيقات في بناء تطبيقات قابلة للنقل بين بنين التشغيل الغير متجانسة. وبالرغم من ذلك فإن الواجهات الخاصة تعاني من مشكلة عدم الكيف الديناميكي لواجهات برمجة التطبيقات (API dynamic adaptation) حيث أن تعديل في الواجهات الخاصة لبرمجة التطبيقات يجعل هذه الواجهات غير قابلة للالخدام. ولذلك فإن هذه الرسالة العلمية تقدم الإطار الذي يمكن أن STAGER التوليد الأوتوماتيك للمحولات الخاصة للواجهات العامة لبرمجة التطبيقات. ويشترط الإطار أن تكون الواجهات العامة لبرمجة التطبيقات موصفة باستخدام تطبيقات الدالة (Semantic Annotations).

بالإضافة إلى ذلك، فإن هذه الرسالة توفرtypescript واجهات عامة لبرمجة التطبيقات يطلق عليها والتي يمكن أن يستخدمها مطورات التطبيقات لبناء تطبيقات عامة. والميزانية الرئيسية في بناء التطبيقات العامة Std-PaaS APIs هي تساعد مطورات التطبيقات على نشر نفس التطبيق في بنين تشغيل مختلفة. تحتوي مجموعة من الواجهات العامة لبرمجة التطبيقات لكل خدمة من خدمات بنية التشغيل. حاليا تقدم الواجهات العامة لبرمجة التطبيقات خدمات تكميل ومراقبة خدمات تشغيل الرسالة على الحوسبة السحابية. وعلاوة على ذلك، تقدم هذه الرسالة فهم إحدى الواجهات العامة لنشر التطبيقات والتي يطلق عليها COAPS-API (Cloud Foundry and OpenShift). Lكي تتشمل على بنية تشغيل جديدة (Cloud Foundry and OpenShift).

كلاً بجانب بنية التشغيل الموجودة بها حاليا (Engine(GAE and Azure) تساعد مطورات التطبيقات على نشر تطبيقاتهم في بنين تشغيل غير متجانسة. وقد تقويم الإطار STAGER بعض المجهد الدور لتبادل المعلومات الدلالية لواجهات برمجة التطبيقات إلا أن النتائج قد أظهرت أن الإطار STAGER قابل لتوئيد المتطلبات الخاصة للواجهات العامة لبرمجة التطبيقات والتي يمكن أن يستخدمها مطورات التطبيقات في بناء تطبيقات عامة ونشرها في بنين تشغيل مختلفة.