Towards Application Portability in Platform as a Service

Stefan Kolb and Guido Wirtz
Distributed Systems Group
University of Bamberg
Bamberg, Germany
{*stefan.kolb, guido.wirtz*}*@uni-bamberg.de*

***Abstract*—Cloud Computing has been one of the most vibrant topics in the last years. Especially *Platform as a Service* (PaaS) is said to be a game changer for future application development. Taking away most of the configuration work, it pledges to foster rapid application development which seems even more important in a world of complex scalable distributed systems. Whereas *Infrastructure as a Service* (IaaS) is in the process of consolidation and standardization, the PaaS market is largely fragmented offering varying ecosystem capabilities. In this situation, application portability is a major concern for companies utilizing PaaS to avoid vendor lock-in and to retain the ability for future strategical decisions. To categorize portability problems of PaaS, we define a model of current PaaS offerings and identify different portability perspectives. Starting from the model, we derive a standardized profile with a common set of capabilities that can be found among PaaS providers and matched with one another to check application portability based on ecosystem capabilities. We validate our findings with a comprehensive data set of 68 PaaS offerings together with a web-based application for portability matching. We also identify further portability problems by porting the application to different PaaS vendors, validating ecosystem portability and giving hints for future research directions.**

***Keywords*—Cloud Computing, Platform as a Service, PaaS, Ecosystem, Comparison, Portability**

I. Introduction

Over the last years, the cloud hype led to the establishment of a large amount of cloud offerings. They span the whole cloud stack from *Infrastructure as a Service* (IaaS) through *Platform as a Service* (PaaS) and *Software as a Service* (SaaS). Especially the PaaS market is said to be crucial with consistent growth over the next years[1]. With its abstraction of the programming platform, including the operating system, runtimes, and middleware, its major promise is that customers can focus on developing applications without the need to maintain the computing environment. This is particularly beneficial in the context of complex interdependencies of highly scalable distributed cloud systems. The outsourcing of vast parts of the IT stack also delivers cost savings to the customers, while vendors in turn can benefit from the economies of scale by efficiently utilizing the underlying infrastructure [5]. A determining influence on the degree of IT commoditization is thereby the standardization of the products [6]. Nevertheless, we see that the PaaS market is driven by differentiation. Currently, there exist a lot of divergent offerings with varying capabilities, system configurations, and vendor-specific restrictions. For vendors, this differentiation is one integral part to attain and retain their market share in the face of market pressure, but for the customers this inevitably leads to some kind of lock-in [7], [8]. In such a scenario, the change to a different provider leads to significant additional costs for necessary migrations [9], [10]. However, business requirements can change over time as well as the capabilities and contract terms of the provider which makes it essential to preserve as much flexibility as possible between different vendors. With the market and cloud offerings steadily evolving, portability is even more important for business continuity [11]. Being a higher level abstraction with more and less standardized ecosystem capabilities than IaaS, PaaS also inherits more potential incompatibilities that make it harder to retain application portability [11], [12]. The term PaaS ecosystem thereby describes the complex system of interdependent components and capabilities that work together to enable a PaaS cloud[2]. We can see increasing demand by users of any PaaS to expand those capabilities, e.g. the number of runtimes or services supported. The common recognition that there exists no one-size-fits-all PaaS[3] means we have to approach portability among PaaS from another perspective than between ordinary standardized middleware implementations. As the offerings do not share one common set of portable capabilities but rather intersect with one another at many different parts, it is better to look at portability between PaaS platforms from a local application view, starting from a particular configuration, and to dynamically identify a set of compatible partners. To achieve this, we define a set of application dependencies from the PaaS ecosystem like language runtime, frameworks, data stores or third-party services that should be portable between vendors. Whereas there are many unstructured PaaS comparisons available throughout the web, there exist few structured approaches to compare PaaS offerings aiming at application portability (See Section VI). In general, there is a lack of comparability between PaaS which makes it hard for customers to decide on one [14]. Cloud offerings like PaaS that rent synthetic entities, such as access to a middleware stack, are less well described by current standards, and hence even common terminology is lacking for describing how such entities might be transferred from one provider to another [11]. In order to categorize these portability problems, we define a model of current PaaS offerings and identify different portability perspectives. Based on this model, we derive a standardized profile with a common set of capabil-

[1] See Gartner Research [1], [2], IDC Research [3] and 451Research [4].

[2] See http://searchcloudprovider.techtarget.com/definition/cloud-ecosystem

[3] See [13] and http://blog.docker.io/2013/08/paas-present-and-future/

ities that are seen among PaaS providers. These profiles can be matched with one another to check ecosystem portability. The profiles are the foundation for different use cases including discovery and lookup, filtered retrieval, and matching with an application dependency profile. Our approach is empirically validated by providing a web-based application for these use cases together with a comprehensive data set of 68 PaaS offerings. We argue that if offerings are ecosystem-compatible, it is generally possible to port the applications with potentially additional adaptation effort between the vendors. In that regard, we also identify further portability problems by migrating our application to five different PaaS vendors validating our notion of ecosystem portability while identifying first results for future research directions.

The remainder of the paper is structured as follows. In Section II we evaluate and define the notion of PaaS and our specific scope of PaaS offerings. Section III introduces a generic model of current PaaS systems. Based on this abstraction, we assess and categorize different portability challenges for PaaS systems in Section IV. Along with the identified portability dimensions derived from the high-level model, we extract important capabilities that form a typical PaaS ecosystem and formalize them into a concrete PaaS profile specification in Section V. Moreover, we present our implementation for portability matchmaking that allows the discovery and matching of those profiles. We validate our idea of ecosystem portability by porting the application to different matching vendors and describe real world use cases that can be targeted with the profiles. Additionally, we identify further portability problems that must be investigated on finer levels of granularity. Section VI reviews related work and gives distinction and rationales for deviating design decisions. Finally, Section VII summarizes the paper and discusses future work.

II. Platform as a Service

The NIST Definition of Cloud Computing defines Platform as a Service as "*the capability provided to the consumer [...] 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*" [15].

This well known and often cited definition already shows that the term Platform as a Service can be applied to cloud services with very different capabilities. In general, the definition only requires the ability to deploy applications which have certain dependencies that are supported by the platform environment. Moreover, it demands the abstraction from the underlying cloud infrastructure while possibly granting access to unspecified configuration settings of the PaaS environment. PaaS clouds are not fundamentally different from traditional computing systems (i.e. platforms) where applications can be developed and run [11]. Although [11] additionally demands an implicit basis for creating scalable applications, other definitions [13] and the current vendor landscape minimize the requirements to an application runtime platform delivered in a pay-per-use, self-service way. In fact, definitions are widely spread and often conducted from contrasting perspectives resulting in diverging PaaS categories missing a common consensus. The result is a crowded market of PaaS offerings that sometimes provide completely different capabilities [16]. Gartner [17] for example defines a lot of *x*PaaS subcategories where *x* specifies the part of the platform functionality that is delivered by the PaaS. Forrester [13] instead, categorizes PaaS by the applicability for different types of developers, namely *DevOps pro*, *coder*, and *rapid developer*. These kinds each come with distinct backgrounds, preferences, and motivations on the controllability of the platform. The situation is also made worse by *cloud washing*, the tendency of jumping on the bandwagon and terming offerings as cloud services that do not even provide typical cloud characteristics (as defined in [15]) [14]. Instead of trying to create another set of new categories, we classify PaaS offerings along dimensions that are high-level and a common denominator, the SPI model (SaaS, PaaS, IaaS). During the research, we found properties from both boundary technologies IaaS and SaaS incorporated into the PaaS offerings. On the one hand this can be more control over the infrastructure and the programming environment, on the other hand the ability to instantly provision applications in a SaaS manner or tools for visual programming of applications tightly bound to SaaS solutions[4]. Currently, we see three distinctive groups of PaaS in between IaaS and SaaS (See Figure 1).

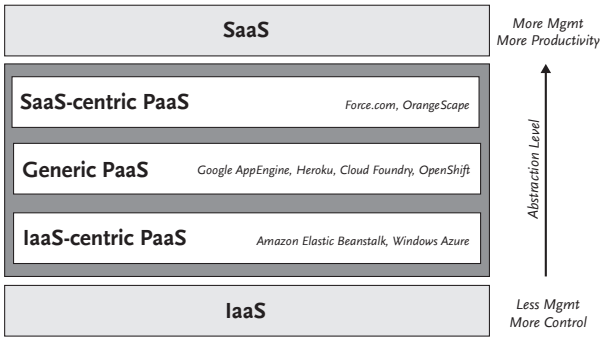

Fig. 1: Types of Platform as a Service

Firstly, there are *IaaS-centric* PaaS that offer streamlined deployment of applications on top of the IaaS stack while still retaining high or full control over the underlying infrastructure. An example of this type of provider is Amazon Elastic Beanstalk which is a simplified composition of Amazon's low-level IaaS services including e.g. EC2 and Elastic Load Balancer. At the other end there are *SaaS-centric* PaaS with a clear focus on productivity and simplicity which are mostly restricted and tailored to a specific SaaS solution. These platforms abstract most of the available middleware and are often composed via visual tooling help without touching the actual application code. One of the first representatives was Force.com that provides the ability to develop applications for Salesforce's *Customer Relationship Management* SaaS solution. *SaaS-centric* PaaS also include very specific platforms that target Business Intelligence or Business Process

[4]See also http://blogs.forrester.com/james_staten/11-01-24-is_the_iaaspaas_line_beginning_to_blur

Management and such. At the center of our model reside the PaaS which we term *Generic PaaS*. All of these supply a more or less classical application platform that consists of a set of language runtimes, frameworks, services, and other components an application can be programmed to. The platform is managed by the PaaS provider while important aspects like scaling can be transparently initiated through a management interface by the developer.

Although as described many types of platforms can be termed PaaS, the scope of this research focuses on *Generic PaaS* and solutions on the lower end, closer to IaaS. That is because these solutions actually include an application platform that is comparable between different providers. As mentioned, *SaaS-centric* solutions are too specific to be compared or switched between. Those platforms come with restrictive vendor lock-in that cannot be bridged because of the nature of the offering's purpose.

Achieving portability between cloud offerings is a diverse challenge. In the literature, the terms portability and interoperability are often confused and falsely used as substitutes [18]. However, both describe different scenarios. Portability is defined as "*[...] the capability of a program to be executed on various types of data processing systems without converting the program to a different language and with little or no modification*" [19]. For the cloud and PaaS this means the ability to write code that works with more than one cloud provider regardless of the differences between them [20]. Interoperability in contrast describes "*[...] the ability of two or more systems or components to exchange information and to use the information that has been exchanged*" [19]. Here, we use the term cloud interoperability only for cloud systems that interoperate with each other in the classical sense. This would be the case for hybrid or federated multicloud deployments that need to interoperate to e.g. synchronize data or exchange other information in order to run one application in multiple sovereign clouds. Whereas we are strictly talking about application portability, some researchers may have termed similar ideas as interoperability. We will still only use the term portability throughout this paper.

III. Platform as a Service Model

Due to their different specifications, each cloud model (IaaS, PaaS, SaaS) needs to be treated separately in terms of portability [21]. Whereas the entities and interfaces of low-level IaaS systems like compute, network, and storage [22] are widely agreed upon, the entities of PaaS offerings are less well described by current standards and lacking common terminology for describing how such entities might be transferred from one provider to another [11]. Therefore, we need a common model of PaaS [23]. Before we have a closer look at the different aspects of PaaS portability, we define such a model of current PaaS offerings (See Figure 2). We develop this PaaS model because existing approaches are either too generic, aiming at the whole SPI model which does not fit the specialties of the PaaS environment, or do not depict the current state of PaaS in enough detail. The results are based and validated by the findings of our analysis of 68 PaaS offerings. Moreover, we aligned our model with related work on models and taxonomies from [14], [24]–[28]. The properties may not be exhaustive but at that level and time they prove to be the most important ones to form a model of a modern PaaS offering. In our notion, PaaS can be divided into three layers: *infrastructure*, *platform* and *management*.

A. Infrastructure

The PaaS infrastructure tier abstracts the physical infrastructure and adds another layer on top of IaaS capabilities or directly abstracts the bare hardware. Whereas with IaaS one can choose from different machine configurations, PaaS hides most of those physical properties. What is left for the customer are concepts like *dynos* (Heroku), *worker units* (AppHarbor), *app cells* (CloudBees) or *gears* (OpenShift) that abstract a specific instance configuration that can be used within the PaaS. The raw CPU power among these concepts will vary and is elusive. Horizontal scalability is achieved by provisioning more instances on the fly. The instance's disk capacity is often negligible as most PaaS only provide ephemeral storage to be stateless and highly scalable. Therefore, all persistent assets except the deployment artifacts must be saved in separate data stores to allow scale-out. The RAM size of those instances, however, is often explicitly given and may be directly configured as part of vertical scalability. In contrast to IaaS where CPU power and usage is a main factor for billing, most PaaS are metered by instance count and RAM size.

Another important factor is the geographical region the application will be deployed in. This is particularly interesting because of legal and performance reasons. As bandwidth capacities keep increasing on the customer's end, latency is one of the main constraining factors for publicly hosted applications[5]. An application deployed in a data center in Europe will have significantly faster response times to European users than an application hosted in the United States. It is therefore beneficial that a PaaS vendor offers several deployment regions or at least the appropriate region for the application's customer base. This is an essential feature for companies serving a particular region or willing to expand to different regions. Even more important than raw speed are legal issues and data security regulations. EU-based companies for example are prohibited by law to transfer or store customer-related data outside of the European Union [29]. With the majority of PaaS and cloud offerings in general being US-based or governed by US rights, those companies are not permitted to record customer-related data at the provider. However, the sole deployment region of the applications does not infer the required rights from this area. This must be ensured by explicit legal agreements with the provider like Safe Harbor[6] or a jurisdiction based in the EU. This is a crucial aspect for cloud vendors in general and even more present with the disclosures of the PRISM surveillance program in mid 2013. Moreover, there are other regulations that limit cloud adoption for certain businesses, like HIPAA and Sarbanes-Oxley compliance, to name a few, that must be provided for corporate data to be moved to the cloud [30]. Although some providers are explicitly EU-based and advertised as EU-compliant, the majority of vendors are just starting to address these issues.

[5] See http://www.igvita.com/2012/07/19/latency-the-new-web-performance-bottleneck

[6] See http://export.gov/safeharbor/

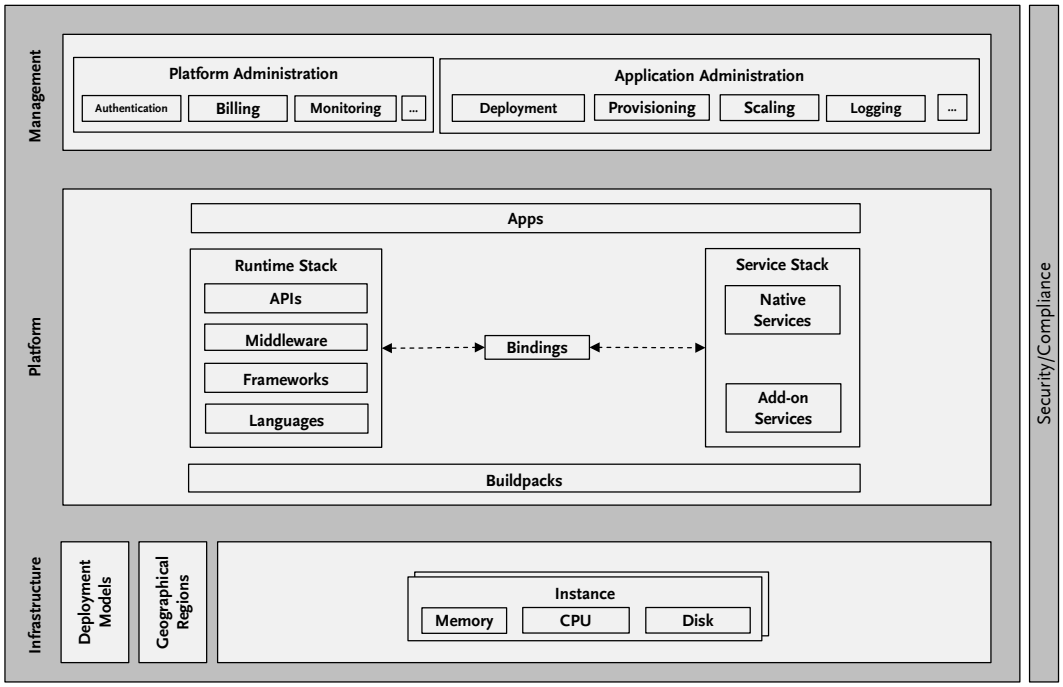

Fig. 2: Platform as a Service Model

PaaS is getting popular in different deployment models [15]. Public PaaS are hosted over the Internet accessible for a vast amount of different customers. A lot of public PaaS vendors tend to use existing IaaS providers like Amazon Web Services for their bare infrastructure management. Whereas public PaaS is still the most popular type of PaaS, companies are moving towards the implementation of private in-house PaaS solutions. With the emergence of open source PaaS like Cloud Foundry and OpenShift, more and more companies try to modernize their infrastructure capabilities and reuse existing in-house hardware for new private clouds. This can result in better workload distribution for these computing clusters while enabling the companies to leverage the productivity improvements and dynamic capabilities of PaaS inside their own security realms.

B. Platform

The platform is the main deliverable of a PaaS offering and includes the application hosting environment delivered as a service. Two stacks of components are decisive: The *runtime stack* and the *service stack*. Both stacks can be combined by the customers via bindings. Those bindings are generally environment variables that include important properties of the services like endpoint URLs, credentials, and other configuration information.

The runtime stack includes the basic runtimes offered by the PaaS, i.e. the programming languages that applications can be written in. Furthermore, we see the vast popularity of language-specific frameworks like Ruby on Rails which are leveraged to develop today's applications. Many customer applications also depend on middleware that may be hosted by the PaaS. Java EE for example is an established technology that requires a middleware product that implements its specification. Most specific are APIs that cover PaaS functionality like Google App Engine's APIs to their proprietary Datastore or Blobstore services. The higher the stack, the more specific the application dependencies become, thus raising the risk of lock-in.

The services stack is divided into native and add-on services. Native services are hosted and operated by the PaaS vendor typically co-located to the PaaS environment inside the same infrastructure. These services include mostly latency and performance critical core services like data stores. Add-on services are supplied by third-party service providers that integrate with the PaaS. They include both competing (e.g. data stores) as well as complementary services like analytics, search engines, messaging services, and many other utilities. The ability to create a large ecosystem of partners is a huge factor of current PaaS offerings. These services can improve the customer's ability to deliver applications along with cross-selling opportunities for the vendors[7]. Add-ons are provisioned from within the PaaS including *Single Sign-on* (SSO) with the add-on provider and are directly billed as additional part of the platform fees. However, add-ons possibly run in another infrastructure that is even geographically different from the PaaS. This must be taken into account when performance critical operations are involved.

Another key part of a modern PaaS is extensibility. Originally developed by Heroku, buildpacks[8] are a collection of scripts that define a generic API for detecting, compiling and releasing runtime languages, frameworks or services. Buildpacks enable the developers to add own packages of services or runtimes to their PaaS environment. They can be

[7] See http://www.infoworld.com/d/cloud-computing/forrester-paas-makes-developers-happy-220963

[8] See https://devcenter.heroku.com/articles/buildpacks

seen as isolated entities that can generate any of the service or runtime stack's capabilities. As of scalability issues with services like data stores (i.e. necessary data replication) this is typically more convenient for parts of the runtime stack. Other vendors have either adopted Heroku's buildpacks or defined their own extensibility mechanisms like OpenShift cartridges[9] or dotCloud custom services[10]. Buildpacks can be manually created by the developer or are also often created and shared by the community. This capability gives the developers greater freedom and possibilities to use the system, blurring the differentiation to IaaS.

Above both stacks, we see that some PaaS are starting to support instant deployment of popular applications like *Content Management Systems* (CMS) which on their part have dependencies on the runtime and the services stack while crossing the boundary to SaaS products.

C. Management

On top of the two previously described layers resides a management layer that allows control over the deployed applications and the configuration settings of the platform. The management layer includes the abilities to deploy and manage the lifecycle of the applications. This encompasses pushing, starting, and stopping of applications. Moreover, the provisioning of all native services and add-ons is initiated from the management tier. All available configuration and administration settings for the applications and the PaaS environment can also be controlled. This includes a wide range of functionality like scaling, logging, down to the creation of domain routes and environment variables. The management layer also covers the resource usage monitoring that is relevant for billing and scaling decisions. All those functionalities are controlled by the management interface. The interface can be a fully fledged RESTful API, console-based or driven via web UIs. Although the mentioned functionalities are shared by all different PaaS to a great extent, procedures and commands are not standardized and differ widely between providers.

IV. Platform as a Service Portability

As we can see from the several layers and the possible manifestations of the inherited components and capabilities, portability between PaaS is a difficult task. Nearly every vendor has its particular management API, platform configuration and restrictions, resulting in a strong dependency to a certain provider and significant costs when migrating from one vendor to another [31]. Yet the cloud was about flexibility, abandoning the burden of expensive in-house data centers and workstations, this enables providers to harvest their locked-in customers by dictating the prices. The EU commissioned study SMART 2011/0045 [32] even says that portability is the second most important obstacle hindering increasing cloud adoption. Portability threats can occur at a variety of different parts of a PaaS.

According to [21] there are two main interfaces that are exposed to the customer that must be investigated when looking at portability problems (See Figure 3). These are the Self-service Management API through which the cloud user

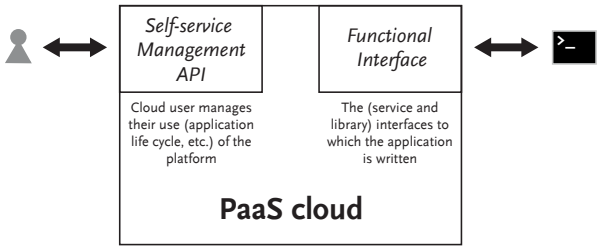

Fig. 3: PaaS Portability Interfaces [21]

manages their use of the cloud and the functional interface provided by what is resident in the cloud. This interface encompasses the primary function of the cloud service. For PaaS, this functional interface is the runtime environment and the set of components to which the application is written. The Self-service Management API in turn manages the application lifecycle and configuration settings of the platform. Both interfaces can be mapped to our model. The management tier includes the Self-service Management API functionality whereas the functional interface corresponds to the platform tier. Portability of the Self-service Management API can be achieved independently from the functional interface. As we want to focus on the portability of application dependencies in this paper, we concentrate on portability approaches for the functional interface and omit efforts on the standardization of the management interface. This must be investigated separately.

[33]–[35] define three different dimensions for cloud portability: service, functional and data portability. Service portability is defined as the ability to add, reconfigure and remove resources on the fly. Functional portability refers to the platform agnostic definition of application functionality. At last, data portability includes import and export functionality for data structures across platforms [33]. Whereas service and functional portability de facto match with the Self-service Management API and the functional interface, data portability is explicitly added. Nonetheless, data portability is strongly dependent on the particular data store solution that needs to supply appropriate export routines to enable portability between databases. Solutions for this problem should be developed independently from the PaaS context.

For portability of the functional interface two scenarios are possible[11]: the portability of application dependencies versus the portability of entire applications. One can either port an application with all its dependencies as a single unit of delivery, take the dependencies with the application through extensibility mechanisms like buildpacks or rely on the native support of application dependencies between PaaS. Almost all PaaS use some kind of container virtualization[12] atop of the operating system to manage and isolate applications. One idea is that standardized containers can encapsulate any payload and will run consistently on virtually any server [36]. Another approach is to standardize the packaging of the application and their dependencies so that they can be consistently run on different platforms. Standardization around the unit of delivery

[9] See https://www.openshift.com/developers/cartridge-authors-guide
[10] See http://docs.dotcloud.com/services/custom/
[11] See https://www.openshift.com/blogs/paas-standards-standardize-on-what
[12] See e.g. http://www.docker.io

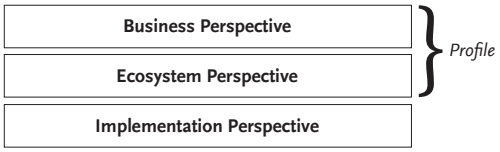

Fig. 4: Perspectives for Categorizing Portability Requirements

would correspond with a uniform virtualization image format like *Open Virtualization Format* (OVF) for IaaS. Consensus on such low-level PaaS artifacts, however, is unlikely and still a long way off. On a higher level, portability can be based on application dependencies. If one wants to port an application to another PaaS, this PaaS has to support either all the application dependencies natively or a developer needs to be able to take these dependencies to the PaaS in a standardized and consistent way. This will enable customers to deploy an application on any PaaS without any drastic changes. Porting application dependencies could be possible if all PaaS agree on a standard format for extensibility, e.g. buildpacks. In this way one can move an application even if the PaaS does not support the dependency by itself. Yet, we can see that vendors also have competing ideas and approaches in this area (See Section III-B). Instead, we want to focus on a no-standards approach that works with the status quo. We actually see consensus in an array of dependencies that are supplied and used for typical application development. As most PaaS already include a wide set of common components for application development, it is not unrealistic to say that there exists a set of dependencies that are used in most of the development cases in the form of a common denominator of core capabilities. This is also motivated by the fact that vendors want to attract as many customers as possible by supporting their development needs. If a collection of PaaS offerings support all necessary application dependencies natively, one should be able to move an application between those vendors. Moreover, this scenario includes both, portability between clouds and application migrations from or to the cloud.

Apart from the aforementioned dimensions, we think we can tackle portability on different levels of granularity. As an example, we take the metaphor of crafting a product. Here, first of all one needs to be sure to have all the components and tools needed to assemble a product. If all parts are present one can be sure that there is a way to build the product but there is most likely not only one way to assemble it. The same applies for an application on a PaaS. If the PaaS offers all the components like runtime languages, services, etc. that an application depends on, one should be able to run the application on the system, but it might be necessary to add some glue here and there to make it happen. These finer details are implementation details. A PaaS may have specific requirements for applications that they must conform to in order to run on the platform. Consequently, we can distinguish between the capabilities we need to craft a product and the conformance to how it must be build. Therefore, we categorize PaaS portability by three different perspectives (See Figure 4).

The most abstract perspective is the business perspective. It includes business relevant nonfunctional and abstract requirements like pricing, compliance or SLAs. The ecosystem perspective describes concrete requirements including application-specific dependencies like runtimes, services, and other capabilities of the platform tier. It can be summarized as all capabilities that form the technical realization of the platform. On the lowest end, we see the implementation perspective. These conformance requirements are portability threats that are implementation-specific requirements or restrictions, e.g. deployment descriptors, restricted usage of runtime APIs or specific management API calls. All capabilities that are specific to the technologies of the ecosystem belong to this layer. Every layer not only has a specific set of portability requirements but also a certain granularity. The properties on the upper two tiers are well-defined capabilities of a PaaS offering that can be mapped to taxonomies. The lower tier includes very specific implementation artifacts and restrictions that need other approaches to formalize and test those requirements like e.g. static analysis or unit tests. We focus on high-level portability of applications in this paper and omit the details of the implementation perspective. Therefore, we formalize capabilities of the two upper tiers, focusing on the ecosystem perspective, into a PaaS profile that can be matched with application requirements. We also incorporate important abstract capabilities from the business perspective to provide a more thorough profile.

V. Platform as a Service Profiles

To make PaaS offerings comparable and matchable, we cluster a set of core properties of the aforementioned business and ecosystem perspectives into a standardized machine-readable PaaS profile[13]. We address the following use cases from [21]. Firstly, the discovery of cloud resources, i.e. selecting an appropriate cloud for an application. In this regard, the profile serves as description language and standard catalog for inter-cloud resource discovery. By means of this discovery we can identify high-level portability for deployment on a single cloud, and the additional use cases of migration from on-premise to cloud as well as the migration between different clouds.

One of the major aspects was the real world applicability of the profiles. They should relieve the current problem of different vendors with diversified capabilities and the incomprehensible status quo. In our opinion, an impartial initiative open for contribution of customers and vendors can help to lead to a better overview and understanding of PaaS. Currently, we face various biased initiatives pushing certain products and a lot of outdated and incomplete information about PaaS offerings. By following the dimensions and components of our model, we also try to solve semantic conflicts between PaaS by providing a common set of capabilities. A major challenge with profiles is to keep them accurate and up-to-date. Current PaaS offerings are changing at a fast pace. Snapshots of the status quo as provided by market researchers are likely to be already outdated when they get published[14]. Even the documentation of the vendors itself is sometimes lagging behind. We tackle this problem with several ideas. First of all, the profiles are open source and can be collectively

[13]The most recent specification can be found at the project homepage at https://www.github.com/stefan-kolb/paas-profiles.

[14]See market research from Forrester [13], [37], Gartner [17] or DZone (http://www.dzone.com/page/comparison-guide-to-cloud-providers-2013).

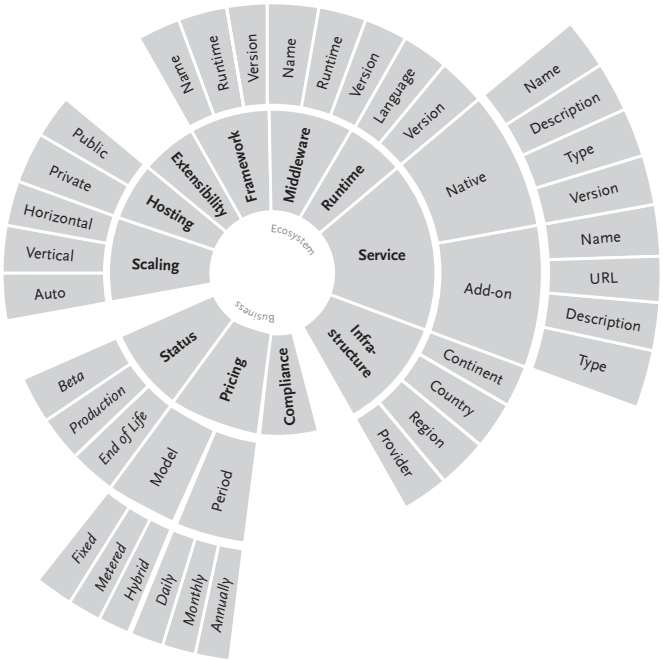

Fig. 5: Platform as a Service Taxonomy

updated and revised. Another idea is that providers can add themselves to the shelf, driven by the fact that they want to become known to the customers. We take respect to this fact by including vendor-verified profiles. Moreover, the profiles and the web interface are continuously updated. If a profile gets updated, it is immediately deployed to production. To our knowledge, this is by far the most recent and comprehensive publicly available collection of PaaS vendors[15].

As a next step, we transform the PaaS model into a concrete taxonomy describing essential parts of a PaaS with the different perspectives in mind. Figure 5 shows the extracted taxonomy on which the profiles are based. Capabilities belonging to either the business or ecosystem perspective are visually clustered. The taxonomy depicts a restricted set of properties that are present for the majority of PaaS offerings in order to avoid missing values in the profiles and to allow for reasonable matching. Not all properties that may be derived from the PaaS model are included in the taxonomy. Some are missing because they cannot be compared or are too specific. For example, APIs are too fine-grained and often nonportable when being vendor-specific and are therefore omitted in the profile. We also include other business-relevant information that is not depicted in the PaaS model but beneficial for a real world comparison, as a simple proof of supplying all application dependencies will not satisfy a business decision in practice. This should make the profile more self-contained and complete.

A. Profile Specification

Listing 1 shows an exemplary PaaS profile definition. Whereas the taxonomy's properties could be specified in any markup language, we explicitly choose the *JavaScript Object Notation* (JSON) because of its wide application in cloud-based systems. It is human-readable, de facto standard for RESTful APIs and appropriate for direct injection in document-oriented databases. The representation also enables the possibility to serve the profile directly through the PaaS API of a certain vendor. This would add more transparency to the offerings as one could retrieve relevant information about the PaaS via a standard API call. We aim at restricting the possible values, so all profiles can be compared against each other. Where possible, we try to rely on commonly known and established concepts in order to have an intuitive profile creation process.

[15] At the time of writing we had 68 vendor profiles.

```
{
  "name": "Uniba Paas",
  "revision": "2014-01-26",
  "vendor_verified": "2014-01-26",
  "url": "http://PaaSify.it",
  "status": "production",
  "status_since": "2013-08-01",
  "pricing": [
    {
      "model": "fixed",
      "period": "monthly"
    }
  ],
  "compliance": [
    "SSAE 16 Type II",
    "ISAE 3402 Type II"
  ],
  "scaling": {
    "vertical": true,
    "horizontal": true,
    "auto": false
  },
  "hosting": {
    "public": true,
    "private": false
  },
  "infrastructures": [
    {
      "continent": "NA",
      "country": "US",
      "region": "Virginia",
      "provider": "Amazon Web Services"
    }
  ],
  "runtimes": [
    {
      "language": "java",
      "versions": [
        "1.6", "1.7"
      ]
    }
  ],
  "middleware": [
    {
      "name": "tomcat",
      "runtime": "java",
      "versions": [
        "6.0.35"
      ]
    }
  ],
  "frameworks": [
    {
      "name": "rails",
      "runtime": "ruby",
      "versions": [
        "3.*", "4.*"
      ]
    }
  ],
```

```
  "services": {
    "native": [
      {
        "name": "mongodb",
        "description": "Document database",
        "type": "datastore",
        "versions": [
          "2.2"
        ]
      }
    ],
    "addon": [
      {
        "name": "mongohq",
        "url": "https://www.mongohq.com/",
        "description": "MongoDB as a Service",
        "type": "datastore"
      }
    ]
  },
  "extensible": false
}
```

Listing 1: An Exemplary PaaS Profile

1) Meta Information: Besides the main concepts from the PaaS taxonomy, we introduce additional meta information to the profiles. In order to verify and keep track of the profile itself it includes the properties *revision* and *vendor_verified*. The property *revision* dates the last change or update of the profile. The property *vendor_verified* denotes if and when the profile was last verified and officially audited by the vendor itself.

2) Business Properties: The business properties either describe the PaaS as a whole or are related to the business perspective of the PaaS. This includes the official *name* of the PaaS offering and the *url* leading to the PaaS' webpage. As a qualifier for the maturity of the PaaS, a profile includes the *status* of the offering and the time when this status became active. This consists of the following lifecycle stages: 'beta' if it is in private or public beta testing, 'production' when the offering is live and generally available, and 'eol' (end of life) if it is discontinued or integrated into another offering. Moreover, the profile includes all available *pricing* options. This can be empty if it is royalty-free open source or no billing is announced yet. Otherwise, this includes an object with the pricing *model* and the billing *period*. All billing options are subscription-based. The model can either be *fixed*, *metered* or *hybrid* billing. Fixed billing describes a one-off fee that is payed for a certain amount of resources (excluding additional bandwidth transfer after a threshold) during a certain period. Metered pricing is based on a sole consumption paradigm. Here, all resources are billed by a fee per unit contract. Hybrid pricing is a mixture of both models where a fixed fee in combination with a metered consumption is applied. The billing periods can typically be daily, monthly or annually. As described in Section III, *compliance* with certain laws or security standards is crucial for enterprises. The profile includes an array of certified standards that are fulfilled by the PaaS.

3) Ecosystem Properties: A major benefit and essential characteristic of cloud environments is their rapid elasticity, in other words *scaling* of the application resources [15]. One does typically differentiate two methods for adding more resources to an application: *horizontal* and *vertical* scaling. Vertical scaling (scale-up) adds more resources to the same logical unit, i.e. instance, in terms of e.g. CPU or RAM capacity. In contrast, horizontal scaling (scale-out) scales the number of application instances that may serve user requests. Both tasks can be done manually or automatically based on policies, according to application demands. The property *auto* scaling describes if the PaaS is capable of scaling any of the above properties automatically.

The *hosting* property conforms to the available deployment models of the PaaS as defined in [15]. Although, values are limited to public or private clouds. A community cloud is just another form of privately managed cloud. An offering is considered capable of being a hybrid cloud if it offers both public and private deployments.

If an offering is available as public PaaS, the profile includes all *infrastructures* an application can be deployed to. The location of any infrastructure can be localized by four properties: The *continent* where the data center is located in, the *country*, the *region*, and an optional *provider* field. The continent must be encoded with one of five continent codes for Africa, Asia, Europe, North America, South America and Oceania. Also, the country codes must conform to the two-letter codes defined in ISO 3166-1 [38]. The property *region* can be used to further clarify the location of the data center. This field is free text and may specify a region or even the city the data center is located in. The *provider* field may indicate the name of an external IaaS provider used by the PaaS vendor, e.g. Amazon Web Services for a PaaS run on Amazon EC2 instances.

The four main components of platform application dependencies are *runtimes*, *middleware*, *frameworks*, and *services*. Runtimes include all language runtimes an application can be written in that are officially supported by the vendor[16]. To further classify these runtimes, the properties *language* and *versions* are used. *Language* identifies the official name of the runtime and is limited to a restricted set of languages in order to enable exact matching between them. As several versions of languages are not necessarily backward compatible and newer versions may offer different features, the property *versions* includes all supported language versions. Wildcards may be used for branches or even marking all versions as supported (e.g. '*.*'). *Middleware* includes an array of preconfigured middleware stacks. These are identified by their official *name* which will be compared based on regular expressions to eliminate syntactic differences. This matching procedure is also applied to any of the other platform components. Similar to the runtimes, the field *middleware* includes a version array that indicates supported versions. To group the middleware products to the correct runtime, they have an additional *runtime* field that ties them to the runtime they are used in. Accordingly, *frameworks* consist of the name of the preinstalled and configured framework, the supported versions, and the base runtime. Frameworks and some middleware products are special in terms of portability requirements. They can often be ported as artifacts included in the application package, too. This can release developers from expecting them to be natively available in the PaaS. Services are divided into *native*

[16]Languages added via buildpacks that are not officially supported must not be added. Extensibility is modeled explicitly.

and *add-on* services (See Section III). Native services have a *name* that identifies them. Moreover, they are classified by a *type* field that assigns a category to them in order to be able to infer the usage of the service. To further describe what the service is offering, it might have an additional *description* field. A *version* field is supplied that defines the release of the service for compatibility reasons. Add-ons are handled slightly differently. They are also referenced by their *name*, *type* and an optional *description* but do not include a *version* property. Many of them do not even have a version number as they supply services like analytics, search, messaging or payment that are not necessarily offered as standalone application. These add-ons are consumed as a service and are independent from any particular PaaS. The internal properties will therefore not vary between PaaS providers if they offer third-party integration with the service provider. To that end, a *url* property references the add-on provider's webpage. The property *extensible* indicates if the PaaS supports any mechanism like buildpacks to add custom components to any of the runtime or service stack.

B. Web Application

To be able to apply the PaaS profiles in practice, we implemented a web application that is capable of viewing, filtering, and matching user and application requirements[17]. We present an overview of all listed PaaS offerings as an entry point. The overview includes the most important high-level characteristics *name*, *status*, *runtimes*, *scaling*, *hosting* and *infrastructures*. This gives the user the ability to get a quick overview over the available offerings. Starting from there, one may navigate to a detail page showing all information provided by the profile, structured and partially visually enriched for better accessibility. One may also directly go to the filtering and matchmaking capabilities. The results can be influenced by applying multiple filters along the properties defined in the profile specification. Additionally, the profiles can be retrieved, searched, and matched via a RESTful API. The implementation itself is PaaS-based. This serves another important aspect of our portability considerations: the validation of the initial portability of applications based on ecosystem capabilities (See Section V-C). The web interface is based on the Sinatra[18] DSL depending on the Ruby runtime. As the JSON profiles can be easily imported and used in a document-oriented database, we choose MongoDB[19]. MongoDB is a scalable, high-performance, open source NoSQL database. As an *Object-Document-Mapper* (ODM) that implements the data mapper pattern, we use Mongoid `3`[20] which has a requirement on Ruby `1.9.3` or `2.0` and MongoDB version greater than `2.2`. This MongoDB version is not widely accessible as native service, so we decide to use an add-on service for this purpose, in that case MongoLab[21].

C. Application Portability Matchmaking

The web application includes a matchmaking capability which returns matching vendors for a definition of capabilities.

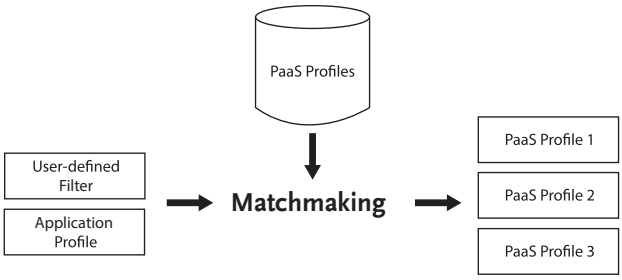

Fig. 6: Application Portability Matchmaking

The application portability matching (See Figure 6) can either be done visually by selecting all necessary dependencies in the web interface or via an application profile that is automatically matched against the PaaS profiles. A query on the PaaS profiles is actually a profile by itself. An application profile can include arbitrary properties that are included in the profile specification (See Section V-A). Of course not all of them will be sensible and needed to describe application dependencies and PaaS capabilities. The matching of all given properties except the *version* properties is `AND` concatenated, i.e. all properties must exactly match with a compared PaaS profile. The *version* attributes, however, are treated as `OR` concatenated in the query because an application is typically only dependent on one specific or any of a set of versions. In order to allow better matching between concepts that are not identified by a restricted set of values like *middleware*, *frameworks*, and *services*, their names are compared based on regular expressions. Partial matching, i.e. differentiating between required and optional capabilities or the inclusion of results that only match a portion of properties is currently not supported by default. We omit this option because we think portability is primarily about must-haves. Still, we can adapt to these scenarios by adding and removing optional capabilities in the filter or by querying alternative configurations. To be able to automatically include feasible partial PaaS matches, the algorithm needs to be enriched with further semantic information about which properties can be manually upgraded even if they are not natively supported, e.g. the ability to use extensibility mechanisms for missing runtime languages or to replace missing services with external add-ons.

Listing 2 shows the application profile for the web application prototype. A suitable PaaS must allow public deployment, has to be horizontally scalable, support the Ruby runtime in either version `1.9.3` or `2.0` and the MongoLab add-on. With these requirements the prototype returned five matches that fit our application profile. These are AppFog[22], CloudFoundry.com[23], cloudControl[24], EngineYard[25] and Heroku[26]. All platforms have quite a few contrarieties and similarities which make the comparison interesting. Although AppFog and CloudFoundry.com are based on the *Cloud Foundry* (CF) open source PaaS, AppFog emerged from the `1.*` branch of CF while CloudFoundry.com has already migrated to the new

[17] The project homepage is https://github.com/stefan-kolb/paas-profiles. An online version of the web interface can be found at http://PaaSify.it.

[18] See http://www.sinatrarb.com/

[19] See http://www.mongodb.org/

[20] See http://mongoid.org

[21] See https://www.mongolab.com/

[22] See https://appfog.com

[23] See https://www.cloudfoundry.com

[24] See https://www.cloudcontrol.com

[25] See https://www.engineyard.com

[26] See https://www.heroku.com

major version 2. Heroku and cloudControl are independent proprietary systems but cloudControl reuses key parts of Heroku's concepts. Both use buildpacks to install and configure different application dependencies. Furthermore, cloudControl has a similar Git-based deployment workflow like Heroku. EngineYard does not have a counterpart in this set.

```
{
    "hosting": {
        "public": true
    },
    "scaling": {
        "horizontal": true
    },
    "runtimes": [
        {
            "language": "ruby",
            "versions": [
                "1.9.3",
                "2.0"
            ]
        }
    ],
    "services": {
        "addon": [
            { "name": "mongolab" }
        ]
    }
}
```

Listing 2: Application Profile for the Web Prototype

While deploying onto the different PaaS, we come across several differences that require partially very different deployment workflows and also code changes to get the application running. In this paragraph we give a nonexhaustive overview of some of the findings. Any of the PaaS use their own *command-line client* (CLI) for communicating with the platform. Not all of them allow a continuous workflow with the CLI but require additional manual deployment steps for certain tasks like add-on provisioning via the Web UI. The API methods of the management interface are generally very different for most tasks between the vendors and the APIs in general are far from compatible. Some PaaS require the application artifacts to be in *Git* revision control in order to deploy them onto the PaaS. Besides very different deployment workflows on the management interface, we also have to adapt the application artifacts. The recognition of the type of application (in this case Ruby/Sinatra) is based on configuration files and code characteristics. Different PaaS are not necessarily able to detect the correct type with the same set of configuration files. Furthermore, standard mechanisms for specifying the required Ruby version via the *Bundler* dependency management are not available in all PaaS. Once, we have to manually specify the version via a CLI parameter due to an old *Bundler* version running on the PaaS. Some PaaS support a direct invocation of shell commands on the environment to populate the associated database via *Rake* build commands while others require to tunnel the services and access them from the local machine. Also, the structure of the environment variables that bind the application to the services is different between vendors which requires reprogramming of parts of the application.

As described in the preceding paragraph, several changes are required to deploy the application to the vendors but it is possible to get the application running on every PaaS. Despite the simplicity of the application which does not make use of any critical system calls within the environment that might be conflicting, we can validate and conclude initial findings from our research. The results support our initial hypotheses that we can actually identify ecosystem portability that allows us to tell if we can run our application on a PaaS from a high-level ecosystem perspective, and that there also is a narrower implementation perspective, which must be investigated independently, that includes various additional requirements and restrictions.

VI. Related Work

Whereas a lot of standardization bodies and groups are working on cloud portability and interoperability standards on IaaS level, only few directly target PaaS. As we have argued in this paper, PaaS' capabilities and functionalities are fundamentally different from IaaS and therefore need to be focused separately in terms of standardization. In general, the PaaS service model benefits less from standardization than IaaS [18]. The platform components and capabilities are too different between vendors and too plenty to be standardized. As stated in Section IV, standardization on the functional interface can also happen on the unit of delivery. The *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [39] specifies a portable description for the structure of applications, their component services and artifacts including management and operational behavior of those. In order to run these standardized application packages a TOSCA-compatible runtime environment is necessary on the target cloud. This approach asserts to be feasible for IaaS and PaaS environments. Whereas on this functional interface only few efforts are currently developed, it is another matter with the management interface. The management API that controls the monitoring and application lifecycle will have widely the same set of functionalities between PaaS. While the *Open Cloud Computing Interface* (OCCI) [40] standard claims to be applicable for all layers, it is still missing a concrete proposal besides the IaaS management API. Another initiative *Cloud Application Management for Platforms* (CAMP) [28] does specifically target the management interface of PaaS. Technically, CAMP defines interfaces for self-service provisioning, monitoring, and control of cloud platforms. While there are standardization efforts ongoing, none of them has gained significant traction in practice. Like with many other cloud standards, an important factor for this situation is the lack of acceptance and disregard from established technology leaders in this area that prevents any widespread adoption of standards. In contrast, our approach is directly applicable to all vendors in practice.

A related but narrower scoped initiative is the *Cloud Foundry Core Definition* (CFCD)[27]. It defines a baseline of common capabilities in order to promote cloud portability among different Cloud Foundry PaaS offerings. However, the definition's capabilities are limited to runtime languages and native services. The CFCD defines a set of specific versions of these runtimes and services that developers can use to build portable applications. The Cloud Foundry team recognizes that application services and runtimes continue to evolve, so they

[27] See http://core.cloudfoundry.org/definition

are willing to introduce a system of deprecated, current, and next versions. These capabilities can be validated by entering the API endpoint of the Cloud Foundry service[28]. Whereas this specification targets the portability of applications considering the PaaS ecosystem, we argue that it does only consider a small scope of capabilities that are needed for compatibility. Compared to our specification, important aspects like e.g. middleware, frameworks, add-ons or scaling capabilities are neglected. Although being explicitly defined as CF definition, the concept could be transferred to other PaaS, too. We tested against our data but did not find any non-CF PaaS that fulfills the specification. In contrast to the fixed set of capabilities and versions of the CFCD, our approach does not need to declare certain property values but does naturally depict the current state of PaaS which is a better fit for user- and application-specific requirements. In our scope, generic portability matching has an edge over one specific specification. Furthermore, it seems that the specification is currently neglected because of the switch to Cloud Foundry v2. It is unclear how v2 will influence the specification as v2 itself is not compatible with the CFCD.

In [41] the authors present a concept of a *Cloud Service Broker* (CSB) that should be able to find a best match for a PaaS application and automatically deploy it through a generalized API to the provider or set up an appropriate PaaS solution on an IaaS provider if no matching provider can be found. To achieve this, the CSB should be aware of application requirements and available cloud offers. However, besides the architecture for the cloud broker and the use cases, the authors do not present a solution how those application requirements should be collected or represented neither than how the existing cloud offerings and their capabilities are structured and matched.

The European Commission funded project Cloud4SOA[29] [42] is equally motivated. Among other capabilities, they also include matchmaking into their work. Even though the underlying *PaaS Semantic Interoperability Framework* (PSIF) model [24] is comparable with our PaaS model, it is relatively high-level. It consists of a PaaS system which may have multiple PaaS offerings (based on language runtime) that provide software components and a management interface that hosts applications inside an IaaS system. The inherited functional, nonfunctional, and execution semantics of the PaaS entities are only roughly specified. An announced detailed specification of the fundamental PaaS entities is not available yet. Consequently, important capabilities and a clear separation of concerns especially at the platform tier is missing. There is no distinction between components of the service and runtime stacks but all functionality is clustered into a software component entity. Their profiles are divided by programming language. Whereas having a profile for each language is reasonable, as some capabilities are only available in a specific language (mapped by a runtime property in our specification), the multitude of profiles will make it even harder to keep the profiles consistent and up-to-date. For matchmaking, Cloud4SOA allows selecting certain required capabilities and optional requirements that resolve into an ordered result on a percentage basis. We omit this option because we think portability is not about options but must-haves. Still, we can adapt to that scenario by adding or removing optional capabilities in our filter interface.

[28]Compatibility is validated by calling specific API targets returning the necessary capability descriptions.

[29]See http://www.cloud4soa.eu/

VII. Conclusion and Future Work

In this paper, we presented a model that describes current Platform as a Service offerings and deducted an ecosystem profile to enable comparison and portability matching based on application dependencies and capabilities. We also investigated more on different portability threats and possible solutions for them from a PaaS point of view. With data from 68 PaaS vendors, we offer a comprehensive overview of the fragmented vendor landscape. Furthermore, we implemented a web interface that allows users to take advantage of the PaaS profiles. Besides giving an overview over available products, it is possible to filter on capabilities and do matchmaking by configuring required capabilities for application portability. Whereas our results allow for validating portability between PaaS on a high-level, this still does not include lower level portability problems in terms of implementation details. We validated this hypothesis by porting our application to five different vendors and identified several low-level problems. Although we could generally port our application, it involved additional (re)programming and significantly different workflows to migrate the application. These problems include platform- and cloud-specific requirements and restrictions[30] as well as management API differences. These factors have impact on the migration of applications from one cloud to another and also from on-premise to cloud environments. As next steps, we would like to investigate more on both of those adjacent perspectives separately.

[30]See e.g. http://12factor.net/

References

[1] Y. V. Natis, B. J. Lheureux, M. Pezzini, D. W. Cearley, E. Knipp, and D. C. Plummer, "PaaS Road Map: A Continent Emerging," Gartner, Tech. Rep., January 2011, http://www.gartner.com/id=1521622.

[2] F. Biscotti, Y. V. Natis, M. Pezzini, T. E. Murphy, P. Malinverno, M. C. D. Feinberg, W. R. Schulte, T. Friedman, J. Thompson, B. J. Lheureux, E. Thoo, and B. Huang, "Market Trends: Platform as a Service, Worldwide, 2012-2016, 2H12 Update," Gartner, Tech. Rep., October 2012, http://www.gartner.com/DisplayDocument?id=2188816.

[3] R. P. Mahowald, C. W. Olofson, M.-C. Ballou, M. Fleming, and A. Hilwa, "Worldwide Competitive Public Platform as a Service 2013–2017 Forecast," IDC, Tech. Rep., November 2013, http://www.idc.com/getdoc.jsp?containerId=243315.

[4] Y. Peraza and G. Zwakman, "Cloud Computing: Overview Report 2013," 451Research, Tech. Rep., August 2013, https://451research.com/report-long?icid=2863.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[6] S. Ortiz, "The Problem with Cloud-Computing Standardization," *Computer*, vol. 44, no. 7, pp. 13–16, 2011.

[7] J. Bitzer, "Commercial versus open source software: the role of product heterogeneity in competition," *Economic Systems*, vol. 28, no. 4, pp. 369–381, 2004.

[8] D. Durkee, "Why Cloud Computing Will Never Be Free," *Communications of the ACM*, vol. 53, no. 5, pp. 62–69, 2010.

[9] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani, "Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 243–254, 2010.

[10] K. Sun and Y. Li, "Effort Estimation in Cloud Migration Process," in *7th IEEE International Symposium on Service-Oriented System Engineering (SOSE)*, 2013, pp. 84–91, San Francisco, United States.

[11] L. Badger, T. Grance, R. Patt-Corner, and J. Voas, "Cloud Computing Synopsis and Recommendations," *NIST Special Publication 800-146*, 2012.

[12] G. S. Machado, D. Hausheer, and B. Stiller, "Considerations on the Interoperability of and between Cloud Computing Standards," in *27th Open Grid Forum (OGF27), G2C-Net Workshop: From Grid to Cloud Networks*, 2009, Banff, Canada.

[13] J. R. Rymer and J. Staten, "The Forrester Wave: Enterprise Public Cloud Platforms, Q2 2013," Forrester, Tech. Rep., June 2013, http://www.forrester.com/The+Forrester+Wave+Enterprise+Public+Cloud+Platforms+Q2+2013/fulltext/-/E-RES86441.

[14] S. Gudenkauf, M. Josefiok, A. Gring, and O. Norkus, "A Reference Architecture for Cloud Service Offers," in *17th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, September 2013, Vancouver, Canada.

[15] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *NIST Special Publication 800-145*, September 2011.

[16] S. Ried, "Multiple PaaS Flavors Hit The Enterprise," Forrester, Tech. Rep., August 2012, http://www.forrester.com/Multiple+PaaS+Flavors+Hit+The+Enterprise/fulltext/-/E-RES78101.

[17] Y. V. Natis, J. Tapadinhas, W. R. Schulte, M. Pezzini, M. Cantara, J. Feiman, D. Feinberg, J. Murphy, T. E. Murphy, P. Malinverno, G. V. Huizen, A. White, B. O'Kane, N. Heudecker, E. Thoo, J. Thompson, G. Phifer, and I. Finley, "Platform as a Service: Definition, Taxonomy and Vendor Landscape, 2013," Gartner, Tech. Rep., June 2013, http://www.gartner.com/id=2515316.

[18] G. Lewis, "The Role of Standards in Cloud-Computing Interoperability," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, United States, Tech. Rep., October 2012.

[19] ISO/IEC/IEEE 24765, *Systems and software engineering – Vocabulary*. International Organization for Standardization, 2010.

[20] Cloud Computing Use Case Discussion Group, "Cloud Computing Use Cases White Paper," July 2010, http://opencloudmanifesto.org/Cloud_Computing_Use_Cases_Whitepaper-4_0.pdf.

[21] M. Hogan, F. Liu, A. Sokol, and J. Tong, "NIST Cloud Computing Standards Roadmap," *NIST Special Publication 500-291*, 2011.

[22] OCCI, *Open Cloud Computing Interface – Infrastructure*. Open Grid Forum, 2011.

[23] N. Loutas, E. Kamateri, F. Bosi, and K. Tarabanis, "Cloud computing interoperability: the state of play," in *3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2011, pp. 752–757, Athens, Greece.

[24] N. Loutas, E. Kamateri, and K. Tarabanis, "A Semantic Interoperability Framework for Cloud Platform as a Service," in *3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2011, pp. 280–287, Athens, Greece.

[25] C. Höfer and G. Karagiannis, "Cloud computing services: taxonomy and comparison," *Journal of Internet Services and Applications*, vol. 2, no. 2, pp. 81–94, 2011.

[26] R. Prodan and S. Ostermann, "A Survey and Taxonomy of Infrastructure as a Service and Web Hosting Cloud Providers," in *10th IEEE/ACM International Conference on Grid Computing*. IEEE, 2009, pp. 17–25, Banff, Alberta.

[27] D. Hilley, "Cloud Computing: A Taxonomy of Platform and Infrastructure-level Offerings," Georgia Institute of Technology, Tech. Rep., April 2009.

[28] OASIS, *Cloud Application Management for Platforms Version 1.1 – Draft 03*. Organization for the Advancement of Structured Information Standards, July 2013.

[29] European Union, "Directive 95/46/EC of the European Parliament and of the Council on the Protection of Individuals with Regard to the Processing of Personal Data and on the Free Movement of Such Data," *Official Journal of the European Communities*, vol. L 281, pp. 31–50, November 1995.

[30] W. Jansen and T. Grance, "Guidelines on Security and Privacy in Public Cloud Computing," *NIST Special Publication 800-144*, 2011.

[31] D. Petcu, "Portability and Interoperability between Clouds: Challenges and Case Study," in *Towards a Service-Based Internet*. Springer, 2011, pp. 62–74.

[32] D. Bradshaw, G. Folco, G. Cattaneo, and M. Kolding, "Quantitative Estimates of the Demand for Cloud Computing in Europe and the Likely Barriers to Up-take – SMART 2011/0045," July 2012, http://ec.europa.eu/information_society/activities/cloudcomputing/docs/quantitative_estimates.pdf.

[33] D. Petcu, G. Macariu, S. Panica, and C. Craciun, "Portable Cloud applications – From theory to practice," *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1417–1430, 2013.

[34] A. Sheth and A. Ranabahu, "Semantic Modeling for Cloud Computing, Part 2," *IEEE Internet Computing*, vol. 14, no. 4, pp. 81–84, 2010.

[35] K. Oberle and M. Fisher, "ETSI CLOUD – Initial Standardization Requirements for Cloud Services," in *Economics of Grids, Clouds, Systems, and Services*. Springer, 2010, pp. 105–115.

[36] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 275–287, 2007.

[37] S. Ried and J. R. Rymer, "The Forrester Wave: Platform-As-A-Service For App Dev And Delivery Professionals, Q2 2011," Forrester, Tech. Rep., May 2011, http://www.forrester.com/The+Forrester+Wave+PlatformAsAService+For+App+Dev+And+Delivery+Professionals+Q2+2011/fulltext/-/E-RES56710.

[38] ISO 3166-1, *Codes for the representation of names of countries and their subdivisions – Part 1: Country codes*. International Organization for Standardization, 2006.

[39] OASIS, *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Organization for the Advancement of Structured Information Standards, November 2013.

[40] OCCI, *Open Cloud Computing Interface – Core*. Open Grid Forum, 2011.

[41] C. Goncalves, D. Cunha, P. Neves, P. Sousa, J. P. Barraca, and D. Gomes, "Towards a Cloud Service Broker for the Meta-Cloud," in *12th Conferencia sobre Redes de Computadores*, November 2012, Aveiro, Portugal.

[42] F. D'Andria, S. Bocconi, J. Cruz, J. Ahtes, and D. Zeginis, "Cloud4SOA: Multi-cloud Application Management Across PaaS Offerings," in *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2012, pp. 407–414, Timisoara, Romania.

APPENDIX

The data set consisting of 68 PaaS vendors on which the results in this paper are based can be found at https://github.com/stefan-kolb/paas-profiles. The web application for portability matching is also available at this URL. Moreover, an online version of the web application is accessible at http://PaaSify.it.