STAGER: Semantic-based Framework for Generating Adapters of Service-based Generic-API for Portable Cloud Applications

Eman Hossny, Sherif Khattab, Fatma A. Omara, and Hesham A. Hassan

Abstract—In PaaS model, providers have different proprietary APIs, which make developers locked inside a specific platform and not able to easily port their applications among different platforms. So, vendor lock-in problem appeared. One solution to this problem is to use generic APIs with specific adapters. However, any update in a PaaS specific-API makes its corresponding adapter unusable which causes, what we call, API synchronization problem. Therefore, STAGER (SemanTic-based GenERation of Generic-API Adapters) framework is proposed. STAGER framework provides a semi-automatic adapter generation process, which generates specific adapters of generic APIs for PaaS services (e.g., blob storage and datastore services) for target PaaS platforms. The adapter generation process is based on semantic annotations of the generic APIs and their corresponding PaaS specific-APIs. In order to evaluate STAGER framework, two generic APIs for blob storage and NoSQL datastore services have been proposed. STAGER framework is used to generate the adapters of these generic APIs for two PaaS platforms: Google App Engine (GAE) and Windows Azure. Although there is some overhead 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.

Index Terms—PaaS heterogeneity, Vendor lock-in, Semantic annotation, Generic API

1 INTRODUCTION

CLOUD computing has become very convenient for many users because they can pay only per demand without worrying about the maintenance and management of resources [2], [3]. Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) are the main service models that are provided by cloud computing. This paper is concentrating on the PaaS model.

PaaS model shifts the charge of installing and maintaining software from cloud developers to cloud providers [4], [5], [6]. Thus, cloud developers can focus only on implementing and managing their applications. However, each PaaS provider has its own proprietary APIs for implementing and managing cloud applications. So, vendor lock-in problem appears. Some examples of the currently available proprietary PaaS solutions are Google App Engine (GAE) [1] and Windows Azure [1]. Other examples of open source PaaS solutions are CloudFoundry [1] and Heroku [1].

The vendor lock-in problem is defined as one of the major obstacles hindering cloud adoption [6]. It makes cloud developers are locked inside a specific PaaS platform and not able to easily port their applications or data to another PaaS platform. However, the applications or data portability might be necessary in several cases, such as a high rate of outage (e.g., Amazon EC2 became down for 12 hours on April 21, 2011 [7]), provisioned resources became expensive, low quality [8], or cloud providers stopped providing the provisioned resources [9].

Three phases are required for provisioning cloud applications: development (for implementing and testing cloud applications); deployment (for uploading cloud applications to a specific PaaS platform); and management (for starting cloud applications and executing management operations, such as billing, scaling, and monitoring) [10]. Because of the PaaS vendor lock-in problem, each PaaS provider provides specific APIs for each platform basic service (e.g., blob storage, messaging, authentication, and mail services) [11] to be used in the development phase (called service APIs) and another specific APIs for deployment and management phases (called management APIs) [12]. Therefore, in order to port an application among different PaaS platforms, the source code of this application may need to be changed during one or more of the provisioning phases. This paper focuses on the application development phase using proprietary PaaS service-APIs.

Several research studies have been done to support application portability and overcome the PaaS vendor lock-in problem [13], [14], [15], [16], [17], [18]. However, the majority of these studies focus on providing generic APIs for only deploying and managing cloud applications. In addition, the adapters of the currently available generic APIs are manually implemented. So, any update in a PaaS specific-API makes its corresponding adapter unusable (e.g., when the parameters of a specific method in a specific API have been changed, all references to this method need to be manually updated in the adapter code). Thus, the generic-APIs adapters are suffering from API synchronization prob-

---

1. https://appengine.google.com/
2. https://azure.microsoft.com/
3. https://pivotal.io/platform
The authors in [1] have solved the API synchronization problem by providing a semantic-based component for Automatic Adapter Generation (AAG) of generic APIs. In the AAG component, it is assumed that the semantic annotations of generic APIs and their corresponding PaaS specific-APIs are stored in an ontology that will be used in the adapter generation process. In this paper, the AAG component is used as a part of an overarching framework for adapter generation. Therefore, a complete framework for semantic-based generation of generic-API adapters for portable cloud application, called STAGER, has been proposed. STAGER framework provides semi-automatic process that is based on semantic annotations and semantic search to generate specific adapters of service-based generic-APIs for target PaaS platforms. Moreover, service-based generic-APIs (called Std-PaaS APIs) that can be used by developers to implement generic cloud applications have been presented. These applications can be deployed on heterogeneous PaaS platforms without the need for re-factoring their source codes. As far as we know, STAGER framework is one of the first works in this area presenting service-based generic-APIs with semi-automatically generated adapters.

Although our approach seems to be shifting the problem to the semantic layer, the effort of semantically annotating the generic and specific APIs is, in many cases, much less than the effort of manually implementing and adjusting the adapter code. 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 of all generic APIs. In this case, instead of adjusting multiple adapters, only the specific APIs need to be annotated once per change.

The rest of this paper is organized as follows. Section 2 presents an overview about the proposed STAGER framework with the involved assumptions and its architecture. The auxiliary components of the STAGER framework is elaborated in Section 3. An example for code generation of the blob storage case study is presented in Section 4. The evaluation of the STAGER framework is discussed in Section 5. Section 6 provides related work. Ultimately, Section 7 presents our conclusion and future work.

2 STAGER FRAMEWORK

This section presents the details of the STAGER framework. Firstly, an overview of STAGER framework is provided. Secondly, two assumptions for using STAGER framework are specified. Finally, the STAGER architecture is clarified.

2.1 STAGER Overview

In this paper, a Semantic-based Generation of Generic-API Adapters framework (called STAGER) is proposed as a step towards overcoming the API synchronization problem. STAGER can semi-automatically generate the specific adapters of semantically annotated service-based generic-APIs. STAGER is called semi-automatic framework because of the adapter generation is based on semantic annotations and utilities API. Therefore, to generate adapters through STAGER framework, the following requirements are needed: (1) the different PaaS service-APIs to be semantically annotated. In the current state, these annotations are done manually and (2) the utilities API, which refers to some manually implemented code. By this way, the adapter generation is done automatically based on the semantic annotations and the utilities API.

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 in the following scenario: (1) A PaaS cloud provider provides a set of service-based specific-APIs; (2) The specific-API annotator annotates the service-based specific-APIs and passes them as an input to STAGER framework; (3) The generic-API designer designs and annotates a set of service-based generic-APIs and passes them as an input to STAGER framework; (4) Also, the generic-API designer designs and annotates the required utilities API and passes them as an input to STAGER; (5) STAGER framework parses all these APIs (specific, generic, and utilities) and inserts them into an ontology; (6) Based on this ontology, STAGER framework generates the source code of a set of specific adapters for the target PaaS platforms; and (7) The cloud developer uses the generated adapters to implement generic applications that can be deployed on multiple heterogeneous PaaS platforms.

STAGER framework can be customized through two use cases: by adding a new service-based generic-API or by generating a new adapter of an existing service-based generic-API. To add a new service-based generic-API, the generic-API designer designs and annotates the prototypes of a set of new generic methods, which satisfies the functionalities of a specific service. For example, to add generic APIs for an email service, the generic-API designer designs and annotates the prototypes of a set of new generic methods, which satisfies the functionalities of the email service. Then, these generic APIs are passed to STAGER framework to generate their adapters. According to the second use case, to generate a new adapter of an existing service-based generic-API, the specific-API annotator annotates the service-based specific APIs of the new PaaS and passes them as an input to STAGER framework, which generates a new adapter for the new PaaS platform. For example, to generate an adapter of the email service generic-APIs for Heroku and Azure platforms, the specific-API annotator annotates the email service specific-APIs of both Heroku and Azure platforms and passes them as inputs to STAGER framework to generate new adapters for Heroku and Azure platforms.

In nutshell, the idea to build a middleware that provides developers with a set of standard APIs, and thus migrate the development overhead from the cloud developers to the middleware, is not new. This idea has already been studied by many previous works such as MetaSync [19], UniDrive [20], and CoCloud [21]. In addition, our work builds on a bunch of existing works such as SCRO (Source Code Representation Ontology), reflection and existing PaaS specific-APIs. The main contribution is the code generation algorithm which is based on SPARQL queries against our ontology. Other contributions include: (1) The glueing together of the previous concepts into an overarching framework to save the development effort needed for updating PaaS adapters every time a specific PaaS-API is changed, (2) the extension
of SCRO ontology by enriching it with domain-specific concepts (e.g., blobs, containers, and storage account in the blob storage service) to help us in the automatic adapter generation process and (3) the Std-PaaS APIs which include generic APIs for several services (e.g., blob storage and datastore).

2.2 STAGER Assumptions

Although STAGER can semi-automatically generate the adapters of semantically annotated service-based generic-APIs, it involves two assumptions that are specified as follows.

The proposed Std-PaaS APIs are JAVA-based. So, only JAVA applications can currently benefit from STAGER framework. In addition, STAGER works only for generating adapters for API wrappers (written in JAVA) because the implementation process 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 [22] and Ruby calls it introspection [23]). Although the proposed Std-PaaS APIs are created as API wrapper and not RESTful API, Std-PaaS APIs are still useful because some providers wrap their REST APIs inside API wrappers written in JAVA for example.

STAGER framework assumes that the semantic annotations of all Object Oriented (OO) methods are available as inputs. These OO methods may be specific or generic methods. Currently, these semantic annotations are done manually and they include annotations of the I/O parameters of each OO method. By this way, STAGER can import any generic APIs and generate their specific adapters provided that these generic APIs are semantically annotated. However, this assumption could be relaxed by building public back-end repository (or marketplace) with contributions from the community. Alternatively, a third party organization can build their business on annotating the APIs and sell the annotated APIs to interested software houses.

2.3 STAGER Architecture

The architecture of STAGER framework is composed of two main layers: API analysis layer and Adapter generation layer. Fig. 1 clarifies the STAGER architecture. Firstly, the API analysis layer parses a set of given APIs and inserts them into an ontology. Secondly, the adapter generation layer automatically searches the ontology to generate the source code of specific adapters for target PaaS platforms.

2.3.1 API Analysis Layer

The Source Code Ontology Population (SCOP) is the main component of the API analysis layer. SCOP is 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 an ontology (called STAGER ontology). Therefore, SCOP requires, as inputs, one or more PaaS API(s) with their semantic annotations and it outputs an updated version of the STAGER ontology. The process flow of STAGER framework is illustrated in Fig. 2. The SCOP component has two main functions:

- **Automatic analysis of a PaaS API**: it uses the reflection mechanism [24] 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, and the I/O parameters of each method.
- **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 manually 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 adapter generation layer.

2.3.2 Adapter Generation Layer

The Semi-Automatic Adapter Generation (SAAG) is the main component of the adapter generation layer. It automatically searches the STAGER ontology to generate the source code of specific adapters for target PaaS platforms. It requires, as inputs, the STAGER ontology and the target PaaS platform. The SAAG component uses the automatic code generation algorithm that is proposed in [1] to generate the source code of a specific adapter for a target PaaS platform. The specific adapter maps the generic methods, which are defined in the STAGER ontology, into their corresponding generic-API calls. Whenever, the SAAG component fails in generating a specific adapter, it will notify developers with a list of generic methods that cannot be generated. The adapter generation could only be failed in the case of no corresponding annotations for the target PaaS platforms.

It should be noted that the SAAG component is a semi-automatic process because it generates the adapters based on the utilities API and the semantic annotations of the PaaS APIs. However, it is useful in two cases: (1) when
a new PaaS platform is added to STAGER framework, it can generate a specific adapter for this new PaaS platform and (s2) when an existing PaaS specific-API is changed or modified, it can generate a new adapter for this updated PaaS platform. Examples of changes that can be handled by the SAAG component are adding a new method, changing a method name, adding a new parameter to an existed method, changing the data types of the I/O parameters of an existed method, merging a set of methods, and splitting a method into a set of methods. However, the SAAG component cannot handle call-back methods, as this is a subject of future work.

To generate the code of a generic method, the SAAG component can automatically generate a SPARQL query to retrieve a path (i.e., a list of specific methods) that starts from the semantic concept of one of the generic method input parameters and ends at the semantic concept of the generic method output parameter. In this case, the SPARQL query is initialized again to detect a path from another input parameter of the generic method. If it is not possible to link between any of the semantic of the generic method input parameters and the semantic of the generic method output parameter, then a list of generic methods which cannot be generated are returned to the cloud developer. Furthermore, if there are cycles in the ontology graph, then it is not possible to detect a right path. So, the algorithm will stop when it reaches the maxDepth. The details about the code generation algorithm is presented in [1].

3 STAGER Auxiliary Components
STAGER framework is composed of two main components: Source Code Ontology Population (SCOP) and Semi-Automatic Adapter Generation (SAAG); and three auxiliary components: Std-PaaS APIs, Utilities API, and STAGER ontology. It should be noted that SCOP and SAAG components are service independent components, where the auxiliary components can be categorized into service dependent components and service partially-dependent. In this section, the STAGER auxiliary components will be elaborated.

3.1 Std-PaaS APIs
The Std-PaaS APIs are service dependent components that provide a real set of PaaS standard-APIs for each service (e.g., database, blob storage, NoSQL datastore, messaging, email, authentication, and authorization). Therefore, our target is to define a set of generic APIs for each service
3.2 Utilities API

For generating adapters of the Std-PaaS APIs, we needed to manually write some code. This manually implemented code is combined as Utilities API, which may need to be extended whenever a new platform is introduced, as well as, a new set of methods is implemented. Therefore, the utilities API is a service dependent component. Most of the utilities API involves type conversions and loops, also, it should be small in size because it requires a manual update for their code.

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 values; 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 used by cloud developers to specify their 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.

3.3 STAGER Ontology

The STAGER ontology is a service partially-dependent component because it is composed of service independent and service dependent ontologies. A sample of STAGER ontology is available online [26]. The STAGER ontology is designed based on SCRO ontology, which represents source codes as an ontology [27], [28]. SCRO ontology has been implemented using Web Ontology Language (OWL) [29], [30]. SCRO has defined a set of concepts and relationships that are used to describe OO programs as ontology. For example, it has defined ClassType concept to represent different OO classes, such as abstract or static classes, and Method concept to represent different OO methods, such as static or constructor methods. In addition, it has defined two relations, hasInputType and hasOutputType, to represent the data types of the I/O parameters of an OO method.

The STAGER ontology has three main objectives. Firstly, it represents all OO concepts that are defined by a set of PaaS specific-APIs, generic-APIs, and utilities API. Secondly, it defines a set of shared vocabularies among different PaaS specific-APIs. Thirdly, it hides syntactical differences and heterogeneity of different PaaS specific-APIs. Fig. 3 illustrates the structure of the STAGER ontology, which includes three main ontologies: SCRO+, domain specific, and Std-PaaS APIs ontologies. They are elaborated as follows.

3.3.1 SCRO+

SCRO+ is a service independent ontology. It is an enhanced version of SCRO ontology [27], [28]. 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 OO 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 String to represent a userName and a password respectively,
and it returns an integer to represent a connectionStatus. Therefore, an ontological instance with the name \texttt{meth1(java.lang.String,java.lang.String)} is added to SCRO ontology under the method concept. Furthermore, two relationships are defined for this instance as \texttt{hasInputType} with the value \texttt{java.lang.String} and \texttt{hasOutputType} with the value \texttt{int}. However, there are three extra problems in the definition of this ontological instance:

1) The method \texttt{meth1} requires two input parameters of type \texttt{java.lang.String}. However, only one relationship with the name \texttt{hasInputType} and the value \texttt{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 did not provide relationships to define the semantic meaning of each input parameter of an OO method. Since these two input parameters of \texttt{meth1} have the same data type, we need to identify the semantic meaning of each one to differentiate between them.

3) SCRO did not provide relationships to define the semantic meaning of an output parameter of an OO method.

Therefore, SCRO$^+$ is introduced to overcome these problems and specify semantic annotations for each OO method. SCRO$^+$ defines three new relationships as follows:

1) \texttt{hasSemanticInput}: an object property which specifies the semantic annotation of an input parameter of a method. For example, for \texttt{meth1}, we need to define two relationships as \texttt{hasSemanticInput} with the value \texttt{username} and \texttt{hasSemanticInput} with the value \texttt{password}.

2) \texttt{hasSignatureSemantic}: a data property which specifies the required order of the semantic annotations for all input parameters of a method. For example, for \texttt{meth1}, we need to specify the order of its input parameters by defining a relationship as \texttt{hasSignatureSemantic} with the value \texttt{[username, password]}.

3) \texttt{hasSemanticOutput}: an object property which specifies the semantic annotation of an output parameter of a method. For example, for \texttt{meth1}, we need to define a relationship as \texttt{hasSemanticOutput} with the value \texttt{connectionStatus}.

### 3.3.2 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 \texttt{bucket} to store a blob; while Azure uses a \texttt{container} to store a blob. So, the \texttt{bucket} and the \texttt{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 Fig. 4. 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).

### 3.3.3 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 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. Fig. 5 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 Fig. 5 highlights the generic methods that are defined for a cloud blob storage service.

### 4 STAGER Feasibility: Code Generation of Blob Storage generic-APIs

STAGER framework has been evaluated using two generic-APIs: one for blob storage and the other for NoSQL datastore. So, STAGER framework is used to validate and generate the specific adapters of these generic-APIs for GAE and Microsoft Azure platforms.

To clarify the feasibility of STAGER, we will go through the process of generating code of a generic method (e.g., \texttt{createBlob}) for a specific adapter (e.g., Azure adapter). Fig. 6 shows a visual representation of a subset of Azure storage API with its semantic annotations. According to Fig. 6, a set of specific methods is presented by yellow ovals. The input arc to each method represents its semantic input; whereas the output arc from each method represents its semantic output.

Fig. 7 clarifies the semantic annotations of the \texttt{createBlob} generic method. This method requires several input parameters with semantic annotations as storage account, container name, blob name, content type, and blob access rights; while it outputs an object with semantic annotations as new blob.

![Fig. 4. Domain Specific Ontologeies for Blob Storage and NoSQL Datastore](image-url)
To generate the code of the createBlob generic method for Azure adapter, a SPARQL query is automatically generated to search about a method which has a semantic input storageAccount, has a semantic output newBlob, and belongs to Azure provider. Unfortunately, as it appears in Fig. 6, 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 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 belong 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 black arcs in Fig. 6. In addition, the seven yellow ovals present the specific methods that form the code of the createBlob generic method.

Finally, after a right path is detected, the sequence of the specific methods in that path is called to form the generated code. Sample of the generated code of the generic-APIs of both the blob storage and the NoSQL datastore services for Azure and GAE platforms are available on GitHub [31], [32].

5 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 following metrics are specified to measure STAGER performance: (1) the time consumed by each component (SCOP and SAAG); (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.

5.1 Metrics of performance

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 six different libraries with different
sizes and the average parsing time is computed after executing the parsing process of each library 20 times. Fig. 8 presents the average parsing time (in seconds) for these six libraries vs. number of parsed classes. According to Fig. 8, 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. Fig. 9 shows that there is a linear relation between the size of the generated ontology (in kilobytes) and the average parsing time for these six libraries.

The SAAG component performance: the SAAG component is tested by generating the adapters of the blob storage and datastore generic-APIs for both Azure and GAE platforms. The average time for generating the code of each generic method is computed by executing the SAAG component 20 times for each method. Fig. 10 and Fig. 11 clarify the average code generation time (in milliseconds) of the blob storage generic-APIs and the datastore 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 parameters, 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 SAAG component starts a new trail 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 specific method parent and has the desired 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`.

Measuring the execution time using generic-APIs versus specific-APIs: Two generic applications are implemented; one that uses the blob storage generic-APIs and the other uses the NoSQL datastore generic-APIs. These generic applications are deployed on both GAE and Azure platforms. In addition, four other specific applications are implemented as follows: (1) an application that uses GCS API, (2) an application that uses Azure blob storage API, (3) an application that uses GAE datastore API, and (4) an
Fig. 11. The Avg. code generation time of Datastore Generic-APIs

application that uses Azure datastore API. Because of space limitation, we will explore the results when using GAE adapter versus the results when using GAE specific APIs for both blob storage and NoSQL datastore. 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.

Fig. 12 and Fig. 13 clarify that there are no statistically significant difference between the average time using both (generic and specific) APIs which means that there is no significant overhead when using the adapters. Also, Fig. 12 and Fig. 13 show large values for confidence interval this is because of the large variation in the execution time of each method. This large variation is produced because of 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.

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

Fig. 12. Execution Time of GAE Blob Storage adapter vs. the specific GCS API

Fig. 13. Execution Time of GAE Datastore adapter vs. the specific GAE datastore API
framework. Therefore, we measured the size of the static Utilities code (by counting no. of lines) and found it indeed significant (~ 35% of total adapter code). By this way, the STAGER framework reduces the effort of manually implementing the adapters by ~ 65% (which is good). However, most of the static code 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 I.

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 manual effort of the semantic annotations of the PaaS APIs is dependent on the complexity of the PaaS APIs and number of inputs of each PaaS method. So, this manual effort can be represented as a function of (no. of parameters per method * no. of methods per API * no. of APIs). According to our work, the average number of methods in GAE was 30 and the required parameters of each method are two or three. Thus, its annotations were 63. However, these annotations are done once, and then, they are used to generate the adapters of 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 the new changes and this update is done once. After that, the SAAG 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 which 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 adapters can be, in some cases, less than the effort for manually implementing the adapters and updating it whenever the specific-APIs are changed.

### 5.2 Factors affecting performance

The previously mentioned metrics may be affected by three main factors: the PaaS specific-API; the nature of the different PaaS APIs (e.g., blob storage API, message API, etc.); and the design of the Std-PaaS APIs.

For a PaaS specific-API, if a PaaS specific-API is 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. As the number of classes in a PaaS specific-API is increased, the time consumed by the SCOP component is increased.

For the nature of the different PaaS APIs, the implementation of the SCOP and SAAG 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 SAAG component is tested with the blob storage generic-APIs and the datastore generic-APIs. It succeeded in generating the adapters of these generic-APIs for two of the leading PaaS platforms; GAE and Azure. Moreover, the SAAG component can generate the adapters of any semantically annotated generic-APIs. In the future, we plan to use the SAAG component to generate adapters of other generic-APIs.

For the design of the Std-PaaS APIs, the Std-PaaS APIs design is independent on STAGER framework. Std-PaaS APIs represent service-based generic-APIs with specific adapters that can be generated by STAGER. Moreover, STAGER can generate the adapters of any semantically annotated generic-APIs.

### 6 Related Work

As stated previously, the vendor lock-in problem is considered one of the major obstacles which prevents cloud adoption. Therefore, many efforts are done to address this problem by building standards or open source to allow the cloud portability. The authors in [33] have categorized the cloud portability into four types: virtual machine (VM) portability, IaaS application portability, PaaS application portability, and data portability. The state of the art for VM and application portability over IaaS clouds will be discussed in Sub-section 6.1, whereas application and data portability over PaaS clouds will be elaborated in Sub-section 6.2. Finally, a brief discussion is presented in Sub-section 6.3.

#### 6.1 Portability over IaaS

IaaS clouds provide two types of portability: virtual machine (VM) and application [33]. A VM portability refers to the ability to migrate a VM from one IaaS provider to another. Open Virtualization Format (OVF) provides a standard format to allow migrating VMs 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 deploy applications on this VM. In addition, it is possible to build the application inside a container (e.g., Docker) and then migrate the container, which contains the application with its dependencies, to other IaaS clouds [34]. Furthermore, IaaS application portability can be done by implementing the applications using open generic-APIs for IaaS platforms, such as Jcloud [35] (JAVA-

7. https://www.docker.com
based), Libcloud\(^\text{\textregistered}\) (Python-based), Fog\(^\text{\textregistered}\) (Ruby-based), and Deltacloud\(^\text{\textregistered}\) (RESTful-based).

### 6.2 Portability over PaaS

This Sub-section introduces some of the available efforts to overcome the PaaS vendor lock-in problem by supporting application or data portability. These efforts can be categorized as follows.

#### 6.2.1 Standardization Efforts

A set of standardization efforts has been done to address the PaaS vendor lock-in problem, such as Cloud Application Management for Platforms (CAMP), Topology and Orchestration Specification for Cloud Applications (TOSCA), and Open Cloud Computing Interface (OCCI).

**CAMP**: OASIS has introduced Cloud Application Management for Platforms (CAMP) standard, which provides standard management API to allow deploying and managing applications over different PaaS platforms\(^1\). CAMP aims to match the application requirements with the PaaS capabilities and generate deployment plans. However, CAMP deployment is only supported by a very small number of new PaaS platforms, such as Solum\(^2\) and Brooklyn\(^3\).

**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\(^4\). In TOSCA, cloud applications can be specified by a topology and a set of management plans. However, most of the currently available PaaS platforms could not support the TOSCA deployment except a very small number of PaaS platforms, such as OpenTOSCA\(^5\).

**OCCI**: Open Grid Forum (OGF) has introduced Open Cloud Computing Interface (OCCI) standard, which provides standard RESTful APIs for managing clouds\(^6\). 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).

#### 6.2.2 Research Projects

**PaaSport**: The authors in\(^7\) have provided a semantically-based cloud-broker which aims to deploy, migrate, and manage cloud applications over heterogeneous PaaS platforms. In addition, it provides a unified API with specific adapters to support PaaS portability. Their unified API is based on the CAMP standard and Cloud4SOA. However, their adapters are manually implemented which will lead to API synchronization problem. The PaaSport is similar to the proposed STAGER framework in that the PaaSport requires the PaaS specific-APIs to be semantically annotated through an OWL ontology. Moreover, the PaaSport provides an ontology-based recommendation algorithm which aims to select the most suitable PaaS platform based on the application requirements, which may be functional or non-functional requirements.

**mO-SAIC**: The authors in\(^8\) have provided the mO-SAIC (Open Source API and platform for Multiple Clouds) project, which aims to support applications portability and interoperability among multiple clouds. mO-SAIC 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 their API is that it was designed based on event-driven, which is a complex programming style. In addition, their API focuses on deploying applications across IaaS clouds. Currently, mO-SAIC proposes adapters for a set of IaaS clouds, such as Amazon EC2, OpenNebula, and Eucalyptus. For more information about the mO-SAIC project see\(^9\), \(^10\).

**MODAClouds**: most of the current research efforts have solved the cloud vendor lock-in problem by providing standards or generic APIs. However, MODAClouds followed a different approach to solve this problem, using model driven engineering (MDE). The main objective of the MDE is to provide abstraction and automation\(^11\). The abstraction is satisfied by making the application development is independent from 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\(^12\) aims to design and execute 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. Then this IDE can semi-automatically map the designed application into a code to be ready for execution.

The authors in\(^13\) provide a MDE framework to hide the heterogeneity of the PaaS specific-APIs. Their framework helps cloud developers to design agnostic cloud applications that can be deployed on heterogeneous PaaS platforms. Their framework is based on semantic ontology and meta-model. It provides an abstract model and generic API (called reference API) for each PaaS service. In addition, it automatically generates the specific adapters of the reference APIs. Their work is similar to our work, however, it differs in two points. Firstly, their framework is based on MDE, while our framework is based on generic-APIs with semantic annotations. Secondly, their framework can only handle the common features of a PaaS specific service among heterogeneous PaaS platforms, while our framework can handle the unique and common features of a PaaS specific service (e.g., the NoSQL datastore requires partition key in case of Azure, while it is not required in case of GAE. So, the partition key is considered a unique feature that is handled in the specific adapters of the datastore generic-APIs).

#### 6.2.3 Generic-APIs

The authors in\(^14\) have proposed a common model for different PaaS platforms to solve the application portability problem. In their model, they have defined the structure of a PaaS platform to be composed of three layers: 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, with the profiles of the different PaaS platforms. The matching process will output a list of PaaS platforms that are suitable for deploying that application. The disadvantage 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 the cloud developer needs to use a PaaS specific-API to implement his/her application. However, the STAGER framework provides cloud developers with a service-based generic-APIs to help them to implement portable cloud applications.

The authors in [16], [43] have proposed a unified interface, called Nucleus, for deploying and managing cloud applications on 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 its API, its corresponding adapter needs to be updated manually (i.e., their adapters suffer from the API synchronization problem).

COAPS API [14] is a generic PaaS API for deploying and managing cloud applications on multiple heterogeneous PaaS platforms. It is based on OCCI standard. 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 allow deploying cloud applications on GAE, besides CF and OS [44], [15]. The COAPS API is similar to IBM Altocumulus project [45] 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.

CDPort (Cloud Data Portability) framework [8], [46] has solved the data portability problem among different cloud databases. They proposed a standard API and a common data model to hide the heterogeneity of different cloud data storage services. They provide an adapter for each one of the supported databases. They have tested their framework by implementing a portable java application for supporting NoSQL databases. One of the limitations of their 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 adaptation code to use the SimpleDBAdapter). Another problem in the CDPort framework is that the adapters' source codes are implemented manually. Thus, whenever a specific database API has been modified, the adapters cannot be used and their code needs to be updated manually.

The authors in [47], [48] have proposed a policy-driven middleware, called PaaSHopper, which aims to develop and deploy 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: blob storage, data storage, and asynchronous tasks. Their work is very close to ours. Therefore, 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 implemented manually. Thus, when a new PaaS platform is added or when an existed PaaS provider changes its API, then a new adapter need to be (re)implemented (i.e., the adapters are suffering from the API synchronizations problem). Furthermore, the PaaSHopper is in its early stage because it only provides a prototype.

The authors in [49] have proposed the PaaS Manager framework in order 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 on different PaaS platforms. Their common API includes a set of specific adapters, which have been implemented manually, for the supported PaaS platforms, such as CF, Heroku, and CloudBees.

6.3 Discussion

It should be noted that most of the currently related work focuses on providing standards or generic APIs for only deploying and managing cloud applications among different PaaS platforms. Whereas a few proposals are existed for developing cloud generic applications. However, they are not open-source (e.g., CDport and PaaSHopper). Moreover, the current standardization efforts are in its early stage and they are not supported by most of the leading PaaS platforms. Whereas a few proposals are existed for developing cloud generic applications. However, they are not open-source (e.g., CDport and PaaSHopper). Moreover, the current standardization efforts are in its early stage and they are not supported by most of the leading PaaS platforms. In addition, the adapters of the currently available generic APIs are implemented manually. So, they are suffering from the API synchronization problem. Therefore, STAGER framework proposes service-based generic-APIs with semi-automatically generated adapters. Because of space limitation, a comparison between STAGER and the currently existing work is available online [50].

7 Conclusion and Future Work

In this research, STAGER framework for semantic-based generation of generic-API adapters for Portable Cloud Applications is presented. In addition, service-based generic-APIs, called Std-PaaS APIs, are proposed. The Std-PaaS APIs provide a set of service-based generic-APIs with specific adapters. Therefore, the Std-PaaS APIs work as a middleware to hide the heterogeneity of service APIs of different PaaS and help cloud developers to implement applications once and deploy them on multiple heterogeneous PaaS platforms. On the other hand, the Std-PaaS APIs are service dependent and the policy of selecting a service to create a generic API for it depends on many factors, such as popularity, available features, the nature of applications, use cases and user profiles, etc. STAGER framework provides the mechanism that uses such generic APIs once selected and created. As a future work, STAGER framework can be
combined with a mechanism to identify the most requested services.

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). In this paper, STAGER framework works with service API. As a future work, we plan to use STAGER with management API besides service API. Moreover, we are generating adapters for similar service APIs of different platforms. For example, if cloud A has provided searching APIs which are not available in cloud B. Then, STAGER cannot generate the adapter for cloud B.

The semi-automatic adapter generation process generates the specific adapters of any generic-APIs. In other words, this process is not dependent on PaaS in particular but can be applied to semantically annotated generic-APIs and specific-APIs. By this way, the generic-APIs can be imported from elsewhere provided that it is semantically annotated. However, to the best of our knowledge, we could not find PaaS generic development APIs as an open source. Therefore, the Std-PaaS APIs are proposed which can include a set of service-based generic-APIs. Currently, the Std-PaaS APIs include generic-APIs of two main services: blob storage and NoSQL datastore. These generic-APIs are used to test and evaluate STAGER framework by generating the adapters for two PaaS platforms: GAE and Azure. In the future, we plan to update the proposed Std-PaaS APIs to provide generic-APIs for other services. Besides the proposed Std-PaaS APIs which is stateless API, we plan to design stateful generic-APIs, as well as, RESTful.

It should be noted that STAGER has two limitations: the adapter generation depends on the utilities API, which is a set of manually implemented code; and the semantic annotations of the generic/specific-APIs, which are not available. So, it has been done manually. However, STAGER is considered a step towards reducing the effort of manually implementing generic-API adapters and updating it whenever the proprietary APIs are changed. Furthermore, the relaxation of these manual efforts is a subject of future work.

REFERENCES


