34 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

On the Portability of Applications in Platform as a Service

Stefan Kolb

University of Bamberg Press

34 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Band 34

University of Bamberg Press
2019

On the Portability of Applications in Platform as a Service

Stefan Kolb

University of Bamberg Press
2019

Bibliographische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Informationen sind im Internet über http://dnb.d-nb.de/ abrufbar.

Diese Arbeit hat der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg als Dissertation vorgelegen.
1. Gutachter: Prof. Dr. Guido Wirtz
2. Gutachter: Prof. Dr. Schahram Dustdar
Tag der mündlichen Prüfung: 20.12.2018

Herstellung und Druck: docupoint, Magdeburg
Umschlaggestaltung: University of Bamberg Press, Larissa Günther

ISSN: 1867-7401
ISBN: 978-3-86309-631-1 (Druckausgabe)
eISBN: 978-3-86309-632-8 (Online-Ausgabe)
URN: urn:nbn:de:bvb:473-opus4-541025
DOI: http://dx.doi.org/10.20378/irbo-54102

To those who inspired it
but will never read it

Acknowledgments

There are so many things I would like to say and so many people I would like to show my appreciation for helping me to reach the point where I am at now that would go beyond the space of this acknowledgment. So before I have even started, I am already sorry for missing out all the stories and some of you here, please still feel included in some way, you know who you are!

First and foremost, I would like to thank my supervisor Prof. Dr. Guido Wirtz for giving me the opportunity to go on this journey. I am grateful to him for giving me the freedom to pursue my research interests, his understanding and support when help was needed. I would also like to thank the other members of my dissertation committee, Prof. Dr. Daniela Nicklas and Prof. Dr. Sven Overhage, for the discussions and feedback during the making of this thesis. Special thanks to Prof. Dr. Schahram Dustdar for serving as external examiner of my thesis. Many thanks to all of my former colleagues at the Distributed Systems Group, Linus Dietz, Matthias Geiger, Simon Harrer, Jörg Lenhard, Robin Lichtenthäler, Johannes Manner, Andreas Schönberger, Stefan Winzinger, and Christian Wilms, for making our workplace such a nice environment to work in. You did not only integrate me with open arms when I started working at the group, but we also became good friends over time. Thank you for sharing your knowledge with me about academia, programming, and life. Next, I am especially thankful for the help with the administration and other tedious organizational tasks to Cornelia Schecher, our secretary, who is the secret gear that keeps our group going. I also appreciate the exchange with many other Ph.D. colleagues and friends in different faculties of the university. Particularly, I would like to thank Carsten Jürck and Stefan Kufer for the discussions about academia and life that kept me going when I was frustrated along the way. I am obliged to many students that contributed to my work through several theses, including Cedric Röck, Michael Wurst, Thomas Aberle, Lars Taubmann, Petr Vasilyev, Mathias Casar, and Robert Müller. Furthermore, I owe many thanks to the people at Blinks Labs GmbH for providing me with their intellectual property to evaluate my research hypothesis in real-world situations. In place of all employees, I would like to thank my friend Sebastian Schleicher for the smooth and open collaboration. But most importantly, lots of love to my family and friends for accompanying me along the journey through life until this point and in the future.

Kurzfassung

Der Cloud-Hype der letzten Jahre hat zu einer Vielzahl von unterschiedlichen Anbietern und Angeboten über den gesamten Cloudmarkt hinweg, von Infrastructure as a Service (IaaS) über Platform as a Service (PaaS) bis hin zu Software as a Service (SaaS), geführt. Trotz der hohen Popularität existieren weiterhin eine Reihe von Problemen und Defiziten. Im Besonderen bei PaaS wird die angepriesene Portabilität zwischen verschiedenen Clouds durch eine heterogene, schwer vergleichbare Angebotslandschaft, technologische Unterschiede zwischen den Anbietern und das Fehlen von gemeinsamen Standards erschwert. Die Auswahl eines geeigneten Anbieters sowie ein möglicher Wechsel zwischen unterschiedlichen Anbietern kann daher mit erheblichen (Migrations-)Kosten verbunden sein.

Dadurch motiviert befasst sich die vorliegende Arbeit mit der Analyse und der Verbesserung der Portabilität von Anwendungen in PaaS-Umgebungen. Im Zuge dessen werden Hindernisse über den typischen Lebenszyklus einer Anwendung hinweg – von der Auswahl eines geeigneten Cloud-Anbieters über das Deployment der Anwendung bis hin zum Betrieb der Anwendung – betrachtet. Als Grundlage für die weiteren Untersuchungen wird zunächst eine verbesserte Abgrenzung und Konzeptualisierung von PaaS durch eine Kategorisierung, die Festlegung eines Referenzmodells und die Aufstellung einer Wissensdatenbank vorgestellt. Wie sich zeigt, ist gerade die heterogene Angebotslandschaft im PaaS-Bereich eine Hürde für die Beurteilung und Realisierbarkeit von Anwendungsportabilität. Zur Lösung dieses Problems stellt die Arbeit ein Entscheidungsunterstützungssystem für die Auswahl und den Vergleich geeigneter Cloud-Plattformen vor. Dabei wird auf Grundlage des PaaS-Modells eine Heuristik vorgeschlagen, die durch Abgleich des technologischen Softwareökosystems der Anbieter und der Anforderungen einer Anwendung oder eines Nutzers potenzielle Partner ermitteln kann. Mithilfe dieses Systems ist es einem Nutzer möglich, Angebote zu identifizieren, welche eine Anwendungsportierung ermöglichen. Zur Validierung des Ansatzes wird eine Fallstudie mit einer produktiven Cloudanwendung, die zu verschiedenen Cloud-Plattformen migriert werden soll, durchgeführt. In diesem Zusammenhang wird zusätzlich ein geeigneter Bewertungsrahmen zur Messung der Migrationsaufwände erarbeitet, der die Unterschiede zwischen kompatiblen Anbietern quantifizierbar macht. Als ein zentraler Aufwandsfaktor der Migration wird die Applikationsmanagementschnittstelle der Anbieter identifiziert. Trotz semantisch gleicher Anwendungsfälle werden von den Anbietern unterschiedliche Schnittstellen

für das Management der Applikation über den Lebenszyklus hinweg benutzt. Um die Aufwände in diesem Bereich zu reduzieren, stellt die Arbeit abschließend eine vereinheitlichte Schnittstelle für das Anwendungsdeployment und -management vor.

Zusammenfassend liefert die Arbeit Belege für bestehende Probleme der Anwendungsportabilität in PaaS-Umgebungen und stellt einen Rahmen vor, um diese frühzeitig zu erkennen und zu vermeiden. Zudem tragen die Ergebnisse der Arbeit zu einer Verminderung des Lock-in-Effektes durch den Vorschlag eines geeigneten Standards im Bereich der Managementschnittstellen bei.

Abstract

In recent years, the cloud hype has led to a multitude of different providers and offerings across the entire cloud market, from Infrastructure as a Service (IaaS) to Platform as a Service (PaaS) to Software as a Service (SaaS). Despite the high popularity, there are still a number of problems and deficiencies. In particular with PaaS, the advertised portability between different clouds is hampered by a heterogeneous, difficult to compare provider landscape, technological differences between providers, and the lack of common standards. Therefore, the selection of a suitable provider and a potential change between different providers may involve substantial (migration) costs.

Thus, the thesis deals with the analysis and improvement of application portability in PaaS environments. In the course of this, obstacles over the typical life cycle of an application – from the selection of a suitable cloud provider, through the deployment of the application, to the operation of the application – are considered. As foundation for further investigations, an improved delimitation and conceptualization of PaaS through a categorization, the definition of a reference model, and the establishment of a knowledge database is presented. As it turns out, in particular the heterogeneous provider landscape in this area is an obstacle for the assessment and feasibility of application portability. To solve this problem, the thesis presents a decision support system for the selection and comparison of suitable cloud platforms. Based on the PaaS model, a heuristic is proposed which can identify potential partners by matching the technological software ecosystem of the providers with the requirements of an application or a user. With the help of this system, it is possible for a user to identify offerings that enable application portability. To validate the approach, a case study with a real-world application is conducted that is migrated to different cloud platforms. In this context, we also develop a suitable assessment framework for measuring migration efforts, which allows making the differences between compatible providers quantifiable. The application management interface of the providers is identified as a central effort factor of the migration. Despite the semantically identical use cases, different interfaces are used by the providers for the management of the application's life cycle. Finally, to reduce the effort in this area, the thesis presents a unified interface for application deployment and management.

In summary, the work provides evidence of application portability problems in PaaS environments and presents a framework for early detection and avoidance. In addition, the results of the work contribute to a reduction of lock-in effects by proposing a suitable standard for management interfaces.

Contents

List of Figures

List of Tables

List of Listings

List of Definitions

List of Abbreviations

ADT Average Deployment Time

AHP Analytic Hierarchy Process

AJAX Asynchronous JavaScript and XML

ANP Analytic Network Process

API Application Programming Interface

ASP Application Service Provider

AWS Amazon Web Services

BaaS Backend as a Service

BI Business Intelligence

BPM Business Process Management

CaaS Container as a Service

CAMP Cloud Application Management for Platforms

CDN Content Delivery Network

CD Continuous Delivery

CFCD Cloud Foundry Core Definition

CF Cloud Foundry

CIMI Cloud Infrastructure Management Interface

CI Continuous Integration

CLI Command Line Interface

COAPS Compatible One Application and Platform Service

CPU Central Processing Unit

CRM Customer Relationship Management

CRUD Create, Read, Update, Delete

CSS	Cascading Style Sheets
CTR	Click-Through Rate
DBMS	Database Management System
DE	Deployment Effort
DNS	Domain Name System
DOM	Document Object Model
DR	Deployment Reliability
DSL	Domain-Specific Language
DSS	Decision Support System
EC2	Elastic Compute Cloud
ECS	Elastic Container Service
EDS	Effort of Deployment Steps
EMF	Eclipse Modeling Framework
EOL	End of Life
ERB	Embedded Ruby
ES	Expert System
FaaS	Function as a Service
FM	Feature Model
FR	Functional Requirement
FTP	File Transfer Protocol
GUI	Graphical User Interface
HATEOAS	Hypermedia As The Engine Of Application State
HIPAA	Health Insurance Portability and Accountability Act
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

IP Internet Protocol

ISV Independent Software Vendor

IT Information Technology

JSON JavaScript Object Notation

JS JavaScript

JVM Java Virtual Machine

KPI Key Performance Indicator

LCS Longest Common Subsequence

LOC Lines of Code

MADM Multi Attribute Decision Making

MBaaS Mobile Backend as a Service

MCDM Multiple Criteria Decision Making

MODM Multi Object Decision Making

NBS Number of Build Steps

NCC Number of Configuration & Code Changes

NDSP Number of Deployment Step Parameters

NDS Number of Deployment Steps

NFR Nonfunctional Requirement

NIST National Institute of Standards and Technology

OAS OpenAPI Specification

OCCI Open Cloud Computing Interface

OCI Open Container Initiative

ODM Object Document Mapper

OS Operating System

OVF Open Virtualization Format

OWL Web Ontology Language

PaaS	Platform as a Service
QoS	Quality of Service
RAM	Random-Access Memory
RDF	Resource Description Framework
REST	Representational State Transfer
RQ	Research Question
RSS	Rich Site Summary
SaaS	Software as a Service
Sass	Syntactically Awesome Style Sheets
SDK	Software Development Kit
SDN	Software-Defined Networking
SECO	Software Ecosystem
SLA	Service-Level Agreement
SME	Small and Medium-Sized Enterprise
SOA	Service-Oriented Architecture
SPI	SaaS, PaaS, and IaaS
SPI	Service Provider Interface
SPL	Software Product Line
SQL	Structured Query Language
SSO	Single Sign-On
TOSCA	Topology and Orchestration Specification for Cloud Applications
UI	User Interface
URL	Uniform Resource Locator
VLAN	Virtual Local Area Network
VM	Virtual Machine
VPN	Virtual Private Network
VPS	Virtual Private Server

WADL Web Application Description Language

WAR Web Archive

WSDL Web Service Description Language

XaaS Everything as a Service

XML eXtensible Markup Language

XSD XML Schema Definition

YAML YAML Ain't Markup Language

Part I.

Background and Problem Identification

1. Introduction

Parts of this chapter have been taken from [190–194].

This chapter introduces and motivates this work[1]. First, Section 1.1 describes the context and the underlying problem statement of the dissertation project. Next, the outline of the thesis, including the individual research questions as well as the overall research methodology and structure, is explained in Section 1.2. Finally, Section 1.3 gives an overview of the main contributions.

1.1. Context

Distributed systems have become ubiquitous in today's IT world [354]. Thereby, cloud computing is an evolution of previous technologies such as clusters and grids [107]. Over the last years, the cloud hype fostered the establishment of a multitude of cloud offerings, revolutionizing the outsourcing of IT services [42]. The majority of technologies and products capitalize on the evocative banner "as a Service," emphasizing the on-demand availability of the products [147]. At the forefront, the best known concepts are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [237].

Whereas SaaS represents the universe of web applications, IaaS delivers virtualized low-level infrastructure such as Virtual Machines (VMs) [18]. After the rise of IaaS, the higher-level cloud model PaaS is finding its way into enterprise systems [38, 64]. PaaS, also known as cloud platforms, provides managed and highly automated application environments which free developers from configuring servers and reduce developer operations and maintenance efforts. As a result, developers can focus on the application development, thus generating the actual business value. This is particularly beneficial in the context of complex interdependencies of distributed cloud systems. Naturally, the advantages of distributed systems come at a price. Managing large, scalable distributed systems is difficult and costly [142]. However, many of these tasks and configurations are no different between companies. A determining idea behind the delivery of managed application environments is the commoditization and standardization of IT. Managing, updating, and keeping application environments secure is an important task these days that does, however, deliver no competitive edge among the competitors [61]. Especially in the field of

[1] All links in the thesis have been last accessed on January 8th, 2019.

web applications, a number of standard technologies and tools have become prevalent. PaaS vendors have specialized on providing and managing popular collections of these technologies, relieving the customers from managing their own environments. Additionally, new trends such as DevOps, which aims at the unification of development and operations, have brought up new paradigms like Continuous Delivery (CD) that streamline the delivery process of applications [155, 163]. This enables customers to automatically test and release their applications into production at any time, achieving a shorter time to market [337]. Next to the runtime environment, providers have tightly integrated these tools with their platforms to allow higher productivity and a seamless user experience for agile application development [391]. For customers, PaaS has several other benefits apart from the operational aspect that are mainly inherited from the cloud context. The outsourcing of vast parts of the IT stack delivers cost savings to customers, whereas vendors in turn can benefit from the economies of scale by efficiently utilizing the underlying infrastructure [18]. Instead of paying for servers, customers only pay for the resources that they are actually using. As demand grows, the cloud enables transparent scaling from one to thousands of users, allowing the IT to grow dynamically with a customer's business.

Even though the cloud and especially PaaS has grown massively over the last years [38, 116], there are also plenty of concerns acting as market barriers or preventing further adoption [87]. A major concern is the lack of standards among cloud providers, which hinders compatibility and fosters the chances of lock-in effects caused by, e.g., incompatible technologies or proprietary interfaces [124, 217, 283, 332]. Likewise, the application environments differ among PaaS providers and there are no standards in place, which define available interfaces or portability guarantees. Whereas for other cloud services, such as IaaS, these aspects are comparatively well studied and established concepts are available, PaaS is a relatively new field where the market and the portability issues are yet to be understood [149]. In the current situation, application portability is a major concern for companies to avoid vendor lock-in and to retain the ability for future strategical decisions. Not only the fact that business requirements change but also the dynamic cloud market with mergers, acquisitions, and bankruptcy challenges the viability of providers [25, 231, 314]. All these circumstances create a setting where application portability is key to competitiveness and the ability to innovate [314].

Typically, application portability is bound to standardization and conformance of implementations to these standards [314]. However, there are also reasons against standardization, and in the past, software standards have failed to achieve portability in various ways [60, 114, 323]. IT is advancing at a fast pace and vendors as well as users often oppose standards as inhibitors of innovation. This is a double-edged challenge for portability: standardizing enough functionality for technological stability while ensuring that innovation

continues in disruptive areas [314]. For the cloud, especially the lack of acceptance by industry leaders prevents adoption, while the market is still fragmented and evolving at the moment [262, 271]. Standardizing too early entails the risk of committing to an approach or technology that does not meet real-world needs, whereof several early cloud standards have suffered from [230, 271, 314]. Nonetheless, as we will show in this thesis, there are ways around this dilemma and the time has come for consolidation and standardization of certain aspects of cloud platforms.

Having no standards in place does not mean that there are no commonalities between the competitors. Innovation and competition naturally shape a common set of functionalities that are perceived as valuable by customers. When standards are not enforceable, concepts of brokers often come to the rescue [95]. The term broker emphasizes the intermediary role that it provides. Without constraining any rules on the independent providers, brokers can act as negotiators between the customers and different providers. They can bridge semantic differences without changes at the vendor side, providing intermediation, aggregation, or arbitrage [214]. Thereby, a broker offers a single view to the users that abstracts the different proprietary ones. In that way, cloud brokers can provide interoperability and portability across multiple cloud providers [95]. The tasks that a cloud broker can fulfill can take multiple facets. It ranges from high-level services such as provider selection to the intermediation of cloud provider APIs. Broker architectures are a means to facilitate and evaluate common capabilities to diminish differences in the absence of standards. As they can be applied without interference with the heterogeneous products, they provide a way for third parties to enable cross-platform compatibility. Therefore, such an architecture supplies an appropriate framework to evaluate the thesis' findings and proposals on application portability in practice.

Motivated by the described drawbacks, the thesis addresses the analysis and improvement of application portability in PaaS. In this context, obstacles along the typical life cycle of an application – from the selection of an appropriate cloud provider to the deployment and operation – should be observed and tackled. Thereby, at first, decisive factors that hinder the portability are identified and afterwards solutions are developed to reduce or prevent these problems in the future. In total, the thesis shall advance the state of knowledge of PaaS and give recommendations of suitable standards and best practices to avoid vendor lock-in and retain application portability.

The following Section 1.2 describes the specific research questions which are addressed in this work and their coherences in detail. Afterwards, the mapping of the research questions to the structure of the thesis is outlined together with an overview of the main contributions in Section 1.3.

1.2. Outline

The thesis is structured into four parts. The problem description and the relevant background are stated in Part I. Part II particularly addresses the concept, technology, and the state of the art of PaaS. Afterwards, Part III introduces the four core contributions of the thesis that shape a PaaS cloud broker focused on application portability. Each chapter specifies the motivation, design, and evaluation of the respective contribution. Part IV concludes the thesis with a detailed consideration of the main competing approaches and a summary of the thesis.

As the title "On the Portability of Applications in Platform as a Service" suggests, the thesis is concerned with a multitude of different problems regarding application portability. The following Section 1.2.1 motivates and introduces the research questions and the solution approaches of the thesis. Section 1.2.2 describes the methodologies used to answer these questions and links them to the structure of the thesis.

1.2.1. Research Questions

Conceptualization

Due to the popularity of the cloud, a multitude of cloud offerings have emerged over the last years. Putting the cloud tag on a product has become a sales argument. The PaaS sector is no exception to this rule. Whereas the acronym PaaS is on everyone's lips, there exists no generally accepted definition and conceptual delimitation from similar cloud technologies. The notion of a hosted software platform, however, is too broadly defined to shape a market of comparable products. This leads to an incomprehensible status quo and confusion among researchers and customers [21, 132, 217, 302, 349]. Therefore, the first step in this thesis is to investigate the current notion and state of the art of PaaS. This is captured by the first research question:

Research Question 1*: How to distinguish and classify Platform as a Service offerings?*

To improve this situation, we first conduct a literature review to gather and analyze existing definitions. Subsequently, we extract different characteristics and anticipated benefits that define PaaS offerings. In addition, we propose three distinct PaaS clusters in between the boundaries of IaaS and SaaS for refining the differentiation between PaaS offerings. To further evaluate the proposed categorization, we also conduct a market analysis that shows the state of the art and developments inside the PaaS market. Beyond that, we explain the typical architecture of PaaS and the conceptual delimitation between PaaS and similarly motivated technologies as well as the universe of Everything as a

Service (XaaS). The work advances the state of knowledge of PaaS in a way that provides us with an appropriate overview of the current state of the art in this area. Furthermore, it enables a better distinction between PaaS and other cloud services, and consequently allows us to study existing portability issues among them.

Knowledge Management

The PaaS market encompasses many conceptually heterogeneous offerings which also provide a variety of technological capabilities. Until recently, the vendors' documentation and advertisements were the only source of information for customers to acquaint themselves with the topic. Since the whole process involves a lot of manual tasks that are costly and as data is only available in an unstructured form, the need for a consolidated knowledge repository evolved [349, 394]. Such knowledge management is important to allow customers to make informed decisions when evaluating and selecting providers. Nonetheless, no publicly accessible repository with a decent data set in terms of amount, actuality, and quality is available. This leads to research question 2:

> **Research Question 2:** *How to model and capture knowledge of Platform as a Service offerings inside a structured knowledge base?*

As Stachowiak [344] noted, an appropriate model can only serve a particular purpose. Therefore, the purpose is a decisive factor for the choice of relevant attributes of the abstraction. In our case, the model shall assist customers to retain application portability among different vendors. The level of abstraction in PaaS, including diverse software stacks, services, and platform features, also opens up new risks of vendor lock-in and inherits more potential incompatibilities that make it harder to preserve application portability [21, 224, 284]. As of now, there are no widely applied standards in the world of PaaS, which requires a new approach based on technological components and capabilities to compare offerings. Here, the offerings do not share one common set of portable capabilities but rather intersect with one another at many places. Therefore, it is preferable to look at portability between PaaS platforms from a local application view and to dynamically identify a set of compatible partners. We argue that if offerings have intersecting capabilities, then this can be used as a heuristic for evaluating application portability based on application requirements. Therefore, we define a model of PaaS offerings including the technological stack and platform capabilities relevant for application portability. Based on this model, we derive a standardized, machine-readable profile with a common set of capabilities that exist among PaaS providers. The profiles are collaboratively maintained in a publicly accessible repository. Our approach

is empirically validated by providing a web-based application together with a comprehensive data set of 71 PaaS offerings[2].

Decision Support and Selection

Due to the multitude of available PaaS offerings and their differences, provider selection is an important but currently not well-supported task for companies trying to benefit from the technology. Besides making an informed choice, a main issue within this task is to avoid a potential vendor lock-in and retain future options for application portability [85, 286, 332]. Therefore, we state the following question:

> **Research Question 3.1:** *How to find matching cloud platforms for particular application and user requirements?*

The knowledge base, including the provider profiles, is the foundation for answering the research question. In that regard, we present a system that is able to match user and application requirements with the knowledge base and assists users in finding a suitable provider for their requirements. The implementing system *PaaSfinder*[3] is publicly available and utilized by many users around the world.

During system operation, we monitored and evaluated several problems and drawbacks of technological knowledge bases. We discovered that a fair amount of queries did not return results despite the fact that the contents of the search requests are believed to be satisfiable by the market. Therefore, we investigate the following question:

> **Research Question 3.2:** *Are there any issues with the accuracy and satisfaction of user queries caused by data and query biases?*

To that end, we analyze our log data and realize that issues with the data quality and the queries of the users lead to unsatisfying results. We also identify that related approaches often neglect these problems and are based on an optimistic view on the quality of the data and the users' selection queries. Whereas feasible algorithms for selection are discussed fairly often, data and query problems as well as semantic knowledge to account for these issues lack appropriate consideration. Closely associated, most approaches only allow an exact matching of user queries with the data. However, many customers are also interested in offerings that only partially fit their initially specified requirements. Especially, since these decisions are usually not made by fully automated systems but by a human assessment of the proposed candidates. To improve on that end, we phrase the question:

[2] https://github.com/stefan-kolb/paas-profiles
[3] https://paasfinder.org

Research Question 3.3: *How to semantically enhance matching algorithms to find and rank cloud platforms that only partially match the defined requirements?*

As a first step, we identify essential patterns for both query and data biases that lead to unsatisfying results. Next, we propose different enhancements to the data governance and the selection process to reduce these effects. We also develop different semantic enhancements to the selection process to allow for better recommendations and partial matching of user queries. The recommended changes are validated on the user queries and show that the response rate and user satisfaction can be improved by applying the introduced measures. The presented work aims to advance these issues in general and as an application of the problem in the context of cloud platform selection.

Migration

Although the fact that lock-in and migration issues are prevalent among PaaS offerings is widely accepted [21, 85, 152, 286, 332], the effort of application migration in PaaS environments and typical issues of this task are hardly understood. Whereas the migration from on-premises applications to the cloud is frequently considered in current research, less work is available for migrations between clouds [165]. Especially migration studies with real-world applications are scarce [165]. Therefore, to evaluate the current situation for migrating applications between PaaS offerings, we challenge the feasibility of a migration:

Research Question 4.1: *Is it possible to move a real-world application between different cloud platforms?*

To evaluate the research question, we port *Blinkist*[4], a cloud-native Ruby on Rails web application, between seven major PaaS systems. In this study, we focus on the portability of the application artifacts and their deployment. Whereas the general task of migrating the application among the ecosystem-compatible providers proves feasible, the amount of effort differs between the platforms. To compare the varying efforts of the migration between the platforms and quantify the differences, we state the research question:

Research Question 4.2: *What is the development effort involved in porting a cloud-native application between cloud platforms?*

For reasons of reproducibility and objectiveness, we are aiming for code-based metrics rather than human factors such as man-hours. In that regard, we ascertain that there exist no standard methods for measuring migration efforts among clouds. Therefore, we decide to adapt the measurement framework

[4] `https://www.blinkist.com`

for evaluating such characteristics for service orchestrations and orchestration engines, based on the ISO/IEC SQuaRE quality model [159] by Lenhard, Harrer, and Wirtz [207]. With our work, we show that this framework can be properly adapted for evaluating the migration effort between PaaS environments. Using this framework, the study identifies key problems during migrations and quantifies differences between the platforms by distinctive metrics.

Interface Unification and Standardization

Application portability between clouds does not only include the portability of application artifacts but ideally also the usage of the same service management interfaces among vendors [152, 283]. During the migration study, we realized that a fair amount of effort that is caused by the migrations is due to the heterogeneity of the management interfaces. Nevertheless, the performed operations and actions were often semantically equivalent between the platforms. Unification or standardization of management interfaces is a solution to enable the consistent management of applications across several providers [175, 231, 262]. By analyzing the literature, we observed that although the need for a unified interface to manage cloud applications is regularly mentioned [72, 77, 82, 193, 217, 283, 324], the majority of approaches target the IaaS model [152, 285], missing out on PaaS. As the entities and use cases of these cloud models are different [94, 152], we phrase the question:

Research Question 5*: How to unify application management interfaces among cloud platforms?*

To answer this question, we analyze the management interfaces of PaaS systems and the existing literature. Subsequently, we merge the core functionalities in our proposal for a unified application management interface for PaaS. As a validation of the proposed interface, we introduce *Nucleus*[5], a reference implementation targeting four leading cloud platforms and evaluate its utility by typical use cases. The results show both the conceptual overlap and compatibility between the vendor interfaces and the practical applicability of our unification approach.

1.2.2. Research Methodology and Structure

Figure 1.1 depicts the research questions mapped to the thesis structure. The methodologies and steps used to answer the research questions are described in the center of the figure. Next, the relation of the research questions to the parts and chapters of the thesis are depicted.

Every contribution and its respective chapter follows a similar pattern for conducting the research. First, the particular topic is motivated and introduced.

[5] https://github.com/stefan-kolb/nucleus

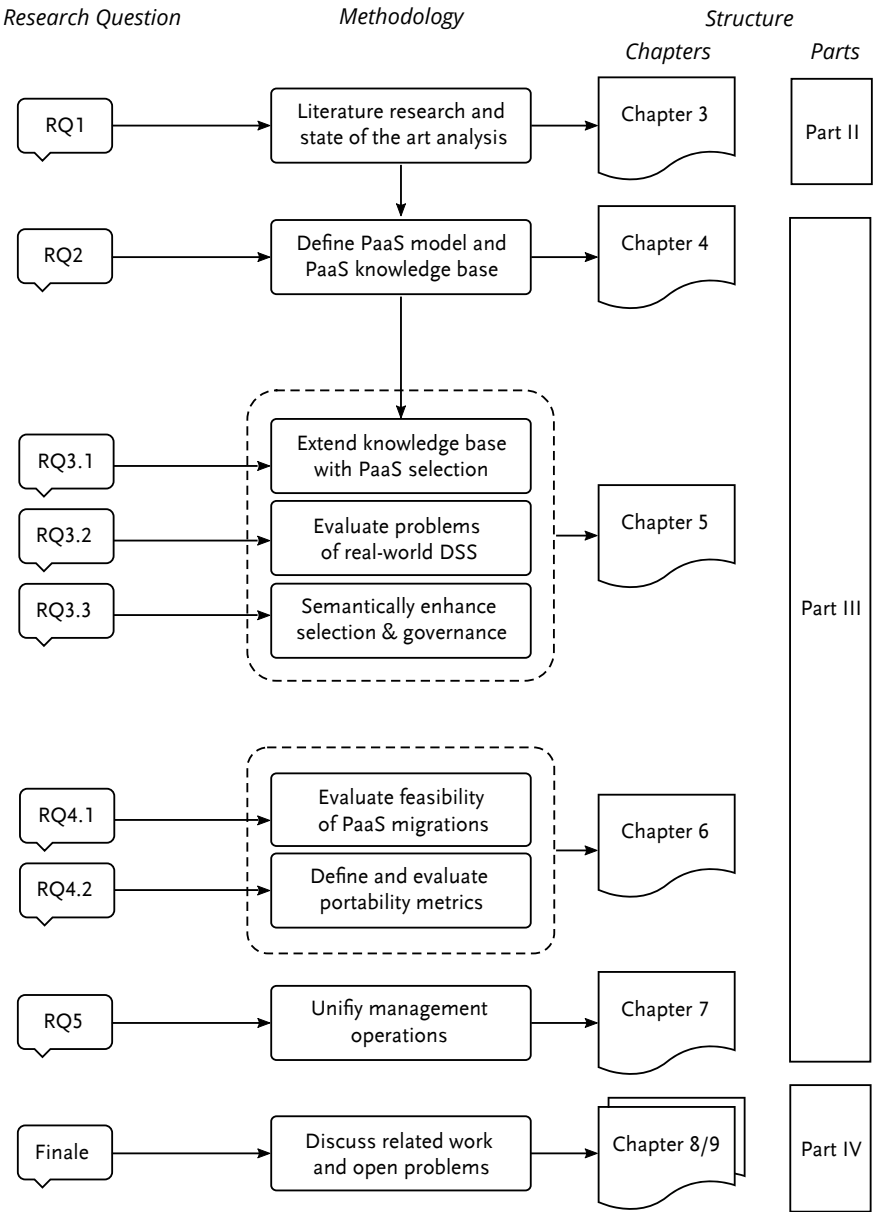


Figure 1.1.: Structure of the Thesis

Where applicable, coherences to and implications from previous chapters are given that determine the follow-up work. Next, the methodology and design of the contribution is described. Afterwards, the work is validated by a practical application of the concept, either by a prototype, a practical evaluation, or both. Related work is presented after the approaches to put the results directly in relation with the state of the art and to show how the approach differs from and enhances existing work. Major competing approaches are additionally discussed in Chapter 8. Finally, limitations, future work, and a summary of the contributions of the chapter are given.

After the general introduction and background in Part I, the following Part II is dedicated to answering RQ 1. Therefore, we commence with an analysis of the existing literature and extract common characteristics and anticipated

benefits of PaaS. Next, we describe the typical architecture of PaaS systems to get a better knowledge of the technical internals to shape the notion of PaaS. Afterwards, we contrast the theoretical view with the actual state of the art via a market analysis and observed developments in the market over time. Finally, we differentiate PaaS from former technologies and other cloud services. In total, this answers the research question and provides a foundation for the following contributions.

Part III includes the main technical contributions and addresses RQs 2–5.

At first, Chapter 4 introduces a generic model of current PaaS systems. Based on this abstraction, we assess different portability challenges of PaaS systems. Influenced by our portability heuristic, based on the intersection of application requirements and the technological ecosystem of the providers, we codify a set of properties into a concrete PaaS profile specification. Afterwards, we collect instances of our formalization to create a knowledge base of current PaaS systems, concluding RQ 2.

In Chapter 5, we present our Decision Support System (DSS) that allows the discovery and matching of those profiles (RQ 3.1). Next, we evaluate real user queries to answer RQ 3.2, before we present our methodologies for data governance and semantically enhanced selection algorithms to account for data and query biases and partial selection (RQ 3.3).

Chapter 6 evaluates the feasibility and major issues when migrating real-world applications between cloud platforms (RQ 4.1). In this context, we develop and present distinct metrics for measuring the effort of application migrations between PaaS systems, targeting RQ 4.2.

Following the results of the migration study, we devote Chapter 7 to the detailed consideration and unification of the management interfaces of PaaS systems. Therefore, we synthesize the literature and the state of the art in our definition of a unified management interface for cloud platforms. We evaluate the feasibility of the proposed approach with a reference implementation supporting RQ 5.

The thesis is concluded by Part IV with a discussion of related work in Chapter 8 and a summary of the contributions in Chapter 9. The final chapter also points out limitations of our approach and open problems that offer areas for future work.

1.3. Contributions

The contributions of this thesis touch several topics in the area of application portability among PaaS systems. All of them are related to the process of evaluating the usage of PaaS for an application, i.e., from the selection of an appropriate vendor to the deployment of an application. Identified problems and gaps were evaluated on the way and the work on an obstacle typically

made us observe subsequent issues. The main contributions of the thesis can be phrased as:

1. *Classification and Model for Platform as a Service*
2. *Platform as a Service Provider Selection and Application Portability Matching*
3. *Evidence for Application Portability Issues and Measurement Framework*
4. *Unified Platform as a Service Application Management Interface*

First of all, we 1) define a more precise classification and model for PaaS that serves as a basis for all investigations and following steps of the thesis. As motivated before, this advances the state of knowledge in a way that provides us with an overview of the current state of the art in this area, enables a better distinction between PaaS and other cloud offerings, and consequently allows us to study and overcome existing portability issues among them.

Next, we present 2) an approach for PaaS provider selection based on a structured data set derived from the model. Besides typical multi-criteria selection, we propose and validate the possibility of application portability matching based on the technological ecosystems of the providers due to the absence of standards in this field. Furthermore, we suggest several enhancements to typical but often neglected problems in technological knowledge bases, such as policies for data governance and semantic enhancements for partial matching of user queries. Such measures should be implemented to optimize the quality of selection results and improve the satisfaction of users.

Consequently, we validate the practicability of our application portability matching with a real-world migration case study among PaaS providers. The evaluation uncovers existing migration efforts and identifies areas of functionality that are particularly problematic with respect to portability. To make migration efforts measurable, we adapt the measurement framework for service orchestrations and orchestration engines based on the ISO/IEC SQuaRE quality model [159] by Lenhard, Harrer, and Wirtz [207] to PaaS environments. The viability of the framework is demonstrated by the utilization in the case study. Overall, the metrics together with the case study form our third main contribution, 3) giving evidence for application portability issues including a suitable measurement framework.

Finally, identified as one of the major effort drivers in the case study, we propose 4) a unified PaaS application management interface as our last contribution. Application portability between clouds not only includes the functional portability of applications but ideally also the usage of the same service management interfaces among vendors [152, 283]. In that regard, unified management interfaces are said to be an important component to accomplish this scenario, as they enable the consistent management of applications across several providers [175, 231, 262]. Despite the fact that management tasks for deploying

Table 1.1.: Publications by Year and Type

	Publisher	Type	Year
[192]	Conference on Cloud Computing	Conference	2017
[312]	University of Bamberg Press	Technical Report	2016
[190]	World Congress on Services	Conference	2016
[194]	Services Transactions on Cloud Computing	Journal	2015
[193]	Conference on Cloud Computing	Conference	2015
[191]	Symposium on Service Oriented System Engineering	Conference	2014

and running applications are semantically equivalent, very different and non-standardized interfaces are used among vendors. Our proposal for a unified API including a prototypical implementation defines itself from related work and the scarce standardization efforts lacking a working reference implementation.

Major parts of the aforementioned contributions have been submitted and subsequently published at scientific forums throughout the dissertation project. This resulted in the following peer-reviewed articles and one non-peer-reviewed technical report. Also, see Table 1.1 for a condensed overview by publication year and type.

[191] S. Kolb and G. Wirtz, "Towards Application Portability in Platform as a Service," in *Proc. Symp. Service-Oriented System Engineering*, 2014.

[193] S. Kolb, J. Lenhard, and G. Wirtz, "Application Migration Effort in the Cloud – The Case of Cloud Platforms," in *Proc. Conf. Cloud Computing*, 2015.

[194] S. Kolb, J. Lenhard, and G. Wirtz, "Application Migration Effort in the Cloud," *Services Transactions on Cloud Computing*, vol. 3, no. 4, 2015.

[190] S. Kolb and C. Röck, "Unified Cloud Application Management," in *Proc. World Congress on Services*, 2016.

[312] C. Röck and S. Kolb, "Nucleus – Unified Deployment and Management for Platform as a Service," University of Bamberg, Tech. Rep., 2016.

[192] S. Kolb and G. Wirtz, "Data Governance and Semantic Recommendation Algorithms for Cloud Platform Selection," in *Proc. Conf. Cloud Computing*, 2017.

The peer-reviewed papers have been published at renowned international conferences. Besides, one of the papers [193] has been extended to a journal publication [194]. The author of the dissertation is the first author in all peer-reviewed publications. This dissertation is based on the research produced and already published in these publications. Consequently, the chapters are to

some extent self-contained and contain slightly overlapping contents, making it eligible to read them in isolation. The thesis puts the presented research into a consistent shape and, beyond that, adds additional insights and discussion. For each upcoming chapter, it is made explicit upon which of these publications the respective chapter is based.

As part of the experimental evaluations of the studies, several software tools and proof-of-concept prototypes have been developed. The author of this work built or significantly contributed to the following tools as a core developer:

- **PaaSfinder** is the main repository for the PaaS broker system, including a user-facing web application and the PaaS provider knowledge base. It implements all viewing and selection capabilities of the broker. The functionality and architecture of PaaSfinder is described in Section 4.5, whereas the matching algorithms are discussed in Chapter 5. The web interface is accessible at `https://paasfinder.org`. The source code is available at `https://github.com/stefan-kolb/paas-profiles`.

  Technologies: Ruby, HTML, JavaScript, MongoDB

- **PaaSfinder Updater** is an enhancement to PaaSfinder to support the editing and update process of the JavaScript Object Notation (JSON)-based provider profiles for nondevelopers. It includes a workflow to visually edit the existing information, review the changes, and automatically create a pull request for the profile inside the PaaSfinder main repository. The necessity and requirements for such an editing workflow are described in Section 5.3. The source code is available at `https://github.com/stefan-kolb/paasfinder-updater`.

  Technologies: Java, Git

- **Nucleus** is a PaaS API abstraction layer implementation of the unified application management interface discussed in Chapter 7. It shows the feasibility of the homogeneous PaaS API by mediating between the proposed and different existing PaaS system APIs. The source code is available at `https://github.com/stefan-kolb/nucleus`.

  Technologies: Ruby

- **PaaSyard** is a Docker-powered deployment system for PaaS. It is used for automating the deployment of applications to PaaS providers in a repeatable and isolated manner. The tool was developed to measure the effort of application migrations discussed in Chapter 6. The source code is available at `https://github.com/stefan-kolb/paasyard`.

  Technologies: Docker, Bash, Ruby

- **PaaSalyzer** is a tool for analyzing the PaaSfinder provider knowledge base. It includes functionality to generate key figures of the current state

of the art of the PaaS landscape and the possibility to extract historical data from the existing data points inside the Git revisions. Statistics and data of the market overview in Section 3.3 are generated with the help of the tool. The source code is available at `https://github.com/stefan-kolb/PaaSalyser`.

Technologies: Java

Each tool is developed as open source and is publicly available. All projects are licensed under the terms of the permissive MIT license[6] and are free to use and distribute. The tools can be used to reproduce and evaluate the results presented in the thesis.

[6] `https://opensource.org/licenses/MIT`

2. Theoretical and Technological Foundations

The thesis is based on various theoretical and technological foundations. This chapter serves as an introduction to important fundamentals. First, Section 2.1 gives an overview of the phenomenon of cloud computing and the different cloud models IaaS, PaaS, and SaaS. The chapter classifies PaaS into the overall context of cloud computing. Next, Section 2.2 considers the challenge of application portability in software systems and particularly the cloud from an abstract point of view. It motivates why application portability is desirable, discusses different subproblems, and highlights the basic problems of application portability in PaaS systems. Section 2.3 introduces the foundations of DSSs and algorithms for solving multi-criteria decisions. DSSs are an important tool for supporting decision makers in heterogeneous markets with a multitude of decision criteria such as PaaS. Finally, Section 2.4 presents the concept of cloud brokerage as a means of mediation to overcome problems of heterogeneity and incompatibility between cloud systems. Cloud brokers can be used as central agents for bridging heterogeneity and as an alternative to the standardization of system functionality.

2.1. Cloud Computing

Already in 1961, John McCarthy envisioned that computation would be available as a public utility in the future [275, 277, 293]. However, until his vision became reality in the form of cloud computing, a few decades had to pass by. One of the first times the term cloud computing was introduced was in a panel discussion with Eric Schmidt, at that time CEO of Google, at the Search Engine Strategies Conference in 2006.[7] In the conversation, he coined cloud computing as an emergent new model where the computational and the data facilities are on servers, accessible to the public via the Internet. The cloud thereby stands for the transparent manifestation and location of the distributed computing systems.

Cloud computing has evolved and overlaps with previous distributed computing paradigms such as clusters and grids [107, 275, 392]. Without going into great detail, Foster et al. [107] see cloud computing as an evolution from

[7] `https://www.google.com/press/podium/ses2006.html`

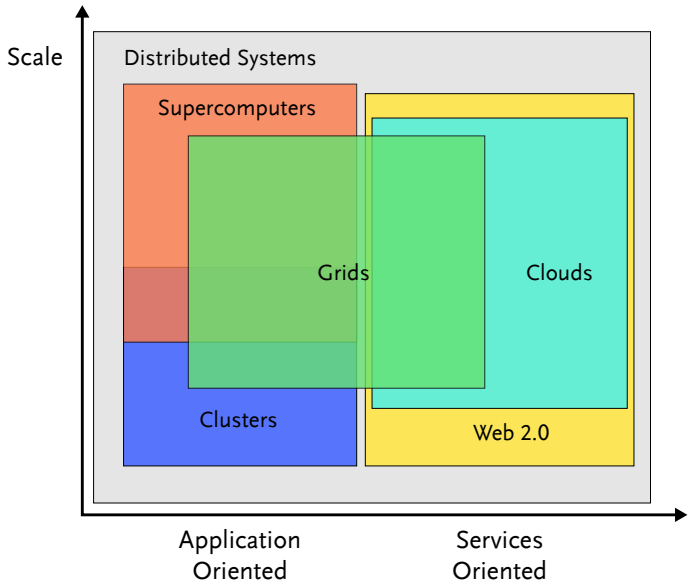


Figure 2.1.: Domains of Distributed Systems [107]

a resource-oriented view, delivering solely storage and compute resources in grids, to a service-based economy with more abstract resources in the cloud. Thereby, cloud computing still relies on the grid and cluster fundamentals as its backbone for infrastructure support [107]. Figure 2.1 depicts the connections between related concepts as envisioned by Foster et al. [107].

Here, Web 2.0 covers nearly the entire spectrum of service-oriented applications, whereas cloud computing is focused on large-scale systems. Supercomputing and cluster computing are more focused on traditional nonservice applications. Grid computing overlaps with all these fields, but it is generally considered of lesser scale than supercomputers and clouds [107].

Overall, there exist many definitions of cloud computing [42, 367]. One of the most cited is the one by the National Institute of Standards and Technology (NIST), which defines cloud computing as:

> **Definition 2.1 (Cloud Computing)** *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* [237]

A main property that distinguishes cloud computing from related technologies such as grids or clusters is the ubiquitous access to the computing resources over the Internet through standard mechanisms that promote use by heterogeneous clients [237]. This *broad network access* happens on demand and can be adapted or configured to the needs of the users, e.g., the amount of resources [18, 237, 293]. Another important factor is the low management effort and provider interaction that is implemented via *self-service interfaces* that can immediately fulfill and provision resources for the users' demands [237]. The *pool of*

computing resources serves multiple users using a multi-tenant model, with different physical and virtual resources that are dynamically assigned and reassigned according to user demand, providing *rapid elasticity* [18, 237]. The cloud user is only charged for the consumed and *measured resources* [18, 237].

Key enablers for these properties are a set of new technological developments and advancements over time. Most importantly, the evolution of the Internet. Better network infrastructures, the ubiquity of broadband and wireless access points, and ever-increasing bandwidths have made it possible to transfer large amounts of data over the Internet, enabling efficient data exchange and access to applications [196, 275]. This is vital not only for the providers but especially for the end users that make use of the services from anywhere, be it on the local computer or via a smartphone. Next, the commoditization of server hardware makes it possible to create large data centers based on the economies of scale [18]. Instead of scaling up via better server hardware, cloud data centers scale out via distributing workload over large numbers of low cost servers and hardware [126, 293]. Finally, the advent of server virtualization technology, including VMs and containers, has enabled to abstract the multitude of servers of a data center and use them as a single transparent resource, where the partition of again virtualized computing environments can happen at any time [24, 80, 339].

These technologies and characteristics provide several benefits to the users. Apart from cost reductions for both IT personal and hardware, due to the outsourcing of IT, the cloud also entails efficiency gains [42]. The appearance of infinite computing resources creates completely new scenarios for users. For example, applications can be flexibly scaled to support fluctuating workloads. Thereby, users have no up-front costs and only pay for the use of computing resources as needed [18]. Using thousands of servers for a short time or a single server for a long time makes no difference on the bill, anymore. Moreover, providers can largely benefit from the economies of scale and less under-utilized resources at data centers compared to conventional server farms [18]. Due to the amount of available cloud offerings, customers have a nearly limitless choice among available services and tools. The streamlined delivery of the IT can also enable users, especially smaller organizations, to achieve shorter time to market for their projects. Furthermore, the cloud can deliver improved availability and security due to optimized, provider-managed environments [167].

Nevertheless, there are also obstacles for using the cloud. One of them is the lifespan of the provider for preserved business continuity, i.e., the existence of a provider over a longer period [25]. The dynamic and fast changing market sees providers coming and going from time to time. This may eventually limit the choice of providers to a very small set of established vendors. Data lock-in is another factor, as only very few resources and operations are standardized in the cloud, especially for more specialized service offerings [264]. Another aspect are security considerations such as data confidentiality [167, 282, 347]. Since

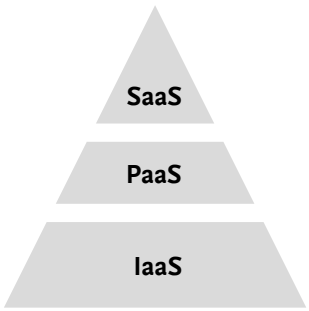


Figure 2.2.: SPI Model Pyramid

data is not stored on premises anymore and no exact storage location of the data may be given due to the virtualization of resources, regulations may prohibit the transfer of sensitive and confidential user data to the cloud [282]. Also, security breaches may make data accessible for unauthorized third parties [347]. A more technological factor is performance unpredictability, which is due to the co-location and shared hardware usage of virtualized environments [18, 293].

The variety of different cloud services are typically characterized by three cloud computing models. These are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). As we can see in Figure 2.2, the relationship between the cloud services models is frequently depicted as a pyramid, suggesting an hierarchical relationship between them. This does not necessarily mean that models at higher levels of the pyramid are based on the lower models but that the technological abstraction is on a higher level. IaaS provides the hardware and low-level capabilities, such as compute, storage, and networking infrastructure [37, 208, 237]. VMs are the most common form for providing computational resources to cloud users at this layer [392]. PaaS delivers a software development environment and components for creating and operating web applications. On the other end, SaaS realizes the delivery of business and web applications over the Internet. IaaS targets the administrators, PaaS the developers, and SaaS the end users or businesses [275]. The emergence of these systems, however, was contrary to today's layering. In the 1990s, vendors started to deliver their software via the Internet following the SaaS model, whereupon IaaS and PaaS became the enablers for the development and operation of such applications in the mid 2000s [33, 150, 293].

For a better understanding, it is worthwhile to examine all three cloud models and the traditional approach of deploying applications, as shown in Figure 2.3.

In a traditional on-premises environment, users are responsible for managing the whole stack from the hardware up to the application. With IaaS, the hardware stack from networking up to the virtualization is provided and managed by the cloud provider. Often, the OS is also delivered or at least pre-configured inside, e.g., a VM. Therefore, the customer only needs to manage the necessary middleware up to the application, thus outsourcing the system administration

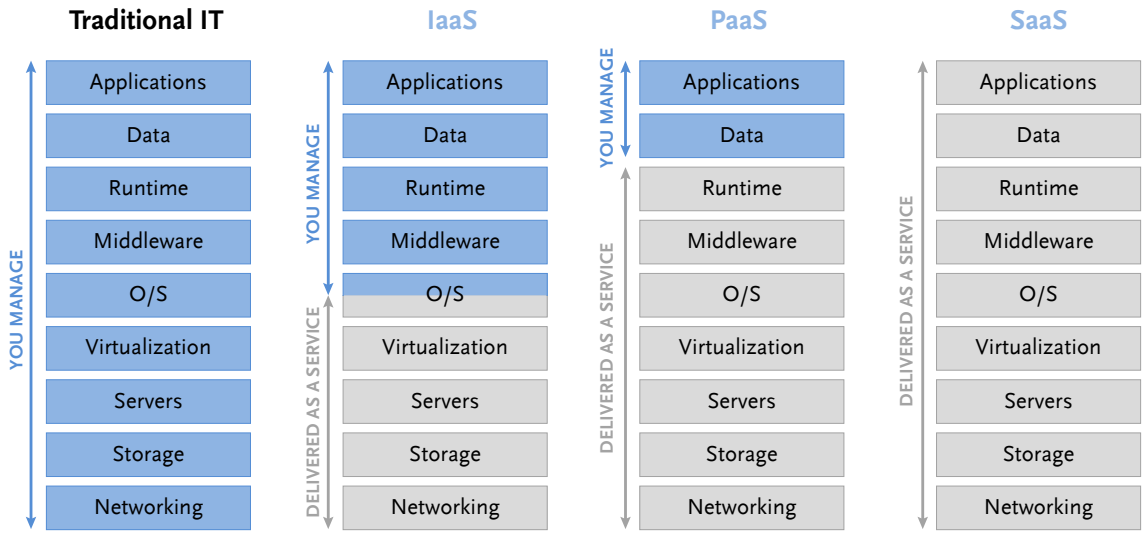


Figure 2.3.: Cloud Stack [142]

part. PaaS additionally provides an application environment including middleware and runtimes to execute applications. Hence, customers can solely concentrate on application development and forward all the administrative and operational work to the provider. With SaaS, the entire computing stack is managed and delivered by the provider, including the application that the customer uses.

The NIST [237] defines four different deployment models for the underlying cloud infrastructure: *public*, *private*, *hybrid*, and *community* cloud. A *public cloud* is a cloud infrastructure that is accessible to the general public. The cloud provider enables access to their infrastructure via the Internet. It is the most common form envisioned when referring to the term "cloud." The *private cloud* is an infrastructure that is provisioned for exclusive use by a single organization. Typically, the infrastructure is owned, managed, and operated by the organization and hosted on premises [237]. This gives the organization a maximum of control over the infrastructure and the security isolation of the system. Private clouds are particularly relevant for sensitive or confidential data [347]. A *virtual private cloud* is a virtual private cloud environment in the public cloud that is reserved and isolated via virtualization for the use of a single customer. Whereas the system appears to the customer as a sole unit, it is only abstracted by Virtual Private Networks (VPNs). The data of the customer still resides in the public cloud off premises [384]. A *hybrid cloud* is a combination of the two concepts public and private. Thereby, the infrastructure of two or multiple clouds are bound together by standardized or proprietary technology to enable data and application portability. This facilitates concepts like cloud bursting for load balancing between clouds. When demand rises, the data is transferred from the private cloud to the public cloud to cope with increasing demands. Another use case is to combine a private and a public cloud to store sensitive data on premises in the private cloud, e.g., user data, and other data

and applications in the public cloud [237]. The *community cloud* is a more open form of a private cloud that is accessible for use by a specific community or group of customers. These customers typically have shared concerns, e.g., mission, security requirements, policy, and compliance considerations [237]. In practice, the term and concept of a community cloud is of little relevance and in most cases blends with the private cloud.

In the following, we describe the three different cloud models IaaS, PaaS, and SaaS in detail.

2.1.1. IaaS

IaaS delivers the most fundamental computing entities, i.e., compute, storage, and networking infrastructure [37, 208, 237]. One of the most popular IaaS services is the provision of virtualized server instances via VMs, essentially introduced by Amazon Web Services (AWS) Elastic Compute Cloud (EC2) in 2006 [293]. Consequently, services such as Virtual Private Server (VPS) are also predecessors of IaaS [374]. The emergence of virtualization technologies such as Xen [24] is a key enabler for today's IaaS. Together with the cloud, and its illusion of infinitely scalable resources, it enabled new possibilities to deliver virtualized server instances compared to the classical partition of a single dedicated server into multiple VPSs. However, IaaS goes beyond only virtualizing entire server instances and includes other low-level entities such as network and storage capabilities. Overall, the NIST defines IaaS as:

> **Definition 2.2 (Infrastructure as a Service)** *"The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls)."* [237]

The general advantages delivered by IaaS are similar to those of the cloud. Through the multitude of fundamental computing resources, it is possible for a user to configure his own virtual data center from the available components variably and on demand.

Processing power is particularly delivered as a virtual CPU (vCPU) share assigned to the VM instances. The users can assign variable amounts of vCPUs and memory to their VM configuration. This also allows extremely powerful and large system configurations.

Moreover, there exists an amount of different storage abstractions that are offered by IaaS. Each abstraction advertises a very high availability, durability, and performance. Fundamental storage facilities are provided by block storage

services, such as AWS Elastic Block Store[8]. Block storages are ideal for applications that require high throughputs and consistent, predictable performance. Most block storages are abstracted by higher level services, e.g., file systems or Database Management Systems (DBMSs) for use by applications and end users. Here, IaaS also provides drop-in replacements for conventional file systems. An example is the AWS Elastic File System[9]. Opposed to block storages, object storages manage data as objects instead of chunks of data. Each object includes the data, metadata, and a unique identifier to address the object. Object storages can save and retrieve any kind of unstructured data and are often used for storing large amounts of text, image, or audio files. A popular example of a cloud object storage is AWS S3[10]. Higher-level data stores, e.g., databases that provide additional sophisticated middleware functionality such as transactions, are an edge case. These are better characterized as Backend as a Service (BaaS) offerings, as described in Section 3.5.2.

Whereas compute and storage resource management is well studied, the use of network resource management in the context of IaaS is still in its early stages. Existing network virtualization technologies alone, e.g., Virtual Local Area Network (VLAN) and VPN, do not provide adequate solutions for today's needs [12]. Software-Defined Networking (SDN) [199] is seen as a promising technology capable of addressing these needs and can be used to make networking even more dynamic for IaaS [12]. OpenFlow [236] is the de facto standard communication protocol to define SDNs.

Other services provided by IaaS providers are firewalls, dynamic load balancers, API gateways, or scalable Domain Name Systems (DNSs).

According to Gartner [251], the IaaS market is currently dominated by AWS[11]. Next to AWS, the most important vendors by market share are Microsoft Azure[12], Alibabacloud[13], and Google Cloud[14]. Besides the proprietary solutions for managing the virtualization of compute, storage, and network, there exist several open source initiatives to implement cloud computing architectures. The best known software is OpenStack[15]. Other representatives are CloudStack[16] and OpenNebula[17].

[8] https://aws.amazon.com/ebs
[9] https://aws.amazon.com/efs
[10] https://aws.amazon.com/s3
[11] https://aws.amazon.com
[12] https://azure.microsoft.com
[13] https://www.alibabacloud.com
[14] https://cloud.google.com
[15] https://www.openstack.org
[16] https://cloudstack.apache.org
[17] https://opennebula.org

2.1.2. PaaS

As PaaS is the main topic of the thesis, we dedicate the entire Part II to a detailed consideration of the conceptualization of the term and technology PaaS. Therefore, we only briefly introduce the concept of PaaS here and refer to Part II for more details.

PaaS is a cloud computing model where the software development and runtime components are delivered by a third-party provider. The NIST [237] defines Platform as a Service as follows:

> **Definition 2.3 (Platform as a Service)** *"The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment."* [237]

The PaaS system includes the runtime environment for user applications, including a stack of runtimes, middleware, and services that are hosted and managed by the provider. Depending on the abstraction of the PaaS system, these components are made explicit or are hidden behind a platform API that must be used by customers to develop applications. Additionally, visual environments for developing applications such as IDE-like UIs are sometimes available. The user is only responsible for developing the application which is then deployed and operated inside the PaaS system including all necessary hardware resources. Examples for PaaS providers are Heroku[18], Cloud Foundry[19], or OpenShift[20].

2.1.3. SaaS

The acronym SaaS reportedly first appeared in a white paper [150] published in February 2001 by the Software & Information Industry Association (SIIA).[21] Nevertheless, on-demand software application delivery models are known since the 1990s and have been described by various acronyms each representing a slightly different approach such as Application Service Provider (ASP) or Business Service Provider (BSP) [33, 150]. In general, the growing interest in on-demand outsourcing models is driven by the will to focus on the business' core competencies and the lack of qualified IT personal and their labor costs. Furthermore, it is fostered by attractive cost models for customers provided by the outsourcing of software [181]. In most cases, the relevant software is not

[18] `https://heroku.com`
[19] `https://cloudfoundry.org`
[20] `https://openshift.org`
[21] `https://en.wikipedia.org/wiki/Software_as_a_service`

the core business of the company and only provides a, nevertheless important, utility function for economic value creation. The concept of SaaS essentially is an alteration and extension of the former idea of the ASP model. The common denominator of all these concepts is the demand-driven application sourcing model that provides a network-based access, e.g., over the Internet or local network, and the management of applications for users and companies [59, 150, 355]. The NIST [237] defines SaaS as:

> **Definition 2.4 (Software as a Service)** *"The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings."* [237]

In the SaaS model, the application provider sells access to an application which shall be hosted in the cloud to the user. The user has no up-front investments for an application license, servers, people, or other resources for acquiring the software but is typically charged based on a subscription model, transforming costs to running expenses [150]. Moreover, the user requires no local installation of the software as the provider is also responsible for the maintenance of the application. This facilitates using software that is complex to set up and ensures that the users are working on a consistent application version on different workstations, as the application is centrally managed and upgraded. The continuous updates by the provider keep all application instances in sync and prevent security as well as compatibility problems between tenants. For the application provider, this provides economies of scale as he can serve a multitude of users. Whereas the software architecture used by early ASPs mandated maintaining a separate instance of an application for each business, SaaS solutions typically utilize a multi-tenant architecture, in which an application serves multiple businesses and users, and partitions its data accordingly.[22] Through modern virtualization technology, this can also be realized via a multi-instance approach, where each tenant gets his own instance of the application. From a development perspective, this is easier to implement but requires more effort for maintenance and upgrades [36]. In the beginning, a lot of ASPs focused on managing and hosting popular third-party software via sublicensing. In contrast, SaaS vendors mostly develop, manage, and sell their own software. Many ASPs offered more traditional client-server applications, which relied on parts of the application being stored or streamed to the client side [150]. Today's SaaS offerings are almost exclusively delivered as web-based applications and only require a web browser on the client side.

[22] `https://en.wikipedia.org/wiki/Software_as_a_service`

This was made possible by new Internet technologies and open web standards such as HTML5, Asynchronous JavaScript and XML (AJAX), and Cascading Style Sheets (CSS). These technologies are browser native and require no additional installation by the users. This goes hand in hand with the evolution away from desktop software to on demand software, e.g., Google Docs or Office Online. Objections against the outsourcing of applications via SaaS are, e.g., risks of reliability, security, and process dependence [33].

2.2. Application Portability

One of the major obstacles when working with software systems is how to prevent or tackle application portability threats. Fundamentally, portability is about retaining the value of previous investments for developing an application [314]. Thereby, the vendor lock-in problem can concern the business, technical, and the legal perspective [264]. With portability, an application can be transferred to different environments without substantial new investments in terms of labor and costs. Here, application portability plays a central role in the cost-effectiveness of IT, while maintaining its quality [314]. Especially the cloud computing landscape consists of a diverse set of products and services in between IaaS, PaaS, and SaaS. Differences between and within the models increase the risk of vendor lock-in for customers [265]. Changing business requirements and the dynamic cloud market with mergers, acquisitions, and bankruptcy further strengthen the necessity for portability [25, 231, 314]. Allowing the customers to migrate applications in response to business values such as faster service, lower cost, or greater reliability is a preference of customers [160]. In that regard, application portability is key to competitiveness and the ability to innovate [314]. The possibility to migrate applications is also critical to future cloud service adoption and the realization of the benefits of computing as a utility [73].

To begin with, we want to take a closer look at the abstract concepts of portability and interoperability. In literature, the terms portability and interoperability are often confused and falsely used as substitutes [210]. Whereas they are related concepts, they both describe different scenarios. The ISO/IEC/IEEE define portability as:

> **Definition 2.5 (Portability)** *"[Portability is 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."* [162]

For the cloud, this means the ability to write code that works with more than one cloud provider regardless of the differences between them [72]. In contrast, interoperability is defined as:

> **Definition 2.6 (Interoperability)** *"[Interoperability is the] degree to which two or more systems, products or components can exchange information and use the information that has been exchanged."* [162]

Hence, interoperability describes the case where two or more systems interact with each other, are able to successfully exchange information according to a prescribed method, and obtain predictable results to fulfill a desired task [161]. An example for interoperability between cloud systems would be a hybrid or federated multi-cloud deployment that needs to interoperate to synchronize data or exchange other information to run one application in multiple sovereign clouds. A more common use case in the context of cloud computing is the ability for a cloud service customer system to interact with a cloud service, e.g., to control the life cycle of an application [161].

A cloud service that is interoperable does not necessarily support portability of applications. Vice versa, a cloud service that supports portability is not inevitably interoperable [160]. Consequently, both perspectives need to be evaluated separately. Nevertheless, application portability between clouds not only includes the functional portability of applications but ideally also the usage of the same service management interfaces among vendors [152, 283]. Therefore, the interoperability of a cloud with the customers' tooling is tightly coupled to a frictionless user experience. Hence, in this thesis, we examine both the portability of applications between PaaS systems and the interoperability of the service management interfaces (see Chapters 4 and 7).

Portability between cloud environments is not a binary decision [160]. That means there typically exist different gradations between portable and nonportable [206]. Often, an application can be ported to another cloud service with equivalent capabilities, if a certain amount of work is invested [160]. In that case, efforts in terms of labor costs caused by changes to the application must be considered and compared to the expected benefits [159]. For that reason, we also investigate the effort of porting an application between PaaS systems. With appropriate metrics, we make the effort measurable and comparable between different providers (see Chapter 6).

An obvious answer to achieve portability and interoperability is often standardization [314]. Naturally, several standardization organizations have addressed cloud standards. In general, standards fall into two categories: de facto standards, which are determined by the market, and de jure, which are defined by procurement bodies [314]. For de jure standards, an open standardization process with democratic influences has gained traction that involves multiple stakeholders and open participation [74, 145]. Such an inclusive standardization process has more chances of getting accepted by the stakeholders than a process that is exclusive [145]. The success of a standard, however, is often not decided by the openness of the standardization process or its technical strength but rather by market share [314]. A major problem for the cloud is

that most initiatives remain disregarded by vendors [271]. Moreover, only very few initiatives consider the PaaS model as their main objective rather than IaaS [124, 217, 283, 332]. Furthermore, the existence of a standard alone does not guarantee portability between implementations and software standards have failed to attain portability in various ways in the past [60]. Examples are a lack of conformance to existing standards or differing interpretations of standardization documents [114, 323]. Especially in disruptive technologies like the cloud, a balance between innovation (nonportability) and standardization (portability) is vital [314]. Standardizing too early entails the risk of committing to an approach or technology that does not meet real-world needs, whereof several early cloud standards have suffered from [230, 271, 314]. The thesis outlines important standards for portability and interoperability in the related work of the respective chapters.

There are several scenarios where portability and interoperability are important for cloud systems [73, 152]:

1. Application migration from on premises to the cloud
2. Application migration between different cloud services
3. Interface with cloud services
4. Interface between cloud services

Portability issues can create initial barriers to cloud entry or for the migration between offerings. A lack of interoperability makes it difficult to interface with multiple clouds or accomplish multi-cloud scenarios through consistent mechanisms. The migration of on-premises applications to the cloud is frequently targeted by current research [15, 34, 138, 182, 274, 368]. As these environments follow different architectures, major changes that are caused by adjusting an application to the cloud paradigm are expected, especially for legacy applications [368]. In contrast, less work is available for application migrations between cloud environments, particularly PaaS. Considering the portability promises of open cloud platforms, consequences of this migration type are less obvious. Of the four scenarios listed, the thesis is concerned with 2) the portability of applications between PaaS environments and 3) the interoperability of the service interfaces to achieve this task.

In the following, we describe important entities involved in the portability and interoperability of cloud services.

Application portability, i.e., the portability of the application artifacts, is closely related to data portability. A migration of an application without the corresponding data rarely occurs. Application portability describes the customer's capability of moving the application between different environments. For the cloud, this is only possible for IaaS and PaaS services, since in SaaS the application is the property of the provider [73]. An application consists of

the source code and typically a set of dependencies that must be supplied to run the application [73, 160]. The dependencies may include runtimes, data stores, or other middleware services. For porting an application, the target environment must support both the application artifacts and the application dependencies [73]. Additionally, the environment must be capable of executing the application equivalently to the original environment, in terms of behavior and possibly further nonfunctional policies [160]. Key to application portability is the ease of moving the application or application components. The application may require recompiling or relinking for the target cloud service, but it should not be necessary to make significant changes to the application code [73, 162].

Data portability describes the ability of a customer to move data into and out of the environment. Usually, this particularly concerns the application data of the customer. Data portability can be an ultimate lock-in risk, if it is not possible to extract the information, making it impossible for the customer to leave the provider [160]. This is especially an issue for SaaS offerings, for which appropriate export routines might be unavailable. Even if the data can be extracted, the content and data schema are created by the service provider and it is questionable if the data can be used inside a comparable system. For IaaS and PaaS, the customer typically is in control of the content and the schema of the data [73]. Solely PaaS offerings with a high level of abstraction of the platform close to SaaS suffer from similar problems like SaaS (see Section 3.2). If the underlying data store is known and appropriate export routines are available, it is more an issue of syntactic and semantic portability, i.e., if the syntactic representation of the data store can be transferred to the new data store, and if the semantic model is equivalent between the applications. Consequently, data portability is often restricted by the data store, which can be investigated independently of the PaaS context. *Therefore, we do not further investigate data portability in this work.*

Interoperability aspects of cloud services mainly relate to the interfaces between the cloud service and the customer as well as integrations with other cloud services. Whereas few cloud services are interoperable with other cloud services, every cloud service includes the ability to control the capabilities of the service on the customer's end. The ISO/IEC [160] distinguishes this functionality by the three interfaces functional, admin, and business. Functional interfaces include the main functional capabilities offered by the cloud service. The admin interfaces involve capabilities for administering the cloud service, including monitoring its behavior and security aspects, e.g., user identities, authentication, and authorizations. The business interfaces comprise features such as subscription information, billing, and invoicing [160]. Whereas this categorization is debatable and not selective, it gives an idea of the concerned functionality. As mentioned, the interoperability of the service interfaces is closely related to the efficient portability of applications [152, 283].

Due to the variety of possible implementations, many issues may arise for both perspectives of portability and interoperability between systems. Regarding application portability between cloud offerings, especially PaaS poses the biggest challenges [73]. For IaaS, several standards, e.g., the Open Virtualization Format (OVF) [158] for enabling the portability of entire VM images, have gained traction. The composition of PaaS environments, including supported application artifacts and dependencies, vary widely between vendors. In the worst case, a PaaS may enforce specific proprietary ways to persist and manage data for, e.g., scalability guarantees that may not be supported by other PaaS platforms, requiring extensive re-engineering of the application code [73]. Also, the environments between the providers differ which makes it impossible to move an application to certain vendors. We investigate solutions to recognize and avoid such scenarios in the thesis (see Chapter 5). Likewise, the interfaces of cloud offerings are rarely standardized. Again, PaaS lacks solutions for unifying semantically comparable workflows and APIs compared to IaaS. Therefore, we target the interfaces that are related to the migration of applications between systems in Chapter 7.

2.3. Software Selection Decisions

As first step, the migration of an application requires the selection of an appropriate provider. A migration process can be divided into the planning phase, the migration execution, and the migration evaluation [165]. The planning phase includes the analysis of application requirements and the selection of appropriate providers. This is both a technical and a business decision. Here, Decision Support Systems (DSSs) can assist users to make an informed decision for an appropriate provider.

In general, DSSs are an area of IT that is focused on supporting and improving managerial decision-making [19]. DSSs evolved from the theoretical studies of organizational decision-making conducted at the Carnegie Institute of Technology during the late 1950s and early 1960s and the technical work carried out at MIT in the 1960s [179]. Fick and Sprague [102] define a DSS as a computerized planning and information system that prepares valuable information, often graphically, from raw data, documents, or personal knowledge for helping to solve problems and make decisions. Thereby, the computer system handles the structured portion of a problem, whereas the judgment of the decision maker deals with the unstructured part, hence constituting a human-machine, problem-solving system [330]. A definitional problem stems from the wide applicability and intuitive validity of the term “decision support.” Here, any system that supports a decision may be characterized as DSS [341]. Therefore, many researchers addressed the definition and distinction of DSS from related concepts which, however, did not lead to a generally accepted, clear distinction [103, 106].

In that regard, an important related concept to DSSs are Expert Systems (ESs). Bramer [52] defines an ES as a computing system that embodies organized knowledge concerning some specific domain of human expertise that is capable to perform as a skillful and cost-effective consultant. Hence, the goal for an ES is to mimic an expert in a well-defined task, e.g., a doctor, a production manager, or a marketing director [103]. In this context, an expert is a person with special skills or experience in the particular area, who is widely recognized as a reliable source of knowledge [113]. Originally, ESs were application programs that drew logical conclusions from a knowledge base [117]. Thus, in the early literature, ES are mostly applications which present their knowledge in the form of logical expressions and are able to derive new findings from existing knowledge [356]. Over time, the concept of an expert system has been broadened to more general areas of application involving domain-specific knowledge representation.

A comparison of the concepts and components of ESs and DSSs reveals that the two types show many areas of substantial overlap [103]. The differences between them are primarily based on the emphasis in the employment of different features and characteristics. A DSS often targets less well-structured problems, whereas an ES's domain is well-defined and narrowly bounded so that a codification of relevant knowledge is feasible. The retrieval of results is rather rule-based and heuristically for ESs than based on formal reasoning and mathematical relations as for DSSs. Inferences must often be made based on incomplete or uncertain information. Both aim at the ability to make the decision convenient and accessible for the users, e.g., via UIs [106]. As the remarks show, it is not easy to clearly distinguish these concepts. From a more abstract point of view, DSS can also be seen as an umbrella term for all decision-making processes. Accordingly, an ESs is a more restricted and focused subvariant of a DSSs. Throughout this thesis, we use this generalized definition of DSSs.

In this thesis, we want to support the selection of an appropriate PaaS provider for users. Here, the domain is limited and can be modeled. The data, however, is rather incomplete and not formally defined. Hence, the definition and concepts of an expert system are a close match to our use case. Consequently, our knowledge-base system (see Chapter 4) together with the semantic rules (see Chapter 5) has many parallels to an ES.

Our system should be available via the Internet, accessible to a wide target audience including end users, managers, and providers. Therefore, it also has similarities to web-based DSSs which deliver support information or decision support tools to a manager or business analyst using a web browser [290]. Web-based DSSs have reduced technological barriers and made it easier and less costly to make decision-relevant information available to managers in geographically distributed locations [289].

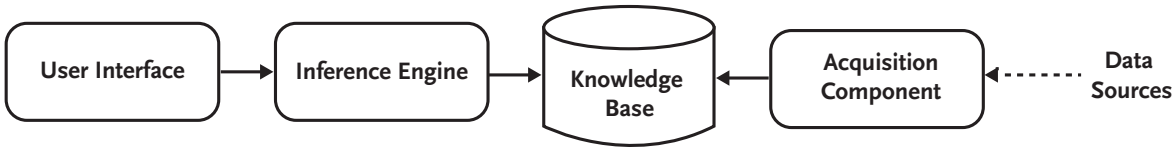


Figure 2.4.: Structure of Expert Systems

Typically, minimal requirements for systems with decision support are components for storing the knowledge data and capabilities to access and enable interactive queries on the data [330]. Figure 2.4 shows important components of an ES. Here, an ES comprises a knowledge base, an acquisition component, an inference engine, and a user interface [103, 106].

The central component of an ES is the knowledge base which encodes the knowledge domain via a defined model. Therefore, an ES builder must first extract the knowledge associated with the chosen domain and then codify it into a machine-readable form. These processes are termed knowledge acquisition and knowledge representation [103]. The data is entered through the acquisition component, either directly by an expert, indirectly via a knowledge engineer, or if possible via automated processes [185]. For our approach, we discuss these processes and its artifacts in Chapter 4.

The procedure by which the expert uses his factual knowledge is termed inference [103]. The part of the ES that performs this task to answer the users' questions is termed inference engine. The users' requirements and questions are entered and controlled inside a UI, which provides an interactive way of using the inference engine, focusing on features which make it easy to use by noncomputer people [341]. In the context of this work, we aim at selecting PaaS providers that fulfill a range of different requirements that can be specified by users. These criteria include application requirements that are necessary for portability and additional nonfunctional requirements. The design of the inference component is addressed in Chapter 5. In our case, the functionality of the inference component is closely related to decision theory.

Harris [144] defines decision-making as the identification and choice of alternatives based on values and preferences of a decision maker. Decision-making helps to sufficiently reduce uncertainty and doubt about possible alternatives to enable a reasonable choice among existing alternatives [144]. Several techniques have been developed to help decision makers with such decision problems. One of the simplest forms are decision tables [189] and decision trees [295]. Here, the acts, states, outcomes, and their relationships to each other are codified either in a flat table structure with rows and columns or as an hierarchical tree structure. Due to the plain visual representation, the techniques are easily accessible for users from every domain. When multiple criteria with different scales need to be considered for a decision, these approaches come to a limit. In our case, a multitude of attributes with qualitative and

quantitative scales need to be considered for a selection of a provider. Hence, more sophisticated methods such as Multiple Criteria Decision Making (MCDM) need to be applied. Generally, MCDM approaches are often used for software selection decisions [195, 201, 301, 378].

Triantaphyllou [360] describes MCDM as the search for the best alternative(s) in a collection of candidates by a set of decision criteria. A major difference to the previously mentioned decision approaches is that often no unique optimal solution for the problem exists and it is therefore necessary to apply the user's preferences to differentiate between solutions. Here, the set of nondominated solutions replaces the otherwise optimal solution. For a nondominated solution, it is not possible to choose another alternative without sacrificing at least one criterion [45, 360]. Each of the criterion's manifestations has either a natural ranking or a user-defined ranking function. MCDM can be further classified into Multi Object Decision Making (MODM) and Multi Attribute Decision Making (MADM), based on whether the available solutions are explicitly or implicitly defined [398]. If the problem space consists of a finite set of alternatives, one talks about MADM. Otherwise, if the space is not explicitly known but defined by, e.g., mathematical models, the term MODM is used. In this thesis, we will solely work with finite sets of alternatives, i.e., MADM, and hence simply use the more common term MCDM [360].

The thesis' approach focuses on application portability between PaaS providers. This requires certain attributes, especially the technical criteria for the application, to be must-haves. Therefore, further adjustments to the selection and ranking process are necessary. Typically, MCDM algorithms do not handle excluding criteria but optimize a target function. Therefore, we need to prepend a filtering step for all must-have criteria before applying the MCDM ranking algorithm. This means if any of those criteria cannot be fulfilled by a candidate provider, it must be excluded from the result set. Afterwards, established MCDM methods can be applied to rank the results based on non-must-have criteria.

A variety of algorithms to solve MCDM problems have been developed [51]. Every method roughly follows three steps with different alterations [360]. As a first step, relevant criteria and alternatives are determined. Next, numerical measures are attached to the relative importance of the criteria and to the impacts of the alternatives on these criteria. Last, the numerical values are processed to define a ranking for the alternatives [360]. Examples for established algorithms are outranking [315], the weighted sum model [105], weighted product model [244], ELECTRE [316], and TOPSIS [393]. The most popular among MCDM algorithms, however, is the Analytic Hierarchy Process (AHP) [318] with its related forms revised AHP [31] and Analytic Network Process (ANP) [319] [58, 363].

The AHP utilizes a hierarchical structure for the criteria used in the decision process. Therefore, a hierarchy with at least three levels (goal, criteria, al-

ternatives) is created. If required, further hierarchies of subcriteria may be introduced. Next, the relative importance of the criteria is evaluated by a pairwise comparison. Additionally, a consistency index measures if the decision maker has been consistent when comparing the criteria with each other. The evaluations are then converted to numerical values representing the priority of the criteria. Finally, a score for each alternative is calculated, leading to an order of the alternatives. The alternative with the highest score is considered the best alternative [318]. Whereas several shortcomings were discovered for special cases and a revised AHP [31] was suggested, the classical AHP still remains the most used algorithm. AHP is used in practice by a multitude of selection approaches [363]. It has also been successfully used to support software selection scenarios, and thus is a candidate for evaluating PaaS providers [201, 202, 378].

2.4. Cloud Brokerage

Due to the popularity of the cloud, the market sees a high number of different and often heterogeneous cloud offerings [94, 95, 349]. For customers, this makes it hard to get a thorough view of the market and difficult for them to predict the effects of a decision which often eventually leads to lock-in [332]. The lack of standards among cloud offerings further aggravates the scenario for users [217]. As with other markets, brokers are able to facilitate the business connections between providers and customers by providing their expertise. A broker is a centralized coordinator that fulfills an intermediary role between the involved parties. Thereby, the nature of a broker can be either human, machine, or a combination of both. The services range from the initiation of a business relationship to the communication and mediation between the contracting parties. The term broker itself is very broad and can be used to refer to different mediation patterns between several parties. Akin, the ISO/IEC define a cloud broker loosely as:

> **Definition 2.7 (Cloud Service Broker)** *"[A cloud service broker is a] cloud service partner that negotiates relationships between cloud service customers and cloud service providers."* [161]

Thereby, the relationships and service exchange between customers, brokers, and providers can be manifold. According to the NIST and Gartner [112, 214] cloud brokerage can be classified into three groups: service intermediation, service aggregation, and service arbitrage. In the case of service intermediation, a cloud broker enhances a given service by reducing barriers or by improving a particular capability of the service and provides value-added services to consumers. Possible improvements are access management to cloud services, identity management, performance reporting, or enhanced security. With

service aggregation, a cloud broker combines and integrates multiple services into one or more new services. Service arbitrage is similar to service aggregation except that the set of services that are aggregated is not fixed but will be dynamically chosen depending on, e.g., current prices [214].

A more intuitive way for identifying the purposes and services of a broker is to look at the challenges that are tackled. In particular, we identified three major motives: market size, vendor lock-in, and value-added functionality [95, 332, 349].

The large number of cloud providers and offerings makes provider and service selection a challenging task. The size of the market and number of available cloud services can overwhelm a human decision maker [95]. Also, there is no single point of information, but data must be gathered from various sources, harmonized, and filtered to make a decision [349, 394]. For this purpose, a cloud broker that aggregates this information and makes it available to customers for provider selection can support human decision-making [94]. Here, brokerage is limited to recommending a cloud service to customers based on their requirements and no further technical mediation occurs.

Further, cloud brokers may focus on bridging existing heterogeneity between vendors and reduce lock-in risks for customers. This includes technical solutions for mediating between proprietary interfaces of cloud providers through a single uniform interface. In that way, cloud brokers can provide interoperability and portability across multiple cloud providers without enforcing any requirements on the independent providers [95]. Hence, brokers are a means to facilitate common capabilities in the absence of standards [73]. The customers can then use and switch between different supported providers without any changes to their application or knowledge of the providers interfaces [285, 324].

Moreover, instead of making existing functionality more open and accessible, a cloud broker might create entirely new value for customers that is not yet delivered by existing offerings. Examples are multi-cloud application management [56], impartial performance measurement and Service-Level Agreement (SLA) assurance [14, 100, 243], or cost-optimized resource placement [220, 334].

For the reasons above, broker architectures supply an appropriate framework to evaluate our findings and proposals on application portability. In this thesis, we want to consider especially the decision support for cloud provider selection (see Chapter 5) and the management interface unification aspects of cloud brokers (see Chapter 7). Both tasks are important and valuable for improving application portability and the ease of an application migration process. Whereas at best, a cloud broker is able to conduct all necessary steps automatically without disruptions and manual tasks, this is hardly viable due to the amount of heterogeneity challenges. The individual support services of a cloud broker can, however, substantially facilitate the migration process for customers.

Different observations of Elhabbash et al. [95] further motivate and confirm our planned approaches. First, cloud brokers for IaaS are most common and there is a lack of comparable approaches for PaaS. Next, there exists no central repository and standardized description format for PaaS provider selection [349]. Also, portability and interoperability in PaaS is a concern that is only recently addressed. Whereas there were hopes that the market would settle on a set of common interfaces, advancements have shown that the providers have little interest in unifying their APIs to allow customers to freely move between offerings. Furthermore, especially for the interoperability of management interfaces, most approaches solely focus on deployment as central functionality, failing to provide life cycle management beyond deployment [95].

Part II.

Platform as a Service

3. Conceptualization of Platform as a Service

Parts of this chapter have been taken from [191].

In this chapter, RQ 1 *("How to distinguish and classify Platform as a Service offerings?")* is supported.

3.1. Motivation

Zimki, the alleged precursor of today's PaaS, which was initially termed Framework as a Service, was first introduced by Fotango, a subsidiary of Canon Europe, in 2005.[23] The basic idea behind Zimki was that IT is becoming a commodity and there is no competitive advantage, i.e., no correlation between IT spending and the value of a company. At that time, the common notion was that all kinds of IT add business value and IT reduces costs through automation. The facts, however, seemed to be that most of the IT spendings were only necessary to keep up with competitors and did not add any new business value. So the majority of money was spent on essential systems which competitors also had to stay in competition. Fotango's conclusion to these observations was that the only advantage a business can get from such IT technologies is to do it cheaper than the competitors. Trends like SaaS, utility computing, and the tendency to replace customized solutions by generic ones fostered this assumption. Inspired by the visions that Carr [61] explains in his book "Does IT matter?," they designed a system where all the commodities for an application, i.e., the application environment an application runs in is delivered as a service. Zimki provided a utility-based web application development and hosting environment using client- and server-side JavaScript (JS) that enabled businesses to easily develop and deploy web sites, applications, and web services. The idea was to use the components of the platform and automation tools to remove all reoccurring tasks in the process of application development and operation. This way, efforts for setting up, maintaining, and running the server before the actual application can be developed or deployed were eliminated. The companies should focus on developing applications which contribute to

[23] https://en.wikipedia.org/wiki/Platform_as_a_service

the business value. Also, the utility aspect was strengthened by delivering the platform as a virtualized environment, only charging for the actual usage of storage, network, and JS operations. The company also aimed at being a portable platform for developing SaaS applications without lock-in by making their platform open source. Eventually, this should result in an array of Zimki providers, with freedom of choice for customers [377]. The goals were set a little too high at that time. Canon shut down Zimki in December 2007, thinking that the business model would not pay off.[24] However, the basic ideas were seized by other companies.

The first successful PaaS products that entered the stage were Force.com[25], EngineYard[26], and Heroku[27] around 2007. Force.com was designed to easily develop extensions for Salesforce's Customer Relationship Management (CRM) SaaS solution based on a low-code tools approach using point and click functionality. EngineYard and Heroku, at that time focused on Ruby applications, in contrast, were two of the first classical application development environments hosted in the cloud similar to Zimki. Shortly after, in 2008, Google launched the Java-based offering App Engine[28].

In the pre-cloud era, there was no popular platform notion yet. However, this does not mean that no such thing did exist. Back then, large organizations have used concepts like Software Product Lines (SPLs) to define a standard set of software components for their enterprise [46, 47, 71]. Those enterprise architecture groups hand-crafted a platform definition standard by combining various products like application servers, web servers, databases, and other middleware products. Therefore, it is likely that the utilized application environments, i.e., the platforms, will have commonalities but also different requirements among companies. The ISO/IEC/IEEE [162] define a platform as:

> **Definition 3.1 (Platform)** *"[A platform is] a type of computer or hardware device and/or associated operating system, or a virtual environment, on which software can be installed or run."* [162]

Accordingly, a platform can be determined by both or either, hardware and software components. A hardware platform, for example, may be defined by a particular machine or processor architecture. An example for a popular instruction set architecture is x64. On the other hand, software platforms can be specified on various levels of the software stack. For the cloud, where the hardware foundation is made transparent, the definition also includes the notion of a virtual environment. For PaaS, a platform is a unified software foundation for developing and running applications in the cloud. The "as a

[24] https://blog.gerv.net/2007/09/zimki_shuts_down
[25] https://www.salesforce.com/products/platform/overview
[26] https://www.engineyard.com
[27] https://www.heroku.com
[28] https://cloud.google.com/appengine

Service" part of the PaaS abbreviation resembles the idea of the delivery of the entity via a third-party provider, i.e., the outsourcing of the IT service. The service concept also includes the on-demand delivery aspect. In most cases, this will be done via the Internet.

The introduced platform definition already shows that a platform can have multiple manifestations that are not necessarily comparable. The same problem applies to the definitions of PaaS, e.g., the popular definition by the NIST [237]. The term PaaS itself has no sharp boundaries and can be applied to cloud services with very different capabilities [21, 30, 119, 217, 349]. The result is a crowded market of PaaS offerings that sometimes provide completely different capabilities [307]. Also, offerings and the notion have changed over time, so the early purposes are not necessarily valid anymore. Nevertheless, the market still sees a broad range of offerings labeling themselves as PaaS connecting to the anticipated benefits of the concept. This makes it hard for customers to get a well-founded overview of the market and even harder to compare them to make an informed decision [132, 302, 303]. Whereas a lot of providers benefited from that confusion in the early phases and during the hype, this situation can only harm the long-term success of PaaS. A look at the literature did only reveal how fuzzy the definition and understanding of the fragmented PaaS landscape still is. This further motivated the necessity for a more structured approach and literature review to investigate which definitions of PaaS exist, what key properties are attributed to PaaS, and how this is reflected by the state of the art. We argue that the current definition and classification is oversimplified and needs to be refined. Hence, before we investigate portability aspects of PaaS, we want to work on a more accurate definition and classification of PaaS. We state the following research question, which will be answered in the different sections of this chapter.

***Research Question 1*:** *How to distinguish and classify Platform as a Service offerings?*

Therefore, Section 3.2 compares existing definitions from literature and extracts important characteristics of PaaS. Section 3.3 contrasts the definitions with the state of the art by providing a market overview. Next, we describe the underlying technical architecture of typical PaaS systems in Section 3.4. Finally, we compare the concept of PaaS to and differentiate it from related technologies in Section 3.5.

3.2. Conceptual Delimitation

Often, the definitions for IaaS, SaaS, and PaaS are taken for granted. However, we have found that, as with most buzz words, a precise definition is lacking, which leads to confusion among customers [21, 217, 302, 349]. For customers, it is vital to have a clear understanding of the concept and the differences in the market to make an informed decision [132, 303]. Overall, not all PaaS

offerings fall into the same homogeneous group. Also, different types of PaaS imply different business values and lock-in risks. Whereas the general notion of PaaS is a starting point to describe the concept, we need to refine it to account for the different existing and emerging subcategories [134].

As a first step, we want to review how literature defines and characterizes PaaS and extract common characteristics. We tried to follow the guidelines of Kitchenham et al. [188] where possible to make the study more thorough and replicable and minimize possible threats to validity. As with literature studies, threats to validity such as biases or internal and external validity, cannot be completely eliminated [188]. There are several reasons why it is not feasible to fully satisfy the suggested guidelines in our case. First of all, looking for definitions of PaaS by the occurrence of the term and its abbreviation returns too many results for an examination. Google Scholar yields nearly 100,000 results for the combination of the term and the abbreviation. This is caused by the fact that the cloud is a vibrant topic that is frequently targeted by research. Nearly every cloud paper includes a short delimitation of the concepts of IaaS, PaaS, and SaaS. Next, current software engineering search engines are not designed well enough to effectively support systematic literature reviews. A contributing factor is that IT and software engineering abstracts are not standardized enough to rely on when selecting primary studies. This includes missing unification and validation of keywords between journals and publishers. Therefore, the full text or at least the conclusion needs to be examined for inclusion or exclusion of papers [54]. Nevertheless, we adhere to the guidelines of Kitchenham et al. [188] for structured literature reviews where possible. In the following, we describe our research protocol, relevant steps, and decisions made during the process.

For identifying relevant research, we used a combined approach of an automatic search in digital libraries and a manual search in the references of seminal papers. We selected the following digital libraries for our literature search. This is a subset of the important libraries as suggested by Brereton et al. [54].

- ACM Guide to Computing Literature[29]
- IEEE Xplore[30]
- Google Scholar[31]

We searched for one of the terms “Platform as a Service” or its abbreviation “PaaS” in combination with the term “definition.” We used the following abstract search string to query the selected digital libraries:

(Platform as a Service `<OR>` PaaS) `<AND>` definition

[29] `https://dl.acm.org`
[30] `https://ieeexplore.ieee.org`
[31] `https://scholar.google.com`

The abstract search term is adapted to the syntax of the respective digital library, as they do not support the same syntax. More general search terms like "Platform as a Service" cannot be used, as the number of results becomes unmanageable. This is especially the case for a full-text search. To compensate, we additionally conducted a manual backward and forward search in papers that were identified as relevant. Due to the amount of results, we extracted the first 100 papers ordered by relevance of Google Scholar.

As the standard two-step approach [188] of selecting the primary studies via title and abstract and subsequently the full text does not work well for software engineering, we immediately consulted the full text for our selection of relevant papers [54]. Whereas there are few papers exclusively focused on defining the term and scope of PaaS [30, 118, 246], the way a paper describes PaaS in the context of the research work defines the authors' anticipated notion of PaaS. To select the primary studies, we decided on the following inclusion and exclusion criteria for this review: First, the source needs to include a definition or conceptualization of the term PaaS. The definition must at least contain a short description of the concept's characteristics and not solely name the term. Next, the used description must not be adapted or cited from another publication. Moreover, we decided not to exclude non-peer-reviewed articles like technical reports as often done in literature reviews. As cloud computing and PaaS is such a vibrant topic, solely defining PaaS by the academic literature seems to be inappropriate.

Table 3.1 shows the final set of selected articles[32]. The data for the literature study comes from 37 sources in between the years 2008 and 2018.

To compare the different definitions, we examined and extracted the key characteristics from each definition. In the following, we exemplify the procedure for the definition as envisioned by Salesforce, the creators of Force.com, one of the first PaaS. Weissmann and Bobrowski [379] define PaaS as:

> **Definition 3.2 (Platform as a Service)** *"Platform as a service (PaaS) [. . .] is an application-centric approach that* ***abstracts the concept of servers*** *altogether. PaaS lets developers* ***focus on core application development*** *from day one and to* ***deploy an application*** *with the push of a button. The* ***[customer] never needs to worry*** *about multitenancy, high-availability, load-balancing,* ***scalability****, system backups, patches and security, and other infrastructure-related concerns."* [379]

In this example, we extracted the concepts *infrastructure abstraction*, *runtime environment*, *scalability*, *managed environment*, and *third-party provider*. Text passages used to derive the attributes are marked in bold in the definition. We applied a similar approach to all other definitions and merged closely related concepts into the resulting set of core properties. For example, we use the

[32]The unfiltered search results, the selected articles, and all text excerpts that were consulted in the study can be found at `https://github.com/stefan-kolb/paas-definition`.

Table 3.1.: Related Works Defining PaaS

Paper	Type	Year
Dubey et al. [91]	Technical Report	2008
Lawton [204]	Magazine & Journal	2008
Mitchell [246]	Technical Report	2008
Vaquero et al. [365]	Magazine & Journal	2008
Wang et al. [375]	Conference	2008
Hilley [148]	Technical Report	2008
Grohmann [128]	Book	2009
Motahari-Nezhad et al. [248]	Magazine & Journal	2009
Prodan and Ostermann [291]	Conference	2009
Rimal et al. [310]	Conference	2009
Weissmann and Bobrowski [379]	Conference	2009
Armbrust et al. [18]	Magazine & Journal	2010
Bhardwaj et al. [37]	Journal	2010
Dillon et al. [87]	Conference	2010
Sriram and Khajeh-Hosseini [342]	Technical Report	2010
Beimborn et al. [30]	Journal	2011
Eurich et al. [98]	Conference	2011
Khalidi [184]	Magazine & Journal	2011
Kim [187]	Conference	2011
Marston et al. [231]	Journal	2011
NIST [237]	Technical report	2011
Subashini and Kavitha [347]	Journal	2011
Giessmann and Stanoevska [118]	Conference	2012
Jadeia and Modi [164]	Conference	2012
Rodero-Merino et al. [313]	Journal	2012
Shao and Wang [325]	Conference	2012
Xu [385]	Journal	2012
Kachele et al. [173]	Conference	2013
Ayad et al. [20]	Conference	2014
Braubach et al. [53]	Conference	2014
ISO/IEC [161]	Standard	2014
Yaqub et al. [388]	Conference	2014
Almorsy et al. [11]	Technical Report	2016
Firozbakht et al. [104]	Conference	2017
Krintz [200]	Book	2017
Carrasco et al. [62]	Journal	2018
EngineYard [96]	Technical Report	2018

broader term elasticity instead of scalability to deal with varying demands [237]. Table 3.2 shows the selected articles and the set of identified characteristics.

Overall, eight features are frequently mentioned. These are *runtime environment*, *development environment*, *third-party provider*, *infrastructure abstraction*, *web applications*, *elasticity*, *online delivery*, and *fully managed environment*. As we show in the following description of the features, a main problem for distinguishing and classifying PaaS offerings is how exactly these properties are shaped. This creates significant differences between the offerings. After introducing these differences, we propose a new classification that allows us to better classify PaaS offerings.

Runtime Environment The application runtime environment is the centerpiece of a PaaS offering. Consequently, all the definitions mention it. Every PaaS delivers an environment where the customer's application can be run. Often,

Table 3.2.: Characteristics of PaaS in Related Works

Characteristic	References	Percentage
Runtime Environment	[11, 18, 20, 30, 37, 53, 62, 87, 91, 96, 98, 104, 118, 128, 148, 161, 164, 173, 184, 187, 200, 204, 231, 237, 246, 248, 291, 310, 313, 325, 342, 347, 365, 375, 379, 385, 388]	100 %
Development Environment	[11, 18, 20, 30, 37, 53, 87, 91, 98, 104, 118, 148, 161, 164, 173, 184, 187, 204, 231, 237, 246, 248, 291, 310, 313, 325, 342, 347, 365, 375, 379, 385, 388]	54 %
Third-Party Provider	[11, 20, 37, 96, 104, 118, 128, 148, 161, 173, 204, 231, 237, 313, 342, 347, 375, 379, 388]	51 %
Infrastructure Abstraction	[11, 37, 62, 96, 98, 128, 148, 164, 173, 184, 187, 237, 342, 347, 365, 379, 388]	46 %
Web Applications	[18, 37, 87, 91, 164, 200, 204, 246, 248, 291, 310, 325, 385]	35 %
Elasticity	[18, 30, 53, 62, 104, 173, 184, 200, 231, 246, 325, 365, 379]	35 %
Online Delivery	[11, 20, 37, 87, 98, 164, 204, 231, 291, 375, 385]	30 %
Fully Managed Environment	[11, 18, 104, 148, 161, 173, 379]	19 %

it is complex and labor-intensive to install and configure these environments. PaaS relieves the developers from this repetitive task by providing a runtime environment for typical or custom application scenarios.

The way these runtime environments are composed is not defined and may widely differ between offerings. Not only from a technical perspective, i.e., what the environment provides, but also from an abstract point of view. The runtime environment may either be more like a black box or a white box. The composition of the environment may be explicit (white box) or transparent (black box) to the user. For example, there are PaaS that involve higher-level programming languages or even template-based software building programs that enable users with little coding experience to create business applications [32]. Consequently, the entire runtime environment is custom-build and abstracted for the user. On the other hand, there is another category of PaaS that host applications created with standard programming languages such as Java or Ruby. Here, the composition of the runtime environment, i.e., programming languages, middleware products, or services, is made explicit, so the user knows which technologies are available for use in an application. Black box runtime environments provide a higher level of abstraction than white box runtime environments, which also typically reflects the overall abstraction level of the PaaS.

Development Environment The development environment includes the software and tools to support the creation phase of an application. Whereas half of the definitions mention such a functionality for PaaS, the market shows that it is very different if such functionality is included or not. PaaS that follow a black box runtime approach often include an appropriate development environment to create applications. Such development environments may span from classical IDEs served via the browser to visual programming IDEs. The specialized use cases of black box PaaS make such an approach sensible. In contrast, most white box PaaS focus on the operational aspect and omit the inclusion of a

development environment. As the runtime environment reflects standard application development scenarios, the assumption is that the development is better done locally with familiar tooling, e.g., an IDE such as Eclipse [32].

Third-Party Provider A majority of definitions refer to a third-party provider that is responsible for the platform. We think this is influenced by the fact that the public availability of the PaaS system is still the default case. For private PaaS, this provider may also be an entity of the same company. Nevertheless, it emphasizes that the delivery and management of the platform is done by a different entity than its users. The PaaS is open to an external group of users capitalizing on the proposed service.

Infrastructure Abstraction Another important factor is the abstraction of the underlying hardware and infrastructure. The complex setup and management of the distributed cloud system is delivered by the provider. Typically, the user does not control the cloud infrastructure except for a restricted set of automated management tasks, e.g., scaling. Again, the level of abstraction differs between the platforms. At best, the system abstracts servers at all and is responsible for managing the whole system and the application operation. In most cases, however, the user still has to deal with the abstract concept of application instances. Both of these abstraction levels are used in conjunction with black box and white box runtime environments. Again, the serverless view is most often applied to higher-level black box approaches. An exception are a few white box PaaS that still abstract the infrastructure but allow full visibility and access for the user if desired.

Web Applications PaaS is built for web applications. Thereby, the emphasis lies on the concept of delivering applications similar to the functional scope of desktop applications via the Internet, often referred to as SaaS. As with a client-server model, data processing and evaluation mainly takes place on a remote server. New technologies and open web standards such as HTML5 and AJAX have facilitated the delivery of applications over the Internet. These technologies are browser-native and require no additional installation by the users. Nevertheless, PaaS is also very well suited for web services and simple web sites.

Elasticity Elasticity is said to be an important factor for PaaS. The main motivation comes from the desired abstraction of the operational aspects and the infrastructure. Today, elasticity is often still manually regulated by the users via controlling the amount of application instances or by defining the compute power of an instance. However, more and more PaaS are supporting fully automatic scalability of applications.

Online Delivery The online delivery is inherent to PaaS. The entire system is consumed via the Internet or a network. All resources are available to the user on demand. No local installation is required, or only very little such as a browser or Command Line Interface (CLI), to consume the services.

Fully Managed Environment Whereas the management aspect is often implicitly attributed to the concept of PaaS, many definitions explicitly highlight it. The provider is responsible for managing the runtime facilities, i.e., the hardware and software, and the use of the application. This releases the user from tedious but important tasks such as system and runtime updates for ensuring the security of the system. Next to the technical foundation, the provider has to manage the access to the application, e.g., reacting to varying load through load balancing and ensuring the availability of the application.

As described, different manifestations of the characteristics lead to different groups of PaaS systems. Others, especially research companies attempting to provide a market overview, have already come to this conclusion and tried to suggest different subcategories of PaaS. Gartner [252] for example defines several *xPaaS* subcategories in which *x* specifies the part of the platform functionality that is delivered by the PaaS. In contrast, Forrester [317] categorizes PaaS by the applicability for different types of developers, namely *DevOps pro*, *coder*, and *rapid developer*. These groups each come with distinct backgrounds, preferences, and motivations on the controllability of the platform. As these new sets of categories are often not self explaining and intuitive, we argue that they create more confusion than enlightenment. Therefore, instead of trying to create another set of new categories, we suggest classifying PaaS offerings along dimensions that are well-known and a common denominator, i.e., the SPI model (SaaS, PaaS, IaaS).

As shown, the differences between the manifestations of the features are often based on the abstraction level of the PaaS. Especially evident is the white box versus black box abstraction of the runtime environment and the infrastructure. The particular manifestation can be well explained by the proximity of the PaaS system to IaaS respectively SaaS. Currently, we see three distinctive groups of PaaS in between IaaS and SaaS (see Figure 3.1).

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. Both the infrastructure and the runtime environment are evident to the user as a white box. An example of this type of provider is AWS Elastic Beanstalk[33] 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 a high level of abstraction. These PaaS are often tailored to a complementary SaaS

[33] `https://aws.amazon.com/de/elasticbeanstalk`

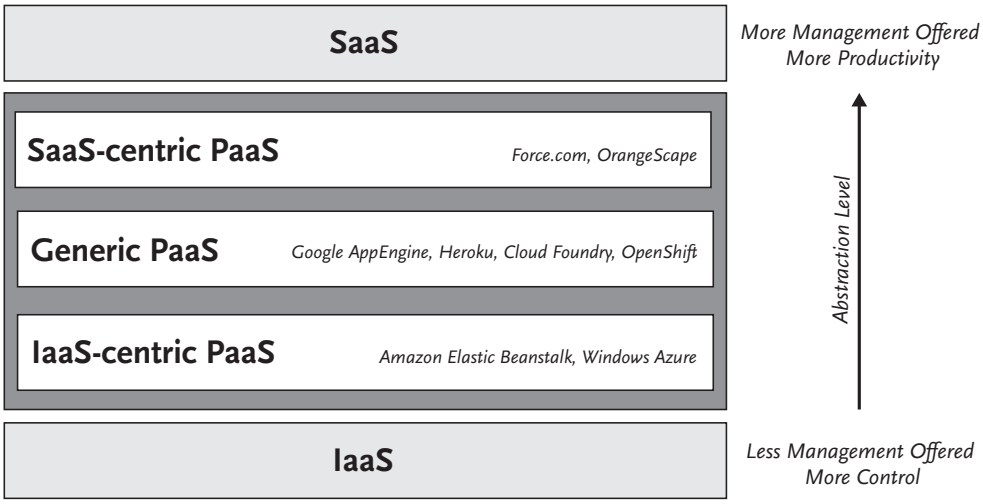


Figure 3.1.: Classification of Platform as a Service

solution and allow developing extensions for an existing SaaS application. The applications can be developed via a proprietary Software Development Kit (SDK) or visual and template-based software building programs [32]. For that reason, SaaS-centric platforms also more often provide a development environment than other groups. The platforms usually follow the black box principle, abstracting most of the runtime environment and infrastructure. A representative is Force.com[34] that provides the ability to develop applications for Salesforce's CRM SaaS solution. *SaaS-centric* PaaS also include very specific platforms that target Business Intelligence (BI) or Business Process Management (BPM). At the center of our classification reside the PaaS which we term *Generic PaaS*. All of these supply a more or less classical white box application platform that consists of a set of programming languages, frameworks, services, and other components an application can be programmed to. Heroku[35] is a popular representative of this group of PaaS. Overall, the abstraction level of the platform components rises from IaaS-centric to SaaS-centric. Here, the provider manages more of the platform and requires only little intervention by the user. The higher abstractions are more geared towards productivity by a more restricted and opinionated framework for running the applications. The lower groups, on the other hand, enable more flexibility and grant the user more control over the underlying environment.

Although as described many types of platforms can be termed PaaS, the scope of this thesis focuses on *Generic* and *IaaS-centric* PaaS. That is because these solutions include a white box application platform that is comparable between different providers. As mentioned, applications created for *SaaS-centric* solutions are tightly bound to the platform, restricting application portability. The runtime environment is mostly a black box, based on proprietary SDKs

[34] https://www.salesforce.com/products/platform/overview
[35] https://www.heroku.com

and APIs, that cannot be compared or altered by the user from a technical perspective. Those platforms come with restrictive vendor lock-in that cannot be bridged because of the nature of the offering's purpose.

3.3. Market Analysis

Next to the conceptual definitions, we identified a lack of research of the state of the art in this area that is freely available. Whereas important market research companies like Gartner or Forrester regularly conduct analysis of current cloud offerings[36], access to the reports is often restricted by a paywall. Furthermore, there exist some more or less comprehensive listings and unstructured studies around the web. However, these snapshots of the status quo are likely to be already outdated when they get published. The lack of comprehensiveness and up-to-dateness are motivating factors for our collaboratively maintained knowledge base, as presented in Chapter 4.

The following survey of the state of the art and developments over the last years serves as a general overview over the topic and as a reference for the subsequent chapters. Again, it tries to shape the notion of PaaS but this time from a practical point of view. Our analysis includes data[37] from July 2013 to August 2018, rendering it capable of depicting changes in the market over this period. For this market analysis, we collected information about 71 different PaaS providers in total. Each profile change is manifested as a Git commit and can be used as a snapshot of the market at this point in time. Whereas this is already a reasonable amount of offerings, we do not expect this list to be exhaustive. Yet, to our knowledge, this data set is the most recent and comprehensive list of PaaS providers on the web. The thesis' decisions can be better justified or comprehended with the presented data.

As an introduction to the market, Figure 3.2 gives an overview of the most popular providers of 2017, based on the profile accesses registered by *PaaSfinder*. In the following, we briefly describe the providers and PaaS systems.

OpenShift[38] is an open source PaaS from Red Hat. Its first version was released in 2011 and the current version is OpenShift 3. In its latest version, it is built around Kubernetes[39], a popular open source system for managing containerized applications, initially developed by Google. It is a polyglot PaaS, i.e., it supports multiple runtime languages and backing services. In the open source PaaS market, OpenShift is the major competitor to Cloud Foundry (CF). OpenShift is represented with two entities in the list. *OpenShift Online*[40] (1) is

[36] See market research from Forrester [308, 317] and Gartner [252].
[37] The snapshot of the used data is provided at `https://github.com/stefan-kolb/paas-profiles/releases/tag/phd18`.
[38] `https://www.openshift.org`
[39] `https://kubernetes.io`
[40] `https://www.openshift.com`

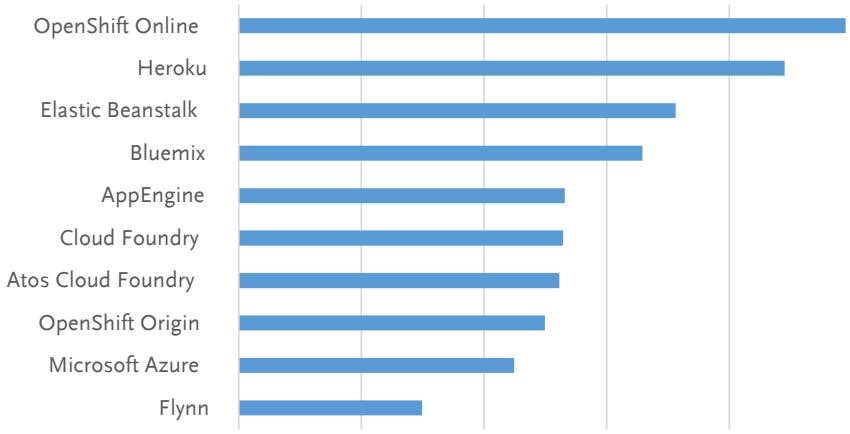


Figure 3.2.: PaaSfinder Most Popular Providers 2017

the publicly hosted, commercial offering of Red Hat. *OpenShift Origin* (8) is the open source foundation of the PaaS system.

Heroku[41] is a proprietary PaaS system founded in 2007. In its beginnings, Heroku was a platform focused on Ruby but transitioned into a polyglot platform early on. Heroku is the originator of the buildpack mechanism[42] for bootstrapping runnable application packages from application sources and the twelve factor methodology[43] for application design. Heroku is one of the pioneers in the market that has also shaped the PaaS technology and system workflows of multiple other vendors in the market. In 2010, Heroku was acquired and since then is a subsidiary of Salesforce.

AWS Elastic Beanstalk[44] is Amazon's IaaS-centric PaaS offering. Elastic Beanstalk builds upon the existing low-level offerings of AWS enhanced by bootstrapping processes to automate the delivery of the infrastructure and the deployment of applications. Therefore, it orchestrates multiple AWS components such as EC2, S3, and Elastic Load Balancer. Compared to other PaaS systems, customers may still access all low-level components such as EC2 VM instances with full administration access rights.

Google AppEngine[45] is one of the first PaaS offerings dating back to 2008. App Engine has a high abstraction level of all infrastructure components and runs sandboxed applications including automatic scaling capabilities transparently across multiple servers. The sandboxed environments are restricted in terms of usable functionality, e.g., APIs, threading or file system access. In exchange, they provide high availability and utmost scalability of the applications. Due to the developments of highly flexible and configurable environments such

[41] https://www.heroku.com
[42] https://buildpacks.io
[43] https://12factor.net
[44] https://aws.amazon.com/de/elasticbeanstalk
[45] https://appengine.google.com

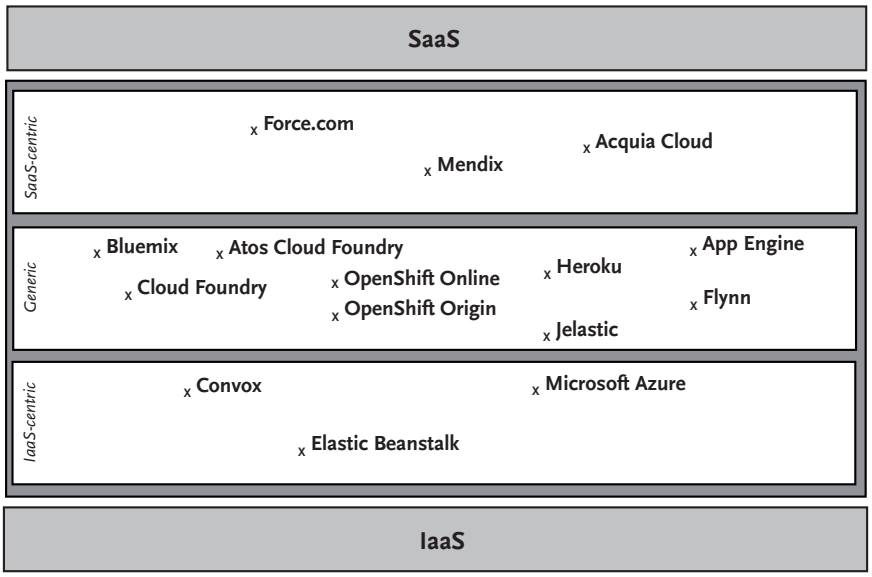


Figure 3.3.: Provider Classification

as provided by the buildpack mechanism, App Engine additionally introduced a flexible environment based on containers with fewer restrictions for developers but inferior scalability guarantees.

Cloud Foundry[46] is an open source PaaS system originally developed by VMWare. It was first released in 2011. In 2015, the CF Foundation was created as an independent Linux Foundation project to ensure an open governance of the project. Like OpenShift, CF is represented with multiple entities in the top list. This also shows the strong position of those two open source PaaS in the market. *Bluemix*[47] (4), now integrated in IBM Cloud, is the commercial CF offering of IBM including many integrations with IBM services. *Cloud Foundry* (6) is the open source PaaS system. *Atos Cloud Foundry*[48] (7) is the commercial offering of Atos, a French IT services corporation.

Microsoft Azure[49] is an assortment of cloud computing services offered by Microsoft since 2010. Among them, there are also multiple services that provide PaaS functionality, e.g., Azure App Service[50]. Similar to Elastic Beanstalk, the offering is based on a fully managed environment of other low-level systems.

Flynn[51] is a Docker-based open source PaaS released in 2016. It is inspired by many paradigms of Heroku and developed in a modularized structure. Most notably, the development of the system was funded by several partners and a crowd funding campaign in 2013.

[46] https://www.cloudfoundry.org
[47] https://www.ibm.com/cloud/cloud-foundry
[48] https://atos.net/en/solutions/application-cloud-enablement-devops/multi-cloud-application-platform
[49] https://azure.microsoft.com
[50] https://azure.microsoft.com/services/app-service
[51] https://flynn.io

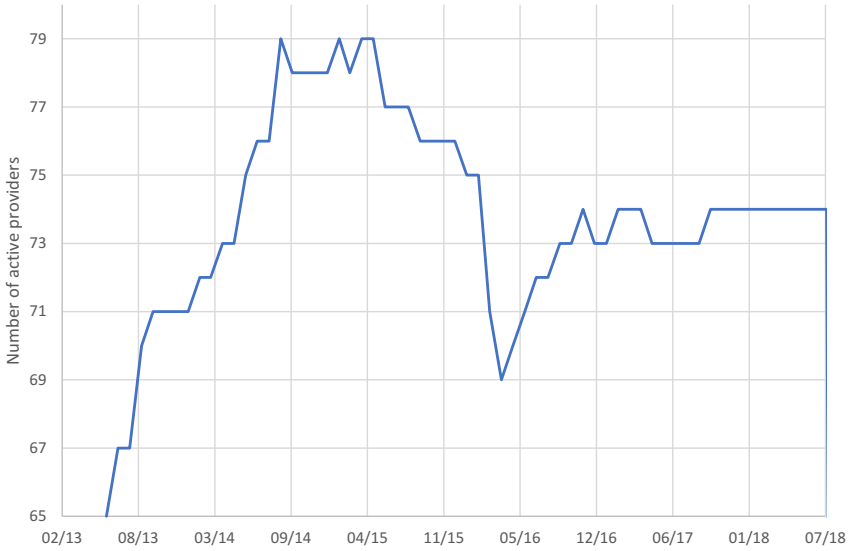


Figure 3.4.: PaaS Providers over Time

Figure 3.3 shows how the providers fit into our proposed categorization scheme. The majority of providers are settled in the Generic and IaaS-centric market segment. However, this is biased by a focus on providers that are candidates to support application portability. In reality, the share of specialized platforms is way higher, but they are also hardly comparable and applications are rarely portable. Hence, the data is especially sound for Generic and IaaS-centric offerings but less for SaaS-centric, which are on the edge of focus. In total, the most recent data record[52] includes 11 SaaS-centric, 12 IaaS-centric, and 49 Generic PaaS providers.

Due to the hype around the cloud and PaaS, it is interesting to look at the increase in the number of providers over time. Typically, one would expect a rise of providers in the hype phase and afterwards a consolidation of providers to a set of established and enduring offerings. In Figure 3.4, we can see that the number of providers increases especially in between 2013 and 2015. This can be attributed to the hype phase of the technology [338]. It also marks the peak of PaaS' growth. Afterwards, there is a slight decrease in numbers with occasional bumps. Eventually, the number of providers is continuing to flatten out until now. What is interesting, however, is that the amount of discontinued offerings rose even during the period of strong growth. Therefore, it can be assumed that while the market volume rises, also a lot of providers are continually going out of business. Events such as bankruptcy, acquisitions, or discontinued offerings due to unprofitability were often observed during that times. It seems to be very difficult for small, new entrants to gain a foothold in the market. This also indicates that only a few providers can actually establish their offering over time. This especially threatens the viability of the

[52]The snapshot of the used data is provided at `https://github.com/stefan-kolb/paas-profiles/releases/tag/phd18`.

providers and aggravates lock-in effects for customers [25]. Right now, new providers are often based on established PaaS systems such as CF or OpenShift and only few new distinct systems enter the market. More and more bigger vendors from the IaaS market include instances of the former two into their portfolio. Currently, the data shows 11 CF and 7 OpenShift providers. Whereas previously many new offerings were released in beta status, now nearly all systems are in production. Back then, a lot of small startups bootstrapping their offerings with small development teams were trying to get a hold of the business. In summary, it becomes clear that there are few to no new entrants in the already consolidated market and that there are a lot of established services. This situation also enables standardization and increases the chance of acceptance [230, 271, 314].

Another factor which shows the consolidation and movement to profitability is the billing model [92]. While a majority of providers offered a free plan to users in 2013, this has diminished to 23 % in 2018, excluding trial plans. Such an offer is particularly interesting for small projects as there is no need for any financial investments. However, this also posed problems for providers that had to deal with scam and criminal sites hosted on their system, making the service unusable for a number of legitimate users. Additionally, providers that still supply a plan at no cost have often limited the amount of uptime per month and put applications to sleep on idle times. All public providers apply the typical subscription model with reoccurring payments.

A main characteristic of cloud computing systems is elasticity [237]. Computing resources can be scaled by the customers, depending on the load. With PaaS, customers can transparently scale their applications. Vertical scaling describes pushing the application to a larger instance with more physical RAM or CPU. Horizontal scaling stands for spawning more instances that can serve user requests in parallel behind a load balancer. Also, some systems allow customers to automatically scale their applications. This can either be done autonomously by the system, e.g., App Engine or with policy- or restriction-based user configuration, e.g., Clever Cloud. 85 % of the providers allow horizontal scaling, 72 % vertical scaling, and 47 % automatic scaling.

A PaaS offering consists of a large technological ecosystem that defines the capabilities that are brought to the customers. Thereby, the ecosystem describes the complex system of interdependent platform components and external providers that work together to form and enable PaaS services. Parts of the ecosystem can be available runtimes, middleware, or geographical deployment regions that are supported by the PaaS. Moreover, many PaaS enhance their capabilities by a deep integration with third-party service providers that offer, e.g., storage or analytics services to enhance customer experience and platform functionality.

Today's PaaS support a variety of different runtime languages as basis for customer applications. Currently, such polyglot platforms officially support a

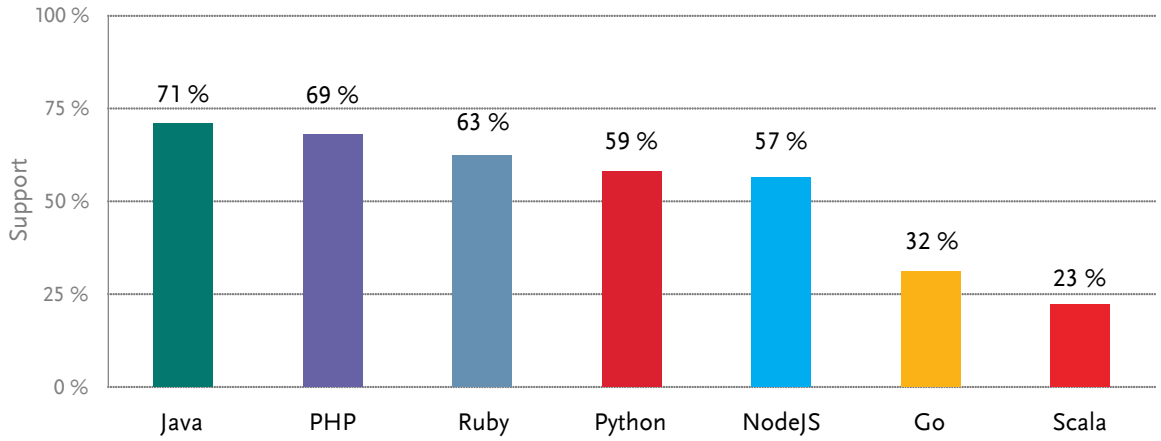


Figure 3.5.: Runtime Languages Support

total of 22 different languages. Language-specific platforms are increasingly developing into a niche market. In 2013, 44 % of the offerings only supported one particular language. Now, solely 25 % are still language-specific. This trend is also confirmed by the constantly rising average value of 2.9 to 4.6 languages per provider. Whereas a specialization on one language may result in better support of that language, more languages attract a larger mass of customers and allow more flexibility for developing in different languages while sticking to one particular PaaS provider. The top five languages based on percentage of support are Java, PHP, Ruby, Python, and NodeJS. Figure 3.5 shows the percentage of support of available languages.

Runtime language support matches with the popular languages as recognized by programming language indexes, e.g., TIOBE[53], PYPL[54], or Redmonk[55]. Out of the most popular languages as recognized by Github[56], the top five languages are even equal. Especially languages that are popular for web programming are supported. Only system programming languages like C-based languages such as C++ and Objective C are hardly supported. Also, wide support for new languages, e.g., Go show the cutting-edge approach.

With the trend of polyglot platforms, the need for an easy mechanism for supporting them evolved. Originally developed by Heroku, buildpacks[57] are a collection of scripts that define a generic API for detecting, compiling, and releasing, e.g., runtime languages or frameworks. Buildpacks enable the developers to add own packages of runtimes or services to their PaaS environment. Other vendors have either adopted Heroku's buildpack concept or defined their own extensibility mechanism. Here, the rise of Docker with its portable packaging format had significant impact on the extensibility of the systems. Also,

[53] https://www.tiobe.com/tiobe-index
[54] https://pypl.github.io/PYPL.html
[55] https://redmonk.com/sogrady/2018/03/07/language-rankings-1-18
[56] https://octoverse.github.com
[57] https://buildpacks.io

the isolation of applications via containers perfectly fits within most PaaS architectures. This capability gives the developers greater freedom and possibilities to configure the system, blurring the differentiation to IaaS. At the beginning of data logging, only 25 % of the providers supported such a mechanism. Followed by a rapid increase, now 59 % of the vendors are extensible. This confirms the assumption of increasing popularity of individual adaptability.

With regard to backing services, we see data stores, in particular database systems, as a first-class member of the ecosystem. As mission- and latency-critical parts of the system, these services must be hosted geographically close to the application. Due to this fact, these services are most often co-located in the environment and supplied by the PaaS provider. More than three quarters of provider-managed services are data stores. Consequently, the five most popular services are MySQL[58] (50 %), MongoDB[59] (39 %), PostgreSQL[60] (38 %), Redis[61] (35 %), and memcached[62] (18 %). However, many PaaS systems also have a deep integration with external value-added services. Platform providers cooperate with other companies that provide complementary products and services, i.e., add-ons, for the platform. Together, they form a value network that increases a customer's ability to deliver applications and the value of the PaaS offering [263]. This not only offers huge cross-selling opportunities but also complements the idea of outsourcing the programming environment and focusing on application code. In general, the partner ecosystem including add-ons and solution providers is an important capability of a modern PaaS offering.

Another evolution inside the market is the deployment model. Whereas in the beginning, the majority of offerings were only available as publicly hosted service, the trend to private infrastructures has substantially altered the deployment options. Overall, private hosting increased from an initial 34 % to 51 %. Although public deployments still dominate the market (76 %), customers now have more options for running PaaS systems within their private boundaries. This was also significantly pushed by events such as the PRISM surveillance program[63] or restrictive data security requirements. European providers, e.g., are bound to legal rights that prohibit the storage of customer data outside the EU [99]. Security considerations have ever since been regarded as a problematic factor by customers [178, 282, 347]. Nevertheless, very few PaaS providers adhere to compliance standards on their own, but solely inherit the one's from the IaaS vendors. The same holds for Quality of Service (QoS) and SLAs.

[58] https://www.mysql.com
[59] https://www.mongodb.com
[60] https://www.postgresql.org
[61] https://redis.io
[62] https://memcached.org
[63] https://en.wikipedia.org/wiki/PRISM_(surveillance_program)

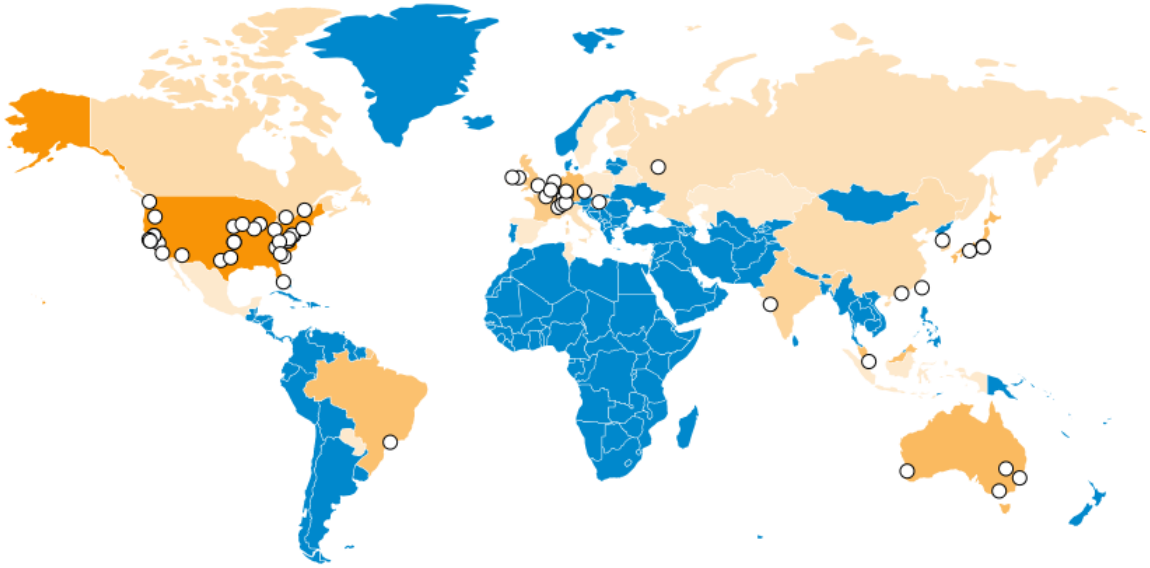


Figure 3.6.: Distribution of PaaS Infrastructures

In this context, an important factor is the geographical region an application will be deployed in. This is relevant not only because of legal but also performance reasons [75, 331]. An application deployed in a data center closer to the end user will have significantly faster response times [127]. Therefore, PaaS providers typically offer several deployment regions to choose from. In the past, few vendors did supply multiple regions. Even as one of the pioneers, Heroku did only expand their offering to a second region in Europe after multiple years in production.[64] Still, this is a crucial feature for companies operating in a particular region or willing to expand their services to different regions. Over time, there is a general tendency of expanding the available deployment regions. The average number of available infrastructure options in public offerings is 5.4 (mean 2.0), which however has a high deviation with a few providers with a lot of options. Most providers (> 49 %) use established IaaS offerings for their infrastructure needs. AWS is by far the most often used IaaS provider of PaaS vendors (81 times).

Figure 3.6 shows a heat map of the world depicting the geographical distribution and density of public PaaS infrastructures. The more saturated the orange countries are, the more offerings support an application deployment in that geographical region. The markers indicate data center locations that are specified more precisely. Infrastructure support on particular continents is different between PaaS. North America is the continent with the best coverage, supported by 72 % of the providers. Next comes Europe with 67 %, followed by Asia (35 %), Oceania (31 %), South America (26 %), and Africa (2 %). Only one provider supports a data center in northern Africa. If a PaaS supports a certain region, it is still not certain if all services and add-ons are available in

[64] https://devcenter.heroku.com/changelog-items/248

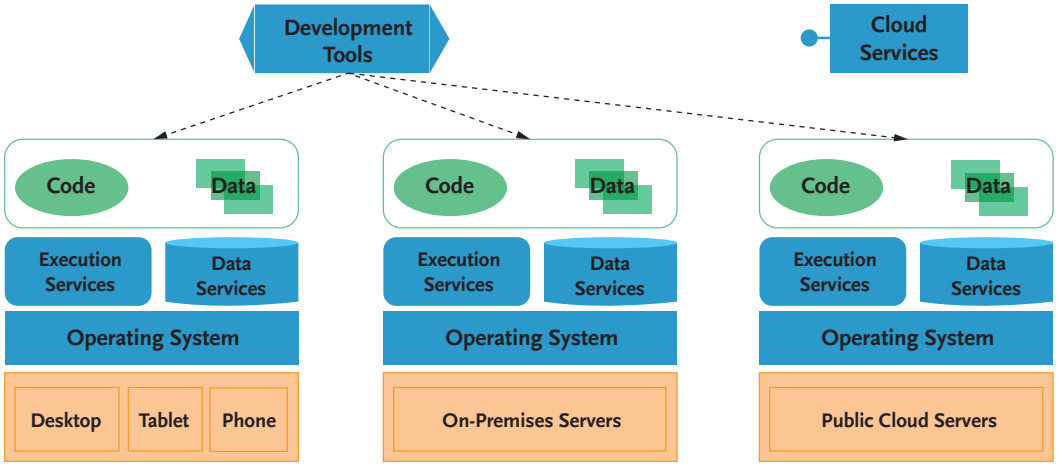


Figure 3.7.: Application Platforms [66]

that region. A lot of providers still lack support for this. For some add-ons this may not be crucial but for latency-critical services like databases this will significantly impact performance. This is also difficult to achieve for add-ons because they are managed by a third party and do not always support multiple regions or even the same data center as the PaaS provider.

The previous observations complement the comprehension of PaaS from a market view and will also be used as a reference for decisions in the course of the thesis.

3.4. Architecture

The following section provides an overview of how a typical PaaS system is build. It shall enhance the understanding of the inner workings of PaaS systems, with regard to the thesis' technical contributions. This includes the high-level architectural components of the system and important internal and user workflows. The focus of this observation is how IaaS-centric and Generic PaaS work, where customers deploy their application source code. SaaS-centric PaaS have a different abstraction of the PaaS system with fewer commonalities between the different systems. This manifests itself in, e.g., less configuration options and a more restricted technological setup. Nevertheless, we try to give insights on how SaaS-centric systems might differ.

In general, the services provided by an application platform can be grouped into five categories [66] (see Figure 3.7): An operating system is the foundation providing basic services on which all applications depend, such as a file system and process scheduling. The building block execution services provides programming language runtimes, application libraries, and more for running the code. Data services are responsible for storing and processing the data that the application is handling. The main technology in this category is a DBMS, but other services for handling, transforming, and analyzing data are becoming

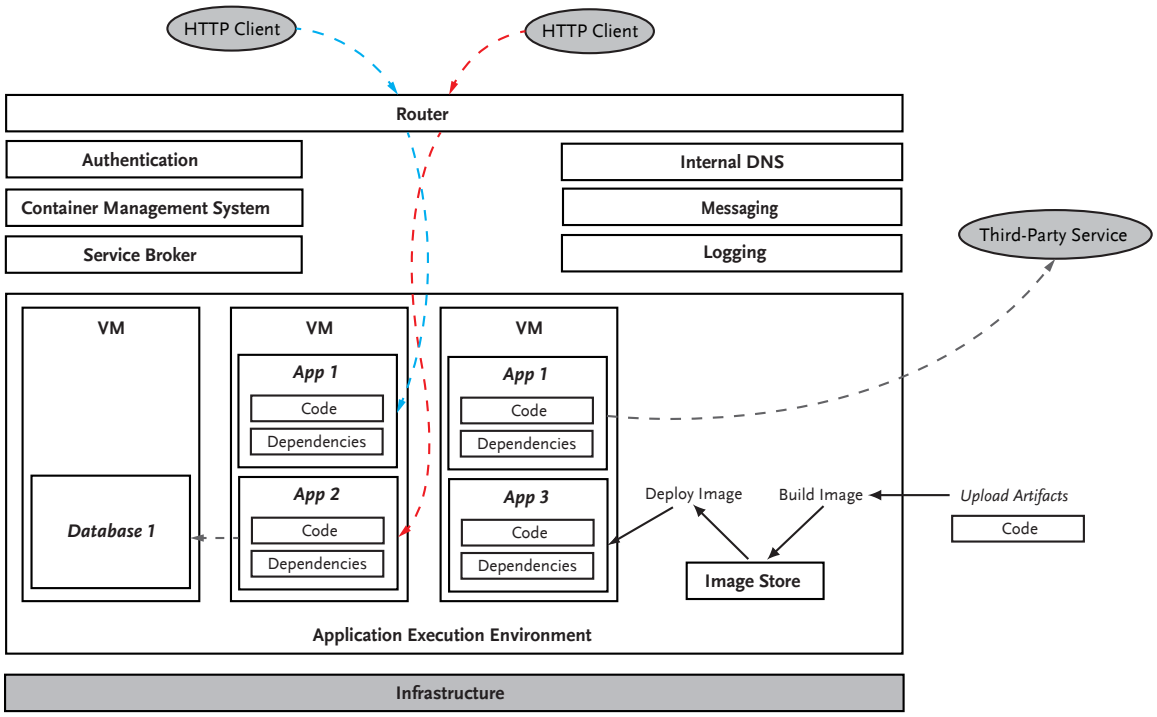


Figure 3.8.: PaaS High-Level Building Blocks

increasingly important. Besides the local stack, external cloud services may be integrated, offering remotely provided functionality that applications can use. For the development phase, development tools may be integrated with the application platform, helping development teams to create and maintain applications. The described software stack may build upon several hardware infrastructures, such as mobile devices, on premises servers, or the cloud [66].

As a next step, we take a closer look at the internals of a PaaS system. We are guided by Heroku's architecture[65] as most of their concepts have been established as de facto standard in the field. Figure 3.8 shows the overall simplified architecture of a PaaS system.

Define an Application

Before an application can be deployed to a PaaS system, several information needs to be provided to define the application. An application itself consists of the source code and the description of any necessary dependencies. Depending on the utilized build system, the application dependencies are declared in a specific build file, such as a `Gemfile` for Ruby with Bundler. Besides the source code and the dependencies of an application, it is important for the PaaS system to know what and how to execute an application. For most of the established application frameworks, such as Ruby on Rails, the PaaS will be able to detect sensible defaults to run the application automatically. If the developers

[65] `https://devcenter.heroku.com/categories/heroku-architecture`

need a custom configuration or the application type cannot be detected, an additional configuration file like the Procfile[66] is necessary. This file specifies the commands that are executed to run the application. Additionally, different process types, e.g., background tasks may be configurable.

Deploy an Application

Deploying an application involves uploading the application artifacts to the PaaS. Whereas there are scenarios where a runnable application image, e.g., Docker image, is deployed to the system, the default method transfers the source code to the PaaS where a runnable image is created for the developer. Therefore, different techniques are available such as deployment via Git, a CLI, or via an API. The prevalent option is to use Git for deploying applications. These days, Git is the most popular version control system and most application code is under Git revision [26]. In this way, the deployment workflow employs the same tools that are used for developing the code and therefore introduces no workflow disruptions. Also, the benefits of pushing only code differences and not the entire code base as well as easy rollbacks between versions come at no additional costs. When a new application is created at the PaaS provider, it associates a new Git remote repository with the local Git repository of the application. As a result, deploying code is equivalent to using the familiar `git push` command but to the provider's remote instead of the main code repository. This workflow makes it easy to automate the deployment in a Continuous Delivery environment that may automatically push to the remote after successful integration testing.

Build an Application

After the code is uploaded to the system, the internal workflow to build the source code is triggered. The automation scripts are responsible for transforming the application source code into an executable package based on the platform's OS stack. The OS stack is the fundamental OS image that will run the application. Most PaaS stacks are based on an existing open source Linux distribution, such as Ubuntu[67]. This base image will be automatically updated by the PaaS provider to keep the environment up to date and secure. The build mechanism follows a similar pattern with some language-specific alterations. This includes retrieving the specified dependencies of an application to creating any necessary assets, from processing style sheets to compiling the application code. A Java application, e.g., may fetch library dependencies using Maven, compile the source code together with those libraries, and produce a Web Archive (WAR) file for execution. Buildpacks[68] are a popular method to unify

[66] `https://devcenter.heroku.com/articles/procfile`
[67] `https://www.ubuntu.com`
[68] `https://buildpacks.io`

this compilation process. Buildpacks take the source code, its dependencies, and the language runtime, and produce runnable packages of an application. The vendors and the community provide buildpacks for a multitude of use cases. These buildpacks combine the efforts of the contributors to serve a means for supporting the variety of available application use cases. This especially became valuable as the majority of PaaS vendors transformed from language-dependent to polyglot platforms which require to support a broad set of runtime languages and application scenarios. The kind of executable package that is created for the platform depends on how the environment handles applications. On Heroku, applications are completely self-contained and do not rely on runtime injection of a web server into the execution environment to create a web-facing service.[69] Therefore, the application needs to declare a dependency to embed a web server library in the application, such as Jetty[70] for JVM-based languages. This happens entirely in the user space, within the application's code. In contrast, other vendors still provide a particular web server as middleware in their system and only require to build a runnable WAR for deployment in a system-resident web server. This makes the application less self-contained and intertwined with the platform's stack, limiting the user's choice for a web server and its configuration. However, the vendors can provide a more optimized web server to the users. The final application packages are typically stored in some kind of blob storage for later execution.

Configure an Application

An application's configuration is everything that may vary between environments, e.g., development and production. This includes resource handles to backing services such as databases, credentials, or any variables that provide specific information to an application. It is reasonable to extract this information from the application code and even from the application artifacts, e.g., configuration files. This enables to customize this information independently of the application code. The configuration values are exposed to a running application via environment variables. Whereas not all PaaS systems did support this separation of concern, it has become the de facto standard in the field. As we can see in Figure 3.8, in this way, links to services such as databases can be configured. Also, external add-on services by third parties can be easily attached to an application. Internally, a *service broker* is responsible for providing access to an appropriate service instance for the application.

[69] `https://devcenter.heroku.com/articles/runtime-principles`
[70] `https://www.eclipse.org/jetty`

Run an Application

The runnable application package, the configuration options, and the necessary backing services form a specific release of an application that can be executed by the platform. The environment where the application runs differs between PaaS systems. In most cases, the application package will be put inside an isolated, virtualized Linux container such as provided by Docker (see Figure 3.8). Multiple of such containers may be executed inside an enclosing VM which provides the OS stack. Nevertheless, there is still the possibility that the application is directly executed inside a dedicated VM. Whereas early PaaS offerings isolated the application instances solely based on VMs, after the rise of Docker and the container technology, most of the vendors changed to the more lightweight and faster isolation model via containers [389]. However, there are still arguments for both of the used technology options, especially regarding security and resource isolation, e.g., performance interference [287]. A *container manager* is responsible for managing the life cycle of the container instances. This includes starting and stopping of container instances during runtime as well as detecting crashed instances or problems with the underlying hardware that require the container to be moved or restarted. When a new version of an application is deployed, new instances are started to replace all currently running instances that are finally terminated.

As we can see in Figure 3.8, the PaaS system is deployed onto and uses some infrastructure for conducting the defined operations. This may be a shared or dedicated infrastructure pool maintained by an IaaS provider or a private infrastructure owned by users. If a customer decides to host his application on a public PaaS, he cannot be sure that all the parts and the data will be operated by and stored at the PaaS provider. Often, different service providers cooperate with each other in a complex and networked business environment. Thus, a cloud service may be facilitated by a mutual dependence among SaaS, PaaS, and IaaS providers [263]. There exist different strategies among providers to supply PaaS services to their customers. Our research has shown that these architectures and involved parties are not always transparent to the customer. A customer in turn may be willing to accept this or not depending on, e.g., legal rights. We identified four major strategies (see Figure 3.9).

We distinguish between a *full-stack provider* that develops and operates all parts of the PaaS system from his own capabilities and providers that involve one or more sub contractors that supply facilities like IaaS base functionality. Examples for full-stack providers are Windows Azure, Google App Engine, and Amazon Elastic Beanstalk. An *upselling IaaS provider* is a provider whose core business is to operate infrastructure services and enhances his portfolio with a proprietary or open PaaS from another party. An example for a so-called white-labeled *proprietary PaaS* is Jelastic[71]. Other providers may customize

[71] `https://jelastic.com`

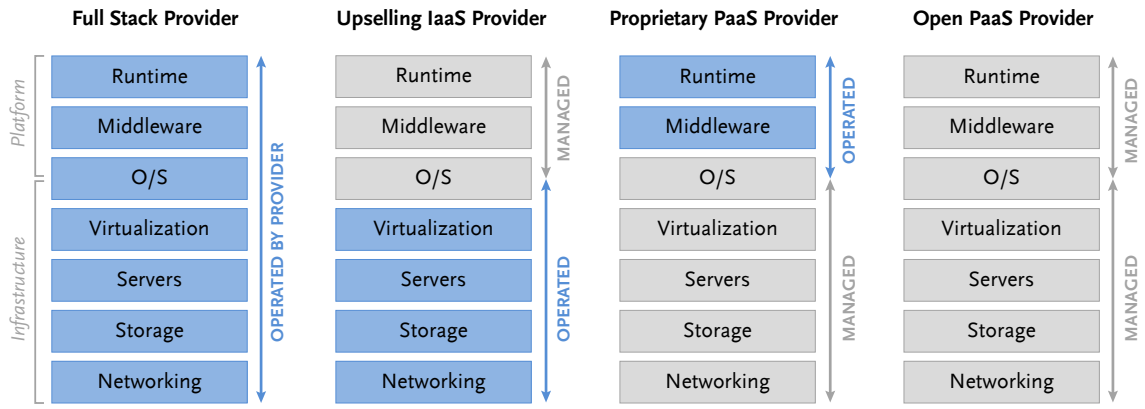


Figure 3.9.: Provider Stacks

open source PaaS like CF on their own infrastructure, e.g., Anynines[72]. The most common approach is to rely on an external IaaS provider and add a proprietary PaaS offering on top of it. AWS is the clear leader in delivering IaaS capabilities for most existing PaaS providers. Another solution is pursued by an *open PaaS provider*. Both, the IaaS and the PaaS is provided and developed by a third party. The provider itself is only responsible for managing the entire offering. This facilitates the provision of a PaaS offer with the help of existing open source PaaS, e.g., CF or OpenShift.

Access an Application

Several system components interact to make a running application accessible to the Internet. The main component that is contacted by an HTTP client for accessing an application is the *router* (see top of Figure 3.8). Together with the *internal DNS*, the router is responsible for directing external HTTP requests to the correct application instance and for balancing between available application instances. Therefore, the router periodically queries or gets a broadcast of the most recent information from the DNS, to determine which VMs and containers each application currently runs on. Using this information, the router creates the latest routing tables for the application instances. The router is then responsible for directing HTTP requests to the application processes. Every time the container manager actively changes the container formation or detects modifications, e.g., starts a new application instance, this information needs to be populated to the DNS and the router. For example, each new deployment causes a new upstream IP and Port for an application instance. Additionally, for monitoring purposes, the system typically includes a *log manager* that collates the stream of logs produced from all the processes inside the application instances providing a single source of activity. Furthermore, an *authentication* system provides the identity management for the entire PaaS.

[72] `https://paas.anynines.com`

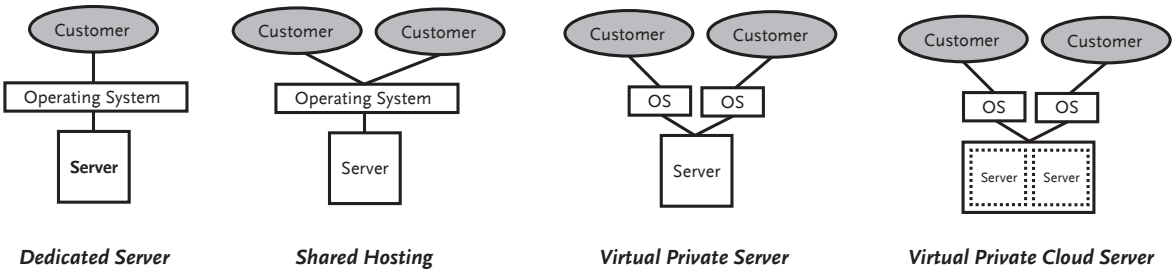


Figure 3.10.: Differentiation of Server Resource Sharing Technologies

3.5. Distinction from Related Technologies

As the three major categories SaaS, PaaS, and IaaS (SPI) all cover a wide range of offerings, several intermediate categories emerged. Some of them are either a subset of one of those categories or constitute an overlapping set between them. In general, a cloud service category is defined as a group of cloud services that possess a common set of qualities [161]. However, often these categories do not have a clear definition, are named exactly as other categories, and can be easily mistaken with other products which leads to the ongoing confusion in this area. Some vendors might also benefit from this confusion as customers cannot compare existing offerings and vendors can sell their offerings with a lot of business buzz. This is also the reason why we limit our categorizations in this thesis to the dimensions of the SPI model, as in our opinion more “as a Service” categories will not help. In the following, we are trying to clarify several terms starting with the pre-cloud era up to a set of XaaS categories. Next to other cloud service categories, we also describe and differentiate technologies which are important to the evolution of PaaS.

3.5.1. Differentiation from Pre-Cloud Technologies

Before the cloud era, there also existed several technologies to enable and assist application developers to serve their applications to customers over the network and particularly over the Internet. At first glance, the qualities that distinguish these technologies and offerings from other cloud services and especially PaaS are not immediately evident. Therefore, we briefly introduce and differentiate important related technologies in the following section. Figure 3.10 visualizes characteristics and differences between the presented technologies.

Dedicated Server

In contrast to the cloud, a dedicated server is a bare metal machine which a client can access and administer as sole customer. This is comparable to a single server machine that is connected to the Internet. All hardware is physically restricted to one customer and there is no shared access by other

customers [215, 299]. Isolation and security is maximal and there are no load interdependencies as the hardware resources are used exclusively. However, vertical scaling is not achieved easily as the hardware is limited and can only be changed manually by upgrading the server hardware. A related problem is resource under-usage and idling of system resources leading to high costs for customers [18, 299]. The software and application runtime environment must be managed by the customer as in the case of IaaS.

Shared Hosting

Shared hosting is a popular option for web hosting. Often based on a dedicated server, several users share the same host system for their applications [361]. In shared hosting, the provider is responsible for managing servers, installing server software, security updates, technical support, and other operational aspects of the service.[73] Hence, the users share the same server operating system and runtime libraries. A well-known example for such offerings is the LAMP software bundle, which is an open source software bundle consisting of the operating system Linux, the web server Apache, the database MySQL, and the runtime language PHP [203]. From the pre-cloud technologies, this option is closest to the platform character of PaaS as the runtime libraries are shared among the users of the server and little to no configuration is possible. So if users want to use another runtime version or have other requirements on the installed software, they need to switch to a VPS or dedicated server. Compared with the other hosting options, it has the least isolation of them. Isolation between users is often done via folder restrictions. Most systems also include a system administration for user and customer configuration options, such as Plesk[74]. Due to the lack of isolation, security breaches between the users are a threat, e.g., if file permissions are improperly set, giving other users or processes access to these files. Sharing the host's resources, such as CPU, RAM, and sometimes also the IP among the users, aims to achieve better resource utilization and to deliver economical cost benefits to the users [215]. Nevertheless, system performance is less isolated and configurable as with VPS which may lead to severe performance issues for co-located users [256, 361]. Another difference to today's PaaS are less automation and integration tools for deploying and operating applications. In most cases, applications have to be manually uploaded inside a folder structure via a UI or File Transfer Protocol (FTP). Thus, we observed a tendency that shared hosting providers took the chance to evolve their offerings into quasi-PaaS services.

[73] https://en.wikipedia.org/wiki/Shared_web_hosting_service
[74] https://www.plesk.com

Virtual Private Server

A Virtual Private Server (VPS) is a VM offered as a service to the customer by a hosting provider. To the user, it provides the illusion of a dedicated server, which is achieved by virtualization software such as VMWare [374] or Xen [24]. The underlying dedicated hardware, such as compute and memory, is shared between multiple VPS. The share of resources that is assigned and usable by a customer can be controlled. This is a major difference to the shared hosting approach, where often only the disk quota is restricted for the users but not the CPU or memory usage. By assigning the different virtualized environments a fixed share of the resources, the impact between the physically co-located applications can be reduced. The operating system software and all configuration are manageable by the user. This enables fine-grained control over the application environment just like on a dedicated server but also introduces the burden of the maintenance and operations work. The major difference between a traditional VPS and a cloud VPS is the underlying infrastructure (see Figure 3.10). Whereas traditional VPS often only share a single dedicated server among multiple clients, a cloud VPS abstracts the underlying hardware as an ephemeral system composed out of multiple dedicated servers, allowing the customers to scale more dynamically. Nevertheless, a VPS may also be based on a cluster of hardware machines. A VPS is also a central component of an IaaS solution besides other low-level capabilities and resources such as storage or network [237]. Another difference between the manifestations of a VPS can be seen in the payment model. VPS based on dedicated hardware are often sold with more long term agreements whereas cloud VPS facilitate the subscription model right up to on-demand instances with metered usage.

3.5.2. Differentiation from XaaS

For the entire amount of offerings that are or can be delivered as a service now or in the future, the acronym XaaS, for Everything as a Service, evolved. In theory, literally every *thing* can be delivered "as a Service." This spans from physical goods to intangible work, i.e., services. Occasionally, this leads to offerings such as Surface as a Service[75]. Here, Microsoft Surface[76] notebooks are rented to enterprise customers. Given such a usage of the term, every pizza delivery service would have done Pizza as a Service for decades. Originally, the "as a Service" pyramid is focused on hardware and software services [90]. Therefore, we not only restrict XaaS to IT technologies but also expect that the delivery must be done via the Internet [237]. So low-level offerings such as Hardware as a Service are per se still included, but they must be facilitated by software that allows users to access these resources on demand over the

[75] https://www.surface-as-a-service.de
[76] https://www.microsoft.com/en-us/surface

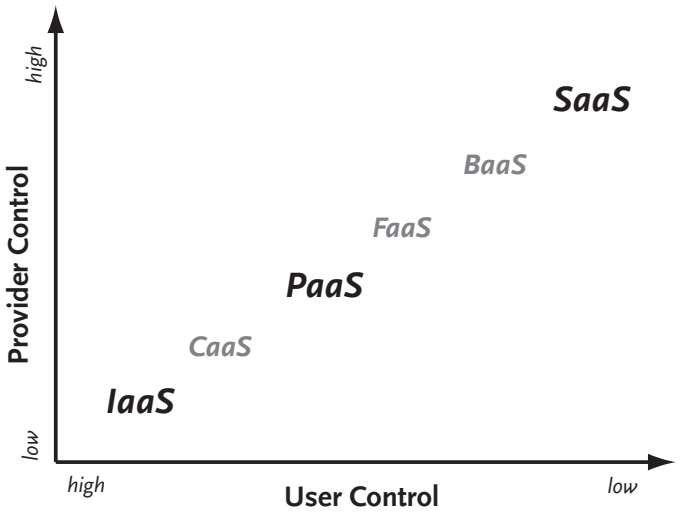


Figure 3.11.: Everything as a Service

Internet. All XaaS offerings share a common set of functionalities that make such a business approach interesting and valuable for customers. Among others, they all offer an on-demand, managed service with self-service capabilities to the users, together with a typically low entry barrier in terms of costs. As there are a multitude of offerings and categorizations termed and categorized within some XaaS cluster, we need to focus on the most important ones. For our case, this is defined by the relation to the enabling of application development and hosting. So categories and technologies that have no closer relation to these aspects are out of focus.

In the following, we want to further examine a few important models that help us to either cluster a larger set of offerings into a service category, i.e., BaaS or are closer related to PaaS, i.e., Function as a Service (FaaS) and Container as a Service (CaaS). Figure 3.11 gives a visual overview of the discussed XaaS categories and offerings. The current notion is bound to the IT paradigm shaped over decades and aligned to IT infrastructure, i.e., parts of the existing IT stack. Other than that, there are also a multitude of other abbreviations that often also collide with the most well-known ones. Actually any service that can be offered over the Internet can be marketed under an artificial “as a Service” abbreviation. Typically, any of them can be categorized in one of the presented categories or in special cases inside the whole schema.

BaaS

BaaS not only includes traditional back end services such as databases offered as a service, but may encompass every service like search, alerts and notifications, media processing, and other features that can be used for application development.[77] These services can be accessed via APIs, SDKs, or their native

[77] https://martinfowler.com/articles/serverless.html

protocols and reduce the effort for the developers by providing services that are often needed and used inside applications. This broad definition makes BaaS a cloud category with a lot of different services. However, it mitigates the confusion of having a multitude of subcategories for every other back end service. Consequently, we would categorize services such as Database as a Service (DBaaS) [5, 129, 136] or Monitoring as a Service (MaaS) [238] solely as BaaS. Like PaaS, BaaS is about reuse and speeding up the development cycle. Also, some tasks are just better of with a third-party provider that is really proficient with the specific technology. PaaS add-on services are typically BaaS offerings integrated within the PaaS marketplace. Here, cross-selling opportunities arise, as PaaS providers can just maintain the most important services natively but not the whole stack of services an application developer might want to use. Whereas some researchers equally delimit BaaS and Mobile Backend as a Service (MBaaS), we see MBaaS as a subtopic of BaaS, specialized on mobile applications. Hence, these services provide multiple modules that realize the most used back end functionalities of mobile applications. Altogether, the idea is that several activities like push notifications, social network integration, user management, analytics, and cloud storage are needed by the majority of applications. Therefore, it makes sense to provide these services instead of rewriting them for every application. Additionally, back end functionality and processing is outsourced to these services to reduce and compensate computing power and time on mobile devices for CPU-intensive jobs. Popular examples for mobile-first BaaS are Kinvey[78], Firebase[79], and CloudKit[80].

FaaS

FaaS, often equated with *serverless*, is the idea of deploying functions instead of entire applications. Whereas the term serverless only suggests the absence of managing a server and the need for resource planning for the user, which arguably also holds true for some PaaS offerings, it is most often used in conjunction with the FaaS paradigm [228]. In the following, we stick to this usage without reopening the ongoing debate. Kanso and Youssef [176] deem FaaS as a natural evolution of PaaS. Similar to a PaaS environment, the developer programs to a predefined and provider-managed runtime environment in the cloud. Currently, the available execution environments are typically restricted to a smaller set of languages compared to PaaS. Extensibility mechanisms and the lack of freedom of choice of the programming language can be compared to the early stages when PaaS was just on the rise. Also, the community is facing a similar problem to our initial motivation for the dissertation project: the lack of a clearly defined terminology, model, and a vision for research opportunities and challenges

[78] `https://www.kinvey.com`
[79] `https://firebase.google.com`
[80] `https://developer.apple.com/icloud/cloudkit`

for the future [364]. FaaS is a progression of the ongoing trend away from centralized, monolithic applications, to smaller domain-driven applications, such as Service-Oriented Architectures (SOAs) [97] or microservices [211]. Its function-based model of small, stateless execution units is more fine-grained and not tailored around an application context. This is particularly fitting for bursty, CPU-intensive workloads, fulfilling the promise of elasticity without any cluster management overhead for the users [170, 364]. Consequently, the billing of FaaS services is even more fine-grained than other cloud services and typically the execution time (in blocks of milliseconds) of the functions is charged. Unlike PaaS, the function code is idle until it is triggered by an external event, so no instances are running continuously [9, 22]. Whereas some PaaS vendors support auto-scaling of applications, operational logic and scaling is inherent to FaaS platforms. Overall, FaaS can be classified in between PaaS and SaaS. Typically, the classification of the cloud models is based on the management of the operational logic (see Figure 2.3). From this perspective, FaaS fits in the gap between PaaS and SaaS, as more management is done by the provider but there is still an amount of management on the user's side, compared to SaaS [364]. Examples of current FaaS platforms are AWS Lambda[81], Google Cloud Functions[82], Azure Functions[83], and OpenWhisk[84]. Also, several frameworks try to unify those efforts and make it easier to deploy cloud functions, such as the Serverless Framework[85].

CaaS

CaaS became popular with the rise of the Docker technology. It makes it easy to use container-based virtualization in combination with service orchestration. In general, it is more low-level than PaaS and not focused on an application but provides infrastructure in the form of isolated containers comparable to what a VM is for IaaS. Thereby, the user has full control over the container image, including the application, but also full responsibility, in terms of packaging, maintenance, and upgrades. In our cloud stack (see Figure 3.11), CaaS can be positioned in between IaaS and PaaS [288]. In its pure form it is closer to IaaS, as the user manages more of the operational logic and has more control over the technological system. Yet, most often, such systems also include a lot of management functionality that makes it easy to manage the deployment and overall life cycle of containers, similar to what PaaS provides. As containers are one of the key enablers of PaaS, there also exist PaaS systems that support the usage of container images instead of application packages directly. CaaS can also be seen as a foundation a PaaS can be build on. PaaS offers an opinionated

[81] https://aws.amazon.com/lambda
[82] https://cloud.google.com/functions
[83] https://azure.microsoft.com/services/functions
[84] https://openwhisk.apache.org
[85] https://serverless.com

developer experience to go from code to a world-wide scalable service in no time, whereas CaaS provides the foundation for this pledge.[86] Examples for CaaS are Amazon Elastic Container Service (ECS)[87] and Kubernetes Engine[88].

3.6. Summary

Although the term and concept of PaaS is widely known, the market is not as homogeneous as the existing definitions let one believe. The lines between IaaS, PaaS, and SaaS are blurring and within these categories, numerous subcategories emerged that describe a whole range of different approaches [18, 134]. Therefore, it is necessary to further study the market to distinguish and classify the different categories within to avoid confusion among researchers and customers [21, 132, 217, 302, 349]. As a foundation for all further investigations, we analyzed the current notion and state of the art of PaaS in this chapter. At first, we conducted a literature study to evaluate definitions and characteristics of PaaS. After examining important characteristics and their manifestations, we proposed three distinct PaaS categories in between the boundaries of IaaS and SaaS for refining the differentiation between PaaS offerings. Afterwards, we contrasted the theoretical considerations with the state of the art by means of a market analysis. Beyond that, we also outlined the typical architecture of PaaS systems. For a thorough understanding, we differentiated PaaS from similarly motivated technologies from both the pre-cloud era and the universe of XaaS. The presented classification, definitions, and the overview of the state of the art serve as input for the following chapters and allow us to draw on the findings to study existing portability issues among them.

[86] https://kubernetes.io/blog/2017/02/caas-the-foundation-for-next-gen-paas
[87] https://aws.amazon.com/en/ecs
[88] https://cloud.google.com/kubernetes-engine

Part III.

Platform as a Service Broker

4. Model and Knowledge Base for Platform as a Service

Parts of this chapter have been taken from [191, 192, 194].

In this chapter, RQ 2 *("How to model and capture knowledge of Platform as a Service offerings inside a structured knowledge base?")* is supported.

4.1. Motivation

Over the last years, the cloud hype led to the emergence of a large amount of cloud offerings. They span the whole cloud stack from IaaS to PaaS to SaaS. Especially in the PaaS market, a number of smaller providers tried to gain a foothold in the cloud market. As we have shown in Chapter 3, the composition and characteristics of PaaS can be customized in many ways. Hence, the competition between providers is not only limited on price differentiation but can be influenced by the capabilities of the system. Consequently, a lot of divergent offerings exist that are not directly comparable for customers [132, 303]. For providers, this differentiation is an integral part to attain and retain their market share in the face of market pressure, but for customers this inevitably leads to some sort of lock-in [39, 92]. In such a scenario, the change to a different provider leads to substantial additional costs for necessary migrations [138, 348]. However, business requirements as well as the capabilities and contract terms of the provider can change over time which makes it essential to preserve as much flexibility as possible to switch between different vendors. With the market and cloud offerings steadily evolving, portability is even more important for business continuity [21].

As the offerings do not share one common set of portable capabilities but rather intersect with one another at different parts, it is an option to look at portability between PaaS platforms from a local application view, starting from a particular configuration, and to identify a set of potential partners. When trying to compare providers, their ecosystem is a good candidate for evaluating differences and commonalities between them. The term ecosystem thereby describes the complex system of interdependent components and capabilities that work together to enable a PaaS cloud. To compare and select providers in a structured way, a common model for describing these properties is necessary.

Until recently, the vendors' white papers and documentation were the only source of data for customers to acquaint themselves with the topic. Since the whole process involves a lot of manual tasks that are costly and as data is only available in an unstructured form, the need for a consolidated knowledge repository evolved [349, 394]. Such knowledge management is important to allow customers to make informed decisions when evaluating and selecting providers. Nonetheless, no publicly accessible repository with a decent data set in terms of amount, actuality, and quality is available to act as a broker between offerings and customers.

To achieve this, we define a model of PaaS offerings including the technological stack and platform capabilities relevant for application portability. We argue that if offerings have intersecting capabilities, the sets of capabilities can be used by a heuristic for evaluating application portability based on application requirements. Based on this model, we derive a standardized, machine-readable profile with a common set of capabilities that exist among PaaS providers. The profiles are the foundation for different use cases including discovery and lookup, filtered retrieval, and matching with an application requirements profile. Our approach is empirically validated by providing a web-based application for these use cases together with a comprehensive data set of 71 PaaS offerings.

The remainder of this chapter is structured as follows: In Section 4.2, we define and evaluate the pragmatics for the design of our PaaS model based on application portability via the technological software ecosystem. Section 4.3 introduces a generic model of current PaaS systems. Based on this abstraction, we assess and categorize different portability challenges for PaaS systems. Alongside 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 4.4. In Section 4.5, we present *PaaSfinder*, a web application that makes the data accessible for customers. Moreover, we initially validate the idea of ecosystem portability by porting the application to different providers and identify further portability problems that must be investigated on finer levels of granularity. Section 4.6 reviews related work and discusses distinctions and rationales for deviating design decisions. Finally, Section 4.7 summarizes the chapter and discusses future work.

4.2. Model Design

In the following, we elaborate on fundamental model properties that are relevant for understanding the boundaries and restrictions of our model. In general, models are an abstract representation of existing natural or artificial entities that can be models themselves [344]. According to Stachowiak [344],

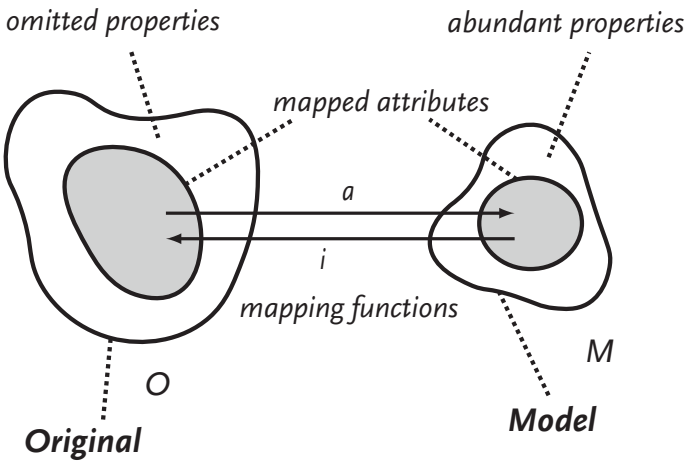


Figure 4.1.: Original to Model Mapping [344]

there exist three main characteristics of a model: *Representation*, *Reduction*, and *Pragmatics*. See Figure 4.1 for an illustration of the coherences.

Representation is defined in the sense that there always exists a function a which assigns a model M to the original O (abstraction). Additionally, there is a nonunique backward mapping i which assigns originals O to each model M (interpretation). Reduction is an important property, as models should not capture all properties of the original, but only those that are relevant to the model creator or users. Consequently, models are a reduced representation of the original [162]. This leads to the third attribute pragmatics. A model replaces the original for a particular purpose. This purpose is also an influencing factor for the choice of relevant attributes for the reduction. Altogether, a model is a simplified mapping for a special purpose. All the described properties are also relevant for our model and influence the modeling outcome.

The purpose of our mapping does directly influence the reduction of the modeling function. Often, related works to our approach do not explicitly state the purpose of their mapping, but just categorize it as the selection of a cloud vendor. In rare cases, they conduct empirical surveys or questionnaires to determine which attributes subjectively matter to the target group [335]. However, this makes the selection of attributes implicit and in the end leads to incomprehensible and arbitrary sets of attributes. This does not mean that there may not be additional attributes inside the model that are not immediately relevant for the specific purpose, but the set of core attributes needs to be appropriate for the selected purpose. Therefore, we want to describe the pragmatics and intentions of our model mapping in the following. The model mapping is based on the notion of application portability and a constrained notion of ecosystem portability for PaaS systems as a portability heuristic in a limited portable world. Section 4.2.1 elaborates on the general notion of application portability in PaaS systems, before Section 4.2.2 describes our notion of ecosystem portability.

4.2.1. Platform as a Service Portability

Achieving portability between cloud offerings poses manifold challenges. The EU commissioned study SMART 2011/0045 [50] even concludes that portability is the second most important obstacle hindering increasing cloud adoption. An obvious solution to this is often standardization [314]. Naturally, several standardization organizations have addressed cloud standards. Whereas some of them have gained traction, e.g., OVF [158], most initiatives remain disregarded by practice. Moreover, only very few of them consider the PaaS model as their main objective rather than IaaS. Yet, we can see that vendors already have competing ideas and approaches for standardization. In the past, software standards have failed to achieve portability in various ways [60, 114, 323]. For the cloud, especially the lack of acceptance by industry leaders prevents adoption. For that reason, we pursue a no-standards approach for application portability with no intermediaries and instant applicability.

According to the NIST [152], there are two main interfaces that are exposed to the customer that must be investigated when looking at portability and interoperability problems (see Figure 4.2). These are the Self-Service Management API, i.e., the management interface, through which the cloud user manages their use of the cloud and the functional interface provided by what is resident in the cloud [152, 264]. 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 life cycle and configuration settings of the platform.

Petcu [286], Sheth [329], and Oberle [262] et al. define three different dimensions for cloud portability: service portability, functional portability, and data portability. Service portability is defined as the ability to add, reconfigure, and remove compute resources on the fly. Functional portability refers to the platform-agnostic definition of application functionality. Finally, data portability includes import and export functionality for data structures across platforms [286]. Whereas service and functional portability 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 and import routines to enable portability between databases. Solutions for this problem should be developed independently of the PaaS context.

Hence, we focus on the interoperability of the Self-Service Management API and the functional portability of applications in this thesis. Interoperability of the management interface can be achieved independently of the functional interface. In this chapter, we focus on the portability of application artifacts first. Therefore, we concentrate on portability approaches for the functional interface and omit efforts on the standardization of the management interface. The

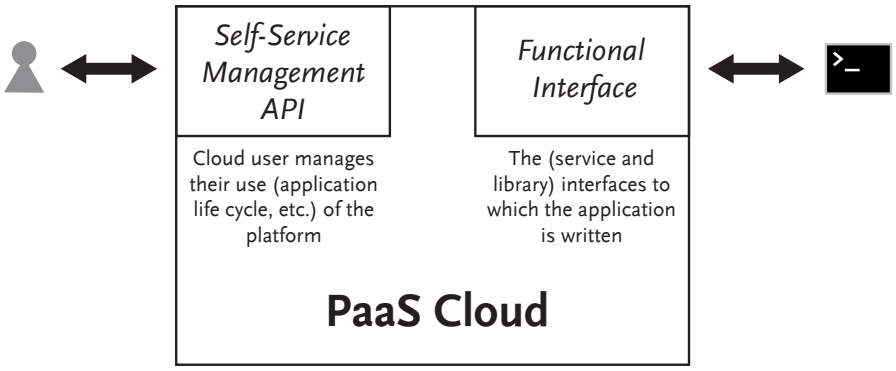


Figure 4.2.: PaaS Portability Interfaces [152]

interoperability of the Self-Service Management API is separately investigated in Chapter 7.

Apart from the aforementioned dimensions, we think that we can tackle portability on different levels of granularity. To illustrate the approach, 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 and services 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 be executable 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 4.3).

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. 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 at the upper two layers are well-defined capabilities of a PaaS offering that can be mapped to taxonomies. The bottom layer includes very specific implementation artifacts and restrictions

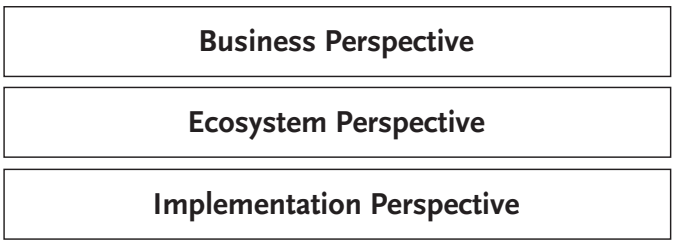


Figure 4.3.: Perspectives for Categorizing Portability Requirements

that require other approaches to formalize and test those requirements, e.g., static analysis or unit tests.

We focus on high-level portability of applications, i.e., the business and ecosystem layers, in this chapter and omit the details of the implementation perspective. Problems and consequences of the implementation layer are discussed in Chapter 6.

4.2.2. Ecosystem Portability

Commonly, portability is based on a set of attributes that bear on the ability of software to be transferred from one environment to another [157]. For portability of the functional interface two scenarios are possible: the portability of application dependencies versus the portability of entire applications.[89] One can either port an application with all its dependencies as a single unit of delivery and take the dependencies with the application through extensibility mechanisms like buildpacks[90] or rely on the native support of application dependencies between PaaS.

One 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 would correspond with a uniform virtualization image format like OVF [158] for IaaS. Almost all PaaS systems use some kind of (container) virtualization, such as provided by Docker, atop of the operating system to manage and isolate applications. Standardized containers can encapsulate any application and will run consistently on virtually any server [339]. Thereby, runtime and image formats for containers as suggested by the Open Container Initiative (OCI) [266, 267] are a step in this direction. However, not every PaaS uses the same virtualization technology and consensus on such system-level technologies is unlikely. Also, establishing portability on this level would require the users to do more of the DevOps work such as creating an appropriate container image for the application and maintaining it. For keeping the promise of a low amount of DevOps work, i.e., only deploying the application artifacts from a folder or packaged as a standard application archive

[89] https://www.openshift.com/blogs/paas-standards-standardize-on-what
[90] https://buildpacks.io

such as a Java WAR, the creation of the instance image has to be standardized. But also here, the vendors have different bootstrapping processes. Despite the fact that many vendors use the buildpack technology, their buildpacks for different runtime languages and therefore the resulting container images are not identical. As an example, compare, e.g., the Java buildpacks of Heroku[91] and CF[92]. Therefore, we want to focus on a no-standards approach that works with the status quo.

On a higher level, portability can be based on application requirements and dependencies which means relying on native support and open technologies. From our observations, we see consensus with regard to a specific array of dependencies that are supplied and used for typical application development. Even before the cloud, organizations have used concepts like SPLs to define a unified platform of software components for intra-organizational software platforms [46, 47, 71]. Once such a software platform is made available outside the organizational boundary, the platform transitions from a SPL to a software ecosystem [47]. Naturally, PaaS vendors want to attract as many customers as possible by supporting their development needs which is why their software ecosystems intersect.

Our usage of the term software ecosystem is related but not equal to the term that emerged with the recently popular topic of Software Ecosystems (SECOs) [47, 140, 226, 241]. According to the earliest definition of Messerschmitt and Szyperski [241], "a software ecosystem refers to a collection of software products that have some given degree of symbiotic relationships." Others like Jansen et al. [166] see it as "a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them. These relationships are frequently under-pinned by a common technological platform or market and operate through the exchange of information, resources and artifacts." The definition by Bosch [47] focuses more on the common interest of the businesses in "the set of software solutions that enable, support and automate the activities and transactions by the actors in the associated social or business ecosystem and the organizations that provide these solutions." A later definition by Bosch and Bosch-Sijtsema [48, 49] states that a "software ecosystem consists of a software platform, a set of internal and external developers, and a community of domain experts in service to a community of users that compose relevant solution elements to satisfy their needs." As we can see and as Manikas and Hansen [226] note in their literature study, there is little consensus on what exactly constitutes a software ecosystem. However, the three main elements *common software platform*, *businesses*, and *connecting relationships* stand out [226]. Therefore, Manikas and Hansen [226] define it as "the interaction of a set of actors on top of a common technological platform that results in a number of software solutions or services." Popular ex-

[91] `https://github.com/heroku/heroku-buildpack-java`
[92] `https://github.com/cloudfoundry/java-buildpack`

amples of software ecosystems are Apple's App Store platform [226], Google's Android platform and the open source development environment Eclipse [140].

Depending on the abstraction level and the perspective from which we look at PaaS, we can identify and distinguish different software ecosystems. Actually, a PaaS system consist of an interplay of multiple software ecosystems, either leveraged or created by the PaaS system. From a remote perspective, a PaaS system offers a hosted software ecosystem for application development to the users as a service. What makes the concept of a software ecosystem interesting is that customers can derive added value if the core product is extended with functions that are outside the core competencies of the particular platform provider that can be delivered by Independent Software Vendors (ISVs) [147]. An example for such symbiotic relationships inside a PaaS ecosystem is the connection of platform vendors and other service providers (add-on providers) via the add-on marketplaces, e.g., the Heroku Elements marketplace[93]. Like Bosch [47], we are focused on the set of software solutions that intersect between the vendors, rather than the connections and business partners that interact within the software ecosystem. We solely target the technological ecosystem that can be used by applications and is supported by the PaaS vendors including third-party services provided by add-on providers. Whereas some PaaS vendors, such as Salesforce, created and defined their own ecosystem of software components which they open to developers, others have opted to only supply existing ecosystems for software development to their customers. Thereby, the set of technologies is not formally fixed or dictated by some ecosystem provider such as Apple but naturally evolves from the market competition and the customers' needs. In that context, Hanssen [140] distinguishes between open and (partially) closed software ecosystems. We are especially interested in open ecosystems, as (partially) closed ecosystems have less chance of portability and result in more lock-in, e.g., Google App Engine or Salesforce. From a closer perspective, one can also say that the PaaS providers take part in a variety of SECOs as they typically support multiple runtimes such as the Java ecosystem. Hence, they leverage and participate in several sub-ecosystems.

In contrast to the general definitions, our notion of the PaaS software ecosystem is focused on and restricted to the portability of applications between PaaS systems. This includes the technological software ecosystem and the capabilities of the system. Therefore, we define the PaaS software ecosystem as:

> **Definition 4.1 (PaaS Software Ecosystem)** *A PaaS software ecosystem is the technological ecosystem of software products and platform capabilities for application development which are supported and provided by a PaaS system, including integrations to external third-party software providers.*

[93] `https://www.heroku.com/elements`

Based on the different technological ecosystems of the PaaS providers, we want to define a notion of application portability that relies on the intersection of the ecosystems among them to avoid lock-in.

When talking about lock-in, we must distinguish certain boundaries that define lock-in. As stated before, we will hardly reach portability in the classical *write once, run anywhere* paradigm between PaaS. The configuration and components of the platform will always vary between providers, and we still have to deal with heterogeneous software ecosystems. This is due to monetary interests as well as the fact that a one-size-fits-all approach never has solved all problems appropriately. In addition, native options may be more powerful, i.e., have greater benefit that can motivate an adoption decision than standardized options [210]. For example, Google App Engine's Datastore[94] offers built-in automatic scalability and replication of data across data centers.

One strategy to avoid lock-in is to rely on a set of open technologies that can be used on virtually any system. That does not necessarily mean that these technologies are standardized itself. For example, NoSQL databases are quite popular in the cloud for being fast and scalable. If we take the document-oriented database MongoDB[95] as a representative, we see that it is not a standardized technology but still open source and available for just about any platform. In consequence, we are taking a low risk of lock-in when basing our system on this database because it can be used on premises as well as with most providers in the cloud. If we instead consider Google's Datastore again, we take a significantly larger risk of lock-in. It is closed source and before recently, it could only be used within Google App Engine hosted applications. Now, it is available as a service, but we still lock ourselves to Google as sole hosting provider of the database solution.

Figure 4.4 exemplifies the ecosystem portability approach for three requirements, including two application dependencies and one platform capability. The overlapping sections of the requirements include sets of providers that can be divided into partially compatible and compatible. Compatible providers support all required demands. Therefore, the application is portable to their system. Partially compatible providers support a subset of the specified requirements and might only be candidates if some application requirements can be relaxed or manually upgraded by the customer. In contrast, incompatible providers, i.e., all providers outside the subsets, do not support any of the requested requirements.

Hence, if all required technological components and capabilities are supported by a platform, we should be able to run our application with little to no additional adaption effort, exactly as demanded by the portability definition [159].

[94] `https://developers.google.com/datastore`
[95] `https://www.mongodb.com`

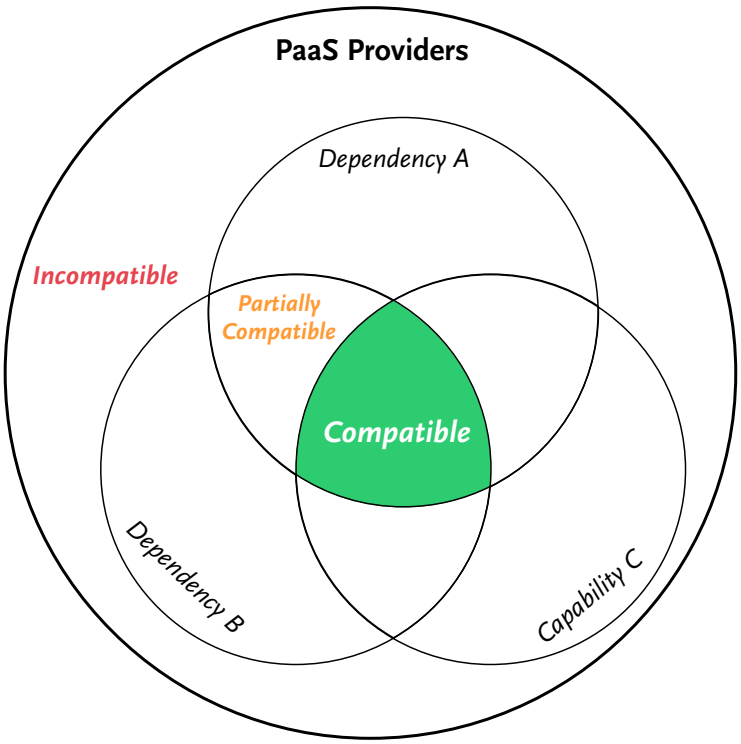


Figure 4.4.: Ecosystem Portability

Next to the components of the software, the characteristics of the service provider become an important factor when selecting a software service [322]. For instance, the location of the provider's data center and the implemented privacy policy are often important nonfunctional criteria [178]. A benefit of our approach is that it can deal with both types of requirements. Whereas this principle weakens the typical *write once, run anywhere* cross-platform benefits of standards, it has a wider range of applicability. Specifically, this scenario covers both, portability between clouds and application migrations from or to the cloud. At best, it has the fewest configuration work, as one does only need to move the plain application artifacts. To sum up, we state our notion of PaaS software ecosystem portability as:

> **Definition 4.2 (Software Ecosystem Portability)** *If all required technological components and capabilities of an application are supported by the PaaS software ecosystem of two PaaS providers, it should be possible to port an application between the two providers with little to no additional adaption effort.*

To apply our concept, two main questions need to be answered in the course of the thesis: What are the most important components and capabilities I need to match, and where can I port my application? The components and capabilities are defined and extracted from the model for PaaS systems which is presented in the following Section 4.3. Afterwards, we extract a taxonomy out of it that constitutes the structure for a knowledge base and is filled with

information for each provider. The knowledge base can then be used for answering the second question by matching application requirements with provider capabilities. Chapter 5 discusses this step in detail. The feasibility of the described approach is validated by two case studies in this thesis, including an extensive real-world application migration (see Section 4.5.2 and Chapter 6).

4.3. Platform as a Service Model

Due to conceptual differences, each cloud service model needs to be treated separately in terms of comparison, selection, or portability [94, 152]. Whereas the entities and interfaces of IaaS systems like compute, network, and storage [269] are widely agreed upon, those of PaaS offerings are less well described by current standards and lack a common terminology or model [21, 217, 302, 349]. Therefore, it is necessary to define a common reference model of PaaS [30, 217]. The model is the foundation for the extraction of capabilities and components of PaaS providers that are utilized to realize the ecosystem portability approach of the previous section. We define such a model of current PaaS offerings in Figure 4.5. 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 the art in enough detail [16]. The presented model is based on and validated by the findings of our initial analysis of 68 PaaS offerings back in 2013 and the ongoing work with the knowledge base over the last years. Moreover, we aligned our model with related work on models and taxonomies from [4, 10, 29, 86, 132, 139, 148, 151, 218, 223, 233, 234, 247, 291, 296, 333, 352]. 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. We further validate this assertion in Section 4.4.2.

In our notion, a PaaS system can be divided into three layers: *infrastructure*, *platform*, and *management*. Every PaaS system is a certain abstraction and subset of the presented model. In the following subsections, we describe and explain the parts of our PaaS model in more detail.

4.3.1. 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 abstract instance concepts that constitute specific virtual resource configurations 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

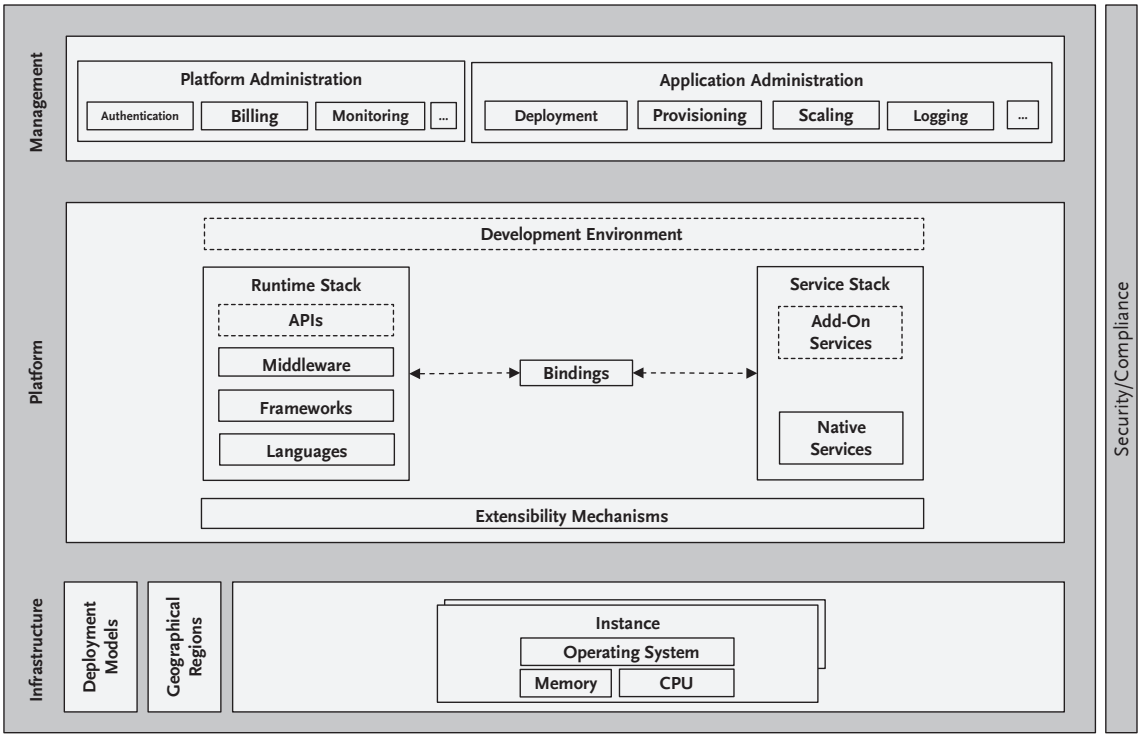


Figure 4.5.: Platform as a Service Model

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.

PaaS is popular in different deployment models [237]. Public PaaS are hosted over the Internet accessible for a vast amount of different customers. A lot of public PaaS providers tend to use existing IaaS offerings like AWS for their 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 CF 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.

In this context, another important factor is the geographical region the application will be deployed in. This is particularly interesting because of legal and performance reasons [75, 331]. As bandwidth capacities keep increasing on the customers' end, latency is one of the main constraining factors for publicly hosted applications [127]. 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. In addition, the distribution of applications across several regions is a factor for reliability and accessibility for customers. Therefore, it is beneficial that a PaaS provider offers several deployment regions. This is an essential feature for companies serving a particular region or willing to expand to different regions. Even more important than performance 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 [99]. 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 the EU-US Privacy Shield[96] or a jurisdiction based in the EU. This is a crucial aspect for cloud providers in general and even more present since the disclosures of the PRISM surveillance program[97] in mid 2013. Moreover, there are other regulations that limit cloud adoption for certain businesses, e.g., the Health Insurance Portability and Accountability Act (HIPAA) or Sarbanes-Oxley compliance that must be provided for corporate data to be moved to the cloud [167]. Although some providers are explicitly EU-based and advertised as EU-compliant, the majority of providers are just starting to address these issues.

4.3.2. Platform

The platform is the main deliverable of a PaaS offering and includes the application hosting environment delivered as a service. Two stacks of software components are decisive: The *runtime stack* and the *service stack*. Both stacks can be combined by the customers via bindings. Those bindings are generally implemented via 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 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.

[96] https://www.privacyshield.gov
[97] https://en.wikipedia.org/wiki/PRISM_(surveillance_program)

The services stack is divided into native and add-on services. Native services are hosted and operated by the PaaS provider, typically co-located with the PaaS environment inside the same infrastructure. These services include mainly latency and performance critical core services like data stores. Add-on services are supplied by third-party service providers that are integrated with the PaaS. They include both competing (e.g. data stores) and complementary services like analytics, search engines, messaging services, and many other utilities. The ability to create a large ecosystem of partners is a huge benefit of current PaaS offerings. These services can improve the customer's ability to deliver applications along with cross-selling opportunities for the vendors.[98] 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 other infrastructures that may even be geographically different from the PaaS. This must be taken into account when performance critical operations are involved.

Another key feature of a modern PaaS is extensibility of the fundamental OS image. Originally developed by Heroku, buildpacks[99] are a collection of scripts that define a generic API for detecting, compiling, and releasing runtime languages, frameworks, or services. Buildpacks enable developers to create customized application environments inside the PaaS environment. The mechanism allows them to create or enhance existing environments with custom runtimes or services. Theoretically, buildpacks can bootstrap software products from both the service or runtime stack. As of scalability issues with services like data stores (i.e. necessary data replication), this is typically more reasonable for parts of the runtime stack. Other vendors have either adopted Heroku's buildpack concept or defined their own extensibility mechanisms like OpenShift cartridges[100]. Buildpacks can be manually created by the developer but are most often created and shared within the community. Next, the rise of Docker with its portable packaging format had significant impact and application for the extensibility of the systems. Also, the isolation of applications via containers perfectly fits within most PaaS architectures. This capability gives the developers greater freedom and possibilities to extend the system, blurring the differentiation to IaaS.

Depending on the abstraction of the PaaS system, a development environment based on the underlying platform stacks may be provided. Generic and IaaS-centric offerings typically focus on the operational aspect and omit the inclusion of a custom-made development environment. Here, the assumption is that the development is better done locally with familiar tooling, as the runtime environment reflects standard application development scenarios [32].

[98] https://www.infoworld.com/article/2611149/paas/forrester--paas-makes-developers-happy.html
[99] https://buildpacks.io
[100] https://docs.openshift.com/enterprise/3.0/whats_new/carts_vs_images.html

4.3.3. Management

A management layer encompasses the two previously described layers that allows controlling the deployed applications and the configuration settings of the platform. The management layer includes the abilities to deploy and manage the life cycle of the applications. This comprises, among others, pushing, starting, and stopping of applications. Moreover, the provisioning of all native services and add-ons is initiated from the management layer. Also, all available configuration and administration settings for the applications and the PaaS environment can be controlled. This includes a wide range of functionality like scaling, logging, 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 RESTful API, console-based, or driven via web UIs. Although the mentioned functionalities are shared by different PaaS to a great extent, procedures and commands are not standardized and differ widely between providers. We analyze and discuss this and the full range of typical management operations in more detail in Chapter 7.

4.4. Platform as a Service Profiles

Semantic technologies such as ontologies and taxonomies are regarded as one of the most efficient solutions for the classification, normalization, and connection of domain knowledge [349]. To supply a central marketplace and compare existing PaaS providers, we first need to agree on an appropriate conceptualization for PaaS to index offerings.

Ontologies are a means to formally model the structure of a system, i.e., the relevant entities and relations that emerge from its observation which are useful to the intended purposes [343]. Gruber [130] defines the notion of an ontology as an "explicit specification of a conceptualization." Borst [44] defines it as a "formal specification of a shared conceptualization," which adds the properties formal and shared to the definition. Expecting a shared view between several parties rather than an individual view is important for a common understanding of an ontology between different stakeholders. Formal means that the conceptualization needs to be in a (formal) machine-readable format, to allow processing and reasoning [343]. Different languages for formalization are possible from rather informal term lists or glossaries to first-order logic [343, 362]. Struder [345] merged these two definitions into "an ontology is a formal, explicit specification of a shared conceptualization." Overall, an ontology is a logical theory to capture the intended models corresponding to a certain conceptualization [343]. As every knowledge base is committed to some

conceptualization, it is beneficial for our approach to capture it with semantic technology [115].

Instead of a pure ontology, we want to use a taxonomy instead. At first glance, both terms are hardly distinguishable, and they are also often used interchangeably in literature. Even in academia, there is no agreement on how to differentiate the terms and concepts. In general, taxonomies are a simpler version of ontologies with a more limited scope on the conceptualization and mapping of the relationships, which are mostly only hierarchical. According to Gruber [131], taxonomies are "hierarchies of classes, class definitions, and the subsumption relation, but ontologies need not be limited to these forms." Following Gruber's definition, taxonomies are made for knowledge classification whereas ontologies can go far beyond this simple task. Ontologies may even consist of multiple taxonomies. In computer-science, taxonomies are often used to describe trees of generalization-specialization relations. We use the word taxonomy, as it fits to describe the simplified nature of the relationships we use in our PaaS taxonomy, i.e., mostly generalization-specialization between the different properties of the profile. Also, the taxonomy does not use named relationships but limits itself to a classification framework. This helps us to keep the conceptualization easy to understand and use, and limited to the absolute necessities.

The foundation for the creation of the taxonomy is our PaaS model (see Figure 4.5) introduced in Section 4.2. As outlined in Section 4.2.1, interoperability of the Self-Service Management API can be achieved independently of the functional interface [152]. The management tier includes the Self-Service Management API functionality whereas the functional interface mainly corresponds to the platform tier. Therefore, we solely address the portability of the functional interface with our profiles. This especially includes the attributes of the platform and infrastructure tiers. The management tier is explicitly addressed in Chapter 7.

The taxonomy and its attributes are the foundation for the structured data set of the knowledge base. As every model and therefore every knowledge base is only an abstraction of reality and limited to a specific point of view, the set of attributes of a derived taxonomy differs [344]. In our case, the selection and definition of relevant attributes is defined by a notion of application portability based on the technological ecosystem of the providers. Thus, the taxonomy's attributes are focused on the available technological ecosystem that is vital for assessing application portability between providers. Other properties like specific business-related QoS or cost details are omitted to some extent. Nevertheless, the specification can be extended by any type of qualitative or quantitative attributes.

As a next step, we transform the PaaS model into a concrete taxonomy describing essential parts of a PaaS. Figure 4.6 shows the extracted taxonomy on which the profiles are based. Capabilities belonging to either the business or

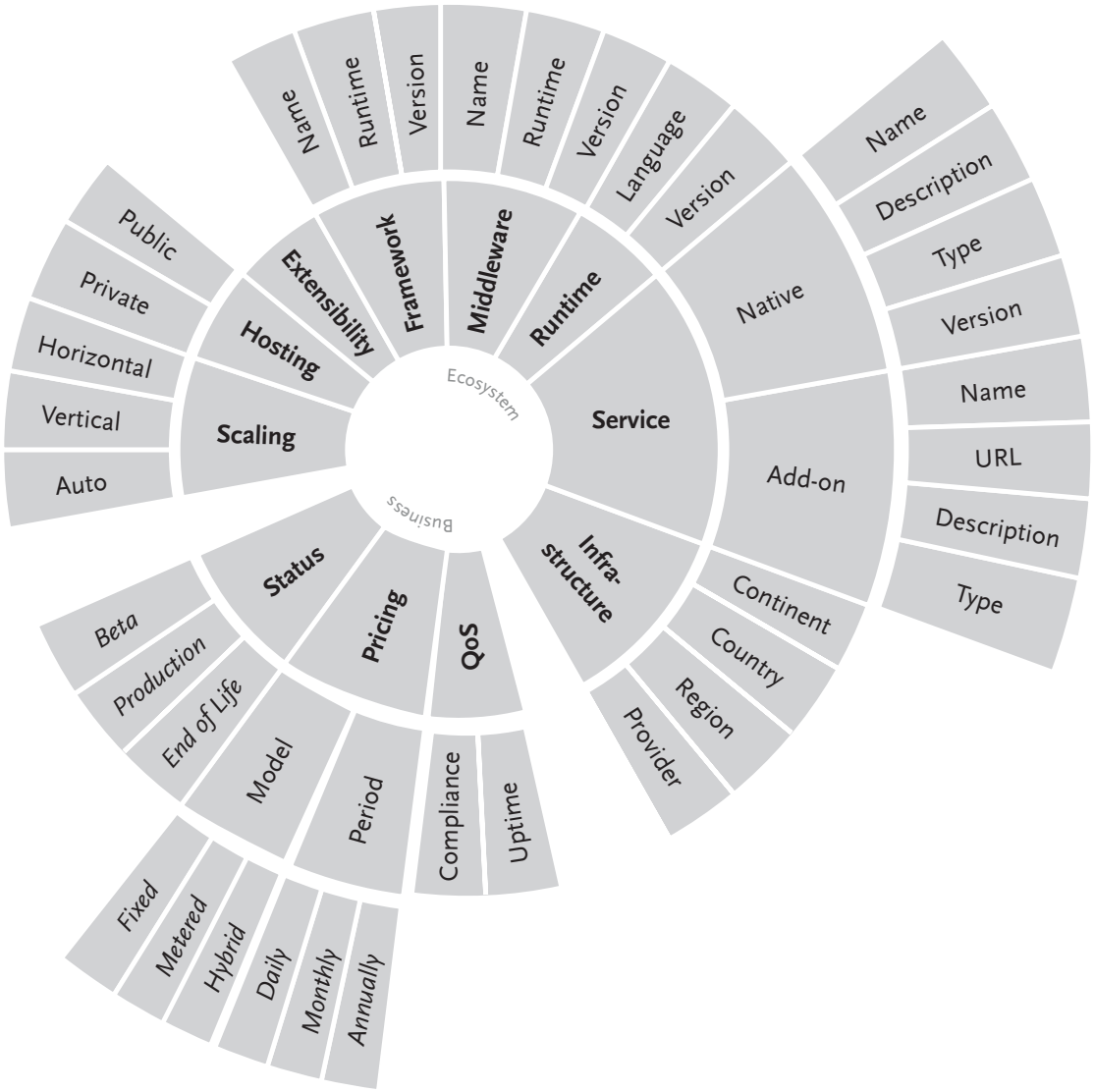


Figure 4.6.: Platform as a Service Taxonomy

ecosystem perspective are visually clustered. The taxonomy depicts a restricted set of properties that are present for the majority of PaaS offerings, 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 tried to restrict the maximum depth and width of the taxonomy to a reasonable amount, as more concepts, object properties, and relationships are antagonistic to the usability of a conceptualization [4]. 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. Table 4.1 shows the coverage of our properties

Table 4.1.: Model Coverage by Literature

Property	Used by Related Work	Total Coverage
status	[218]	8 %
pricing	[4, 10, 29, 86, 132, 148, 151, 218, 223, 291, 352]	92 %
compliance	[10, 29, 86, 132, 139, 151, 218, 247, 333]	75 %
uptime	[4, 10, 29, 86, 132, 148, 151, 218, 223, 247, 291, 352]	100 %
scaling	[29, 132, 139, 148, 218, 247, 291, 352]	73 %
hosting	[4, 86, 132, 139, 247, 352]	55 %
infrastructures	[29, 139, 148, 218, 291, 296, 333]	64 %
runtimes	[4, 29, 139, 148, 151, 218, 247, 291, 296, 352]	83 %
middleware	[10, 151, 218, 291, 296]	42 %
frameworks	[29, 151, 247]	25 %
services	[10, 29, 151, 218, 223, 233, 234, 296, 333]	75 %
add-ons		0 %
extensibility		0 %

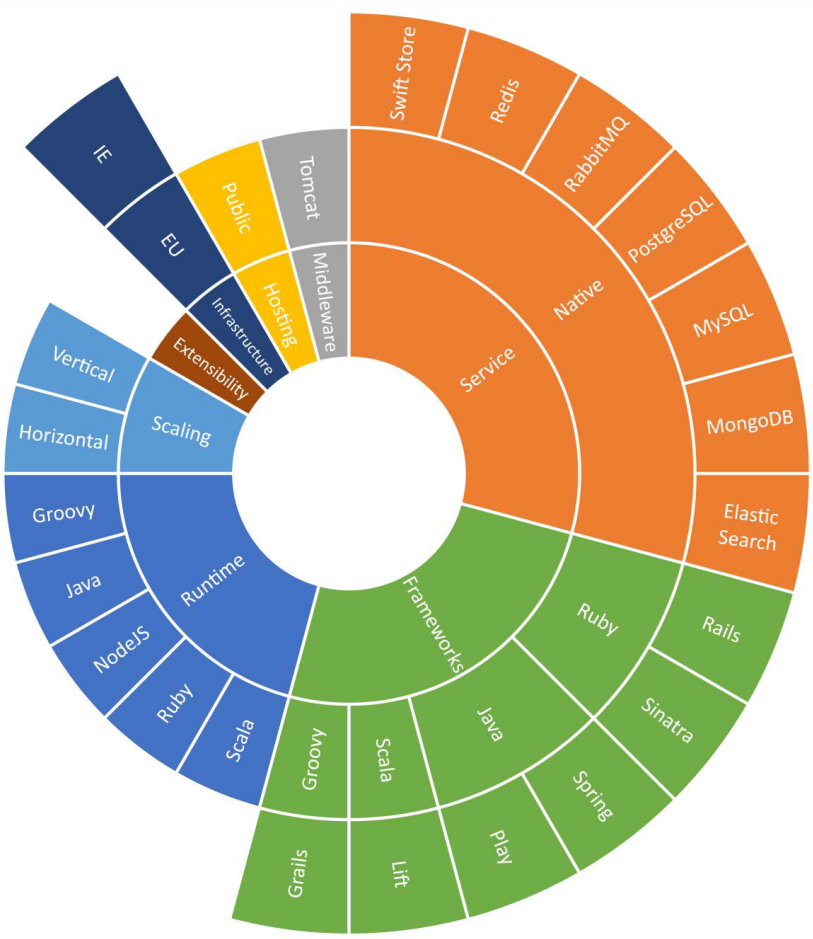


Figure 4.7.: PaaS Taxonomy Instance for Anynines

by related literature. It reveals that the sets of attributes are differing between the approaches, but also shows that the overall set of our selection is seen as relevant by a large amount of related approaches.

Figure 4.7 shows a specific instance of the generic taxonomy for the PaaS provider Anynines.

4.4.1. Profile Specification

To make PaaS offerings comparable and matchable, we transform the abstract taxonomy into a standardized, machine-readable PaaS profile[101].

Whereas the profile's properties could be specified in any markup language, we deliberately choose JSON. 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.

There is a multitude of service description approaches especially in the field of semantic web services [121, 346]. Besides traditional syntax ontology languages like CycL [205] or F-Logic [186], several markup ontology languages emerged out of the semantic web scene. Examples are XOL [177], OML [180], SHOE [146, 221], and most importantly the Web Ontology Language (OWL) [369, 371] which is based on the Resource Description Framework (RDF) [373]. These languages use a markup scheme to encode knowledge, most commonly XML. OWL can be used in combination with SPARQL [372], an RDF query language, that makes it easy to retrieve and manipulate data stored in the RDF format. Some of these semantic technologies are also supported by tools like semantic editors, e.g., Protégé[102] and semantic reasoners such as RacerPro [135]. Nevertheless, like Slawik et al. [335, 336], we decided against these toolsets in favor of a more basic solution. First of all, the concepts and tools are little known to our target group which are developers, software architects, and even business personal. These technologies also have their own complexity and expressiveness, which go beyond our requirements. Therefore, we tried to focus on a simplistic approach and tooling with a low entry barrier that is well-known to our target group. Exactly this is provided by JSON in combination with web technologies for reasoning capabilities.

Listing 4.1 shows an exemplary PaaS profile definition. 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 to achieve an intuitive profile creation process. In the following, we describe the profile specification in detail.

Meta Information

Besides the main concepts from the PaaS taxonomy, we introduce additional meta information to the profiles. To verify and keep track of the profile itself, it includes the properties REVISION and VENDOR_VERIFIED. The property REVISION

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

[102] `https://protege.stanford.edu`

Listing 4.1.: Exemplary PaaS Profile

```
{
  "name": "PaaSfinder",
  "revision": "2018-08-29",
  "vendor_verified": "2018-08-15",
  "url": "https://PaaSfinder.org",
  "status": "production",
  "status_since": "2013-08-01",
  "pricing": [
    {
      "model": "fixed", "period": "monthly"
    }
  ],
  "qos": {
    "uptime": 99.8,
    "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": [ "1.8", "1.9" ]
    }
  ],
  "middleware": [
    {
      "name": "tomcat", "runtime": "java", "versions": [ "8.5" ]
    }
  ],
  "frameworks": [
    {
      "name": "spring", "runtime": "java", "versions": [ "5.*" ]
    }
  ],
  "extensible": false,
  "services": {
    "native": [
      {
        "name": "mongodb",
        "description": "Document database", "type": "datastore",
        "versions": [ "3.6" ]
      }
    ],
    "addon": [
      {
        "name": "mlab", "url": "https://www.mlab.com",
        "description": "MongoDB as a Service", "type": "datastore"
      }
    ]
  }
}
```

dates the last change of the profile. The property VENDOR_VERIFIED denotes if and when the profile was last verified and officially audited by the vendor itself. This should indicate additional certainty of data validity to the users and similar concepts were used before for indicating the authenticity of profiles, e.g., by Twitter[103] or Facebook[104].

Business Properties

The business properties either describe the PaaS offering as a whole or are related to the business perspective of the PaaS. This includes the official NAME of the PaaS and the URL leading to the offering's web page. 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. In the beginning, there were a lot of offerings that only started their business, so for customers it was especially important to separate them from established providers. The value serves as an indicator for the provider's business stability [75]. It consists of the following life cycle stages: *beta* if the PaaS is in private or public beta testing, *production* when it is live and generally available, and End of Life (EOL) if it is discontinued or integrated into another offering [79]. Moreover, the profile includes available PRICING options. In contrast to IaaS, no uniform price structure has yet been established in the PaaS market. A multitude of different parameters need to be considered when calculating prices among providers. Due to the complexity of available price calculations among providers, the profile is limited to the pricing models and does not include exact prices, yet. Overall, pricing is defined by a pricing MODEL and the billing PERIOD. All billing options are subscription-based. The billing model can either be *free*, *fixed*, *metered*, or *hybrid* [7, 84, 250, 272]. A provider with a free plan offers a certain amount of resources to customers at no cost. Fixed billing describes a one-off fee that is payed for a certain amount of resources (excluding additional resources after a threshold) during a certain period [7]. Metered pricing is based on a sole consumption paradigm, e.g., instance hours [7, 84]. 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 [7]. The billing periods are typically *daily*, *monthly*, or *annually*. As described in Section 4.2, COMPLIANCE with certain security standards or laws is crucial for enterprises. The profile includes an array of certified standards that are fulfilled by the PaaS. Moreover, if applicable SLAs in the form of UPTIME guarantees can be denoted via the provided schema. In the future more business-relevant KPIs from, e.g., the SMI [331] could be added.

[103] https://help.twitter.com/en/managing-your-account/about-twitter-verified-accounts

[104] https://www.facebook.com/help/196050490547892

Ecosystem Properties

The essential capabilities and software components of the PaaS system that are relevant for the application and our ecosystem portability approach are encoded in the main part of the profile.

A major benefit and integral characteristic of cloud environments is their rapid elasticity, in other words SCALING of the application resources [237]. 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 (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 [237]. Nevertheless, values are limited to PUBLIC and PRIVATE clouds. A community cloud is just another form of privately managed cloud. Additionally, we allow a property VPC for virtual private cloud which is a deployment by a public cloud provider but physically isolated from other customer deployments through a virtual network, i.e., a private PaaS instance inside a public cloud [384]. 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, COUNTRY, and the REGION where the data center is located in and an optional PROVIDER field. The continent must be encoded with one of six continent codes for Africa, Asia, Europe, North America, South America, and Oceania. Also, the country codes must conform to the two-letter codes defined by ISO 3166-1 [156]. 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 service used by the PaaS vendor, e.g., AWS. When we created the knowledge base, this critical information was particularly difficult to obtain from the providers.

The four main components of 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 provider. Languages that may be added manually and are not officially supported must not be added, since extensibility is explicitly modeled. 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 names to enable exact matching between offerings. 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 to mark all versions as supported (e.g. `2.*`).

MIDDLEWARE includes an array of preconfigured middleware stacks. These are identified by their official NAME. Similar to runtimes, the field MIDDLEWARE includes a version array that indicates supported versions. To associate the middleware products to the correct runtime, they have an additional RUNTIME field that ties them to the runtime they are used in. For middleware products, e.g., web servers such as nginx[105] that only rely on the operating system this field can be omitted.

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. This can relieve developers from expecting them to be natively available in the PaaS.

SERVICES are divided into NATIVE and ADD-ON services (see Section 4.2). Native services have a NAME that identifies them. Moreover, they are classified by a TYPE field that assigns a category to them 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 reasons of compatibility. 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 applications. These add-ons are consumed as a service and are independent of 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, an URL property references the add-on provider's web page.

Finally, the property EXTENSIBLE indicates if the PaaS supports any mechanism like buildpacks to add custom components to any of the runtime or service stacks.

4.4.2. Data Evaluation

One of the major aspects for the validation of the presented approach is the real-world applicability of the profiles. They should mitigate the current problem of different providers with diversified capabilities and the incomprehensible status quo.

A major challenge for knowledge data in general is to keep it accurate and up-to-date. PaaS offerings are changing at a fast pace. Market overviews

[105] `https://www.nginx.com`

Table 4.2.: Model Coverage by PaaS Profile Data

Property	Used by Number of Profiles	Total Coverage
name	71	100 %
revision	71	100 %
vendor verified	28	39 %
url	71	100 %
status	71	100 %
pricing	67	94 %
uptime	6	8 %
compliance	3	4 %
scaling	71	100 %
hosting	71	100 %
infrastructures	53	75 %
runtimes	71	100 %
middleware	34	48 %
frameworks	45	63 %
native services	53	75 %
add-ons	15	21 %
extensible	71	100 %
$\sum$		72 %

as provided by market researchers are likely to be already outdated when they get published. Even the documentation of the providers is sometimes lagging behind. Related works never really tackled or mentioned any of those problems in their approaches. Moreover, all identified studies are limited to a relatively small amount of providers compared to the actual variety on the market [93, 252, 291, 308, 310, 317]. Apart from the aforementioned studies, the web provides a lot of small comparisons between PaaS that are mainly limited to a handful of providers and a subjective set of properties. In general, actuality and verification is a major drawback of most studies.

We tackle these problems with several ideas. First of all, the profiles are open source and can be collectively updated and revised. Another measure is that providers can add themselves to the shelf, driven by the fact that they want to become known to the customers. We take this fact into account by including vendor-verified profiles. As another quality assessment step, all profiles are automatically tested for syntactic validity and conformance to the current specification before they are released to the public. Moreover, the profiles and the web interface are continuously updated. If a profile gets updated, it is immediately deployed to production. To our knowledge, our data set of 71 profiles is the most recent and most comprehensive publicly available collection of PaaS providers.

By following the dimensions and components of our model and taxonomy, we also try to solve semantic conflicts between offerings by providing a common set of capabilities. To prove the point by numbers, we evaluated the overall coverage of the defined attributes by the existing provider profiles. Table 4.2 shows the numbers on August 10th, 2018.

Whereas the size of our taxonomy and profile might look small, we tried to limit it to the most important properties that can also be fulfilled for current providers. Overall, we restrict the maximum depth and width of the taxonomy to a reasonable amount, as more concepts, object properties, and relationships can contrast the usability of a conceptualization [4]. Related works often define fine-grained trees of attributes that cannot be covered with actual data values for the state of the art. In general, we can confirm that we achieve a high coverage of all properties in our data (72 % coverage overall), which confirms the suitability of the attribute selection in our model and taxonomy. For values with low coverage rates, there are reasons that can explain why the values are low. For the attributes *vendor verified* and *add-ons*, it is expected that only a portion of providers support a service marketplace or get in touch with us for profile verification. Features higher up in the runtime stack such as middleware and frameworks are more specific information that often cannot be enumerated in their entirety. Therefore, we observed the tendency that this information is not given at all by data suppliers. The same rationale applies to subattributes such as software versions. The table also helps us to prove some points which we criticize in related works. Many works conducted comparisons and selections on nonfunctional characteristics like QoS and SLAs [6, 28, 247, 351]. This is a popular set of attributes that is applied to evaluate cloud offerings and mentioned in nearly all the related works [94]. However, as we can see in our data, there are very little providers that give such guarantees like service response times [28, 247, 351], availability [28, 247], or compliance certifications [28]. The majority of providers work on best effort delivery[106] and compliances are mostly only available for the underlying infrastructure that is provided by IaaS providers, such as AWS, but not for the PaaS offering itself.

Overall, the identified problems further motivate our planned enhancements to the data governance and the selection process of the knowledge base in Chapter 5.

4.5. PaaSfinder Web Application

Although the presented PaaS profiles include all the necessary raw data for supporting users with the evaluation and overview of the PaaS market, the representation is not suitable for end users. Besides validating the model-data fit of the raw data (see Section 4.4.2), a validation by the practical usage of end users is indispensable for the evaluation. It was shown that an appropriate aggregation and visualization of data is necessary for users to understand complex information [390]. This helps users to make an informed decision

[106] For example, see Heroku's customer promises at `https://www.heroku.com/policy/promise`.

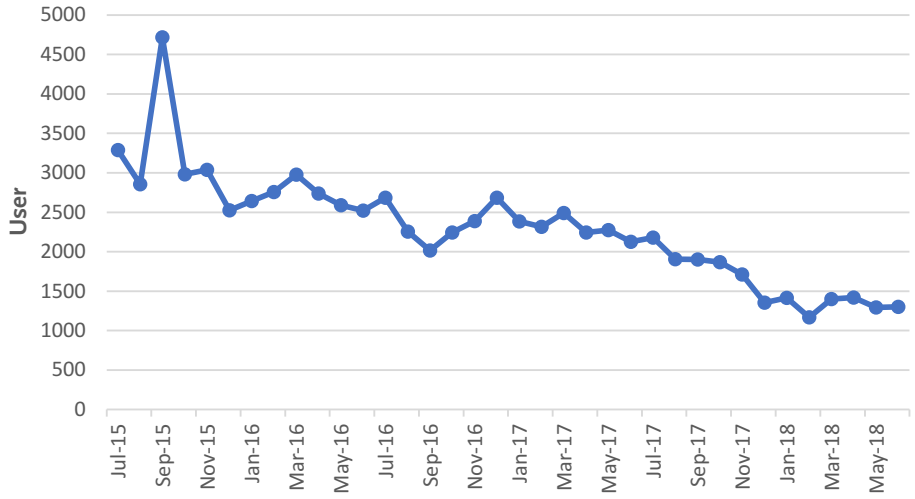


Figure 4.8.: PaaSfinder Web Analytics

with respect to their requirements. In that regard, we think that a provider directory in form of a web application is suitable for presenting the data and to support selection decisions. The focus of this user interface is to enable end users to access, analyze, and use the data conveniently. Therefore, we implemented a web application[107] that is capable of displaying the profiles. The web application is a fundamental part of the evaluation of the PaaS model in practice. It validates the profiles with real users. Figure 4.8 shows the unique users per month from July 2015 to June 2018.

Around 2300 users were registered every month. This shows that the data is used by a substantial amount of people over the last years. The website has also established itself in the top five Google search positions for relevant search queries such as "PaaS comparison" (see Table 4.3). The table[108] also shows that the page is frequently used as resource for comparisons of popular PaaS providers such as Heroku and OpenShift. For organic search results, the Click-Through Rate (CTR) (clicks/impressions) approximately follows Zipf's law (CTR $\approx 1/\text{position}$) [2]. In practice, however, we see lower CTRs and more distribution especially for the first positions. The CTR for the first search result position in Google search is around 36 % in April 2018.[109] The CTR is also influenced by the relative relevance of the presented search result to the user, although factors like search rank influence the decision [63, 169, 306]. Consequently, our relatively high CTR indicates that the users find our offering relevant to their search queries, intentions, and information demands. Table 4.3 shows that the average position in the search results for PaaSfinder is 29.4 (3.7 % CTR). When looking at queries that include the term "PaaS," we rank

[107] The project homepage is `https://github.com/stefan-kolb/paas-profiles`. An online version of the web interface can be found at `https://paasfinder.org`.
[108] Data of the last 90 days before 5/22/2018.
[109] `https://www.advancedwebranking.com/cloud/ctrstudy`

Table 4.3.: PaaSfinder Google Search Top Queries (90 Days)

Search Query	Clicks	Impressions	CTR	Position
heroku vs openshift	170	584	29.11 %	1.7
openshift vs heroku	152	534	28.46 %	1.8
paas comparison	139	327	42.51 %	1.2
free paas	58	917	6.32 %	4.8
paas providers	55	2,773	1.98 %	5.8
paasfinder	51	63	80.95 %	1.0
paas providers comparison	49	166	29.52 %	1.4
paas free	45	367	12.26 %	4.4
php paas	38	363	10.47 %	4.7
openshift heroku	33	191	17.28 %	1.4
heroku vs cloud foundry	31	254	12.2 %	3.8
paas provider	31	676	4.59 %	4.3
heroku vs pivotal	28	65	43.08 %	1.2
platform as a service providers	27	1,165	2.32 %	6.4
paas php	26	186	13.98 %	3.2
cloud foundry vs openshift	26	931	2.79 %	9.1
paas hosting	22	297	7.41 %	6.0
cheapest paas	22	94	23.4 %	2.4
heroku openshift	20	82	24.39 %	1.4
cheap paas	20	50	40 %	1.8
heroku vs bluemix	20	49	40.82 %	1.1
bluemix heroku	20	129	15.5 %	2.9
pivotal vs heroku	19	47	40.43 %	1.2
paas comparison matrix	19	86	22.09 %	1.1
free cloud paas	18	90	20 %	2.9
$\sum$	3,572	96,652	3.7 %	29.4
$\sum$(“paas”)	914	19,392	4.71 %	16.5
$\sum$(“paas + comparison”)	188	494	38.06 %	1.2

16.5 on average with a CTR of 4.71 %. For the primary aim of the application, the combination of the terms “PaaS” and “comparison,” we rank position 1.2 on average with a CTR of 38.06 %. Additionally, we received numerous comments by users and companies via email and social media that also confirm that the data properties and quality are considered as beneficial by the users. Throughout the project, 34 of 71 providers contacted us at least once to verify their profile.

4.5.1. Application Design

At first, the application is based on a web directory which is subsequently extended with search features and other functionality in the course of the thesis. The data for the web application comes directly from our crowd-sourced cloud service registry [336]. We intend to collect, summarize, and present the most important information to users for information or decision processes, similar to a dashboard [101].

Therefore, different user stories were collected before the implementation of the application. User stories [321] were popularized by the agile software method Scrum [311]. In our case, there are three different stakeholders of the application: *user*, *provider*, and *developer*. A user is anyone who wants to use or

find information about PaaS systems and accesses the web application to gather information through the browser of any device such as a smartphone, tablet, or computer. The user is looking for information on the available PaaS that can fulfill the needs of a particular application that he wants to deploy. The user may also want to compare competing products to make an informed decision. The provider wants to make his product known to users and compare his offering with the competition. The developer is the creator and administrator of the web application and has different requirements for the design of the system.

In our research group, we have collected the most important Functional Requirements (FRs) and Nonfunctional Requirements (NFRs) for the PaaS directory in advance. The FRs are split up into FRs for users and providers. Both stakeholders might have similar requirements which are phrased differently because of their perspective (see *FR.User.1/FR.Provider.3*, *FR.User.3/FR.Provider.4*, and *FR.User.4/FR.Provider.5*). Both approach equal requirements from different perspectives and aims.

FR.User.1 – compare all aggregated: As a user, **I want to** see aggregated information on all providers in one place **so that I can** quickly gain an overview of all providers as an entry point for further investigations.

FR.User.2 – get updates: As a user, **I want to** get updates if new providers are listed or existing providers change their capabilities **so that I can** reassess my requirements and stay updated on the state of the art.

FR.User.3 – get details: As a user, **I want to** see an aggregated overview of information for a particular provider in one place **so that I can** quickly gain an overview of the capabilities of the provider to better assess requirements and support the decision-making.

FR.User.4 – compare to one another: As a user, **I want to** be able to compare the capabilities of providers with another side-by-side **so that I can** quickly gain an overview of how providers intersect and how they differ from one another for supporting selection decisions.

FR.Provider.1 – add a profile: As a provider, **I want to** be able to add my offering to the catalog of PaaS providers **so that I can** make my offering known to the users of the directory and the corpus of knowledge and improve my user base and recognition.

FR.Provider.2 – update a profile: As a provider, **I want to** be able to update my profile **so that I can** keep my information updated to inform the users about the current capabilities of the system and to be listed correctly on searches.

FR.Provider.3 – compare all aggregated: As a provider, **I want to** see aggregated information on all providers in one place **so that I can**

quickly gain an overview of all competing providers as an entry point for market analysis.

FR.Provider.4 – get details: As a provider, **I want to** see an aggregated overview of information for a particular provider in one place **so that I can** quickly gain an overview of the capabilities of the provider to better assess the capabilities of the competing provider.

FR.Provider.5 – compare to one another: As a provider, **I want to** be able to compare the capabilities of providers with another side-by-side **so that I can** quickly gain an overview of how competing providers intersect and how they differ from my offering to support decisions for enhancing the own offering.

NFR.1 – hosting: As a developer, **I want to** avoid any server administration and installation of the runtime environment **so that I can** focus on the development of the application and the knowledge base.

NFR.2 – cloud native: As a developer, **I want to** implement the application as a cloud-native application **so that I can** get experience with the paradigm and use the application in case studies to evaluate different cloud offerings.

NFR.3 – digital archiving: As a developer, **I want to** be able to digitally archive the data of the application on different points in time **so that I can** analyze the evolution and trends over time.

Figure 4.9 shows the overall system architecture of the web application *PaaSfinder*. The system has to adhere to all requirements listed before. It is bootstrapped out of several services to fulfill all defined FRs and NFRs.

The main system, the web application (see the landing page in Figure 4.10) can be seen on the right of Figure 4.9. The web interface is based on Sinatra[110], a Ruby framework and Domain-Specific Language (DSL) for building web applications [143]. As the JSON-based PaaS profiles can be easily imported and used in a document-oriented database, we choose MongoDB[111] for data persistence. 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[112] which requires Ruby `1.9.3` or `2.0` and a MongoDB version greater than `2.2`. This MongoDB version was not widely accessible as a native service, so we decided to use an add-on service for this purpose, for which we chose mLab[113]. The web interface uses HTML5, CSS, JS, and Embedded Ruby (ERB). It is built upon the Bootstrap framework[114] and several JS libraries. The web interface can be accessed with any modern

[110] `http://www.sinatrarb.com`
[111] `https://www.mongodb.org`
[112] `https://mongoid.github.io/old/en/mongoid/v3/index.html`
[113] `https://mlab.com`
[114] `https://getbootstrap.com`

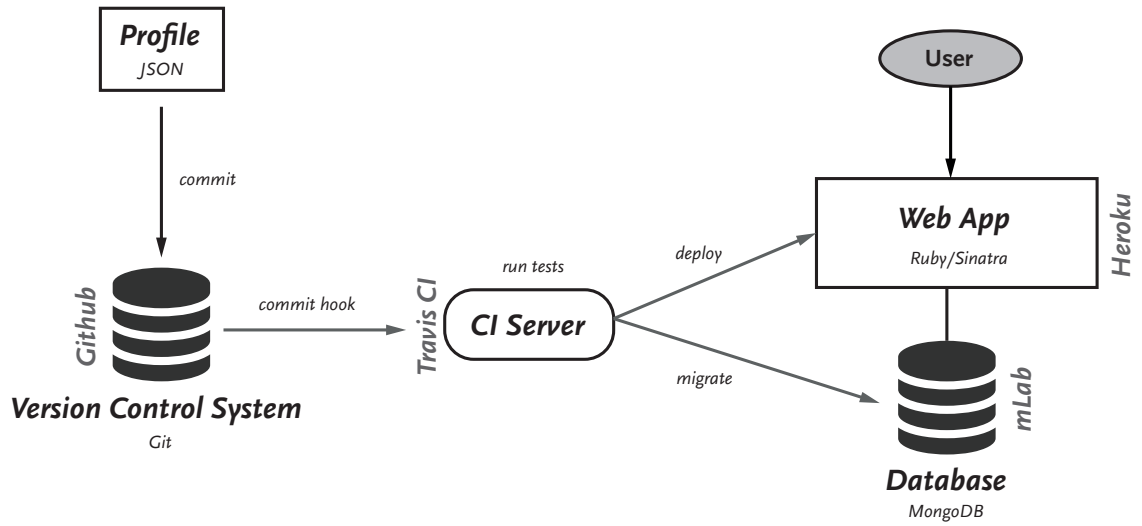


Figure 4.9.: PaaSfinder Architecture

web browser. The application itself is cloud native and adheres to the principles of the twelve-factor app[115] (*NFR.2*). Consequently, it is also hosted by a PaaS provider, in our case Heroku[116] (*NFR.1*). This serves another important aspect of our portability considerations: the validation of the initial portability of applications based on ecosystem capabilities. Therefore, we can use the application in our initial use case to experiment with the migration between PaaS providers to validate our notion of ecosystem portability (see Section 4.2.2), which is described in the following Section 4.5.2 (*NFR.2*).

The data for the web application is saved inside the MongoDB but it only serves as a snapshot of the current status quo. To easily store different snapshots of the state of the art for time analysis, we use the capabilities of Git to implement a digital archive of the provider profiles (*NFR.3*). Every change to a profile is stored as a separate Git commit and can be accessed later on. This allows us to do time series analysis of the provider landscape in the future to show trends and developments in the PaaS market, as done in Section 3.3.

A new Git commit to the master branch of the repository triggers a commit hook that is received by our Continuous Integration (CI) service Travis CI[117]. The CI server receives the application artifacts and runs the test suite to ensure functionality and data consistency. If it succeeds, it deploys the application to Heroku in production. Next, the new profile data is migrated to the MongoDB by executing the database migrations. Changes to the profiles are instantly deployed into the production system so that the data is always up to date.

Every technical requirement and web hosting is provided as a service, which adheres to *NFR.1*. No server setup and maintenance needs to be done by the developer or operator of the service.

[115] `https://12factor.net`
[116] `https://heroku.com`
[117] `https://travis-ci.org`

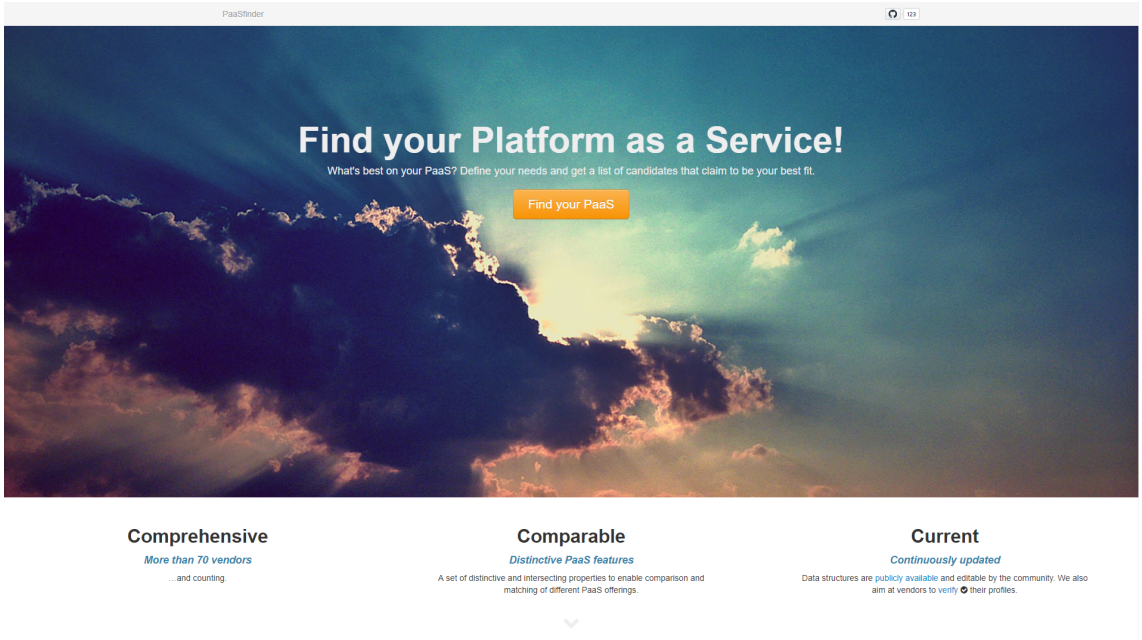


Figure 4.10.: PaaSfinder Landing Page

Name	Status	Runtimes	Scaling	Hosting	Infrastructures	
iKnode	Production	dotnet			NA	Details
Cloudn PaaS	Production	java node php python ruby			AS NA	Details
OpenShift Online	Production	dotnet java node perl php python ruby extensible			EU NA	Details
ElasticBox	Beta	php ruby extensible			AS EU NA OC SA	Details
Dokkur	Production	clojure go java node php python ruby scala extensible			EU NA	Details
Cloudways	Production	php			AS EU NA OC SA	Details
brightbox	Production	ruby			EU	Details
Cloud 66	Production	docker extensible				Details
Force.com	Production	apex			AS EU NA	Details
Cloudify	Production	java php ruby extensible				Details
d2c	Production	go node php python ruby extensible				Details
Getup Cloud	Production	java node perl php python ruby extensible			NA SA	Details
Scalingo	Production	clojure go groovy java node php python ruby scala extensible			EU	Details
MoPaaS	Production	erlang java node php python ruby extensible			AS	Details
Microsoft Azure	Production	dotnet java node php python ruby extensible			AS EU NA OC SA	Details

Figure 4.11.: PaaSfinder Provider Overview (Page Excerpt)

The FRs are all fulfilled by the web interface. We present an overview of all listed PaaS offerings as an entry point (*FR.User.1*). The overview (see Figure 4.11) includes the important high-level characteristics *name*, *status*, *runtimes*, *scaling*, *hosting*, and *infrastructures*. This gives the user the ability to get a quick overview of the available offerings. Starting from there, a user may navigate to a detail page (see Figure 4.12) which shows all information provided by the profile, structured and in parts visually enriched for better

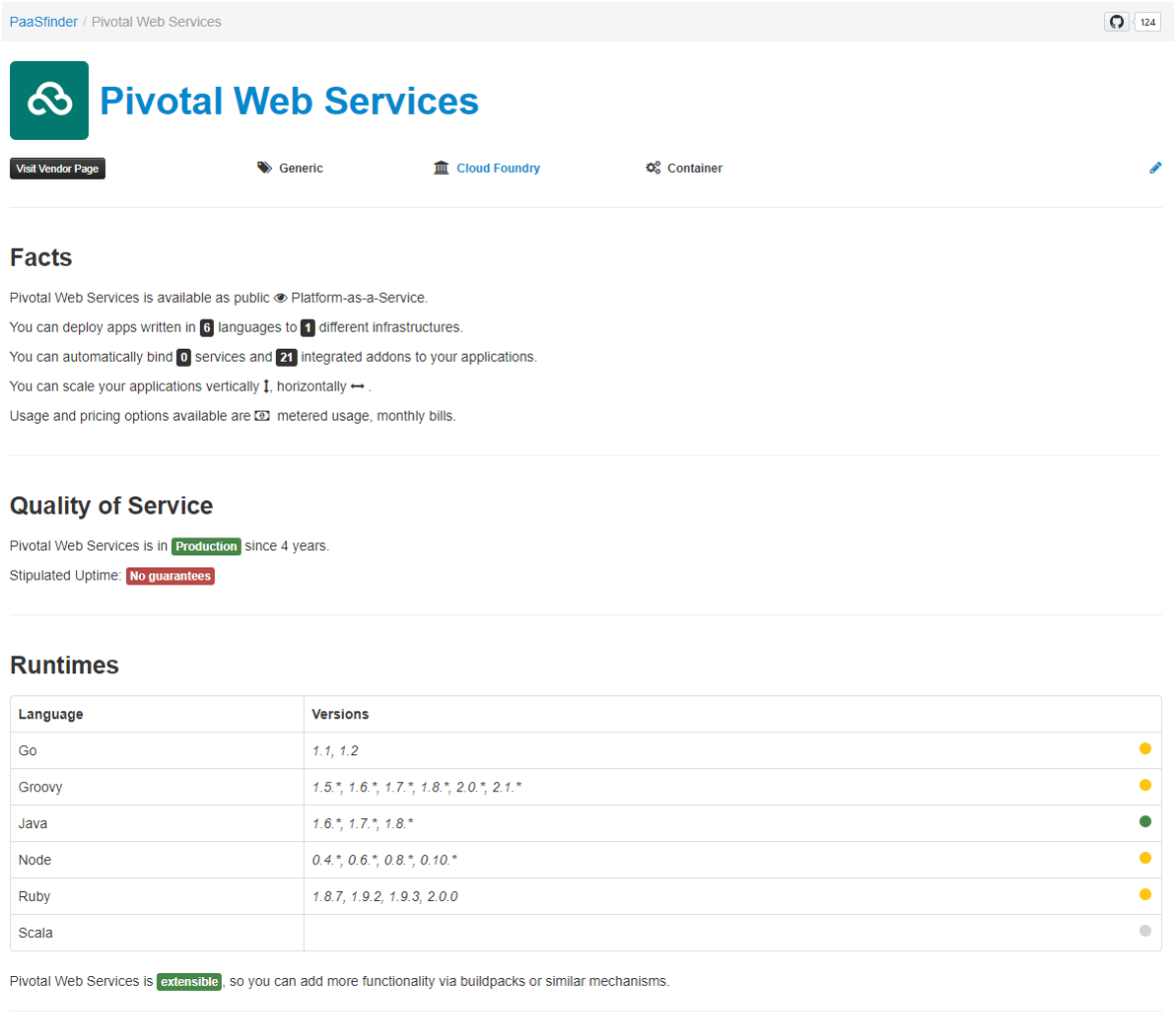


Figure 4.12.: PaaSfinder Provider Details

accessibility (*FR.User.3*). Moreover, it is possible to visually compare (see Figure 4.13) the properties of two offerings directly with each other (*FR.User.4*). Additionally, the profiles can be retrieved via a RESTful API. New offerings and updates to offerings can be received by the users via an RSS feed (*FR.User.2*).

For providers, we implemented functionality to add and update their PaaS profile, either via the Git repository or via a graphical UI (*FR.provider.1*, *FR.provider.2*). The other three FRs can be targeted by functionality that does already serve the user requirements (*FR.User.1*, *FR.User.3*, *FR.User.4*). The provider's information need for an aggregated overview of the provider landscape is realized as for the user through our provider overview page (*FR.Provider.3*). Also, the detail page can be leveraged to show the details for a particular provider (*FR.Provider.4*). The one-on-one comparison of two offerings (see Figure 4.13) enables the provider to either compare its offering to another competing offering or to compare offerings from different competing providers (*FR.Provider.5*). Although some functionality could use some modifications

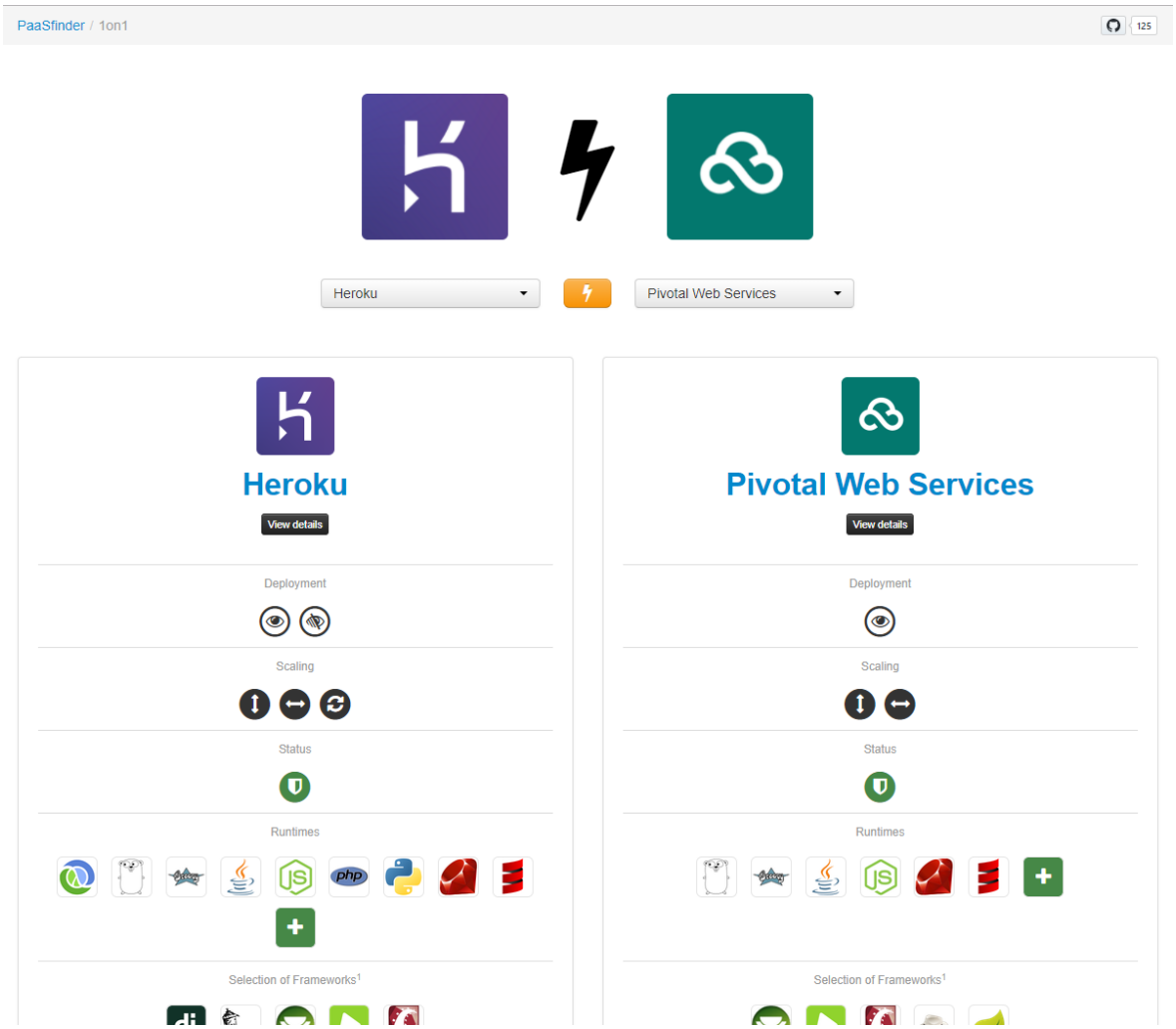


Figure 4.13.: PaaSfinder One-On-One Comparison

for special needs and perspectives of the providers, our focus lies on the user perspective and therefore, we intend to reuse the existing functionalities.

The remarks show that all the initially defined requirements can be fulfilled by our prototype. However, it does not prove how well these are fulfilled. The use of the application by numerous users confirms its usefulness to a certain extent. Nevertheless, to answer questions like the ease of use, an empirical study needs to be carried out.

4.5.2. Validation of Ecosystem Application Portability

As mentioned in the previous section, the web application shall be used for an initial validation of the notion of ecosystem portability on which the profiles are based. In other words: The application should be portable between PaaS providers if they support the defined application requirements. A suitable PaaS for our prototype must allow public deployment, has to be horizontally scalable, support the Ruby runtime in either version `1.9.3` or `2.0`, and the *mLab* add-on.

Based on these requirements, five providers[118] that fit our application profile were identified. These were AppFog[119], CloudFoundry.com, cloudControl, EngineYard[120], and Heroku[121]. All platforms had quite a few contrarieties and similarities which made the comparison interesting. Although AppFog and CloudFoundry.com were based on the CF open source PaaS, AppFog emerged from the `1.*` branch of CF while CloudFoundry.com was already migrated to the new major version `2`. Heroku and cloudControl were independent proprietary systems, but cloudControl reused key parts of Heroku's concepts. Both used buildpacks to install and configure different application dependencies. Furthermore, cloudControl had a similar Git-based deployment workflow as Heroku. EngineYard did not have a counterpart in this set.

While deploying to the different PaaS, we came across several differences that required partially very different deployment workflows and also code changes to get the application running. In this paragraph, we give a nonexhaustive overview of some findings. Any of the PaaS used their own CLI for communicating with the platform. Not all of them allowed a continuous workflow with the CLI but required additional manual deployment steps for certain tasks like add-on provisioning via a web UI. The API methods of the management interface were generally very different for most tasks between the providers, and the APIs in general were far from compatible. Some PaaS required the application artifacts to be in Git revision control to deploy them onto the PaaS. Besides very different deployment workflows of the management interface, we also had to adapt the application artifacts. The recognition of the application type (in this case Ruby with the Sinatra framework) is based on configuration files and code characteristics. Different PaaS were 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 were not available in all PaaS. Once, we had to manually specify the version via a CLI parameter due to an old *Bundler* version running on the PaaS. Some PaaS supported a direct invocation of shell commands in the environment to populate the associated database via *Rake*[122] build commands whereas others required 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 was different between providers which required reprogramming of parts of the application.

As described in the previous paragraph, several changes were required to deploy the application to the providers, but it was possible to run the application

[118] All data and results are latest at the time of the study in April 2014. cloudControl was shutdown due to bankruptcy end of February 2016. CloudFoundry.com is now Pivotal Web Services.

[119] `https://www.ctl.io/appfog`

[120] `https://www.engineyard.com`

[121] `https://www.heroku.com`

[122] `https://ruby.github.io/rake`

on every PaaS. Despite the simplicity of the application, which did not make use of any critical system calls within the environment that might be conflicting, we could validate and conclude initial findings from our research. The results supported our initial hypotheses: We can identify ecosystem portability that allows us to tell if we can run our application on a PaaS from a high-level ecosystem perspective. Furthermore, there is a narrower implementation perspective, which must be investigated independently, that includes various additional requirements and restrictions. As this unstructured study already indicated, there exist portability issues between the providers that needed to be investigated. We also found initial pain points like the management interfaces that needed to be investigated and improved later on. Driven by the findings, we conduct a structured case study on a larger real-world application in Chapter 6 to show measurable evidence for the portability differences.

4.6. Related Work

Portability and Interoperability

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 chapter, capabilities and functionalities of PaaS are fundamentally different from IaaS and therefore need to be assessed separately in terms of standardization. In general, the PaaS service model benefits less from standardization than IaaS [210]. The platform components and capabilities are too different between providers, and too plenty to be standardized [124]. As stated in Section 4.2.1, standardization on the functional interface can also happen on the unit of delivery. TOSCA [259] specifies a portable description for the structure of applications, their component services, and artifacts including management and operational behavior of those. To run these standardized application packages, a TOSCA-compatible runtime environment is necessary on the target cloud [197]. The approach claims to be feasible for IaaS and PaaS environments.

Another approach is to use standardized containers, as defined by Docker and OCI [266, 267], that can encapsulate any payload and will run consistently on virtually any server [339]. Docker was actually extracted from the PaaS system *dotCloud* which was the primary business of the Docker company before their business pivot. Whereas standardized containers have already significantly improved the portability of applications in the cloud, they still do not enable the portability of the application alone. Also, not every PaaS supports the usage of containers as virtualization technology. Next, the packaging, creation, and maintenance of the container is comparable to a lightweight VM. So the users need to take care of the application environment again which takes away one of the key benefits of PaaS.

A related but narrower scoped vendor-driven initiative was the Cloud Foundry Core Definition (CFCD)[123]. It defined a baseline of common capabilities to promote portability between different CF offerings. However, the definition's capabilities were limited to runtime languages and native services. The CFCD defined a set of specific versions of these runtimes and services that developers could use to build portable applications. At the time, five providers were listed as CFCD-compatible. The CF team recognized that application services and runtimes continue to evolve, so they planned to introduce a system of deprecated, current, and next versions. These capabilities could be validated by entering the API endpoint of the CF service.[124] 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-Cloud Foundry PaaS that fulfilled 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. We argue that in our case, generic portability matching has an edge over one definite specification. During the transition from CF v1 to CF v2, the specification was neglected as v2 itself was not compatible with the CFCD. Later, the ideas of the CFCD were rebranded to the *Cloud Foundry Certified Platform* program[125]. Here, all certified offerings are using the same core CF software to establish reliable portability across multiple CF providers. Currently, nine vendors are certified in the annual certification program. Standardization is even more definite than for the CFCD and includes fixed specific releases and components of the Cloud Foundry system, which are not explicitly communicated to the users.

mOSAIC [247, 286] is an academic framework and intermediary layer to ease application portability across cloud platforms. It provides a common set of APIs that are vendor-independent and can be used by the developers to implement platform-neutral applications. At runtime, the mOSAIC platform is responsible for translating the vendor-agnostic calls to the proprietary technology that is used on the target vendor. Apart from the abstraction layer, the project also targets the selection of cloud services for an application.

Silva et al. [333] present a DSL that uses a common cloud vocabulary for describing IaaS cloud services in a way that they can be mapped to each other to

[123] `https://www.cloudfoundry.org/cloud-application-portability-made-easy-introducing-cloud-foundry-core`

[124] Technically, compatibility was validated by calling specific API targets returning the necessary capability descriptions.

[125] `https://www.cloudfoundry.org/certified-platforms`

foster migration between them. They also try to combine their DSL with TOSCA to reduce the workload for creating TOSCA specifications of cloud descriptions.

Apart from the described publications that explicitly work on the portability of applications, there are various approaches that implicitly cover portability. The majority of them work on cloud brokers that target the selection and migration of applications among clouds. The underlying idea is based on a similar notion like ours that tries to match application requirements with platform capabilities. Whereas most of the approaches target ranking and selections based on nonfunctional characteristics such as cost comparisons, a few of them also match technological application requirements. Cloud brokers that target PaaS are proposed, e.g., by Quinton et al. [296], Surajbali and Juan-Verdejo [351], and Goncalves et al. [122]. Also, several EU research projects investigate cloud brokerage such as Cloud4SOA [82], SeaClouds [56], and the PaaSport project [6, 28]. A more detailed overview of cloud brokerage approaches is given in Section 5.5.

Whereas only few efforts are currently developed on the functional interface, it is another matter with the management interface. The management API that controls the application life cycle has widely the same set of functionalities between different PaaS. Several standardization efforts have targeted the interfaces of PaaS cloud offerings, e.g., the Open Cloud Computing Interface (OCCI) [268, 270], Cloud Application Management for Platforms (CAMP) [261], and the DIN SPEC 91337 [88]. Besides those approaches, a set of academic and open source abstraction layer approaches are available [56, 77, 82, 174, 175, 324, 399]. A detailed consideration of the related work for the management interface is done in Section 7.4, which is the respective chapter about management unification.

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. In contrast, our approach is directly applicable to all vendors in practice.

Models and Ontologies

During the creation of our model and the taxonomy, we consulted various related work on cloud models and ontologies. Different from the entities and interfaces of IaaS systems like compute, network, and storage [269], PaaS systems are less well described by current standards and lack a common terminology or model [21, 217, 302, 349]. The main reasons for the development of the PaaS model and taxonomy stem from the fact that existing approaches are often too generic and aim at the whole SPI model, instead of defining each of these models in appropriate detail [16]. The vast majority of papers, e.g.,

[4, 10, 86, 132, 139, 151, 234, 247, 352] provide a generic view of all cloud offerings. Repschlaeger et al. [303] present a well-evaluated classification framework for IaaS. The authors define the target dimensions from a customer perspective, based on expert interviews and a literature review. Afterwards, the relevance and applicability of the resulting framework was evaluated with an additional survey conducted among IT managers and a provider market analysis. Prodan and Ostermann [291], Hilley [148], and Quinton et al. [296] describe both IaaS and PaaS offerings. A model solely for PaaS is discussed by Loutas et al. [218] and Bassiliades et al. [29]. Additionally, Di Martino et al. [233] covers PaaS and SaaS offerings. Moreover, most related work does not explicitly extract the properties for a specific purpose, i.e., application portability or pricing comparison, but rather tries to cover all purposes of cloud provider selection in one model. This leads to incomplete or even arbitrary sets of model attributes. We also claim that these models do not appropriately depict the current state of the art. Next, we argue that the practical applicability of these approaches is not validated in enough detail, as too few actual model instances are created by the researchers. This is particularly evident with ontologies that include a lot of concepts, relationships, and an overall very broad and deep graph structure. This makes it difficult to fill the nodes with information for a larger set of providers with adequate and consistent graph coverage among the ontology instances. For reference, see Table 4.1 again, where we present a detailed comparison of the literature with our taxonomy in its design phase. It reveals that the sets of attributes are diverse between the approaches but also shows that the overall set of our selection is seen as relevant by a large amount of related approaches.

4.7. Summary and Future Work

In this chapter, we presented a new model that describes current PaaS offerings. Next, we deducted and codified a PaaS profile from the model to enable comparison and portability matching based on application dependencies and platform capabilities. We also investigated different portability threats and possible solutions for them from a PaaS point of view. With data from 71 PaaS providers, we offer a comprehensive knowledge base of the provider market. Furthermore, we implemented a web application that allows users to take advantage of the PaaS profiles. Besides giving an overview of available products, it is possible to compare the offerings with one another.

Next, in Chapter 5, the application is extended with filter and selection functionalities, to allow matchmaking by configuring required capabilities for application portability. In that regard, we also reflect upon the data governance of the knowledge base to ensure appropriate data quality.

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 as well as management API differences. These factors have impact on the migration of applications from one cloud to another and also from on premises to cloud environments. Accordingly, we investigate more on both of those adjacent perspectives separately in Chapters 6 and 7.

5. Decision Support for Platform as a Service Selection

Parts of this chapter have been taken from [191, 192].

In this chapter, RQ 3.1 *("How to find matching cloud platforms for particular application and user requirements?")*, RQ 3.2 *("Are there any issues with the accuracy and satisfaction of user queries caused by data and query biases?")*, and RQ 3.3 *("How to semantically enhance matching algorithms to find and rank cloud platforms that only partially match the defined requirements?")* are supported.

5.1. Motivation

Platform as a Service is a major technology to improve development productivity of today's agile development cycles [391]. The managed and highly automated application environments free developers from configuring servers and reduce developer operations and maintenance efforts. As a result, developers can focus on application development, which creates the actual business value. As we showed in the previous chapter, the PaaS market is fragmented and offerings are differing conceptually as well as in their supported technological ecosystem. Therefore, provider selection is an important but currently not well-supported task for companies trying to benefit from the technology. In this context, it is a challenging task to make an informed choice. Besides that, a main issue is to avoid potential vendor lock-in and retain future options for application portability [85, 95, 286, 332]. In contrast to IaaS, decisions on appropriate PaaS providers typically involve more criteria due to their diverse ecosystems. As of now, there are no widely applied standards in the world of PaaS which requires a different approach based on technological components and capabilities to compare offerings. To assist a customer's decision, the need for a consolidated repository evolved [349, 394]. Since existing works regularly lack a decent data set in terms of amount, actuality, and quality, we proposed a new data model and knowledge base for PaaS in the previous chapter. Moreover, existing approaches targeting cloud provider selection often neglect important real-world problems and are based on an optimistic view of the data quality and the users' selection queries. In the following, we show

that these assumptions are often not sufficient for the practical applicability of provider selection and ignore the evident problem of data and query biases. Whereas optimistic assumptions simplify the overall selection problem, this comes at the cost of sacrificing results. Closely associated, most approaches only allow an exact matching of the users' queries with the data which can lead to a substantial amount of unsatisfied queries. However, in certain scenarios there is a chance that partial matches are still able to fit the user's needs, when they are constrained by a set of semantic rules for the specific domain.

Whereas feasible algorithms for selection are discussed fairly often, data and query problems as well as semantic knowledge lack appropriate consideration. The following chapter aims to improve on these shortcomings in general and specifically by an application of the problem in the context of cloud platform selection. Hence, the focus of this chapter is on cloud properties and semantic matching for cloud platforms rather than on generic decision algorithms that are sufficiently discussed in related works such as [219, 360]. We emphasize that existing works on cloud selection are of limited value in practice because they are missing additional validation and semantic enhancements to improve selection accuracy. To that end, we validate our hypotheses in practice via real-world data from our cloud platform knowledge base *PaaSfinder*.

The main research questions for this chapter are:

***Research Question 3.1*:** *How to find matching cloud platforms for particular application and user requirements?*

***Research Question 3.2*:** *Are there any issues with the accuracy and satisfaction of user queries caused by data and query biases?*

***Research Question 3.3*:** *How to semantically enhance matching algorithms to find and rank cloud platforms that only partially match the defined requirements?*

To elaborate the questions, the chapter is structured as follows: In Section 5.2, we describe our fundamental selection approach targeting *RQ 3.1*. Next, we evaluate real user queries to answer *RQ 3.2*. Afterwards, we present our methodologies for data governance and semantically enhanced selection algorithms (*RQ 3.3*) to account for data and query biases in Sections 5.3 and 5.4. In Section 5.5, we contrast our approach with existing related work. Section 5.6 discusses limitations and future work. Finally, Section 5.7 summarizes the contributions of the chapter.

5.2. Decision Support System

Parts of this section have been taken from [191].

According to Slawik et al. [336], three activities need to be carried out when enterprises contract and consume cloud services. These activities are the discovery of services, the assessment of potential services, and finally the selection of an appropriate cloud provider. The tasks are complicated by various issues

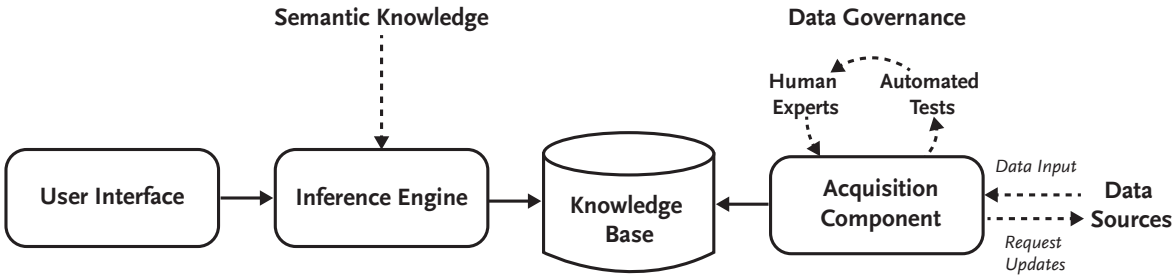


Figure 5.1.: Decision Support System Architecture

that are present in the state of the art of cloud service selection which are targeted by this work. First of all, the cloud market is vast and evolves fast, so new offerings are launched frequently while others are discontinued [336]. For a customer, it is hardly possible to get an overview as a foundation for an informed decision without a central marketplace for service publication and registry [349]. Nevertheless, problems of ambiguous criteria and complex and incomparable features can be targeted by a normalization of cloud service descriptions and their use in cloud brokers [336, 349]. We already addressed the lack of a marketplace for service publication with *PaaSfinder*. We enhance the current system with querying and rating mechanisms in this chapter. Likewise, a solution for the lack of normalization of cloud service descriptions and their use in cloud brokers is suggested by the utilization of our PaaS taxonomy.

5.2.1. System Architecture

Figure 5.1 shows the general architecture of the DSS *PaaSfinder*. The central part of the system is the knowledge base with the PaaS provider data, as defined in Section 4.4. The data is entered and updated via the acquisition component from different data sources and stakeholders. Several quality control stages are implemented along this way which are described in the following to ensure data quality. The interface to the inference engine of the knowledge base supports human as well as automatic decision makers, which we believe is essential for a decision that is both automatable and technically influenced, but finally mostly driven by human decision makers. Moreover, it realizes the matchmaking capability which returns matching providers for a query of required capabilities. The requirements matching (see Figure 5.2) can either be done visually by selecting all needs on the web interface or via an application profile that is automatically matched against the PaaS profiles. Technically, a query on the PaaS profiles is a profile by itself (query by example). An application profile can include arbitrary properties that are included in the profile specification (see Section 4.4). A subset of these properties can be used to describe the required application dependencies and PaaS capabilities. The system implements two different strategies for matching the requirements with the data. For the default strategy, the exact matching step, all properties of the request are matched with

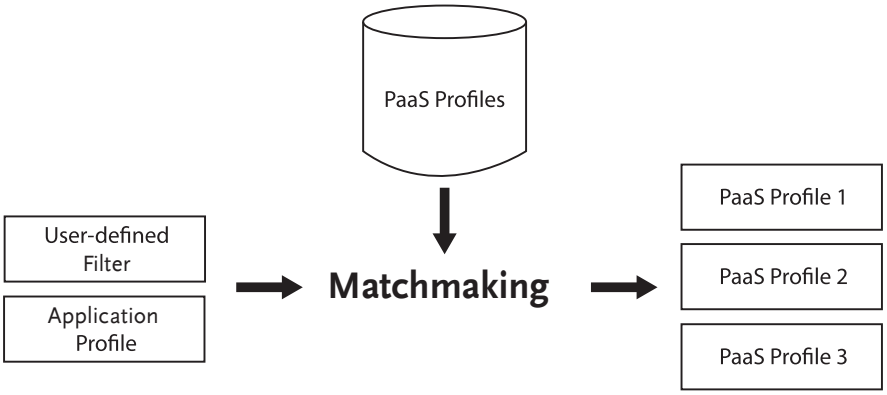


Figure 5.2.: Application Requirements Matchmaking

the data and only results that completely fulfill them are returned. Additionally, a partial matching is available which is able to adapt the requirements and to find offerings that nearly match the request. The details of the algorithms are explained in Sections 5.4.1 and 5.4.2.

Listing 5.1 shows the application profile for the web application prototype from Section 4.5. In that case, a suitable PaaS must allow public deployment, has to be horizontally scalable, support the Ruby runtime in version `2.4.2`, and a stable MongoDB version between `2.6` and `3.6`.

The presented system addresses RQ 3.1 to find matching cloud platforms for particular application and user requirements based on our PaaS knowledge base.

Listing 5.1.: Application Profile for the Web Prototype

```
{
    "hosting": {
        "public": true
    },
    "scaling": {
        "horizontal": true
    },
    "runtimes": [
        {
            "language": "ruby", "versions": [ "2.4.2" ]
        }
    ],
    "services": {
        "native": [
            {
              "name": "mongodb",
              "versions": [ "2.6", "3.0", "3.2", "3.4", "3.6" ]
            }
        ]
    }
}
```

5.2.2. User Query Behavior

In the introduction of Chapter 5, we asserted that the quality of the data and the users' queries have a possibly negative influence on the selection result, leading to fewer appropriate results or even no results. To elaborate this hypothesis, we state RQ 3.2:

Are there any issues with the accuracy and satisfaction of user queries caused by data and query biases?

To evaluate this question, Table 5.1 shows aggregated statistics of user queries recorded by *PaaSfinder*. The queries were conducted by real users with an interface for exact matching as described in Section 5.4.1. A total amount of 7910 interactions from 4200 users were recorded in between 8/2/2016 and 01/18/2017. Unique users are identified based on their IP address. Technically induced duplicate requests, caused by repeated clicks on the filter button leading to multiple identical requests by one user in a short time span, are removed from the data[126]. We define *satisfied* queries as queries that return at least one PaaS provider that completely satisfies the requirements specified by the user as result. This does not necessarily mean that the user found the appropriate provider for his needs, but the query itself is satisfied. Consequently, all queries that return no result are categorized as *unsatisfied*. As we can see, 73.78 % of the user queries provided at least one result. Then again, this means that more than a quarter of them did not satisfy the user's query. If we take an even closer look at the distribution of the result sets in Figure 5.3, we can see that there also exists a tendency towards result sets with very few results, which leave the user with no real choice ($N(results \leq 1) = 2845; 35.97\ \%$ of total queries).

We can observe a positive skew in Figure 5.3, i.e., the distribution of the result sets is right-skewed. The majority of queries return relatively few results. We also see very few queries that return very large result numbers. This shows that only a few queries are very generic and ask for a broadly-supported feature of PaaS. We expect that these features are taken for granted and provide no new insights for the users. However, one outlier at $results = 25$ is notable. The corresponding query searches for providers that offer a free hosting option. Naturally, this is a very popular search term for customers, especially for private end users. The frequency of result set sizes between three to 20 results is distributed relatively equal. This also implies that there are differences between the providers as the result sets slightly change for the queries and reinforce the necessity for assisted provider selection.

By carefully examining the set of unsatisfied queries, we discovered that there is a larger share of queries that could have been satisfied with additional expert knowledge. In this context, an expert is a person with special skills

[126]The anonymized raw data can be accessed at `https://github.com/stefan-kolb/paas-profiles/releases/tag/cloud17`.

Table 5.1.: Statistics for User Interactions with Exact Matching Algorithm

	Exact matching
Number of satisfied queries	5836 (73.78 %)
Number of unsatisfied queries	2074 (26.22 %)
∑ Total queries	7910

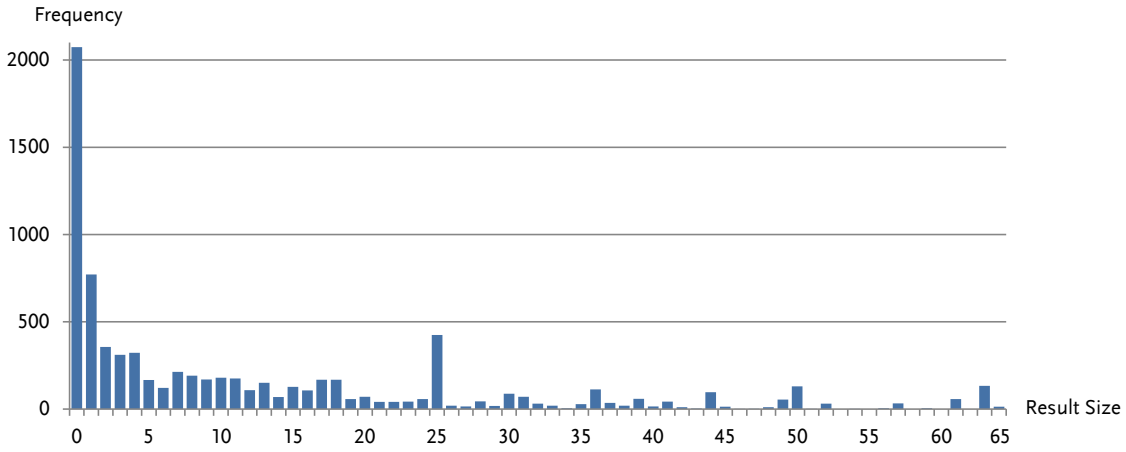


Figure 5.3.: Distribution of Result Set Sizes for User Interactions

or experience in the particular area, who is widely recognized as a reliable source of knowledge in that field [113]. Nevertheless, such experts are costly and therefore it would be beneficial to externalize parts of such knowledge once and then reapply it automatically to different queries on the system. For this, we first need to detect the problems that prevent a satisfying query result. Next, we want to evaluate if they can be prevented by utilizing semantically enhanced selection algorithms. Therefore, we analyze the queries, the data, and the results, and categorize what led to the unsatisfied query result. Thereby, we identify two main categories of problems that cause the mismatches: *query* and *data* problems. Both types influence each other to create unwanted effects on the query results, e.g., no results. Figure 5.4 summarizes these potential problems for queries on knowledge bases intertwined with our approach to include semantic knowledge to account for these issues. As we see, these problems are often neglected in related work on cloud provider selection and limit the applicability of the research results in practice. In the following, we describe the identified problems and present possible solutions for preventing or mitigating them both in general and in the specific application to our knowledge base.

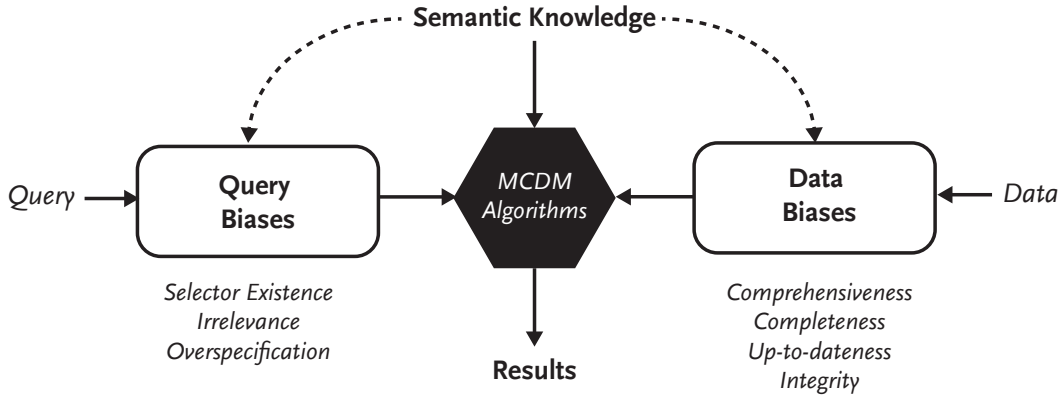


Figure 5.4.: Selection Problems Impacting MCDM Scenarios

5.3. Data Biases and Data Governance

A comprehensive, correct, and consistent knowledge base is the foundation of a methodical comparison and selection of cloud platforms. Hence, an important step during any MCDM [360] approach is to continuously secure the data quality through appropriate data governance measures [133]. This has to be enforced by a carefully designed data model at design time and quality assurance measures inside the acquisition component at runtime.

Several of the data problems depicted in Figure 5.4 that occur at runtime already manifest themselves the during design time of the data model. As discussed in Section 4.2, the decision problem for the selection plays an important role for the scope of the model's attributes [344]. This point of view also defines which data is relevant for the knowledge base. For cloud platforms, we found a conflict in related work between the aims of knowledge bases and their practical applicability. Not all properties that seem relevant for a particular problem can be fulfilled or are available for a majority of entities in the real-world, e.g., SLAs or CPU power values. Hence, typical data problems that occur at runtime are missing or incomplete data (*comprehensiveness, completeness*). Sometimes, possible candidates are not available inside the recent data set (*comprehensiveness*) due to the fast changing business. Moreover, specific information is not listed due to the amount and complexity of the model's properties (*completeness*). The up-to-dateness of data is often limited as providers are frequently changing their offerings but corresponding data is not available in a structured form and must be manually updated by humans (*up-to-dateness*). Also, threats to the *integrity* of the data due to consistency problems and errors are common. Not all domain-specific data consistency rules can be defined inside the data model. Consequently, missing information on derived or related data entries such as CF-based platforms, different naming terminologies, or faulty values cannot be completely eliminated [302, 394]. Also, there exist several threats that exacerbate the identified data biases. First of all, stale data and infrequent updates are problems of knowledge bases that often occur. Next, we experience

data biases due to provider interests and misinformation that are caused by different knowledge levels and opinions of data suppliers.

Due to the presented problems, special attention must be paid to data governance in knowledge-based systems. In our case, one might think that an automatic information gathering process of the platforms' specifications is feasible. However, there is virtually no data available in a standardized and structured form that would allow us to crawl information automatically. Instead, data comes from heterogeneous sources and most often in natural language. Examples are the provider websites, user documentations, and changelogs. Only rarely, the sources are APIs. Consequently, the data is typically gathered and entered through the acquisition component by experts or knowledge engineers. Due to the amount, diversity, and continuous changes to the PaaS provider market, this is not feasible. Therefore, we use a more open process with additional knowledge providers to enhance data completeness and actuality. We decide to apply crowd-sourcing as a concept for data collection and update of the cloud service registry [336]. Hence, the data is available in an open repository that allows contributions from various stakeholders that participate in the PaaS market. Stakeholders are domain experts, vendors, consultants, end users, and automatic crawlers. As the various knowledge suppliers have different levels of expertise and intentions, we needed to implement several techniques and barriers inside the acquisition component to ensure good data quality.

First and foremost, data structure requirements and limitations for available properties and values are enforced by our generic PaaS taxonomy. Additionally, we implemented a large set of automated semantic rules for the data inside the acquisition component that cannot be checked by syntactical model constraints. We use the tests to avoid duplicates and intersecting concepts [276, 302]. Furthermore, they help to ensure the completeness and consistency of concepts and values where possible. Measures include a restricted set of runtime languages, no duplicate software within multiple categories, only one unique runtime per framework, and consistency between providers of a shared base platform, such as CF. Nevertheless, a second human quality control stage must be passed before the data is merged. Solely ensuring the structural and semantic integrity of the data via automatic tests is not sufficient. There still is the problem that wrong information might be given unintentionally or intentionally. Both can only be reasonably validated by human experts. As an example, we experienced that IaaS vendors which did not provide specific PaaS features tried to add their offering to the knowledge base for marketing reasons. Additionally, a feedback loop to the knowledge engineers to trigger a revalidation or an update of the data after a certain threshold is desirable to avoid stale data. All the data is kept in version control, to be able to revert changes if necessary and keep a replicable record of the changes.

With the presented data governance measures, we prevent numerous typically unhandled threats to data quality in advance. However, since absolute

completeness and consistency of data is hard to achieve, this must be complemented by additional semantic enhancements to the query. An approach for this is described in the following section.

5.4. Query Biases and Semantic Query Enhancements

Query problems are caused by a user or a machine that sends a query to the knowledge base. In some cases, query data may be incomplete, inaccurate, or simply irrelevant to the problem that is being investigated [113]. This includes nonexistent query attributes or values that must be prevented by structural query validation to avoid unsatisfied queries (*selector existence*). An example for this would be a user looking for an inexistent runtime language version of Java. In our case, a query is validated by our PaaS feature model by translating it into a virtual provider model. Other effects are harder to detect and handle. For example, if a user does not ask the right question for his problem, e.g., a user is looking for hosting in North America but does not include the option for private hosting in his query. If the model does not cover the relevant properties of the decision problem, the query cannot be answered with an appropriate result. Also, the more detailed a query is, the more complete the data set needs to be. As a state of complete information is not achievable in reality, we need to assume the imperfectness of a data set. A too specific query in connection with an imperfect data set can lead to an unsatisfied query whereas the actual requirements might still be satisfiable in reality (*overspecification*). In general, there is a correlation between very specific queries and a no result. Our data shows that unsatisfied queries have an average of 5.38 query keys, whereas satisfied queries have only 2.72 keys per query selector. The typical search behavior applied by users can often be described by a back and forth of specialization and generalization of the search query [43, 309]. One of our ideas to tackle this problem is to automatically assist the user on his way in this process via semantic knowledge to make it feasible to relax possibly overspecified queries and show the effects of query relaxations to the user.

Therefore, to mitigate the observed effects, we propose an enhanced multi-step selection process (see Figure 5.5). Overall, we try to answer a typical MCDM question: “given a set of alternatives and a set of decision criteria, then what [are] the best alternative[s]?” [360, p. XXV]. First, we try to fully satisfy the user query. In case we could not find a suitable provider and want to provide possible alternatives, we relax and adapt the query based on externalized semantic knowledge. In the following, we first discuss the exact matching algorithm. Afterwards, our semantic algorithms that allow such a partial matching (*RQ 3.3*) are outlined. Finally, we evaluate the impact of the

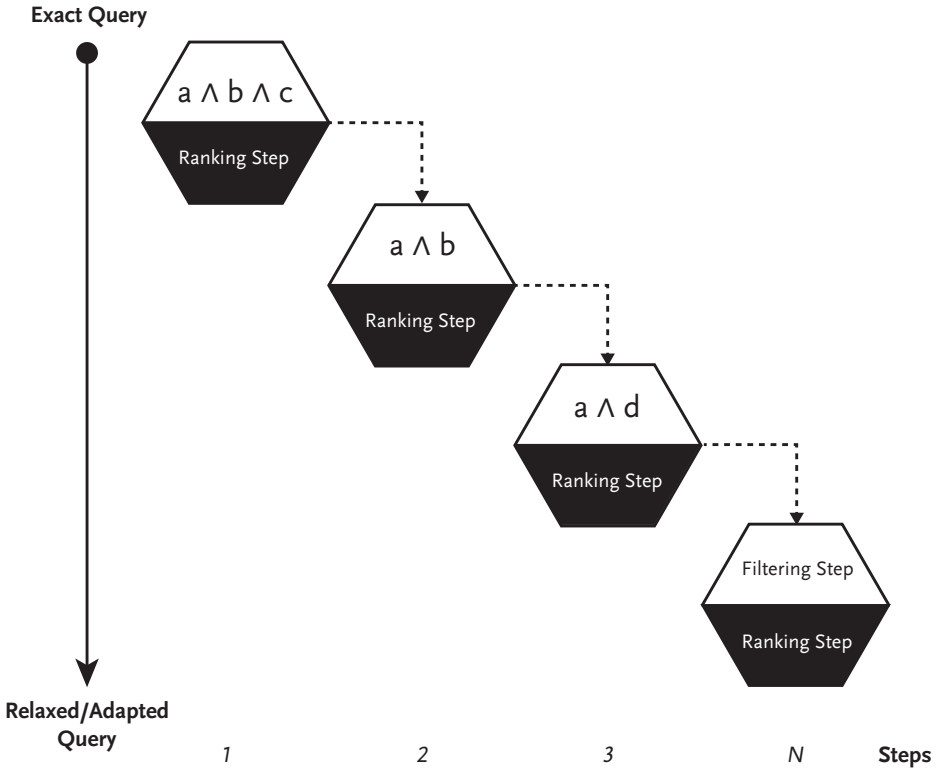


Figure 5.5.: Multi-Step Selection Process

proposed algorithms on the recorded user queries in comparison to the exact matching.

5.4.1. Exact Matching Step

Ultimately, a query result that fully satisfies the user's demands is what we always should aim for. This is especially necessary for automatic application migration scenarios, where we cannot relax the query and risk incompatibilities that would potentially break the portability by forcing adaptations to the application. In fact, the exact matching step can be divided into two steps again, a filtering and a ranking step (see step one in Figure 5.5). As our focus for the selection and the properties of our PaaS model is on application portability, most of the criteria are must-haves. This means if any of those criteria cannot be fulfilled by a candidate provider, it must be excluded from the result set (*filtering step*). Most of the popular MCDM algorithms do not handle such must-have criteria well but optimize a target function. Such a step can be best applied after the initial filtering of portable candidates to rank the results based on non-must-have criteria, e.g., pricing or uptime, with established MCDM methods (*ranking step*). To this end, several well-known algorithms such as the AHP [318], outranking [315], or the weighted sum model [105] can be used. By now, inside the filtering step, all requirements are equally important for the selection algorithm. In our case, this is reasonable as most requirements

are relevant for application portability and therefore must-haves. Also, user-defined preferences require more knowledge by the user with regard to what implications they have on the result set. Furthermore, wrong configurations might have unwanted side effects on the results. Therefore, in this scenario, we follow a different approach and try to make it as easy as possible for the user to define his must-haves and let the algorithms do the hard work. However, we could add this functionality later to feed the algorithms with additional information about what can be safely relaxed to expand the result set.

Algorithm 1 shows the exact matching as conceptual algorithm. The matching of all given requirements except the *version* attributes is `AND` concatenated, i.e., all requirements must exactly match 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. 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.

Algorithm 1 Exact Matching Step

```
providers ← {p_1, p_2, ..., p_n}                    ▷ universe of PaaS providers

function FINDPROVIDERFOR(requirements)
    demands ← requirements          ▷ set of user or application requirements
    prospects ← ∅                   ▷ set of provider prospects that fulfill the demands

    for all p in providers do
        for all d in demands do
            if !p.supports(d) then  ▷ test if demand is supported by provider
                next p              ▷ break inner loop and test next provider
            end if
        end for
        prospects.add(d)            ▷ add matching provider to set of prospects
    end for

    return prospects                                 ▷ return matching providers
end function
```

5.4.2. Semantic Matching Steps

As discussed, there are data and query problems that cause unsatisfied queries or reduce the amount of possible alternatives that are displayed. Therefore, as a second step of our selection approach, we introduce a partial matching stage (*RQ 3.3*). It is suitable for semi-automatic or manual application migration search or exploration with user interaction. This is reasonable as of now, in

Listing 5.2.: PaaSfinder Logged User Query Data

```
{
  "_id" : "57a10026fd346d0006166291",
  "ip" : "141.101.104.0",
  "query_selector" : "{
    'runtimes.language' => { '$all' => ['python'] },
    'frameworks.name' => { '$all' => ['django'] }
  }",
  "result_size" : 23,
  "created_at" : "2016-08-02T20:18:46.945Z"
}
```

most cases, a fully automatic migration between cloud platforms is not possible anyhow (see Chapter 6 for details). Again, see Figure 5.5 (steps two and three) for an illustration of how we gradually relax and adapt the user's query to find new sets of (additional) appropriate selection results. In that case, we make use of certain rules and algorithms based on semantic information to alter the initial user query. This can consist of removing certain query attributes ($a \wedge b \wedge c \rightarrow a \wedge b$) or adding new ones ($a \wedge b \rightarrow a \wedge d$). These rules account for the data and query problems discussed before. Overall, there exist some general algorithms that can be applied to nearly all knowledge bases of similar type as ours, e.g., similarity based on edit distances of query attributes. However, most of the general concepts need to be specifically tailored to the exact domain of the knowledge base [113, 302]. In the following, we present a set of strategies for the aforementioned problems applied to the case of cloud platform selection. By using different domain-specific rules, several restrictions can be relaxed while still retaining a perfect fit or at least the possibility to find a fit. Even when there is no exact match, such a result can satisfy the user more than no result at all. Some strategies are portability preserving while others imply the chance that porting is possible but not guaranteed or involves more effort to achieve the same result. Therefore, these results must be marked with a hint and an explanation of the changes for the user. Here, we present a selection of suitable strategies for our main findings inside the PaaSfinder system.

When a user searches for a PaaS in the UI or via the API, the search query gets transformed into a MongoDB query on the database. Like all data in MongoDB, the user query is encoded as a JSON-like data structure[127]. Listing 5.2 shows an example of the logged data for a user query. It includes the anonymized IP of the user, the query selector of the request, the size of the result set that was presented to the user, and the time when the query was executed. To evaluate our semantic algorithms against the original query, we can extract and manipulate the query selector from this structure, and execute the modified query on the database. Later, in the final implementation, the enhancements

[127] https://docs.mongodb.com/manual/tutorial/query-documents

can be done directly on the virtual provider model that translates to the query. Algorithm 2 shows the general pseudo algorithm for relaxing the initial query.

Algorithm 2 Semantic Matching Step

Require: FINDPROVIDERFOR($requirements$) $= \emptyset$ ▷ unsatisfied query

```
function RELAXQUERYFOR(requirements)
    demands ← requirements          ▷ set of user or application requirements

    if !QUALIFIESFOR(demands) then          ▷ check if query qualifies
        return ∅
    end if

    relaxedDemands ← RELAXQUERY(demands)    ▷ apply a specific algorithm

    return FINDPROVIDERFOR(relaxedDemands)
end function
```

In the following, we discuss several specific strategies.

Generalization

What we learned from our work with the knowledge base is that the more specific the information is, the less likely it is that it is given or even up-to-date. Therefore, specific information that comes in a hierarchical context with other data can be relaxed more safely without influencing the selection result than generic information. Also, as discussed before, users tend to overspecify their needs which also fosters the negative impact caused by missing or outdated data. Hierarchical relationships between data are frequent in technological knowledge bases. As an example, see Relation 5.1. Here, the hierarchical relationship between runtime, framework, and version is shown. A framework can be attributed to a specific runtime, whereas a framework has a (possibly larger) set of framework versions.

$$Runtime \rightsquigarrow Framework \rightsquigarrow Version \tag{5.1}$$

A concrete instance of this rule from our data would be: Ruby $\rightsquigarrow$ Rails $\rightsquigarrow$ 5. In the course of the partial matching stage, we can relax a query that includes very specific information, e.g, a framework version, so that it only requests the framework and implies that the specific version may be supported. Even further, we can then relax the need for the framework to a requirement for the corresponding runtime and imply that this framework will be supported. Although this line of argument and relationship might sound ambitious, tests with our data show that it is actually often feasible.

Customization

Although today, all cloud platforms come with a large ecosystem of technological assets, there is still room for customization and the addition of technologies by a user. This trend was initially introduced by the buildpack concept of Heroku that allows users to customize and add specific technologies to the preconfigured application environments just like they were used to with IaaS offerings. This was necessary as more and more offerings evolved from specialized language-based PaaS to polyglot PaaS supporting multiple runtimes and therefore, the state space of ecosystems to be supported grew exponentially. Platforms that offer such an extensibility concept are capable of running more technologies than officially supported by the provider. Due to scalability issues with services like data stores (i.e. necessary data replication), this is typically best applicable to the addition of other runtime languages and framework support. The selection algorithms can make use of that and include these providers into the extended result set even when runtimes are not supported according to the data. This especially helps with queries that look for a larger set of supported runtimes which indicates the need for extensibility of the platform. Our data gives evidence of 505 (6.4 %) queries leading to unsatisfied queries that are candidates which can benefit from this algorithm ($N(unsatisfied)_{runtimes>1} = 505$).

Compatibility

The concept of compatibility or replaceability can be applied in multiple ways inside the selection algorithms. The ISO/IEC define replaceability as:

> **Definition 5.1 (Replaceability)** *"[Replaceability is the] degree to which a product can replace another specified software product for the same purpose in the same environment."* [159]

Ideally, this would mean that a product can replace another product without any modifications. Standards aim at such a degree of replaceability. In reality, products are at most partially replaceable, which we term *compatible*, and require additional modification efforts. In practice, we often find products that implement the same base technology or a standardized language such as SQL, e.g., MySQL[128], PostgreSQL[129], and MariaDB[130]. Sometimes, providers also have their own names for a base system as it is tweaked differently, e.g., Pivotal GemFire[131] which essentially is Apache Geode[132]. Additionally, the ecosystem of cloud platforms can replace required native services through their

[128] https://www.mysql.com
[129] https://www.postgresql.org
[130] https://mariadb.org
[131] https://pivotal.io/pivotal-gemfire
[132] https://geode.apache.org

third-party add-on marketplaces. It can be beneficial to use a specialized and managed add-on service for specific service dependencies rather than relying on a native service inside the platform. By defining a set of compatible services and add-ons, we are able to suggest alternative configurations and providers to users.

Terminology

Even with a stringent data model and automated tests as presented in Section 5.3, it cannot be guaranteed that every technology has the same consistent name throughout the data. Due to the large and diverse ecosystems, attribute values cannot be completely restricted and enumerated. In our case, this applies to frameworks (64), middleware products (30), services (165), and add-on names (287).[133] Therefore, we also need to find similarities between them to correct inconsistencies and also add probable duplicates to the result sets. To that end, we can apply edit-distance-based algorithms to identify identical technologies [253]. The most well-known set by Levenshtein [209] defines the three operations insertion, deletion, and substitution that can be performed on a string. All of them have their own unit cost for performing the task. In its simplest form, it is a value of one per operation, which results in a metric value for transforming one string into another. Other variants of edit distances are defined by a different set of operations. Adding transpositions to the allowed set of operations leads to the Damerau–Levenshtein [81] distance. The Longest Common Subsequence (LCS) [17, 254] is an edit distance with insertion and deletion as the only two edit operations. The Hamming distance [320] allows only substitutions which limits its applicability to equal-length strings. Last, the Jaro–Winkler [382] distance can be obtained from an edit distance where only transpositions are allowed. For the following evaluations, we use the Levenshtein [209] distance.

Portability

As mentioned before, every selection approach has its specific scope that manifests itself inside the knowledge base's model [344]. Most of the times, several additional attributes are added to the model that contribute to the practical applicability of the selection and are often requested by users. Nevertheless, when using an excluding matching step, all criteria contribute equally to the candidate filtering regardless of their importance for the selection scope. To not require the users to apply this specific knowledge manually, the algorithms should be aware which attributes can be safely relaxed. In our case, we mainly focus on the portability of application dependencies. Therefore, it can be be-

[133] Number of distinct values obtained at 5/28/2018.

Table 5.2.: Evaluation of Semantic Query Enhancements

Algorithm	Target	Query selector(s)	Avg. results	Number of unsatisfied queries	Percentage of unsatisfied queries
Exact matching			5	2074	26.22 %
Generalization	completeness, up-to-dateness, overspecification	*frameworks*	8	1792 (-282)	22.65 % (-3.57 %)
Customization	completeness, up-to-dateness	*runtimes*, *frameworks*	16	1345 (-729)	17.00 % (-9.22 %)
Terminology	integrity	*middleware*, *frameworks*, *native services*, *add-ons*	5.5	2010 (-64)	25.41 % (-0.81 %)
Compatibility	integrity	*native services*	6	1992 (-82)	25.18 % (-1.04 %)
Portability	overspecification, irrelevance	*status*	7	1873 (-201)	23.68 % (-2.54 %)
		pricing	9	1686 (-388)	21.31 % (-4.91 %)

neficial to relax criteria that are not immediately important for application portability, e.g., pricing.

5.4.3. Evaluation

In Section 5.2, we already showed that there are issues with the accuracy and satisfaction of user queries caused by data and user query biases. To assess the effects of our proposed semantic enhancements, we need to analyze whether the algorithms improve the satisfaction and accuracy of real user queries. For evaluation, we apply the suggested strategies[134] on the recorded user interactions to evaluate how the user experience would have been altered with our proposed semantic query enhancements. Table 5.2 shows the results of both the exact query matching and the semantically enhanced queries in relation to each other.

We do not only see that the number of unsatisfied queries decreases substantially due to the semantic enhancements but also that the number of average results (median) rises. Generally, more choices are not necessarily desirable as the decision problem for the user gets more complicated with a larger result set. Therefore, additional semantic steps are best applied to empty result sets or very few results, which give the user little choice. Exactly these cases are frequent in our data ($N(results <= 1) = 2845; 35.97\ \%$) which strengthens the necessity for the semantic matching steps (see Figure 5.3 in Section 5.2.2).

The *generalization* strategy proves to be an effective query enhancement. Apparently, the sizes of the platforms' technological ecosystems pose a problem

[134]The implemented algorithms can be found at `https://github.com/stefan-kolb/paas-profiles/releases/tag/cloud17`.

for the completeness and up-to-dateness of the data set. The presented enhancements smooth out these inconsistencies while still preserving these frequently requested model attributes. As expected, the *customization* strategy has the highest impact on the result set size, as it appends every extensible platform to all runtime queries. Whereas it does retain application portability, due to its impact on the query, it should only be used sparingly and for the final stages of the selection process. The *terminology* results show that our data governance measures from Section 5.3 are working quite effectively, as we experience only a few syntactic duplicates which sum up for less than one percent of result set changes. In our experiments, we use the Levenshtein [209] distance to detect probable duplicates. As a result, we identified 8 (1.5 % of total concepts) semantic duplicates. For the *compatibility* mappings, we identified nine sets of compatible native services (35 services in total). Even with this small set of relationships, we can improve the amount of satisfied queries by one percent. This could be further enhanced by identifying appropriate mappings to available third-party add-on services. Last, we evaluated two examples for relaxing query attributes, i.e., status and pricing, that do not directly contribute to the *portability* of application dependencies. Both highly impact the user's request but can be valuable to satisfy queries that need to focus strongly on application dependencies first. Overall, we can conclude that the more detailed queries a knowledge base allows, the more beneficial our semantic enhancements and matching stages will be.

5.4.4. Prototype

Next to the definition of the multi-step selection process, we implemented a UI and API for the selection of PaaS providers inside our production system.[135] Again, we collect several user stories for the planned functionality in advance. The work of this chapter adds two additional FRs to our web application *PaaSfinder* (see Section 4.5.1).

FR.User.5 – filter: As a user, **I want to** be able to filter and search the structured set of information of the PaaS profiles **so that I can** quickly extract possible candidates of providers that fulfill my requirements as an entry point for further investigations.

FR.User.6 – suggestions: As a user, **I want to** get recommendations for alternative provider prospects if my search does not return matching providers **so that I can** evaluate possible candidates of providers that partially fulfill my requirements as an entry point for further investigations.

From the landing page of the web application, a user may directly navigate to the filtering and matchmaking capabilities (*FR.User.5*). Figure 5.6 shows

[135] An online version of the web interface can be found at `https://paasfinder.org/filter`.

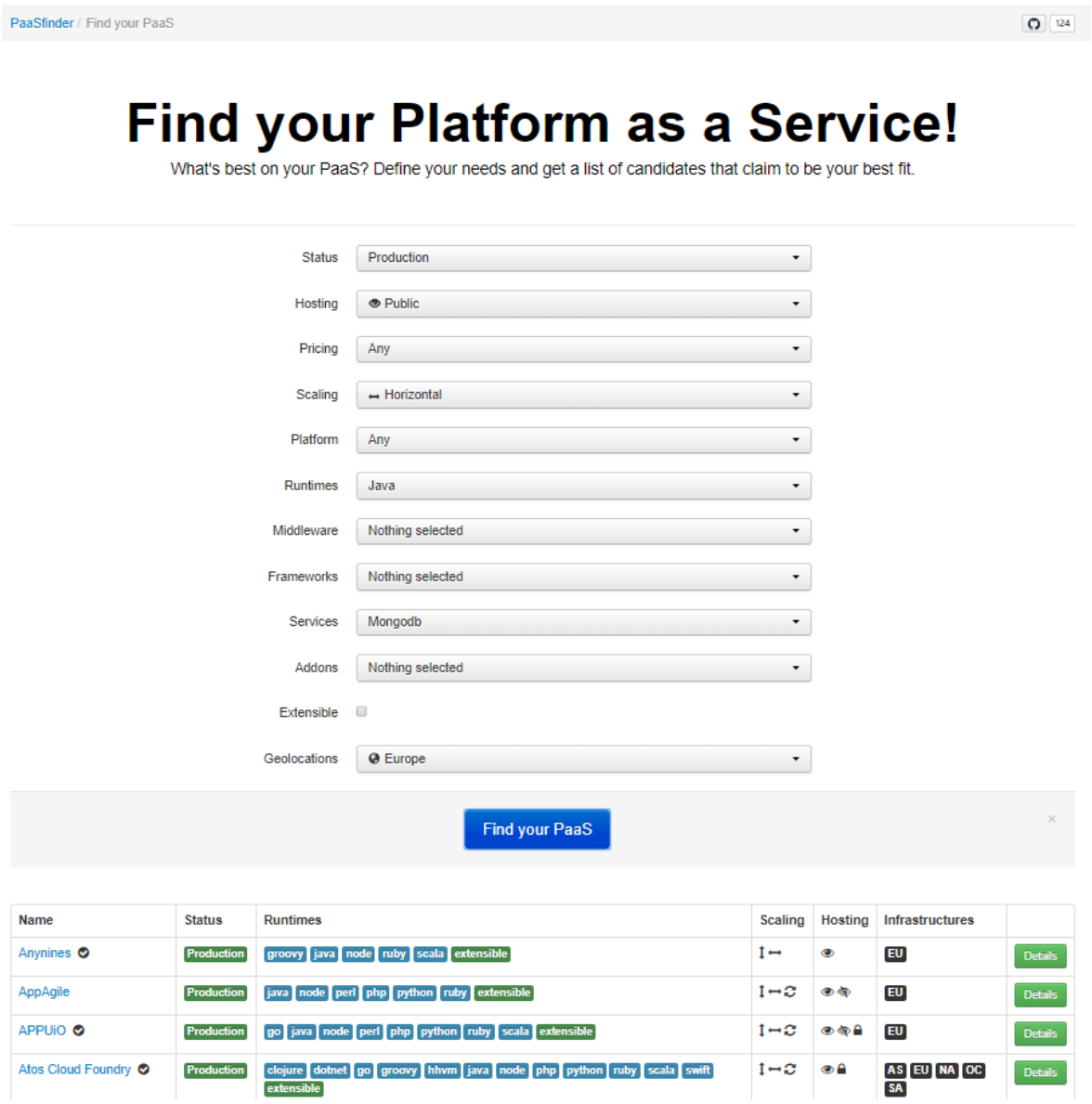


Figure 5.6.: PaaSfinder Provider Selection

a snapshot of the UI for provider selection. The results can be influenced by applying multiple filters along the properties defined in the profile specification. We depict additional contextual information where the properties may not be self-descriptive. After executing a filter request, we present a list of matching providers to the user below the form. The results are displayed as done for *FR.User.1* by a list of important characteristics to allow a quick overview of the offerings. Additionally, we supply an implementation of the functionality via a RESTful API. The user or a software system may query the API with a virtual provider model to request matching provider profiles. This enables external tools to query the broker programmatically for, e.g., automatic provider discovery.

If no providers are found for the query, an alert is shown to the user. In that case, results produced by the semantic matching steps can be displayed

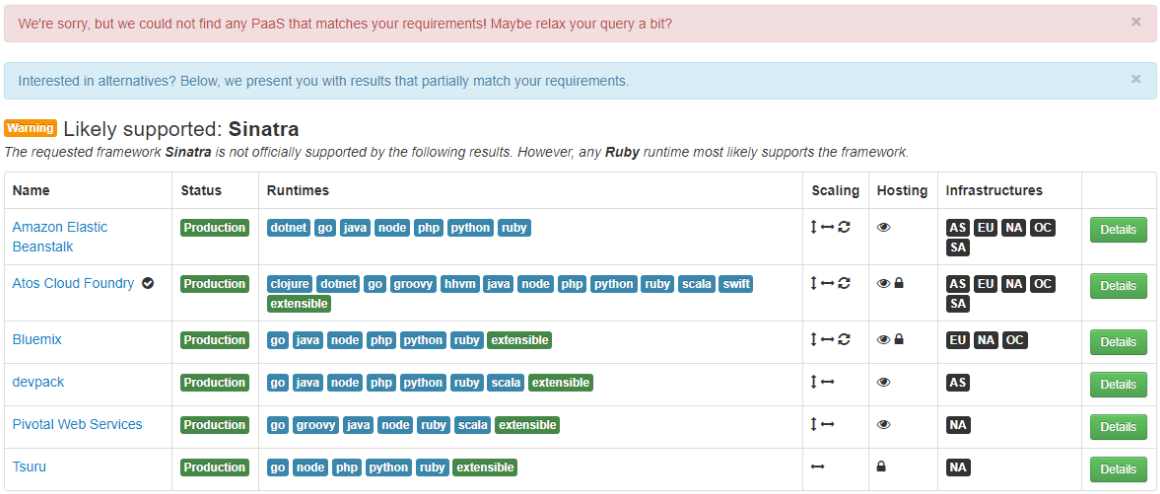


Figure 5.7.: PaaSfinder Provider Suggestions

below (*FR.User.6*). Visually, a result may be depicted similar to the normal result table enhanced by an explanation of the changes and effects on the query (see Figure 5.7). This directly explains to the user what has changed and why. Currently, the semantic algorithms and provider suggestions are solely evaluated manually and not yet implemented inside the system. This remains future work.

5.5. Related Work

Existing literature targeting cloud provider selection can be best distinguished based on the cloud model [152]. Essentially, a majority of papers focus on IaaS, whereas little work has been conducted on PaaS cloud service selection [95, 349]. In general, recent work on cloud service selection is primarily focused on rankings based on cost comparisons and other nonfunctional characteristics [300, 349]. Functional aspects are mostly only considered through virtual machine capabilities in the IaaS context. Moreover, all the referenced works validate their approaches on a very limited data set of cloud providers compared to our study.

Generic Approaches

Despite the differences between cloud types, many existing approaches intend to suggest a generic solution for cloud services. However, this often limits their practical applicability due to the conceptual differences between cloud types and therefore small set of intersecting properties [152].

Spillner and Schill [340] propose a generic broker for XaaS including a set of base and specific domain ontologies. Base ontologies encompass physical units, metrics, system and service properties as well as nontechnical properties

for business, legal, and context aspects. Domain ontologies make use of these concepts and describe domain-specific service concepts. These domain ontologies are finally instantiated to describe actual services. Wittern et al. [383] use Feature Models (FMs) as a representation mechanism for requirements elicitation within a cloud service selection process. As a specific application of their generic selection approach, they present models for cloud storage selection. Sundareswaran et al. [350] are focused on the performance of the selection process rather than the semantic accuracy of the approach. Several papers are concerned with the selection of cloud services based on QoS and SLA parameters [13, 14, 23, 111, 172, 278, 294, 396]. Whereas some of them primarily target IaaS offerings, most of the attributes are generalizable and applicable to any XaaS offering. Amato et al. [13] present an approach focused on SLA negotiation between cloud providers. Jung et al. [172] present a recommender platform focused on QoS properties like cost, performance expectation, and energy efficiency. Similarly, Garg et al. [111] concentrate on a set of quantitative QoS attributes taken from the Service Measurement Index [331]. Based on Garg et al. [111], Patiniotakis et al. [278] provide a cloud service ranking technique that is capable to exploit both exact and fuzzy information. Thereby, requirement values can be specified as a set of numbers, intervals, or even linguistic terms that are eventually mapped to fuzzy numbers to ensure a unified processing. For optimization, the authors derive fuzzy comparison matrices and subsequently use a fuzzy AHP method to rank cloud services. Related, Zilci et al. [397] present an approach to solve the service matching problem with soft QoS constraints based on constraint programming [40]. The constraint programming approach of Zilci et al. [397] is used in the work of Slawik et al. [335, 336] on the Open Service Compendium (OSC). The OSC is a crowd-sourced cloud service registry focused on business-relevant attributes. The business vocabulary reflects common cloud service selection criteria such as the cloud service model, pricing, security, and QoS. The models are based on SDL-NG, a self-developed DSL implemented in Ruby. As a specialty, the service descriptions support automatic parsing of properties from, e.g., web sites. This is due to the ability of SDL-NG to include Ruby code inside the models, which is interesting as some properties often change and are rarely available in a structured form. Examples of business vocabularies for cloud storage and IaaS offerings are available in their source code repository[136]. Mezni and Abdeljaoued [242] propose a cloud service recommendation system that accounts for data biases such as limited data for quantitative data sets. The approach is based on fuzzy formal concept analysis. This method is applied to transform a cloud service repository into a set of small clusters, where the relations between high quality services and users with the best experiences are organized using fuzzy formal concepts. In contrast to most existing recommendation approaches, they use a hierarchical

[136] https://github.com/TU-Berlin-SNET/sdl-ng

data representation instead of a matrix of vector structures, to allow a better grouping and analysis of information. They validate their general approach in the context of an IaaS recommendation scenario. Menzel et al. [240] suggest a step-by-step process to build a specific customized evaluation method for any decision scenario. As an application of their framework, they demonstrate an approach for IT infrastructure decisions. Juan-Verdeijo and Surajbali [171] suggest an architecture for an XaaS multi-cloud marketplace in support of industry 4.0 concepts. The idea is to enable customers to choose and select a suitable provider for their needs independently of the three cloud service models. Their motivation is to tackle the heterogeneity of the decision and the lock-in problem between vendors. By looking at the given attributes and dimensions for each of the models, however, it is questionable if the given dimensions can even be attributed to different offerings within a service boundary. As we argue, it remains a challenge to provide a solution spanning the boundaries of the cloud models beyond high-level business properties.

IaaS

A multitude of approaches have targeted the IaaS model. Zhang et al. [394] include properties such as compute, storage, and network which are eventually ranked by entity costs. Semantically, they claim to perform some basic input validation of a user's query. Rehman et al. [301] present an approach for IaaS cloud selection using MCDM methods. Their method is based on five cost and performance-related criteria for thirteen cloud services. Pawluk et al. [281] introduce an IaaS cloud broker focused on cost minimization. Moreover, they try to address lock-in issues by ranking multi-cloud application scenarios while relaxing the cost objective. Andrikopoulos et al. [15] present a DSS for assisting the migration between cloud providers, again focusing on cost minimization. Further, Javed et al. [168] propose a cloud marketplace for dynamic price negotiation enhanced with QoS feedback of customers. Gong and Sim [123] suggest a centroid-based search engine with the help of a k-means clustering algorithm for similarity search. In that regard, a user query is transformed into a temporary entity and compared to the existing provider vectors. To mitigate biases caused by missing data, they replace absent data with default, most frequent, or similar values. Whereas this approach allows exploring similar providers based on semantic equivalences, it is not feasible for selections targeting portability which demand must-have requirements. García-Galán et al. [110] present a vendor-specific IaaS selection focusing solely on Amazon EC2 configurations. Based on their feature models, they validate user queries before the optimization process is executed. Dastjerdi et al. [83] use a set of typical IaaS hardware characteristics like CPU, RAM, and storage in their selection approach. Furthermore, they take selected information such as the OS type, location, and pricing into consideration.

In addition to academic publications, several industry-driven web services for IaaS cost comparisons exist, such as Cloudorado[137], RightCloudz[138], or Rightscale Optima[139].

PaaS

Among other brokering capabilities, the EU-funded project Cloud4SOA [82] includes matchmaking into their work. Even though the underlying model [218] is comparable with our PaaS model, it is relatively high-level. It consists of a PaaS system which may have multiple PaaS offerings that provide software components and a management interface that hosts applications inside an IaaS system. For matchmaking, Cloud4SOA allows selecting certain required capabilities and optional requirements that are resolved into an ordered result on a percentage basis. We omit this option on the user side because we think portability is not about options but must-haves. Still, we can adapt our approach by adding or removing optional capabilities in our filter interface or automatically via appropriate algorithms. The recommendation algorithm of PaaSport, another EU project, is discussed in Bassiliades et al. [28]. Comparable to our approach, the selection algorithm is a two-step process that first selects all matching providers based on functional requirements and later ranks them based on nonfunctional requirements using an aggregation scoring function. It utilizes OWL models and the RDF query language SPARQL [27]. Whereas the algorithm scores on very specific data values like runtime versions or database storage sizes, it does not use any semantic rules to account for data biases which, as we showed, strongly interrelate with detailed queries. Problems caused by missing data are likely, especially since nonfunctional attributes suggested by the approach such as uptime guarantees or processing cores are not known for a lot of PaaS offerings. Quinton et al. [296, 297] propose a SPLs-based approach. They use FMs to describe different PaaS environments. As one FM describes a single cloud environment, FMs will differ from each other. To bridge the semantic gap between these FMs, a generic cloud knowledge model is introduced to describe all concepts relevant to the domain. Next, several mappings based on the concepts of the cloud knowledge model to features with the same semantics of different FMs are introduced. This approach is comparable to our unified PaaS model complemented by the terminology mappings, but in contrast, all of these rules need to be explicitly specified. This requires a lot of manual efforts and risks missing assignments. Additionally, their feature models can be annotated with assets like configuration files and executable scripts. After the selection of a provider, the framework can generate all necessary configuration files and provides an executable script to deploy the

[137] https://www.cloudorado.com
[138] https://www.rightcloudz.com
[139] https://www.rightscale.com/products-and-services/products/rightscale-optima

application to the selected provider. Surajbali and Juan-Verdejo [351] propose a high-level architecture for a PaaS broker that includes provider selection. Their DSS builds on the AHP to recommend and rank available provider alternatives. However, they do not give more insights on how they adapted their formalized selection approach for PaaS recommendation. Goncalves et al. [122] present a concept of a cloud broker that is able to find a best match for a PaaS application. Afterwards, they automatically deploy the application 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 broker has to be aware of application requirements and available cloud offers. However, besides the architecture of the cloud broker and the use cases, the authors do neither present a solution how those application requirements should be collected or represented nor how the existing cloud offerings and their capabilities are structured and matched.

SaaS

In the field of SaaS, offerings are merely comparable based on nonfunctional requirements. Godse and Mulik [120] present a SaaS selection focused on Sales Force Automation for CRM, based on the AHP. Schlauderer and Overhage [322] suggest an assessment framework for evaluating software service providers in general. They define and evaluate a set of important assessment criteria and requirements but no selection process by itself. Zickau et al. [395] present the TRESOR broker and marketplace for contracting cloud services specialized on the health care sector. Examples of the proposed requirements are supported interfaces and standards, data portability, set-up and provisioning time, price components and contract terms, provider certifications, and support options. Instead of OWL or a proprietary data format, the authors use the Eclipse Modeling Framework (EMF)[140] as description language for their models.

5.6. Limitations and Future Work

Due to the focus on application portability, currently all requirements are equally important for the selection and ordering of the result set. Nevertheless, there exist properties that can be relaxed and weighted with a certain importance dependent on the preferences of a user. We could further enhance our work by adding a means to specify preferences that serve as user-defined semantic input for the selection algorithms. The order of the results can then be based on these factors. As of now, all user and application requirements must be specified manually as input for the selection algorithms. In the future, it may be possible to assist the selection process by automating parts of the application

[140] `https://www.eclipse.org/modeling/emf`

requirements detection, e.g., by finding application dependencies through static code analysis.

5.7. Summary

Extending our work from Chapter 4, we proposed a system for selecting cloud platforms for particular application and user requirements (*RQ 3.1*). By examining real user queries from the PaaS knowledge base *PaaSfinder*, we showed that there are issues with the accuracy and satisfaction of user queries due to query and data biases in knowledge bases (*RQ 3.2*). Both of these factors appear in the process of selecting alternatives from knowledge bases in general and especially in MCDM scenarios where multiple dimensions complicate the data and selection scenario. However, current research has not considered these problems sufficiently. To that end, we proposed various preventive measures for data governance in technological knowledge bases and a modified selection process. We presented a multi-step selection process and suggested different semantic algorithms applied to the domain of cloud platform selection that account for the identified data and query biases. These algorithms smooth data and query inconsistencies and allow a partial matching of application requirements (*RQ 3.3*). Our evaluations show that the application of the proposed semantics enhance the user satisfaction and lead to a more accurate selection result. Whereas we validated our approach for a specific selection domain, the findings and general ideas can be beneficial for a wide range of multi-criteria selection approaches.

6. Application Migration Effort in the Cloud

Parts of this chapter have been taken from [193, 194].

In this chapter, RQ 4.1 *("Is it possible to move a real-world application between different cloud platforms?")* and RQ 4.2 *("What is the development effort involved in porting a cloud-native application between cloud platforms?")* are supported.

6.1. Motivation

Due to the dynamic and fast changing market, new challenges of application portability between cloud platforms emerge. This is problematic because migrations to and between clouds require development effort. The higher level of abstraction in PaaS, including diverse software stacks, services, and platform features, also opens up new risks of vendor lock-in [284]. Even with the emergence of cloud platforms based on an orchestration of open technologies, application portability is still an issue that cannot be neglected and remains a drawback often mentioned in literature [21, 85, 152, 286, 332]. As we have shown in Chapter 4, our selection approach based on the technological ecosystem of the providers enables us to apply portability matchmaking between PaaS offerings. However, it is still unclear if the suggested approach is also feasible for a larger real-world application. Moreover, it remains to be clarified where and how the anticipated migration efforts manifest themselves. So far, the effort of application migration in PaaS environments and typical issues experienced in this task are hardly understood. Whereas the migration from on-premises applications to the cloud is frequently considered in current research [15, 34, 138, 182, 274, 368], less work is available for migrations between clouds. To improve this situation, we present a cloud-to-cloud migration of a cloud-native application between seven public cloud platforms.[141] Kratzke and Quint [198] provide us with a definition of cloud-native applications:

[141] All data and results are obtained at the time of the study in November 2015.

> **Definition 6.1 (Cloud-Native Application)** *"A cloud-native application [...] is a distributed, elastic and horizontal scalable system composed of (micro)services which isolates state in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform."* [198]

Essentially, in contrast to an on-premises application, this kind of software is already built to run in the cloud and to be delivered as a service. It follows important cloud-focused design patterns as, e.g., defined by the popular twelve factor methodology[142] authored by the PaaS pioneers from Heroku. Leymann et al. [212] narrow down these patterns to the five properties isolation of state, distribution, elasticity, automated management, and loose coupling. As our application conforms to these patterns, we primarily investigate application portability between cloud vendors, rather than changes that are caused by adjusting the application to the cloud paradigm. Considering the portability promises of open cloud platforms, consequences of this migration type are less obvious.

Application portability between clouds not only includes the functional portability of applications but ideally also the usage of the same service management interfaces among vendors [152, 283]. This means that migration effort is not limited to code changes, which we also consider here, but includes effort for performing application deployment. Therefore, we put a special focus on effort caused by the deployment of the application in this study. We derive our main research questions from the preliminary results of previous work presented in Section 4.5.2:

***Research Question 4.1*:** *Is it possible to move a real-world application between different cloud platforms?*

***Research Question 4.2*:** *What is the development effort involved in porting a cloud-native application between cloud platforms?*

The utilized application, *Blinkist*, is a web application developed by Blinks Labs GmbH. The set of selected PaaS providers includes *IBM Bluemix*[143], *cloudControl*[144], *AWS Elastic Beanstalk*, *EngineYard*, *Heroku*, *OpenShift Online*, and *Pivotal Web Services*. More information on the used application and the PaaS providers is given in Sections 6.2.1 and 6.2.2. We analyze the feasibility of the migration in terms of portability and the effort for this task. Besides, we present a *Docker*-based deployment system that provides the ability of isolated and reproducible deployment measurements for several platforms. It enables us to compare deployments between different platforms for a particular application. Using this system, the study identifies key problems during migrations

[142] `https://12factor.net`
[143] IBM's Cloud Foundry offering Bluemix is now a part of IBM Cloud.
[144] cloudControl was shutdown due to bankruptcy end of February 2016.

Figure 6.1.: Migration Evaluation Process [165]

and quantifies differences between the platforms by distinctive metrics. In this study, we target the implementation portability [191, 284] of a migration execution, i.e., the application transformation and the deployment. We focus on functional portability of the application. Data portability must be investigated separately, especially since popular database technologies, such as NoSQL databases, impose substantial lock-in problems which are not directly related to the application execution environment provided by the cloud platform. What we do not measure is man-hours and organizational costs for a migration. This is hardly quantifiable without a larger study and influenced by a lot of other factors, e.g., knowledge and expertise of involved workers. With our results, we are able to compare migration efforts between different cloud platforms and to identify existing portability problems. The study also helps us to determine major deficiencies that can reasonably be targeted in the course of this thesis.

The remainder of this chapter is structured as follows: In Section 6.2, we describe our study design including details of the used application, the process of provider selection, the automation of deployment, and the measurement of deployment effort. Section 6.3 presents the results of our measurements and describes problems that occurred during the execution of the migrations. In Section 6.4, we review related work on cloud application migration. Section 6.5 discusses limitations and future work that can be derived from the results. Finally, Section 6.6 summarizes the contributions of the migration study.

6.2. Migration Study Design

The goal of this study is to analyze the task of migrating a cloud-based application with respect to the effort from the point of view of a developer/operator. To achieve this, we follow the process defined in Figure 6.1.

The first step is *migration planning*, which includes the analysis of application requirements and the selection of cloud providers. Next comes the *migration execution* for all providers, including code changes and application deployment. After manually migrating the application to the providers, the necessary steps and modifications are automated to enable a reproducible and comparable deployment among them. To compare the main effort drivers of the execution phase, i.e., code changes and application deployment, we define several metrics that allow a measurement of the tasks performed during the migration execution step. As discussed before, application portability between clouds not only concerns the functional portability of applications but also the interoperability

of service management interfaces between vendors [152, 283]. In our case, due to the use of open technologies and a cloud-native application, this effort is mainly associated with application deployment. With open technologies, we refer to software that is commonly developed as open source. It does not need to be distributed royalty-free, but has to be available within a large amount of software ecosystems. This is in contrast to proprietary software that is customized and tailor-made for a specific scenario implying a strong lock-in effect, e.g., a data store which is only available for a particular platform [380]. Today, open technologies and open standards are key for delivering portability between software systems [74, 145, 210]. We can also observe this fact in the composition and overlap of software ecosystems among PaaS providers in our knowledge base's data. For the reasons above, in this study, we put a special focus on the effort caused by the deployment of the application, next to application code changes. In times of agile and iterative development paradigms, this implies that also the effort of redeployment must be considered [232]. Therefore, our deployment workflow includes a redeployment of an updated version of the study's application. To validate that the application is operating as expected in the platform environment, we can draw on a large set of unit, integration, and functional tests. As concluding step, we evaluate our findings in the *migration evaluation*. This includes a discussion of the measured results and about problems and differences between providers.

The primary focus of this study is on the migration execution and evaluation. The initial planning step can be largely assisted by the knowledge base and cloud brokering tool *PaaSfinder* presented in Chapters 4 and 5 that covers the details of provider brokering and application requirements matching.

6.2.1. Migrated Application

The application *Blinkist* is built by a Berlin-based mobile learning company launched in January 2013. It distills key insights from nonfiction books into fifteen-minute reads and audio casts. At the time of this study, Blinkist includes summaries of over 1,300 books in its digital library. Blinkist has a user count of more than 500,000 registered customers worldwide. The product is created by a team of 21 full-time employees and is available for Android, iPhone, iPad, and web. See Figure 6.2 for three product shots, showing the digital library, the entry page for a book, and a chapter of a book summary, a so-called *blink*.

In our study, we target the web application[145], which is built in Ruby on Rails[146], a Ruby-based web application framework. The high-level architecture relevant for this study can be seen in Figure 6.3. The figure depicts a subset of important components of the overall application architecture. Parts of the system that are not relevant to the service provision for the user, such as

[145]The recent application version can be accessed at `https://www.blinkist.com`.
[146]`http://rubyonrails.org`

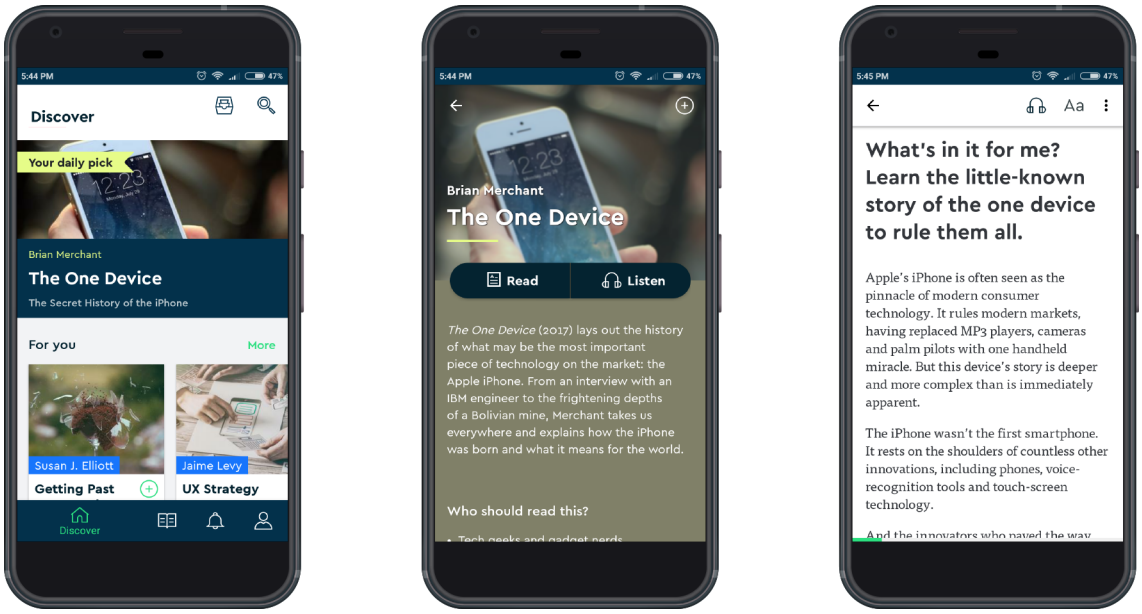


Figure 6.2.: Blinkist Product Shots

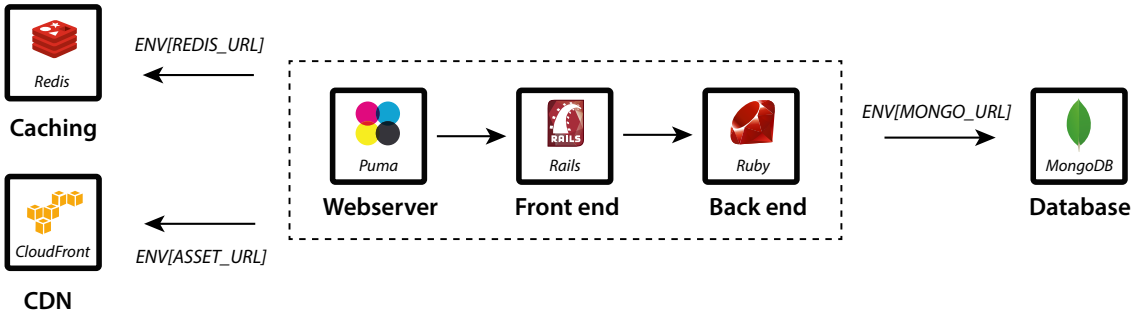


Figure 6.3.: Blinkist Web Application Architecture

administration services or business analytics, are omitted in the overview. The front end is a Rails 4 application with access to decoupled business logic written in Ruby. The application uses a MongoDB[147] NoSQL database for persistence of user data and book summaries. Moreover, page caching and distribution of static application assets, e.g., images, is implemented via a Redis[148] in-memory data store and Amazon's CloudFront[149] Content Delivery Network (CDN). The application configuration to resource handles such as the database is stored inside environment variables as defined by the twelve factor methodology. The web interface is run with at least two application instances in parallel, hosted by a Puma[150] web server.

The study uses Blinkist's application version from May 2014 for the initial deployment and a subsequent release after a major code sprint for redeployment.

[147] https://www.mongodb.com
[148] https://redis.io
[149] https://aws.amazon.com/en/cloudfront
[150] https://puma.io

Table 6.1.: Blinkist Code Statistics Front End

Deployment Version

Language	Files	Blank	Comment	Code
CSS	11	1531	350	7345
JavaScript	50	1564	2412	6550
Sass	116	1559	69	5838
Ruby	184	1929	505	5279
ERB	109	455	2	2839
HTML	17	111	121	2533
YAML	113	76	260	1269
Markdown	3	79	0	212
XML	1	0	0	4
$\sum$	604	7304	3719	31869

376 files changed, 8759 insertions (+), 2869 deletions (-)

Redeployment Version

Language	Files	Blank	Comment	Code
CSS	11	1551	350	7445
JavaScript	62	1841	2636	7813
Sass	129	1868	75	7019
Ruby	170	2086	566	5551
ERB	153	769	2	4223
HTML	18	119	138	2609
YAML	137	76	260	1597
Markdown	3	55	0	102
XML	1	0	0	4
$\sum$	684	8365	4027	36363

The application part totals about 50,000 Lines of Code (LOC). See Tables 6.1 and 6.2 for a detailed breakdown of code composition of the web front end and the business logic back end.[151]

The web project mainly consist of ERB templates in combination with JS and CSS. ERB is a templating system implementation that is used to embed Ruby into HTML documents that are interpreted by the web server. Syntactically Awesome Style Sheets (Sass)[152] is a style sheet language that acts as a CSS preprocessor which is interpreted into CSS style sheets. It extends CSS with features like variables, nesting, or inheritance that aim at making large and complex CSS projects easier to maintain. The large amount of YAML files contain the localization of the front end. The back end code is written in pure Ruby. The changes between the two application versions not only affect the code base, as depicted in the tables, but also update, add, or remove external code dependencies of the application. In total Blinkist uses 45 (49) dependencies in

[151] Code statistics are generated via `https://github.com/AlDanial/cloc` and Git.
[152] `https://sass-lang.com`

Table 6.2.: Blinkist Code Statistics Back End

Deployment Version

Language	Files	Blank	Comment	Code
Ruby	398	3331	372	12804
Markdown	2	29	0	44
YAML	1	1	0	22
$\sum$	401	3361	372	12870

277 files changed, 2917 insertions (+), 6900 deletions (-)

Redeployment Version

Language	Files	Blank	Comment	Code
Ruby	470	4278	422	15759
Markdown	2	30	0	44
YAML	1	1	0	22
$\sum$	473	4309	422	15825

the web project and 10 (10) inside the back end service. All changes to static assets like images are not mapped by the code statistics.

6.2.2. Vendor Selection

As hosting environment for the application, we aim for a production-ready, public PaaS that supports horizontal application scalability. The application itself depends on support for Ruby 2.0.0 and Rails 4. The necessary services and data stores are provided by external service providers (see Figure 6.3).

The decision on possible candidates for the application can be assisted by *PaaSfinder* as presented in Chapters 4 and 5. The underlying knowledge base is founded on a taxonomy describing essential components and capabilities of PaaS providers including the necessary requirements described before. The assumption of the matching strategy is that an application can be ported between providers that support the required application dependencies natively. This approach compensates the lack of commonly accepted portability standards in the cloud context. Listing 6.1 shows the desired profile, as defined in Chapter 4, for the application requirements of the case study application.

The broker tool allows filtering from the multitude of available platform offerings based on the defined ecosystem capabilities and requirements. The filtering can either be done manually via a web interface or in an automated fashion by querying the RESTful broker API with the data from Listing 6.1. With the help of our tool, we were able to filter from a total of 75 offerings to a candidate set of 22 offerings, based on the chosen platform capabilities and runtime support. This means that 70 % of the providers have already

Listing 6.1.: Blinkist's Application Requirements Profile

```
{
  "status": "production",
  "scaling": {
    "horizontal": true
  },
  "hosting": {
    "public": true
  },
  "runtimes": [
    {
      "language": "ruby",
      "versions": [ "2.0.0" ]
    }
  ],
  "frameworks": [
    {
      "name": "rails",
      "versions": [ "4.*" ]
    }
  ]
}
```

been excluded due to ecosystem portability mismatches, i.e., failing support for specific requirements. Thereafter, we also filtered out vendors that are based on the same base platform technology, e.g., Cloud Foundry, except for one control pair (Pivotal and Bluemix). The expectable compliance between the control pair is used to validate the consistency of the proposed effort metrics. The final selection of the seven providers, presented in Tables 6.3 and 6.4, was based on a concluding relevance assessment of the remaining offerings. Here, we tried to factor in the providers' impact and popularity within the market. Thereby, our selection can be reinforced by the fact that OpenShift, Heroku, and Cloud Foundry were the top three vendors queried on *PaaSfinder* as recorded by our application metrics. We also included the strong German contender cloudControl that had just acquired dotCloud (the founders of Docker) at that time. For reasons of comparability, we selected equal instance configurations and geographical locations among the different vendors, grouped by virtualization technology. At the time when the case study was executed, this was possible for all but two vendors, i.e., cloudControl and Bluemix, which only supported application deployment in Dublin, IE, and respectively Dallas, US.

As a first observation in Tables 6.3 and 6.4, we can see that there are substantial price differences between the providers. Prices are based on equivalent production-grade configurations dependent on the technology descriptions and

Table 6.3.: PaaS Vendors and Selected Configurations (Container-Based)

	Heroku	**cloudControl**	**Pivotal Web Services**	**Bluemix**	**OpenShift**
Company	Salesforce	cloudControl	Pivotal Software	IBM	RedHat
Initial Release	2007	2009	2013	2014	2011
Platform	Proprietary	Proprietary	Open Source	Open Source	Open Source
Isolation	Container	Container	Container	Container	Container
RAM (instance)	512 MB	512 MB	512 MB	512 MB	512 MB
Geo location	Virgina, US	Dublin, IE	Virgina, US	Dallas, US	Virgina, US
Pricing	$ 0.035/h	$ 0.04/h	$ 0.015/h	$ 0.035/h	$ 0.025/h
Σ(2 instances/month)	$ 50	$ 50.10	$ 21.60	$ 24.15	$ 36

Table 6.4.: PaaS Vendors and Selected Configurations (VM-Based)

	Elastic Beanstalk	**EngineYard**
Company	Amazon	EngineYard
Initial Release	2011	2006
Platform	Proprietary	Proprietary
Isolation	Virtual Machine	Virtual Machine
RAM (instance)	3.75 GB	3.75 GB
Geo location	Virgina, US	Virgina, US
Pricing	$ 0.067/h	$ 0.12/h
Σ(1 VM/month)	$ 48.24	$ 86.40

specifications of the providers.[153] Nevertheless, our tests reveal performance differences which are not included in this consideration. Currently, a price-performance value can hardly be investigated by a customer in advance. In general, container-based PaaS are cheaper to start with than VM-based ones. Still, instance performance is lower with respect to the technological setup. When looking at instance prices of container-based PaaS per hour, the most expensive provider charges over two and a half times more than the cheapest one. However, it is common among PaaS providers that there is a contingent of free instance hours per month included. Therefore, the total amount of savings is dependent on the number of running application instances. For example, the differences between Bluemix and cloudControl, which are caused by a higher free hour quota of Bluemix, will level up with increasing instance count. Pricing among VM-based offerings is more complex with dedicated pricing for platform components like IP services, bandwidth, or storage, which makes it difficult for customers to compare the prices of different providers.

6.2.3. Deployment Automation

In this study, we want to measure the effort of a customer migrating an application to specific platforms. As discussed, in our case, this effort is mainly

[153] Pricing is based on selected RAM usage, respectively instance type. 720 h/month usage estimate. No additional bandwidth and support options included. Free quotas deducted. Dollar pricing of cloudControl is taken from their US subsidiary dotCloud. Date: 11/11/2015.

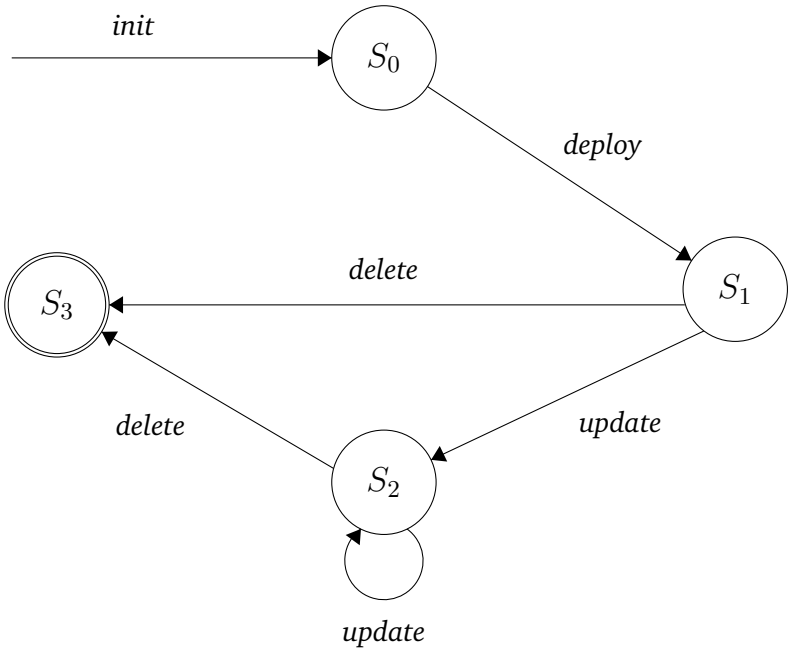


Figure 6.4.: Unified Application Life Cycle States

associated with application deployment. To measure and compare this effort, we automate the deployment workflows by using the provider's CLI tools. This kind of interaction is supported by the majority of providers and therefore seems appropriate for a comparative measurement in contrast to other mechanisms like APIs. Although all selected providers offer client tools, not all steps can be automated for every provider. The amount of manual steps via other interfaces like a web UI is denoted explicitly. The automation of the workflows helps to better understand, measure, and reproduce the presented results. We implement an automatic deployment system, called *PaaSYard*[154], that works similar for every provider and prevents errors due to repeatable deployment workflows. This enables a direct comparison of deployments between providers.

PaaSYard consists of a set of modules which automate the deployment for specific providers. To abstract from differences between providers, we define a unified interface paradigm that each module has to implement to realize the application life cycle depicted in Figure 6.4. To conform to the interface, every module needs to implement one `init`, `deploy`, `update`, and `delete` script that encapsulates necessary substeps. This approach offers a unified and provider-independent way to conduct deployment. Accordingly, the `init` script must execute all steps that are required to bootstrap the provider tools for application deployment, e.g., install the client tools. The `deploy` script contains the logic for creating a new application, including application and platform configuration. Typically, this involves authenticating with the provider's platform, creating a new application space, setting necessary environment variables, deploying the application code, and finally verifying the availability of the remote applica-

[154]The source code is available at `https://github.com/stefan-kolb/paasyard`.

Listing 6.2.: Deployment Script for Heroku

```
#!/bin/bash
echo "-----> Initializing application space..."
# authentication
heroku login <<END
$HEROKU_USERNAME
$HEROKU_PASSWORD
END
# create app space
heroku create $APPNAME
# environment variables
heroku config:set MONGO_URL=$MONGO_URL
                  REDIS_URL=$REDIS_URL
                  ASSET_URL=$ASSET_URL

echo "-----> Deploying application..."
git push heroku master

echo "-----> Checking availability..."
./is_up "https://$APPNAME.herokuapp.com"
```

tion. Updates to an existing application are performed inside the `update` script. Finally, the `delete` script is responsible for deleting any previously created artifacts and authentication information from the particular provider. Any necessary provider-specific artifacts, such as deployment manifests or configuration files, must be kept in a subfolder adjacent to the deployment scripts and will be merged into the main application repository by *PaaSYard* before any script execution. The deployments are automated via *Bash* scripts. User input is inserted automatically via *Here Documents*[155] or *Expect*[156] scripts. This guarantees that user input is supplied consistently for every deployment. As an example, Listing 6.2 shows the `deploy` script for Heroku.

First, the script authenticates the CLI with the platform. Any provider credentials and other variables, e.g., `$HEROKU_USERNAME`, used inside the scripts must be defined in a configuration file. After the login, a new application space is created and necessary environment variables referencing the external caching and database services are set. Next, the application code is pushed to the platform via a Git remote which automatically triggers the build process inside the platform. Finally, a helper script polls the remote URL until the application is up and successfully responds to requests.

Since the system is intended to be used for independent deployment measurements, we must make sure that we achieve both local and remote isolation between different deployment runs. Consequently, the previously described set of scripts must allow for an application installation in a clean platform environment and reset it to default settings by running the `delete` script. The

[155] `https://linux.die.net/man/1/bash`
[156] `https://linux.die.net/man/1/expect`

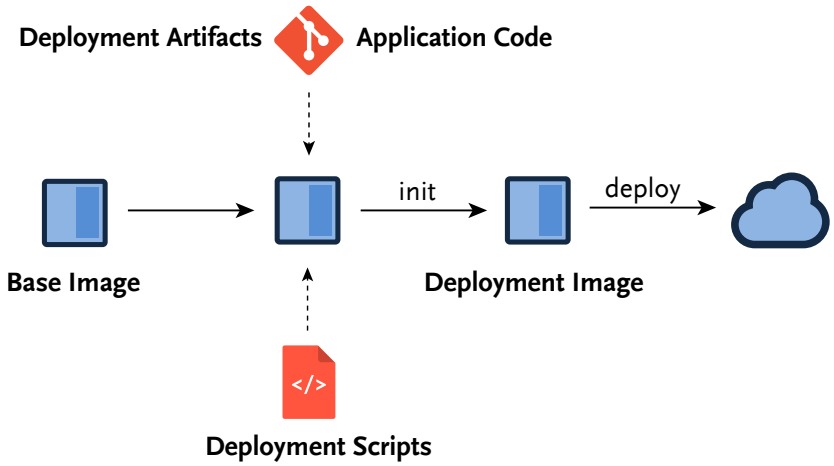


Figure 6.5.: PaaSYard Deployment Workflow

set of scripts must ensure that subsequent deployments are not influenced by settings made to the remote environment by previous runs. As the different build steps and deployment tools possibly write configuration files, tokens, or host verifications to the local file system, we need to enhance our approach with extra local isolation. Thus, the deployments are run inside *Docker* containers for maximum isolation between different deployments. Docker provides lightweight, isolated containers by means of an abstraction layer of operating-system-level virtualization features.[157] Hence, each script invocation uses a fresh container instance to avoid any interferences at OS level.

A graphical overview of the deployment workflow of *PaaSYard* can be seen in Figure 6.5. For each container, a base image is used that only consists of a minimal Ubuntu[158] Linux installation, including Python and Ruby runtimes. This base image can be varied, if one does not want to include specific libraries or runtimes pre-installed. From the base image, a deployment image is created that bootstraps the necessary provider tool dependencies. This is achieved by executing the `init` script of each provider module inside the base image, which results in a new container image. Additionally, the application code and the deployment artifacts are directly merged into a common repository. This is done to avoid additional bootstrapping before each deployment, which could influence the timing results of the deployment run. The resulting image can be used to deploy the code to different providers from every Docker-compatible environment via a console command. For convenience, the tool additionally provides a CLI script that handles the invocation of the different deployment scripts.

[157] See `https://www.docker.com/whatisdocker` for more details.
[158] `https://www.ubuntu.com`

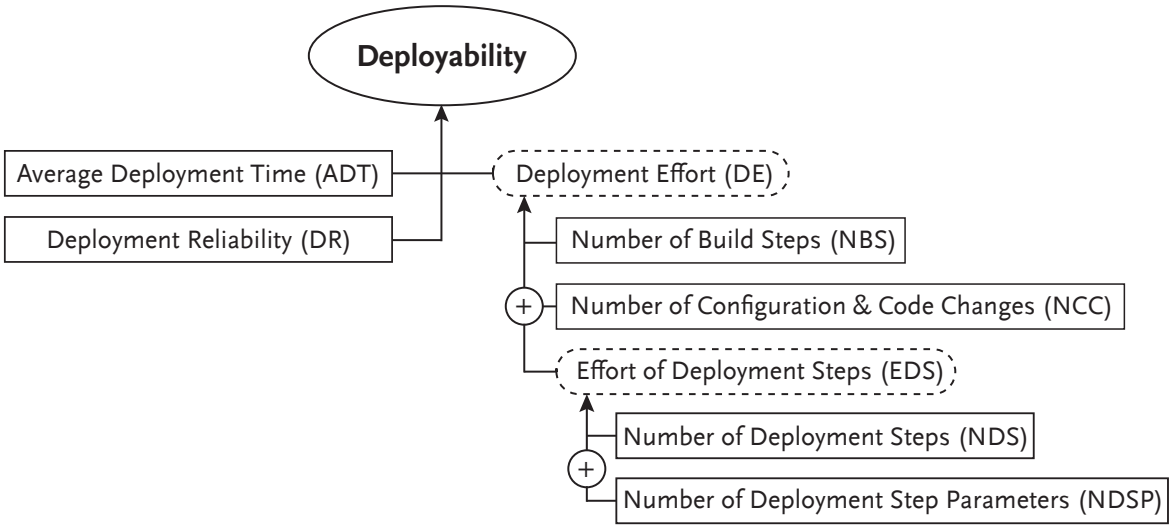


Figure 6.6.: Deployment Metrics Framework

6.2.4. Measurement of Deployment Effort

As discussed before, migration effort in our case translates into *effort for installing* the application on a new cloud platform, i.e., into *effort for deploying* the application. Hence, we need metrics that enable us to measure installability or deployability. Lenhard, Harrer, and Wirtz [207] proposed and validated a measurement framework for evaluating these characteristics for service orchestrations and orchestration engines, based on the ISO/IEC SQuaRE quality model [159]. Despite the differences between service orchestrations and cloud applications, this framework can be adapted for evaluating the deployability of applications in PaaS environments by modifying existing metrics and defining new ones. A major benefit of the chosen code-based metrics is their reproducibility and objectiveness. Currently, we do not consider human factors, e.g., effort in terms of man-hours. Such aspects are hardly quantifiable without a larger empirical study and influenced by a lot of other factors, like the expertise of the involved workers. However, it is possible to introduce such aspects by adding weighting factors to the metrics computation, as for instance done by Sun and Li [348].

As cloud platforms are preconfigured and managed environments, there is no need to consider the installability of the environment itself, as done in [207]. Instead, the focus lies on the deployability of an application to a cloud platform. Figure 6.6 outlines the adapted framework for deployability. We capture this quality attribute with the direct metrics Average Deployment Time (ADT), Deployment Reliability (DR), Number of Deployment Steps (NDS), Number of Deployment Step Parameters (NDSP), Number of Configuration & Code Changes (NCC), and the Number of Build Steps (NBS). The last four metrics are aggregated to an overall Effort of Deployment Steps (EDS) and Deployment Effort (DE). All metrics but ADT and DR are classic size metrics in the sense of

Briand et al. [55]. This means, they are nonnegative, additive, and have a null value. They are internal metrics that can be computed by statically analyzing code artifacts and are defined on a ratio scale. ADT and DR are external metrics, since they are computed by observing execution times and reliability. ADT is defined on a ratio scale and DR is defined by the interval scale of $[0; 1]$. The following paragraphs introduce the metrics in detail.

Average Deployment Time

This metric describes the average duration between the initiation of a deployment by the client and its completion, making the application ready to serve user requests. This can be computed by timing the duration of the deployment on the client side and by repeating this process a suitable number of times. To compute the average amount of time spent until an application is deployed, we use the median, i.e., the 50$^{\text{th}}$ percentile, as measure of central tendency. The median is more robust to outliers which are common in our scenario than other central tendencies that would otherwise distort the overall picture. The following equation gives our definition of the ADT for an application a.

Definition 6.2 (Average Deployment Time (ADT))

$$ADT(a) = \begin{cases} time(s_{\frac{n+1}{2}}) & \text{if } n \text{ is odd} \\ \frac{1}{2}(time(s_{\frac{n}{2}}) + time(s_{\frac{n}{2}+1})) & \text{if } n \text{ is even} \end{cases}$$

- $d_1, \ldots, d_n$ *sequence of application deployments* 1 *to* n
- $time(d_i)$ *deployment time of deployment* d_i *for* $1 \leq i \leq n$
- $s_1, \ldots, s_n$ *sequence of application deployments* 1 *to* n *sorted by* $time(d_i)$
- $ADT(a)$ *is undefined for* $n = 0$

Deployment Reliability

Deployment reliability captures the reliability of an application deployment to a particular provider. It is computed by repeating the deployment a specified amount of times and by dividing the number of successful deployments (N^a_{succ}) of an application a with the total number of attempted deployments (N^a_{total}). If all deployments succeed, $DR(a)$ will be equal to one. Otherwise, the DR will lie inside the interval scale $[0; 1]$.

Definition 6.3 (Deployment Reliability (DR))

$$DR(a) = \begin{cases} 0 & \text{for } N^a_{total} = 0 \\ N^a_{succ}/N^a_{total} & \text{otherwise} \end{cases}$$

- N^a_{succ} *number of successful deployments of application* a
- N^a_{total} *total number of attempted deployments of application* a

Number of Deployment Steps

The effort of deploying an application is related to the amount of operations or steps that have to be performed for a deployment. In our case, deployment is automated, so this effort is encoded in the deployment scripts (see Section 6.2.3). A deployment step refers to a number of related programmatic operations, excluding comments or logging. The larger the amount of such steps, the higher is the effort.

Definition 6.4 (Number of Deployment Steps (NDS))

$NDS(a)$ *is the number of steps required for deploying application* a

- $NDS(a) \in \mathbb{N}$

Usually, there are different ways to deploy an application. Therefore, to determine this number, heuristic evaluation can be used [207, 255]. This translates to the examination and judgment of the implementation complexity of an application deployment by domain experts. In our study, we consistently focused on finding the most concise way in terms of step count, while favoring command options over nonportable deployment artifacts that may silently break the deployment on different platforms. Eventually, the experts count the number of steps inside the approved installation scripts to determine the NDS. Whereas the number of installation steps can possibly be computed automatically, the consistency and comparability between different deployments can only be achieved by involving human inspection. As an example, the value of NDS for the deployment script in Listing 6.2 sums up to $NDS(heroku) = 4$:

1. Authentication: `heroku login`
2. Create application space: `heroku create`
3. Set environment variables: `heroku config:set`
4. Deploy code: `git push heroku master`

Number of Deployment Step Parameters

The number of steps for a deployment is only one side of the coin. Deployment steps often require user input (variables in scripts) or custom parameter configuration that need to be set, thereby causing effort. We consider this effort with the metric deployment step parameters, which counts all user input and command parameters that are necessary for deployment. The NDSP can be determined in the same fashion as previously described for the NDS. To exemplify, the deployment script in Listing 6.2 uses six different variables and requires no additional command line parameters, resulting in $NDSP(heroku) = 6$.

Definition 6.5 (Number of Deployment Step Parameters (NDSP))

$NDSP(a)$ *is the number of configuration parameters required for deploying application* a

- $NDSP(a) \in \mathbb{N}_0$

Effort of Deployment Steps

The two direct metrics NDS and NDSP count the effort for achieving a deployment. Since they are closely related, we aggregate the two to the indirect metric EDS by summing them up. For our running example, this amounts to $EDS(heroku) = 10$.

Definition 6.6 (Effort of Deployment Steps (EDS))

$$EDS(a) = NDS(a) + NDSP(a)$$

- NDS *as defined in Definition 6.4*
- $NDSP$ *as defined in Definition 6.5*

Number of Configuration & Code Changes

The deployment of an application to a particular platform may require the construction of different vendor-specific configuration artifacts, i.e., deployment descriptors. This includes platform configuration files and files that adjust the execution of the application, e.g., a *Procfile*[159]. Again, the construction of these files results in effort related to their size [207]. For all configuration files, every nonempty and noncomment line is typically a key-value pair with a configuration setting, such as an option name and value, needed for deployment. We consider each such line using a Lines of Code (LOC) function. Furthermore, it might be necessary to modify source files to solve incompatibilities between

[159] `https://devcenter.heroku.com/articles/procfile`

different platforms. This can be due to unsupported dependencies that must be adjusted, e.g., native libraries or middleware versions. Any of those changes are measured via a LOC difference function (LOC_{diff}). The idea behind this operation is similar to the `git diff`[160] function, known from the popular version control system *Git*. It shows differences between the existing committed code and the local changes to it via new, removed, and changed lines. In such a way, our LOC difference function aggregates the amount of touched lines between two code files. The sum of the size of all configuration files and the amount of code changes corresponds to the configuration and code changes metric. For an application a that consists of the configuration files $file_i, \ldots, file_{N_{conf}}$ and the code files $file_j, \ldots, file_{N_{code}}$, along with their platform-adjusted versions $file'_j, \ldots, file'_{N_{code}}$, the NCC can be computed as follows:

Definition 6.7 (Number of Configuration & Code Changes (NCC))

$$NCC(a) = \sum_{i=1}^{N_{conf}} LOC(file_i) + \sum_{j=1}^{N_{code}} LOC_{diff}(file_j, file'_j)$$

- $file_i, \ldots, file_{N_{conf}}$ *is the sequence of configuration files needed for the deployment for* $1 \leq i \leq N_{conf}$*;* $N_{conf} \in \mathbb{N}_0$ *is the number of these files*
- $file_j, \ldots, file_{N_{code}}$ *is the sequence of code files that need to be altered for a successful deployment for* $1 \leq j \leq N_{code}$*;* $N_{code} \in \mathbb{N}_0$ *is the number of these files*
- $file'_j, \ldots, file'_{N_{code}}$ *is the sequence of platform-adjusted code files*
- $LOC(file) : File \rightarrow \mathbb{N}_0$ *is a function that returns the number of nonempty and noncomment lines of a file*
- $LOC_{diff}(file_j, file'_j) : (File \times File) \rightarrow \mathbb{N}_0$ *is a function that returns the number of line differences of nonempty and noncomment lines between the files* $file_j$ *and* $file'_j$

Number of Build Steps

Another effort driver in traditional application deployment is the number of build steps, i.e., source compilation and packaging of artifacts into an archive [207]. This is less of an issue for cloud platforms, where most of this work can be bypassed with the help of platform automation, e.g., buildpacks. At best, a direct deployment of application artifacts is possible ($NBS(a) = 0$), shifting the responsibility of package construction to the platform. For some platforms and runtime languages it is still necessary, which is why we capture it in the same fashion as the number of deployment steps.

[160] `https://git-scm.com/docs/git-diff`

Definition 6.8 (Number of Build Steps (NBS))

$NBS(a)$ *is the number of build steps required for preparing application* a *for deployment*

- $NBS(a) \in \mathbb{N}_0$

Deployment Effort

Finally, to provide a comprehensive indicator for effort associated with deployment, we define an aggregated deployment effort, computed as the sum of the previous metrics: $DE(a) = EDS(a) + NCC(a) + NBS(a)$. It is arguable to weight the severity of different deployment efforts by introducing a weighting factor in this equation. As we cannot determine a reasonable factor without a larger study, they are considered as coequal here.

Definition 6.9 (Deployment Effort (DE))

$$DE(a) = EDS(a) + NCC(a) + NBS(a)$$

- EDS *as defined in Definition 6.6*
- NCC *as defined in Definition 6.7*
- NBS *as defined in Definition 6.8*

6.3. Migration Evaluation

In Section 6.3.1, we first describe the execution of the measurements, followed by a presentation, discussion, and interpretation of the results in Section 6.3.2, and a summary in Section 6.3.3.

6.3.1. Execution of Measurements

As part of our migration experiment, we need to compute values for the deployment metrics from the preceding Section 6.2.4. The timing for the ADT of an individual deployment run can be calculated by prefixing the script invocation with the Unix `time` command[161], which returns the elapsed real time between the invocation and termination of the command. One distinct test is the execution of a sequence of an initial deployment, followed by an application redeployment, and concluded by the deletion of the application. Each provider was evaluated via 100 runs of this test. Every successful run was included in

[161] `https://linux.die.net/man/1/time`

the ADT calculation and the amount of successful and failed runs were used to compute deployment reliability. Runs with deployment failures that could not be attributed to the respective platforms, e.g., temporary unavailability of external resources such as the central *RubyGems*[162] dependency repository, were excluded from the calculation. EngineYard forms an exception in the measurement setting, with a total of 50 runs. The reason for this is that the deployment could not be fully automated and each run involved manual steps. The measurements were conducted at varying times during workdays, to simulate a normal deployment cycle inside a company. To minimize effects of external load-induced influences, e.g., *RubyGems* mirror, on the measurement, the deployments were run in parallel. The significance of potential problems can be further attenuated as we are not primarily looking for exact times but magnitudes that can show differences between providers. Such differences are separately identified with the help of significance tests. All deployments were measured with a single instance deployment at first, i.e., without application scaling. The values for each metric were evaluated and validated by a peer review inside the research group. The gathered metrics can be seen in Tables 6.5 and 6.6.

Even though we could successfully migrate the application to all but one provider, a substantial amount of work was required. Whereas the number of source code changes might look low, the effort for detecting problems and finding an appropriate solution is not to be underestimated. Besides the captured effort values, additional important obstacles are incomplete documentation of the providers and missing direct instance access for debugging, especially with container-based PaaS. Even with this common kind of application, it was difficult to get the application running and compromises with certain technology setups, e.g., web servers, were needed. Whereas some of these problems are to be expected and can only be prevented by unified container environments, major parts of the interaction with the system should be homogenized by, e.g., unified management interfaces.

During the case study, a number of bugs had to be fixed inside the cloud platforms. In total, we discovered four confirmed bugs on different platforms that prevented the application from running correctly. The majority was related to the bootstrapping of the platform environment, e.g., server startup and environment variable scopes, and could be resolved by the providers promptly. As a downside, one provider supported a successful deployment but did not allow us to run the application correctly, due to an internal security convention that prevented the database library from connecting to the database. These issues show that even with common application setups, cloud platforms cannot yet be considered fully mature.

[162] `https://rubygems.org`

Table 6.5.: Deployment Efforts

	Heroku	cloudControl	OpenShift	Pivotal	Bluemix	Beanstalk	EngineYard
DR	0.96	0.72	0.78	1	0.89	0.99	1
ADT	6.75 min	9.13 min	8.42 min	5.83 min	7.03 min	15.94 min	28.44 min
EDS	10	15	24	17	17	12	23
NDS	4	5	6	6	6	2	8
Automated	4	5	6	6	6	2	4
Manual	0	0	0	0	0	0	4
NDSP	6	10	18	11	11	10	15
NCC	1	1	0	1	1	40	7
Deployment artifacts	1	1	0	1	1	40	7
Application code	0	0	0	0	0	0	0
NBS	3	3	3	0	0	3	4
DE	14	19	27	18	18	55	34

Table 6.6.: Redeployment Efforts

	Heroku	cloudControl	OpenShift	Pivotal	Bluemix	Elastic Beanstalk	EngineYard
DR	0.96	1	0.97	1	0.93	0.98	0.96
ADT	6.69 min	5.71 min	7.41 min	5.73 min	6.61 min	8.71 min	8.25 min
EDS	1	2	1	2	2	1	2
NDS	1	1	1	1	1	1	1
NDSP	0	1	0	1	1	0	1

6.3.2. Effort Analysis

The following section describes the results of our case study in detail. Again, see Tables 6.5 and 6.6 for the captured metrics. We discuss the metric values and their implications and give insights into the problems that did occur during the migrations.

Effort of Deployment Steps

As a first result, we can state that although deployment steps are semantically similar among vendors, they are all carried out by proprietary CLI tools in no standardized way. This results in recurring effort for learning to use new tools for every vendor and to adapt existing automation. Figure 6.7 depicts the effort of deployment steps of all providers. On average, deployment takes 17 steps with a maximum spread of 14 and a standard deviation of 5. Some providers require more steps, whereas others require fewer steps but more parameters. Heroku, cloudControl, Pivotal, and Bluemix are driven by a similarly concise deployment workflow. We do indeed measure the same EDS for the control pair Bluemix and Pivotal which both use CF as base platform. This confirms the consistency of our metrics and the applied in-group review process. In contrast,

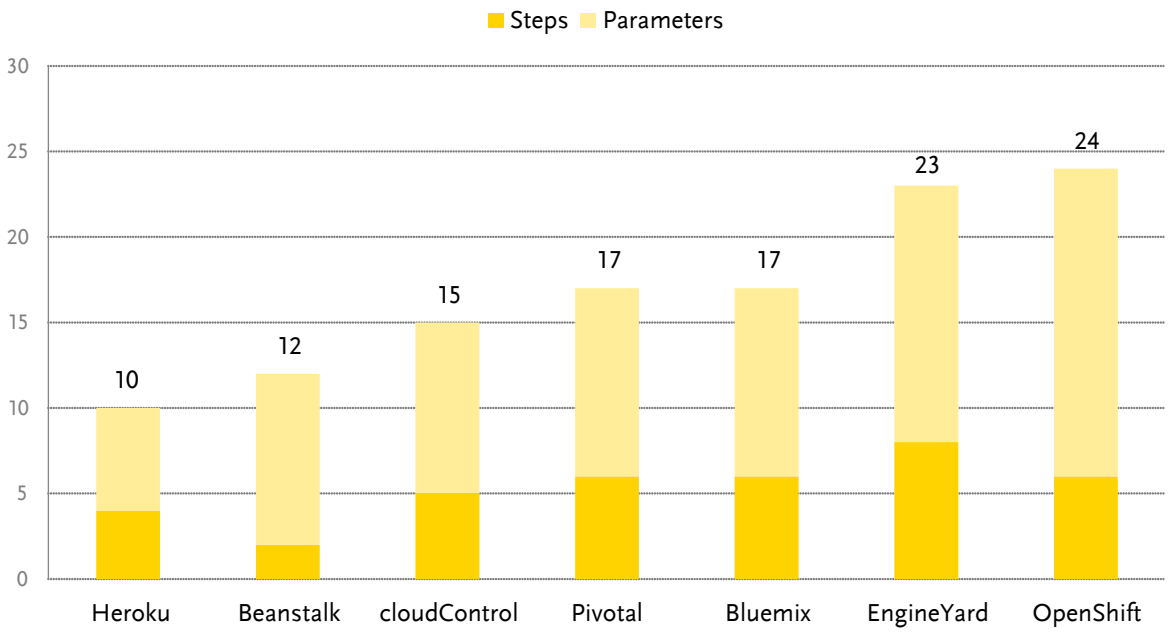


Figure 6.7.: Effort of Deployment Steps

OpenShift requires a cumbersome configuration of the initial code repository. Only the deployment for EngineYard could not be automated entirely. This is because the creation of VM instances must be initiated via a web interface, whereas the application deployment can be triggered by the client tools. As instance setup is normally performed once and not repetitively, this has less negative influence in practice than other steps would have. In the case of Elastic Beanstalk, the low EDS value of 12 is contrasted by a large configuration file. The majority of modern container-based PaaS reduce effort with respect to the EDS through an intelligent application type detection. Thereby, necessary settings for a particular application type will be automatically set in the platform environment and appropriate bootstrapping steps are initiated. In comparison, this must be explicitly configured in advance for the VM-based offerings. The EDS for a redeployment are roughly the same between vendors and only involve pushing the new code to the platform, typically via one console command, i.e., $EDS \in \{1, 2\}$.

Number of Configuration & Code Changes

Particularly the container-based platforms can be used with only few deployment artifacts (see Figure 6.8). Four out of five providers support a Procfile-based deployment for specifying application startup commands ($NCC = 1$). Whereas this compatibility helps to reproduce the application and server startup between those providers, it is a major problem with the others. Especially custom server configuration inside the Procfile, i.e., the Puma web server, is a source of portability problems between platforms. Two platforms only support a preconfigured native system installation of Puma and one does not support

the web server in combination with the necessary Ruby version at all. Moreover, the native installations can lead to dependency conflicts, if the provider uses another version than specified in the application's dependencies, resulting in inevitable code modifications. The only two providers for which more configuration is needed are both VM-based offerings. In the case of EngineYard, the deployment descriptor can be kept small in a minimal configuration. Additionally, in contrast to other providers, a custom recipe repository must be cloned to use environment variables. These variables have to be configured inside a script file. The recipes can be uploaded and applied to the server environment afterwards. Elastic Beanstalk proved to be more problematic to achieve a working platform configuration. We needed a rather large configuration file that modifies installed Linux packages, platform configuration values, and environment variables. Apart from that, we even had to override a set of server-side scripts to modify the *Bundler* dependency scopes and enable dependency caching.

In general, we tried to avoid the use of configuration files or proprietary manifests. If options were mandatory to be configured for a provider, where possible, this was done using CLI commands and parameters instead of proprietary manifests. In either case, the value of EDS and the size of configuration files is in a close relation with each other.

Most notably, for the case study's application, we could achieve portability without changing application code, solely by adapting the runtime environment, i.e., deployment configuration, application, and server startup. This is closely linked to the cloud-native application based on open technologies. Furthermore, all providers that did not support required technologies were excluded in the initial migration planning step. If the application made use of proprietary APIs or unavailable services, this would have caused a large amount of application changes. Apart from that, further tests showed that especially native Gems (Ruby code packages) cause portability problems between PaaS offerings. These Gems may depend on special system libraries that are not available in every PaaS system and cannot always be installed afterwards. An example for this is *Nokogiri*[163], a Document Object Model (DOM) parser that depends on libxml[164] system libraries. Buildpacks can help to prevent such problems by unifying the environment bootstrapping and also making it customizable if needed. This helps providers to better support special dependencies that would otherwise be hard to maintain due to the sheer amount of possibilities. There is a tendency that coverage of these dependencies is better in more specialized PaaS that only support one runtime language or expanded their portfolio from a particular language. Therefore, experience with a special language may in fact result in benefits for customers requiring more customized setups.

[163] `https://github.com/sparklemotion/nokogiri`
[164] `http://xmlsoft.org`

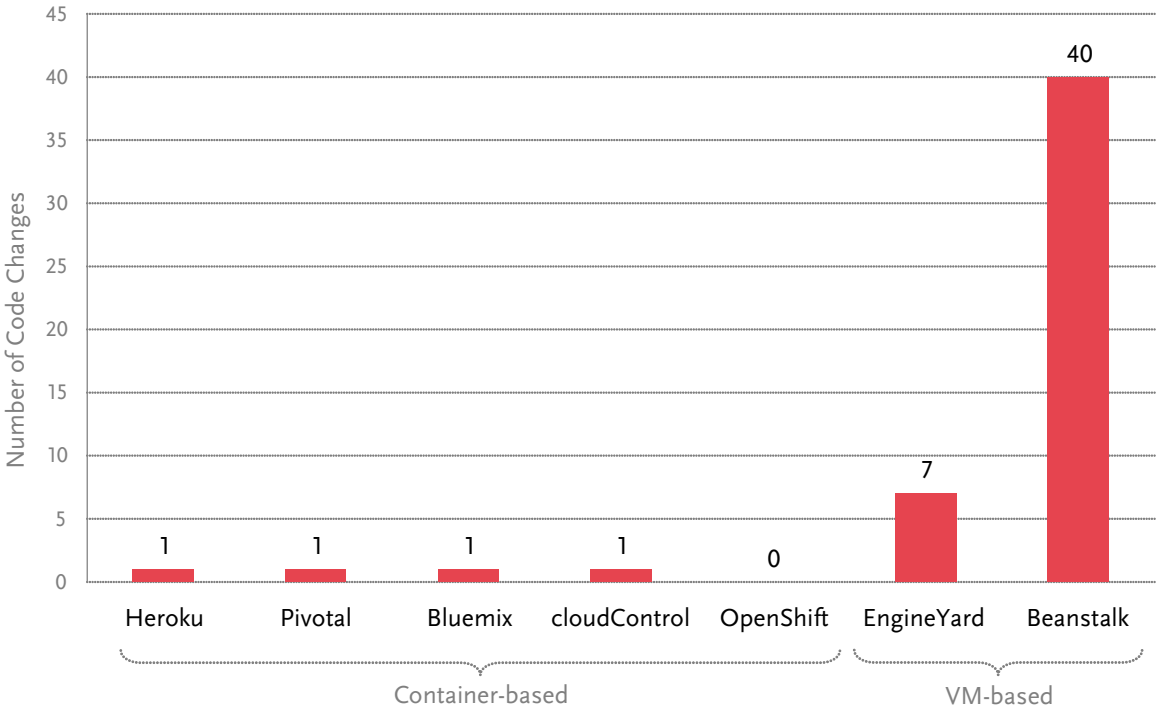


Figure 6.8.: Configuration and Code Changes

Number of Build Steps

The NBS for deployment is similar among providers. As sole packaging requirement, most providers mandate that the source code is organized in a *Git* repository, either locally or remotely ($NBS \in \{3, 4\}$). This is often naturally the case but must be counted as build effort. For the providers, it is convenient to manage the remote deployment of the application as a Git remote repository. Among others, this has the advantage that only the code differences are received via the Git system. Also, the Git commit hook can be intercepted and subsequently the necessary build and deployment steps can be triggered in the remote system.

Deployment Reliability

For some providers, we experienced rather frequent deployment failures, resulting in lower DR values, especially during the initial creation of applications. Often, these failures were provoked by recurring problems, e.g., permission problems with SSH keys or other platform configuration problems. From the descriptive data in Table 6.5, it seems that container-based systems experience more frequent failures than VM-based systems. To examine this assumption, we used a test to check if the amount of deployment successes for container-based systems is significantly lower.[165] Since deployment success is coded in a binary fashion, i.e., either *success* or *failure*, it is possible to apply a binomial test. We

[165] All statistical tests in this chapter were executed using the R software [298].

aggregated the amount of successes and failures for all container- and VM-based systems, respectively. Thereafter, we computed the binomial test, comparing the amount of successful runs for container-based systems (433) and the total amount of runs for container-based systems (497) to the success probability for VM-based systems (0.99). The null hypothesis is that both system types have an equal success probability. The alternative hypothesis is that the success probability of container-based systems is lower. In this case, the null hypothesis can be safely rejected with a p-value of $2.2e^{-16}$. As a result, it can be said that VM-based systems are more reliable in initial application deployment.

In the case of redeploying existing applications, we experienced fewer failures on average, resulting in higher DR values. We used a binomial test in the same fashion as in the previous paragraph to check if VM-based systems are still more reliable. This time, there were 423 redeployment runs for container-based systems in total, of which 411 were successful. At the same time, success probability for VM-based systems is 0.96. The p-value of 0.78 resulting from the binomial test does not reach a significant level and we cannot diagnose significant differences in the success probability for container- and VM-based systems. Also the reverse test, checking if the success probability of VM-based systems is lower, did not reach a significant level.

To sum up this paragraph, VM-based systems are significantly more reliable on initial deployment than container-based systems but this difference vanishes after the initial deployment phase. This can be explained by the anomalies associated with the platform configuration which we mentioned at the beginning of this section and shows room for improving the maturity of the platforms.

Average Deployment Time

Figure 6.9 visualizes the observed average deployment times. The mean of the deployment time is 11.65 min, but it deviates by 7.52 min. Differences between container-based offerings are small, only ranging within a deviation of 71 seconds. Container-based deployments are on average almost 3 times faster than VM-based platforms. This is partly because of the different startup times of containers and VMs. Mao and Humphrey [229] measured an average startup time for Amazon's EC2 VM instances of 96.9 seconds. Tests with the case study's instance configurations confirm this magnitude. This amount of time is contrasted with a duration of only a few seconds for creating a new container. Even when deducting this overhead from the measurements, the creation of the VM-based environments takes considerably longer than the one of container-based PaaS environments. Overall, a majority of the deployment time ($\approx$ 46 %) is spent for installing necessary application dependencies with *Bundler*[166]. Another considerably large part is the asset precompilation[167] of

[166] `https://bundler.io`

[167] `http://guides.rubyonrails.org/asset_pipeline.html`

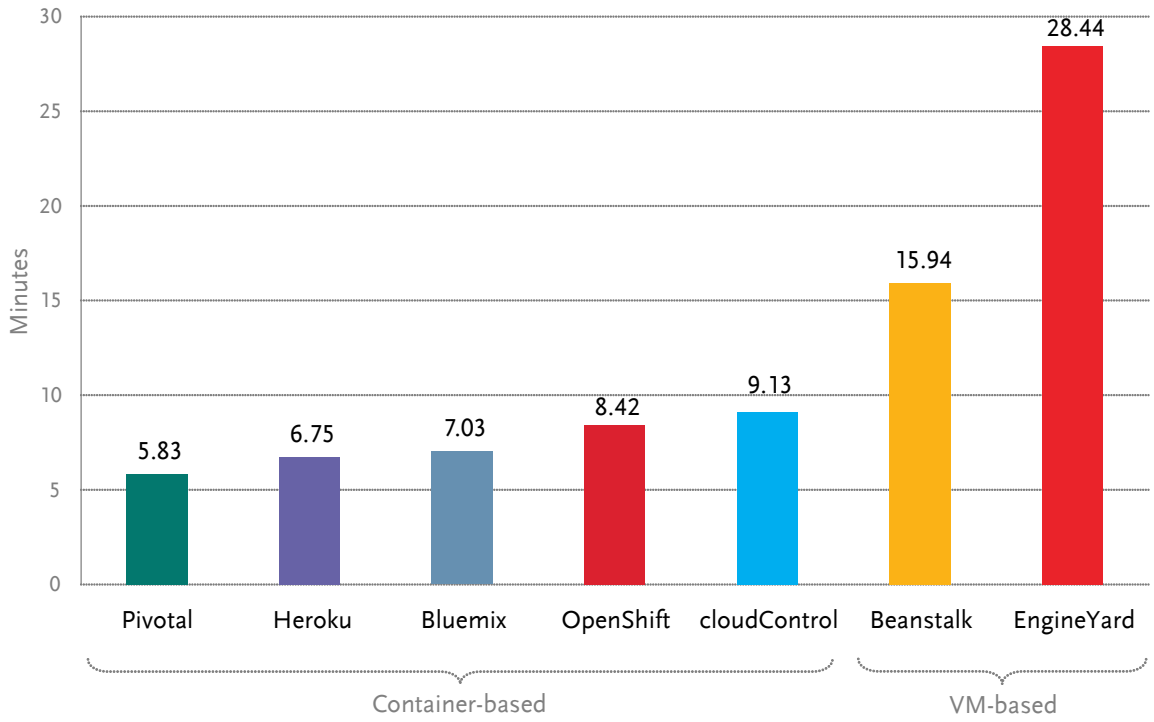


Figure 6.9.: Average Deployment Times

CSS files, JS files, and static assets (≈ 18 %). The remaining time (≈ 35 %) is consumed by other tasks of the build process and the platform configuration.

As before, we used statistical tests to confirm if the differences in deployment times between the two types of environments are significant. To begin with, we used the Shapiro-Wilk test [326] to check if deployment times follow a normal distribution. This can be safely rejected for both container-based and VM-based environments. Consequently, we applied a nonparametric test, the Mann-Whitney U test [227], for comparing deployment times. Our null hypothesis is that there are no significant differences in the deployment times of VM-based and container-based environments. The alternative hypothesis is that deployment times for VM-based environments are greater. The p-value resulting from the test ($N_{vm} : 150, N_{container} : 497, U : 64513$) is $2.2e^{-16}$. Thus, the null hypothesis can be clearly rejected. Container-based environments deploy significantly faster than VM-based environments.

The measured time values are also interesting for the case of redeployment. Besides code changes, the updated application version also includes new and updated versions of dependencies as well as asset changes (see Tables 6.1 and 6.2). In general, the redeployment times are lower than the initial deployment, which can be mainly attributed to dependency caching. In total, the installation of updated or new dependencies takes ≈ 50 % less time than in the initial deployment. During redeployment, there are more assets to process than in the initial deployment, resulting in a slightly longer precompilation time. For redeployment, all timings of the providers are in a close range (see Figure 6.10). Here, VM-based offerings catch up with container-based PaaS due

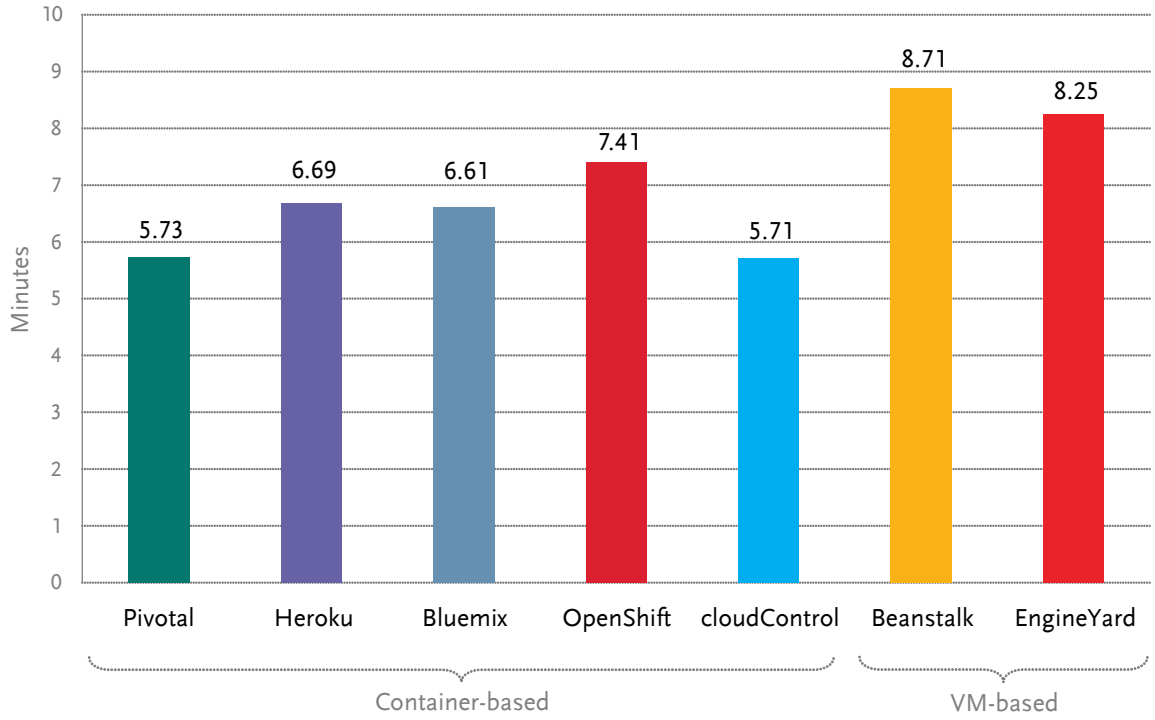


Figure 6.10.: Average Redeployment Times

to the absence of environmental changes. The average redeployment time for all offerings is 7.02 min and only deviates by 65 seconds. Some providers still benefit from a better deployment configuration, e.g., parallelized *Bundler* runs. Providers that were fast during the initial deployment confirm this tendency in the redeployment measurements. Based on these observations, it is interesting to check if there are still significant differences between VM-based and container-based environments when it comes to redeployment. We used the Shapiro-Wilk and Mann-Whitney U tests in the same fashion as above to confirm this. As before, the distribution of redeployment times is clearly nonnormal. The resulting Mann-Whitney U test ($N_{vm} : 146, N_{container} : 423, U : 53089.5$) again allows to reject the null hypothesis with a p-value of $2.2e^{-16}$ in favor of the alternative: Container-based environments also redeploy significantly faster than VM-based environments.

In a final step, we compared the deployment and redeployment times of all pairs of providers with each other using the Mann-Whitney U test as above. The aim of this comparison is to investigate if there is a performance gain in choosing a particular provider or if it is sufficient to decide between VM-based and container-based platforms. Put differently, we checked if there are significant differences among the container-based providers as well. We omit a detailed presentation of the results here due to the amount of comparisons necessary (each pair of providers needs to be tested for deployment and redeployment times, i.e., 42 combinations), but the results are unambiguous: There are significant differences in the deployment times of all providers, except for one combination of two container-based environments. Almost the same holds

for redeployment times, where significant differences can be diagnosed for all but two pairs of container-based environments. This observation also holds for our control pair Bluemix and Pivotal which both use CF as base platform. This indicates that platform and infrastructure configuration can also make a difference for customers even when just switching the hosting provider of the same PaaS system. To sum up this paragraph, even container-based environments differ significantly in their deployment performance and, thus, a performance gain can be obtained by using the fastest provider. Whereas this observation was only validated for application deployment in this study, it can be suspected that this also holds for application response times, which should be investigated separately.

Deployment Effort

Figure 6.11 shows that the values for total deployment effort are substantially different between the platforms, with a maximum spread of 41 and a standard deviation of 13. This leaves us with quantifiable evidence that there are portability issues between PaaS systems. The submetrics EDS, NCC, and NBS also show where this effort manifests itself the most among providers. The considerably large and varying amount of effort revealed by the EDS metric creates the need for a closer examination of the deployment steps among providers. Although these steps have an equal and clear functional definition among all providers, the effort varies notably. This shows that the deployment and other management tasks of PaaS systems need a closer look in terms of unification possibilities, which we investigate in the following Chapter 7. The majority of container-based platforms are within a close range to each other, only deviating by a value of 4, whereas VM-based platforms generally require more effort. When comparing both platform types, the additional effort for VM-based PaaS buys a higher degree of flexibility with the platform configuration if desired. Again, Pivotal's and Bluemix's values for DE are equivalent as expected. As there are no further code changes and build steps necessary after the application is adapted in the deployment step, the overall effort for redeployment is equal to the EDS among providers.

6.3.3. Summary

With the help of this study, we could answer both of our initial research questions. To begin with, it is possible to migrate a real-world application to the majority, although not to all, of the providers (*RQ 4.1*). Only one provider could not run our application due to a security restriction caused by a software fault, which cannot be seen as general restriction that prevents the portability of the application. However, we could not reproduce the exact application setup on all providers. We had to make trade-offs and changes to the technology setup,

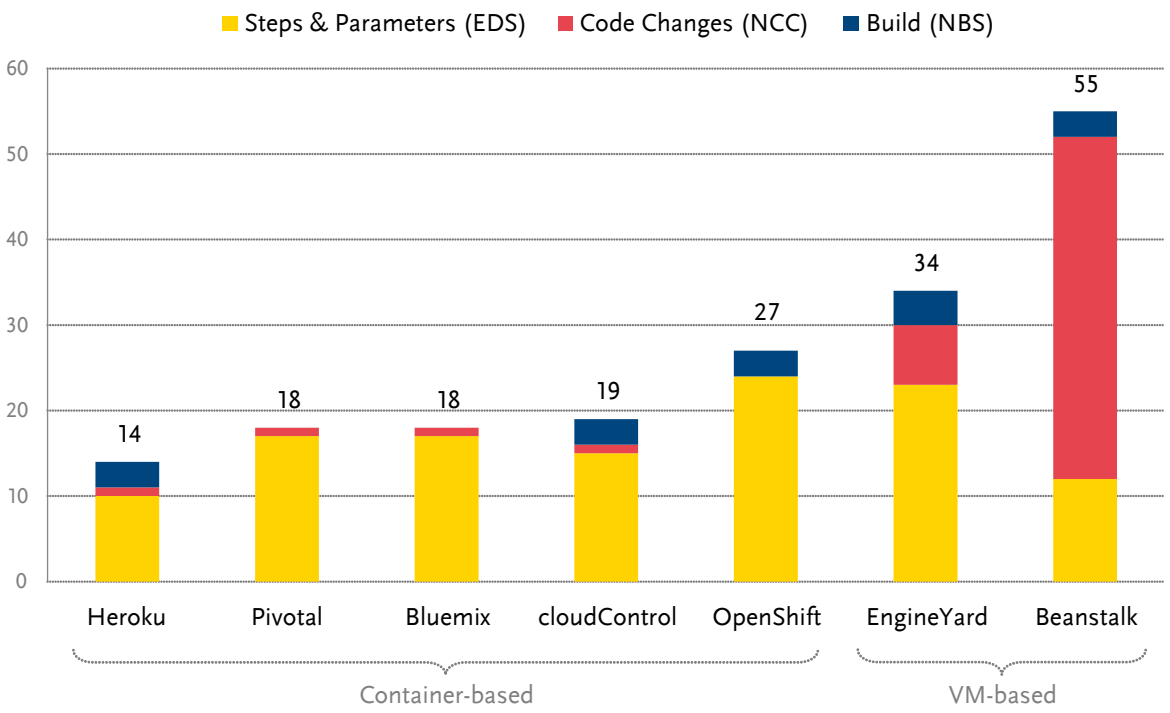


Figure 6.11.: Overall Deployment Effort

especially the server startup. Due to the automation of the migration, together with the presented toolkit and deployment metrics, we could quantify the effort of the migration (*RQ 4.2*). Our results show that there are considerable differences between the providers, especially between VM-based and container-based offerings. Our measurements provide multiple insights into migration effort. They quantify the developer effort caused by deployment steps and code changes and the effort created by deployment and redeployment times of an application.

6.4. Related Work

Jamshidi et al. [165] identified that cloud migration research is still in its early stages and further structured work is required, especially on cloud migration evaluation with real-world case studies. Whereas this structured literature review focuses on legacy-to-cloud migration, our own investigations reveal even more gaps in the cloud-to-cloud migration field. Most of the existing work is published on migrations between on-premises solutions and the cloud, primarily IaaS. Few papers focus on PaaS and even less on cloud-to-cloud migrations, despite the fact that portability issues between clouds are often addressed in literature [21, 85, 152, 286, 332]. Our study is a first step towards filling the identified gaps.

In Section 4.5.2, we already ported a small application between five PaaS providers in an unstructured way and gathered first insights into portability

problems and migration efforts. These initial results revealed that more research has to be carried out in a larger context. Likewise, a large proportion of existing cloud migration studies are confined to feasibility and experience reports, e.g., [67, 68, 368]. These studies typically describe a migration case study including basic considerations of provider selection and application requirements. Afterwards, they present a compilation of occurred problem points and necessary implementation changes during the application migration. Nevertheless, all of them omit a quantification or a more detailed comparison of migration effort.

A large part of more structured research on cloud migration prioritizes migration planning over the actual migration execution and observation. These studies focus on abstracting and supporting the migration process with decision frameworks rather than quantifying and examining actual migrations with metrics. Pahl and Xiong [273] introduce a generic PaaS migration process for on-premises applications. Their framework is mainly motivated by a view on different organizational and technological changes between the systems but not focused on a detailed case study or measurement. Others, like Hajjat et al. [138] and Bessera et al. [34], focus on minimizing cost aspects in their migration decision processes. A broader set of target variables is presented by Menzel and Ranjan [239] who propose an approach for cloud migration based on multi-criteria decision-making, specifically for use with web server migration.

In contrast to these abstract migration processes, several studies exist to assist automatic application inspection and transformation for migration execution. Sharma et al. [328] utilize a set of repositories containing patterns of technical capabilities and services for on-premises applications and PaaS offerings. By analyzing the source code as well as the configuration files, they try to extract application requirements and map them with the capabilities of target cloud platforms. The approach results in a report that describes which parts of the system can be migrated as is, which parts require changes, as well as a listing of those that cannot be migrated due to the limitations of the target platform. Similar to our study, Beslic et al. [35] discuss an approach for an application migration among PaaS vendors. Their scenario includes vendor discovery, application transformation, and deployment. In this regard, they propose to use pattern recognition via static source code analysis and automatic transformations between different vendor-specific APIs. Nonetheless, besides outlining their migration processes, none of the referenced papers quantifies the effort of the described transformations.

When it comes to the measurement of migration effort, most existing research is focused on estimating expected costs in an early phase of the development cycle, whereas we are evaluating factual changes after the implementation phase. Popular examples for generic algorithmic model estimation approaches are COCOMO [41] or Putnam [292]. However, such traditional algorithmic

models were developed in the context of software development projects, not for on-premises or cloud migration [348]. Nevertheless, Tran et al. [359] define Cloud Migration Point (CMP), a metric based on the accepted estimation model Function Points [8], for effort estimation of cloud migrations. Another study by Sun and Li [348] estimates expected effort in terms of man-hours for an infrastructure-level migration. Similar, Miranda et al. [245] conduct a cloud-to-cloud migration between two IaaS offerings that uses software metrics to calculate the estimated migration costs in man-hours rather than making migration efforts explicit. When assessing occurred effort, the focus is often on operational cost comparisons [15, 182, 183, 353], e.g., infrastructure costs, support, and maintenance or migration effort in man-hours [225, 358]. Solely Ward et al. [376] mention migration metrics related to the effort to create build automation and server provisioning time comparable to our deployment metrics.

6.5. Limitations and Future Work

As common for a case study, several limitations exist, which also provide potential areas of future work. First of all, the presented study was conducted with a particular Ruby on Rails application. In future work, we want to investigate the generalizability of the conclusions drawn, i.e., if they also apply for applications built with other runtime languages. Initial experiments back up the presented results and indicate that other languages potentially require an even higher migration effort. Due to their general applicability, our methodology and provided tools can be used to obtain results for other migration scenarios as well. Another main topic for further research is the unification of management interfaces for application deployment and management of cloud platforms. Despite semantically equivalent workflows, the current solutions are invariably proprietary at the expense of recurring developer effort when moving between vendors. We identified this topic as one of the major opportunities that need to be investigated further and discuss our results for a unification of core management functions of PaaS systems in Chapter 7. As revealed by our study, further work is needed regarding the unification of runtime environments between cloud platforms for improved portability of applications. Buildpacks are a promising step in that direction. Another need for research is the performance evaluation of cloud platforms. During our tests, we observed performance differences between the providers that are hard to quantify from the viewpoint of a customer at this time. However, this is vital for a well-founded cost assessment and, hence, should be investigated further.

6.6. Summary

In this chapter, we conducted and evaluated the migration process for a real-world application among seven cloud platforms. As a first step, we examined the feasibility of the application migration by manually porting the application to the platforms. We were able to move the application to a majority of providers but were forced to make trade-offs and changes to the technology setup. During this process, we discovered existing problems regarding the unification of management interfaces and platform environments. To allow for a comparable measurement of the effort involved in the migration process, we presented *PaaSYard*, a Docker-based deployment system that is able to deploy source code to different platforms via isolated containers. *PaaSYard* also includes a small abstraction layer for unified creation, deployment, and deletion of applications throughout the providers. With the help of the tool, we evaluated the deployment effort in terms of duration and amount of necessary steps. This includes a comparison of deployment operations and artifacts between the providers, aggregated to different formal effort metrics. The results show that there are major differences between the providers and the associated effort of the migration. Thereby, the case study gives evidence for portability issues between PaaS providers and enables to quantify them with the presented effort metrics. In general, VM-based platforms require more effort than container-based platforms, which is caused to some extent by the flexibility of the environment configuration. Within the study, we identified problems that prevented the portability of the application between providers and gave suggestions how they can be avoided or solved. The results show that despite trying to design applications as vendor-neutral as possible, the unification of runtime environments and management interfaces between cloud vendors is an important topic.

7. Unified Cloud Application Management

Parts of this chapter have been taken from [190, 312].

In this chapter, RQ 5 *("How to unify application management interfaces among cloud platforms?")* is supported.

7.1. Motivation

Even though the cloud, including PaaS, is said to grow massively over the years [38, 116], there are also plenty of concerns acting as market barriers or preventing further adoption. As we already outlined, a major concern is the lack of standards among cloud providers which hinders portability and fosters the chances of lock-in effects caused by, e.g., incompatible technologies or proprietary interfaces. Associated application migration costs do not only occur when voluntarily switching the provider but can also arise rather unexpectedly in case of takeovers or the bankruptcy of a provider, making the provider change inevitable. Acquisitions[168], bankruptcy[169], and business pivots[170] over the last years show that the PaaS market is under consolidation which highlights that such circumstances are likely [25, 231]. To enable a truly competitive market and unfold the full potential of cloud services, portability and interoperability between offerings must be enhanced [262].

Thereby, application portability between clouds not only includes the functional portability of applications but ideally also the usage of the same service management interfaces among vendors [152, 283]. Unified management interfaces are said to be an important component to accomplish this scenario, as they enable the consistent management of applications across several providers [175, 231, 262]. As we have shown in Chapter 6, a great amount of effort for application migration, manifested in the EDS metric, is caused by the dissimilarity of the management interfaces. Whereas the need for a unified

[168] https://techcrunch.com/2014/08/04/docker-sells-dotcloud-to-cloudcontrol-to-focus-on-core-container-business

[169] https://twitter.com/cloudcontrolled/status/699530071481196544

[170] https://www.cloudbees.com/press/cloudbees-becomes-enterprise-jenkins-company

interface to manage applications in the cloud is often mentioned in literature [72, 77, 82, 193, 217, 283, 324], we argue that, until now, the majority of unified interfaces target the IaaS model [152, 285]. However, due to differing value propositions and a fundamentally different set of resources and services, cloud platforms must be assessed separately [152]. Therefore, we put a special focus on the portability of the management interfaces in this chapter while deferring the consideration of the application artifacts to future work. The vast majority of PaaS providers offer self-developed proprietary APIs and tooling suites [230, 285]. Hence, a provider change does not only require the application to be adapted but also urges the developers and operators from familiarizing with different tooling to adapting existing DevOps automation to new management interfaces [381]. DevOps is a metaphor for the collaboration of the development and IT operation units inside a company, including high task automation to streamline the software delivery process [155, 163, 337]. To mitigate vendor lock-in effects, this chapter presents a unified interface for application deployment and management among cloud platforms. The interface gathers and standardizes core functionalities along the development and application life cycle supported by cloud platforms. We validate our proposal with reference to both a study of related work and an evaluation of the state of the art. Thereby, we stress that existing approaches do not adequately model necessities for application management in cloud platforms. Additionally, we introduce *Nucleus*, a reference implementation of the presented unified interface targeting four leading cloud platforms, and evaluate its utility against typical use cases. The results show the feasibility of our approach and enhance the possibility of portable DevOps scenarios in PaaS environments.

The remainder of the chapter is structured as follows: In Section 7.2, we further define our methodology and present our definition of a unified management interface for cloud platforms. Section 7.3 presents details of our reference implementation based on the introduced interface and evaluates the feasibility of the proposed approach. In Section 7.4, we discuss how our approach differs from related work. Section 7.5 discusses existing limitations and future directions. Finally, Section 7.6 summarizes the contributions of this chapter.

7.2. Unified Management Interface

The aim of the presented interface is to unify core management functions of cloud platforms. Rather than attempting to create a complex combined matchmaking and migration solution, we focus solely on the creation of a harmonized deployment and management interface. Therefore, all technical dependencies, e.g., supported runtimes as well as contract-specific details such as SLAs, are neglected. Those aspects are already targeted by our work in Chapters 4 and 5 and brokering approaches such as [56, 82, 351]. Due to the variety of PaaS systems and their diverging scopes, we argue that an interface covering all

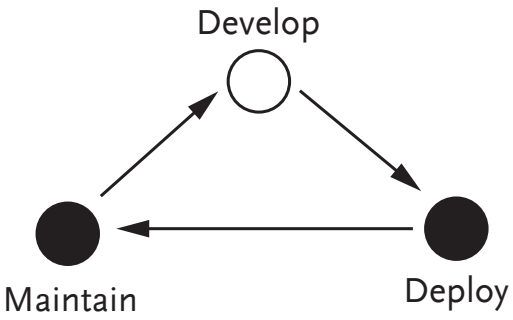


Figure 7.1.: Application Life Cycle

available offerings can hardly be defined. Cloud platforms can be classified by their proximity to SaaS and IaaS boundaries (see Chapter 3). Some products are more closely related to SaaS, whereas others have evolved from a more infrastructure-based approach. The majority of offerings, which we address here, are systems that supply a classical application platform that is composed of a set of runtimes, services, and other components an application can be programmed to. Especially platforms that are designed towards extending SaaS solutions or visual programming, e.g., Salesforce App Cloud[171], will have requirements for a different, presumably more narrow set of management operations. The unification of the management interface targets the Self-service Management API of the PaaS system (see Figure 4.2 in Section 4.2.1). This is the functionality with which the user manages the use of the platform and the application. Processes that are needed by the provider of the service to administer the PaaS are not taken into consideration, e.g, billing or user management.

Figure 7.1 shows the management focus of PaaS inside a typical application life cycle (marked by black circles). The ISO/IEC/IEEE [162] defines application management as a "domain responsible for all of the tasks and activities that are aimed at managing, supporting, maintaining, and renewing existing applications and related data structures." Application management operations of cloud platforms are focused on the operations part of the life cycle, i.e., deployment and maintenance. This includes the creation of the application environment, the deployment of the application itself as well as necessary actions in the maintenance phase. Typical maintenance tasks are monitoring the application's status and initiating reactions to increasing user demand, i.e., scaling the application at runtime. Agile development methods often reiterate the depicted life cycle to update existing applications with new versions [232]. Continuous Delivery is a major enabler for reducing effort of these recurring operation tasks [154]. Cloud platforms provide a high amount of automation of these actions along the software life cycle and are therefore also well-suited for agile development methods [222, 391].

[171] https://www.salesforce.com/products/platform/overview

In PaaS, not only the application life cycle operations are highly automated but also the delivery of the application environment itself. One of PaaS' key values is to supply a managed environment in which application code can be deployed, freeing the developers and operators from managing the application's runtime environment and connected services. Dependent on the application and its required runtime environment, the platform instantly provisions a container or VM instance with an operating system and all necessary runtimes installed where the application package can be hosted. An application is typically not self-contained but needs additional services to store information or to outsource tasks for processing. PaaS focuses on the requirements of applications and supplies a wide variety of preconfigured services that can be provisioned on demand for use by applications. This relieves operators from tasks like setting up databases as well as handling their availability and scalability. Cloud platforms can handle both user-faced applications and applications acting as services as part of a larger application architecture. Due to the high automation of the environment and the deployment, cloud platforms are very well suited for microservices architectures which are composed of an array of many small, isolated services [211].

Both of the mentioned perspectives, the management of the operations during the application life cycle and the management of the application environment itself are targeted by the management interfaces of cloud platforms. The intended interface unification should be viable given that a set of management operations of PaaS systems share the same semantics but only use different syntax among vendors [216, 329]. Currently, every vendor provides its own, nonstandardized interface with varying sets of supported operations and distinctions in the overall functionality offered to the users. For this reason, the collection of operations that shall be included in the core set of the unified interface must be supported by a wide range of vendors. To define such a set of core management functions, we conduct a comprehensive evaluation of 70 vendors from our PaaS knowledge base to homogenize the currently offered capabilities. Besides that, we also analyze and compare our proposal to existing works (see Section 7.4). Initially, the described approach leads to the identification of the typical application management use cases which are depicted in Figure 7.2.

Thereby, the diagram is not intended to be an exhaustive state diagram with all possible transitions but shall introduce important management tasks along an application life cycle inside a PaaS system. Here, a typical application life cycle starts with the *creation* of an application inside the PaaS environment. Most of the times, it is sufficient to provide a unique application name to execute this process step. Before the actual application code can be deployed, the environment needs to be *configured* and necessary services for the application need to be *provisioned*. In few cases, an application can exist without additional services, e.g., databases. Therefore, all PaaS systems provide at least a set of

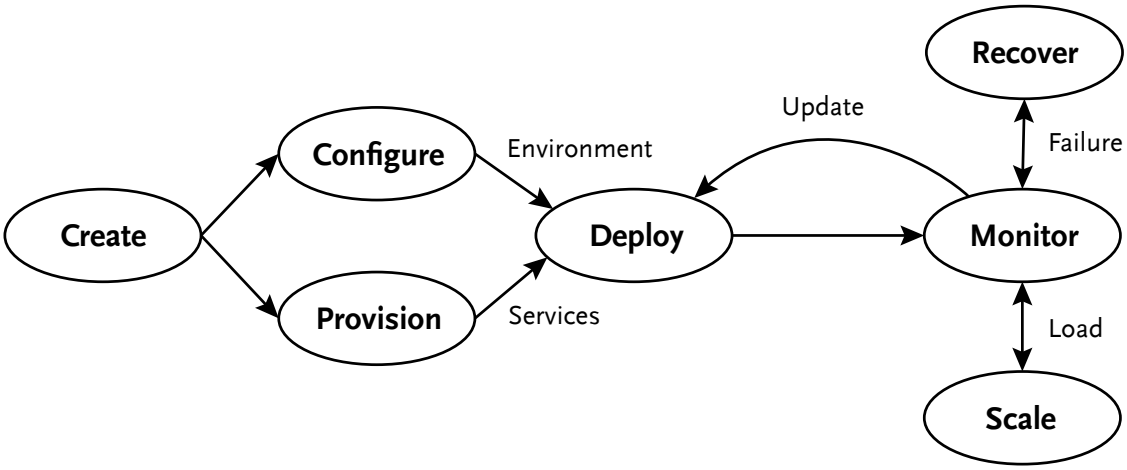


Figure 7.2.: Application Management Use Cases

essential back end services that can be provisioned on demand for use by an application. Access to these services is typically configured via environment variables. Also, PaaS systems grant several other configuration parameters inside the application space, e.g., the region where an application is deployed or the domains on which it is available to users. Afterwards, the application can be *deployed* into production. When an application is running, *monitoring* tasks such as logging need to be available to customers. As part of this, functionality for certain ordinary and exceptional traces must be available. In the event of increasing or decreasing usage from the user base, an operator needs to be able to *scale* the number of instances to handle application traffic appropriately. Likewise, if any failures occur during runtime, application instances may need to be restarted to *recover* to normal operation. Naturally, to enhance and evolve the functionality of an application, operators need to *update* the application code to a new version on a regular basis. In addition to the described actions, several CRUD operations are required for most of the entities that are involved in the steps.

For our unified management interface, we derived the selection of operations presented in Table 7.1. It depicts the proposed operations and their references in the existing literature as well as their support by current vendors[172]. Table 7.1 also shows that a substantial amount of fundamental and well-supported operations of modern cloud application management is not adequately considered by existing approaches. Here, we only depict compatibility with the defined operations for vendors that are also implemented in our prototype (see Section 7.3) to exemplify real-world usage. Nevertheless, the overall picture can be applied to more vendors.

The operations are divided into two groups: *general* operations and *application* operations. General operations include all tasks that target the management of the platform environment, whereas application operations all relate to a

[172]To highlight missing functionalities, they are visually emphasized by a red background.

Table 7.1.: Unified Interface Operations

		Functionality	Description	Cloud Foundry v2	Heroku	cloudControl	OpenShift v2	Literature
App operations	App	GET	Get app details	✓	✓	✓	✓	[77, 82, 174, 324, 399]
		UPDATE	Update the app	✓	✓	✗	✗	[77, 82, 174, 324]
		DELETE	Delete the app	✓	✓	✓	✓	[77, 82, 174, 324, 399]
	Life Cycle	DEPLOY	Upload app package	✓	✓	✓	✓	[77, 82, 174, 324, 399]
		REBUILD	Rebuild app package	✓	✓	✓	✓	
		DOWNLOAD	Download app package	✓	✓	✓	✓	[174]
		START	Start the app	✓	✓	✗	✓	[77, 82, 174, 324]
		STOP	Stop the app	✓	✓	✗	✓	[77, 82, 174, 324]
		RESTART	Restart the app	✓	✓	✗	✓	[77, 174, 324]
	Scaling	ADD INSTANCE	Scale-out	✓	✓	✓	✓	[77, 174, 324, 399]
		DELETE INSTANCE	Scale-in	✓	✓	✓	✓	[77, 174, 324, 399]
		SCALE INSTANCE	Scale-up	✓	✓	✓	✗	[324]
	Domains	LIST	List all app domains	✓	✓	✓	✓	[174]
		GET	Get domain entity	✓	✓	✓	✓	[174]
		ADD	Assign domain to the app	✓	✓	✓	✓	[174]
		DELETE	Remove domain	✓	✓	✓	✓	[174]
		UPDATE	Update domain settings	✓	✓	✓	✓	[174]
	Variables	LIST	List all env. variables	✓	✓	✓	✓	[324]
		GET	Get an environment variable	✓	✓	✓	✓	[324]
		CREATE	Create and set variable	✓	✓	✓	✓	[324]
		UPDATE	Update variable value	✓	✓	✓	✓	[324]
		DELETE	Remove a variable	✓	✓	✓	✓	[324]
	Logging	LIST	Collect all app log files	✓	✓	✓	✓	[77]
		GET	Get a specific log file	✓	✓	✓	✓	
		DOWNLOAD	Download all logs as archive	✓	✓	✓	✓	
	Services	LIST	List all installed services	✓	✓	✓	✓	[174]
		GET	Get bound service entity	✓	✓	✓	✓	[174]
		ADD	Install and bind to the app	✓	✓	✓	✓	[77, 82, 174]
		UPDATE	Update bound service settings	✓	✓	✓	✓	[174]
		REMOVE	Remove bound service	✓	✓	✓	✓	[77, 82, 174]
General	App	LIST	List all apps	✓	✓	✓	✓	[77, 82, 174, 324, 399]
		CREATE	Create the app	✓	✓	✓	✓	[77, 82, 174, 324, 399]
	Service	LIST SERVICES	List all available services	✓	✓	✓	✓	[77, 82, 174]
		GET SERVICE	Get available service entity	✓	✓	✓	✓	[77, 174]
		LIST PLANS	List available service plans	✓	✓	✓	✓	
		GET PLAN	Get service plan entity	✓	✓	✓	✓	
	Region	LIST	List all available regions	✗	✓	✗	✓	
		GET	Get available region entity	✗	✓	✗	✓	
				95%	100%	84%	95%	

specific application instance created inside the platform environment. The three resources that are administered within the general group are the list of user applications, services, and available deployment regions. The general application operations are user scoped, i.e., listing all applications will only return applications that are accessible by a particular user. Also, the creation of a new application environment is initiated inside the user's platform space. The services resource enumerates all available services that can be bound and used by an application. Although the existing literature considers application services, except for [174], these are limited to database services [77, 82]. However, services provisioned in cloud platforms have gone far beyond only

providing database back ends for applications but nearly offer everything as a service, from monitoring over messaging to payment services. This evolution was accelerated by the concept of add-on services provided by third-party vendors. Typically, services are accounted separately from the normal platform fees of an application which is why an interface must also list their associated payment plans. As native and add-on services cannot be distinguished at most vendors, these are managed by a single interface. Another important capability are the deployment regions for applications. Often, platforms do not only allow the deployment of applications to one geographical server location but offer multiple regions. This is particularly important for customers because of legal and performance reasons [213, 282]. Table 7.1 indicates that this functionality is only supported by half of the evaluated vendors directly through their management interface. This is caused by a generally missing multi-region support of Cloud Foundry and cloudControl. Both of these vendors require an independent endpoint instance of the PaaS system for multiple regions. However, our multi-provider concept, which is explained in the next paragraphs, compensates for the different approach of the two providers to implement this functionality (see Figure 7.3). Other capabilities such as runtime and framework support must be targeted by a knowledge-base-backed brokering solution (see Chapter 4) as the vendors do not offer this data through their interfaces.

Operations that belong to a specific application resource are the main part of the proposed interface. We agree with the literature on typical actions of the application's life cycle. The life cycle of an application involves deploying the actual application data, starting, stopping, and restarting the application. This includes getting detailed information, e.g., the status of an application as well as updating its data, deleting the deployment or the entire application space. This life cycle is fully supported by all vendors except cloudControl which directly starts an application at deployment time and allows no further manual state changes. Additionally, the interface includes an action for rebuilding an application whose properties were changed which require a redeployment of the application. Moreover, a download of the current application artifacts is provided. As an essential characteristic of cloud systems [237], elasticity is represented by both horizontal scaling (number of instances) and vertical scaling (instance power). Referring to the services category of the general capabilities, a dedicated group of operations is targeted at the management of services that are bound to an application. First, it is possible to provision or remove a service instance on demand from the services pool of the platform. Furthermore, the array of bound services or details of a dedicated service can be listed.

As mentioned before, existing literature is lacking or is divided over a wide range of fundamental operations of cloud platforms. Especially the remaining groups logging, variables, and domains, are often neglected in the literature despite their wide support in current cloud platforms. Logging plays an import-

ant role for debugging applications and monitoring application health in the maintenance phase of an application's life cycle. The variety of third-party offerings in the service category that target more sophisticated logging and alerts are dependent on the logging functionalities of the main system. Therefore, the interface provides operations for listing available log files, retrieving a particular log file, or the whole set of available logs. Using environment variables for configuring properties that are likely to vary between application deployments has become the de facto standard for cloud platforms. This includes resource handles and credentials to services or external add-ons that the application consumes. Only Sellami et al. [324] provide the ability to manage environment variables, although they can only be added as part of a complex environment description model and not as a separate resource. Last, the ability to assign and manage domains is crucial to address applications and provide them to end users. Typically, cloud platforms manage plenty of customer applications on one physical host and no application receives its own dedicated public IP address. Routing to application instances via domains is a suitable approach for both inter-application communication and for providing applications to end users.

Aside from adding new functionalities to our interface that are not mentioned in related work, we also omit actions previously stated in publications. Application environments [82, 324], monitoring [77], and database actions [82] are intentionally left out. In our opinion, database actions including data extraction are too specific to the particular data store technology to be unified. Furthermore, they are a technical requirement for the back end service and not the platform itself. Monitoring is out of scope, as it is not planned to evaluate the application or platform performance. Application environments would be a candidate for the set of management functionalities but are not widely supported yet and postponed to future revisions of the interface. Also, actions that are restricted to the provider's access such as the addition of new services, runtime support, or security policies for exposing network ports to the public [174] are not part of our user-facing API.

Even with extensive evaluation, it is challenging to define the right array of operations for a unified interface that satisfies all demands. In that regard, the NIST [152] states that one needs to define minimal standards and avoid overspecification that inhibits innovation. Consequently, we ensure that our selected set of operations is supported by the majority of vendors (see Table 7.1). Yet, a problem often experienced in unifying approaches which is caused by this rationale is the attempt to wrap the smallest common denominator of the competing provider APIs for thorough operation support. As a consequence, application developers are faced with a dilemma, being forced to either pick a feature-full API created by the chosen cloud provider and risk getting locked-in, or to favor a unified interface limited to a narrow range of functionality [152, 285]. To mitigate this problem, we include the ability

Figure 7.3.: Multi-Provider PaaS Support

to gain access to proprietary capabilities. Therefore, our interface supports a native loop-through functionality to execute arbitrary commands against the endpoint's API. This allows the developers to use a combination of unified interactions and proprietary functionalities if necessary.

As we intend to consolidate application management among cloud platforms with our interface, besides the unification of the operations, the need to create an interface concept that provides access to different platforms and adds multi-provider support was identified. Figure 7.3 illustrates the concept and the associations between vendors, providers, and endpoints. For each platform vendor, there can be an arbitrary number of providers delivering the platform and a provider can offer any number of endpoints. Both, endpoint and provider are strictly associated with a parent provider or vendor, respectively. Each provider needs to operate the vendor's proprietary API so that all providers and endpoints can be served by one adapter implementation. As an example, the vendor Pivotal develops the platform Cloud Foundry (CF). A provider offers a platform to its customers but does not necessarily need to have developed it. In this context, IBM Bluemix is an example for a CF provider. An endpoint is the API access point defined by the provider. One provider may offer multiple endpoints. For instance, IBM Bluemix offers two API endpoints to its customers, one serving a CF instance in the United States, the other one providing the European counterpart. With this approach, IBM accounts for a lacking direct multi-region support of CF (see Table 7.1).

As a summary, Figure 7.4 shows how the defined operations fit into the resource map of our unified API. The figure does not show all operations that are available but presents an overview of the relations between the resources and how they can be created, resolved, or updated. Resource properties are also neglected, except for the associations with other API objects. The figure is separated into four dedicated API groups. The platform group on the top left is responsible for managing available vendors, providers, and endpoints. All other resources are nested below the endpoint to which they belong. The services group is responsible for providing information about the available services within a cloud platform. All subordinated service plans, which belong to exactly one service, are nested below the services resource path. Equally, the available deployment regions are managed inside the region group. Both of these resources are read-only and belong to the currently connected endpoint. At the bottom of the figure, the application group includes all application operations, i.e., the list and instance retrieval for all applications, domains, logs,

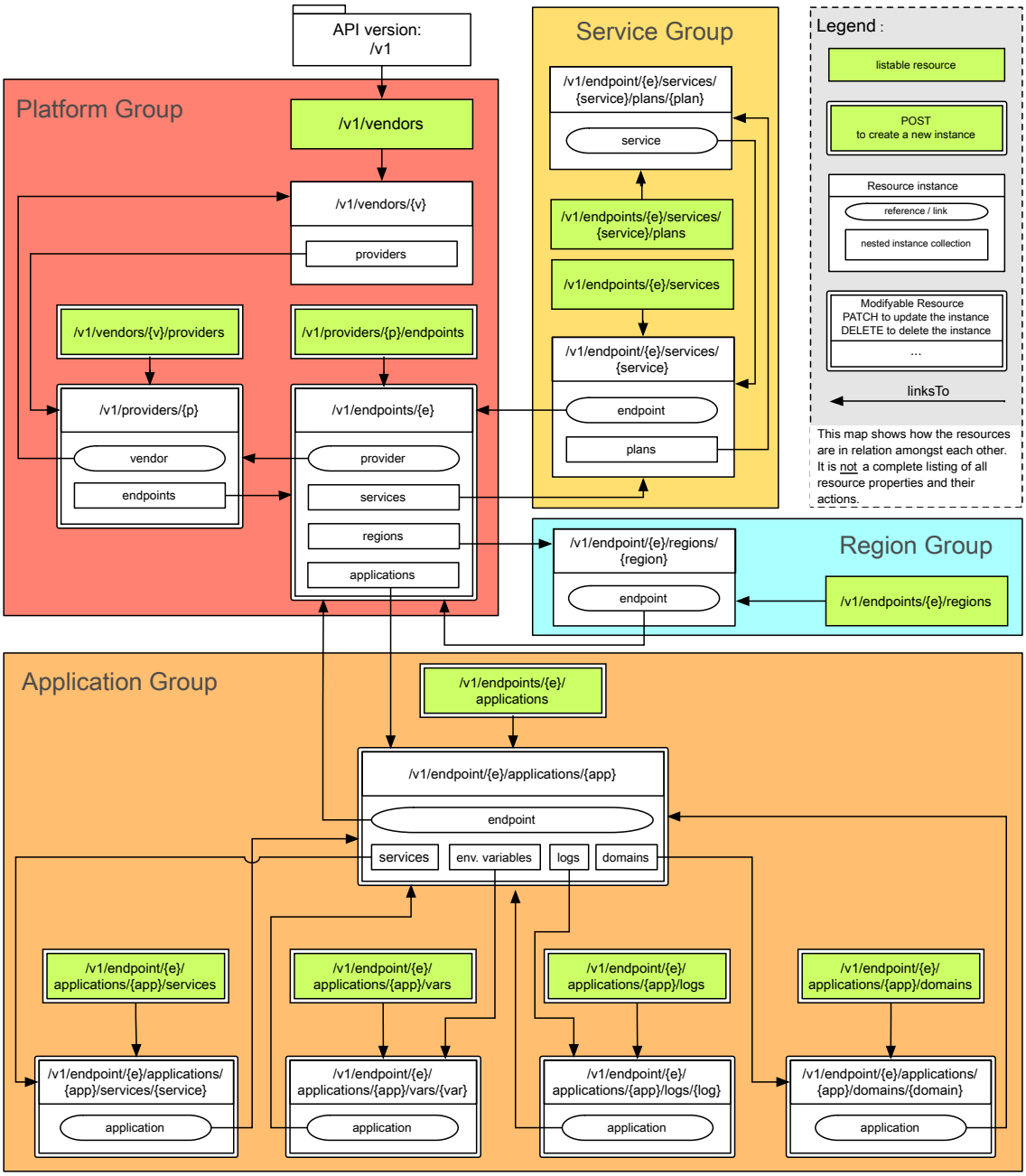


Figure 7.4.: Unified Interface Resource Map

variables, and installed services. Applications are nested below the endpoint, whereas the domains, logs, variables, and installed services are subresources belonging to an application.

The unified interface also introduces accordingly named and structured API objects for all the resources. All identified objects, including their associations and relationships, are visualized in the diagram in Figure 7.5. Following various best practices that describe how to properly build an API [235, 279], respectively a RESTful one, the abstraction layer's API utilizes several common concepts. Consequently, some of the presented ideas are already influenced by the decision to align the interface conceptually with RESTful APIs. If applicable,

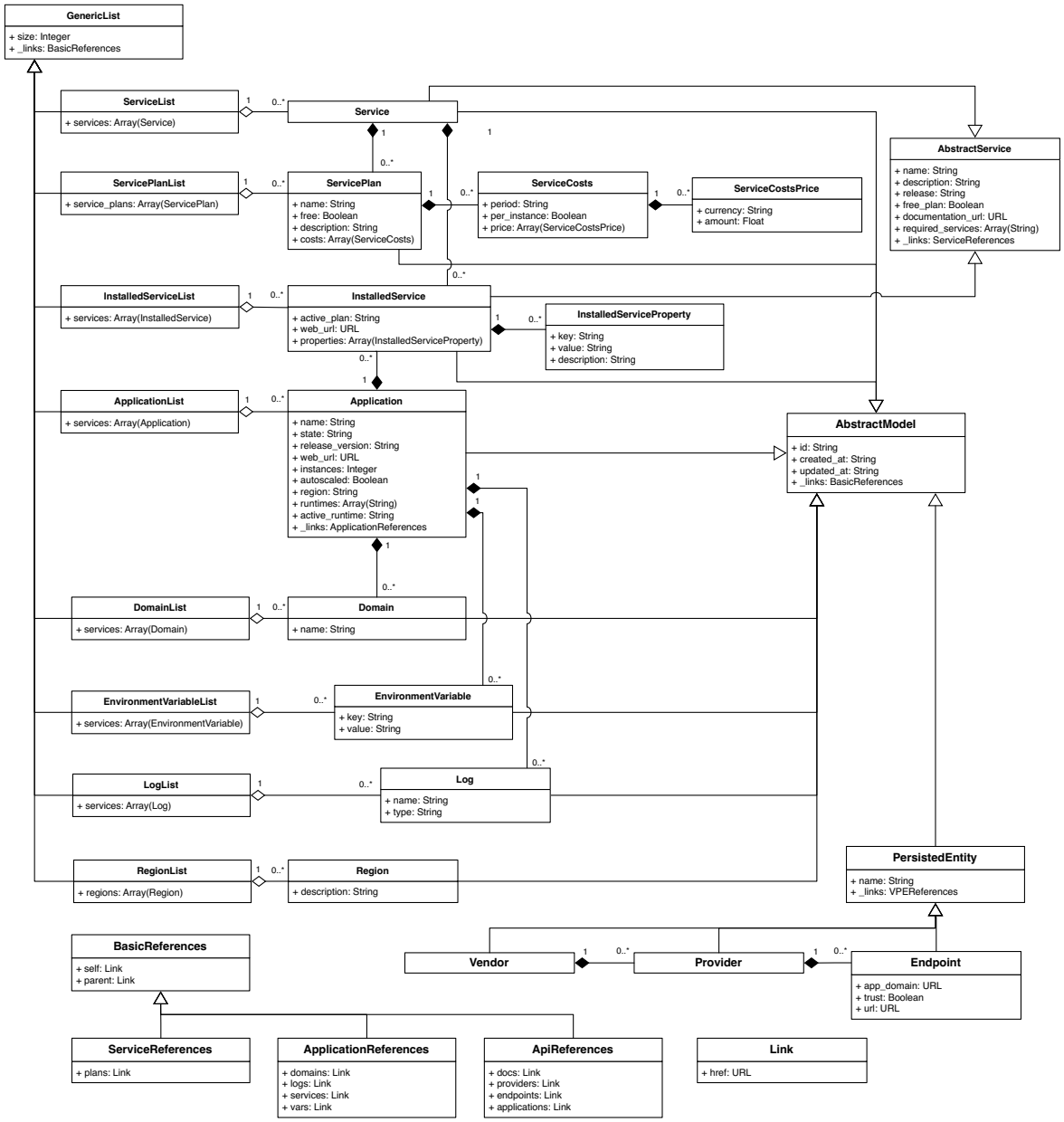


Figure 7.5.: Unified Interface Resource Diagram

the API's response objects have a unique identifier and timestamps that reveal when the object was created and when it was updated for the last time. Together with the `_links` property that is used to implement the Hypermedia As The Engine Of Application State (HATEOAS) principle of REST, those requirements are combined inside the `AbstractModel` which possesses those four properties and must be inherited by all other response objects. References are required to be of the type `BasicReferences` or one of its inheriting resources. The entity `BasicReferences` has two links, a `self` and a `parent` link. Its self-reference is a URL that tells where this specific object can be retrieved from. The parental reference reveals whether the object is ancillary to another object. Both links must always be set, except for the API's root node, which does not have a parental reference. All subtypes of the `BasicReferences` provide additional relations, e.g.,

in case of the resource `ApplicationReferences` to all child object collections of an application. In addition to the `Application` entity as core component of the interface, Figure 7.5 also includes `Region`, `Service`, `ServicePlan`, `Domain`, `EnvironmentVariable`, `Log`, and `InstalledService` resources corresponding to the groups of operations in Table 7.1. Beside those, the diagram also contains dedicated `List` objects for all of the previously mentioned resources. Their purpose is to standardize the way how the API returns object collections. Therefore, all `List` objects inherit from `GenericList` including its `size` property to indicate the number of available elements and the `_links` property for HATEOAS. Additional features, for instance pagination within the API, can easily be added to the generic resource later on. The concrete mappings of the proprietary vendor resource objects to the unified interface objects will be described and exemplified in the following Section 7.3.

Additionally, a unified application life cycle is defined. An application undergoes several stages, all of which are observable via the state property of the application object. Managing an application and its states requires detailed knowledge of the state transitions and the overall life cycle of the application. Without knowing the current application state, an API cannot properly handle necessary transitions. Existing approaches [77, 82, 270, 324] are based on a too simplistic application life cycle or do not target it at all. There is a reason for that, as one of the most problematic points while defining the unified interface was to identify existing application states between vendors and appropriate state detection rules. Relying on a very least denominator is not reasonable in this case as we cannot signal correct states to the user and recover appropriately. The life cycle, which is illustrated in Figure 7.6, differentiates between a total of five states complying with all four examined platforms. Yet, some of them only use a subset of these states, as not every platform supports all states, e.g., the IDLE state.

The cycle can be initiated as soon as an application space is reserved inside the platform environment. To transition into the DEPLOYED state, the application data must be uploaded to the platform. Afterwards, usually the build for an instance image is initiated. If anything goes wrong, the cycle terminates with an error. Otherwise, a runnable application image is created and the state changes to DEPLOYED. Even though it can be argued that the DEPLOYED state is not necessarily required and applications can directly switch into the RUNNING state, we decided to include it. It provides additional flexibility, as for instance data migrations or background jobs can be triggered before the application is made available to the public. From the DEPLOYED state, the instances of an application can be started. If at least one instance of the application is successfully started, its state changes from DEPLOYED to RUNNING. A failed startup can cause a state transition of the application instance into the CRASHED state. With the stop command, an application can be shutdown, whereupon the new state of the application is STOPPED. Similar, after a period of inactivity, some PaaS vendors,

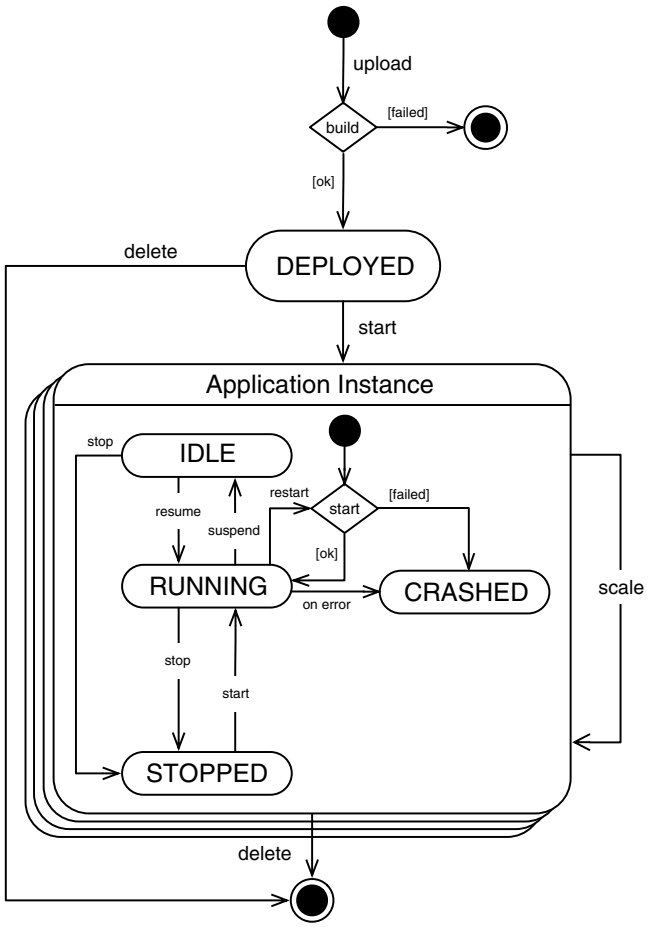


Figure 7.6.: Application States

e.g., Heroku and OpenShift, may put an application to sleep, i.e., suspend the instances, changing the state of all application instances from RUNNING to IDLE. Technically, the IDLE and STOPPED states could be seen as equal since instances are suspended. However, semantically they give different information to the user. In the first case, the application was scaled to zero instances as no incoming traffic was received for a specific period, whereas in the second case the application was stopped explicitly. Also, in the IDLE state, the application will return into the RUNNING state automatically if a new request arrives for the application, whereas STOPPED applications do no longer serve requests until they are manually transitioned into the RUNNING state again. Additionally, all transitions returning to the RUNNING state may change into the CRASHED state if an error occurs, similar to the choice at the initial transition of an application instance. We omitted these choices to reduce the visual complexity of the state diagram. The global state of the application can be deducted by the individual application instance states. The application is said to be RUNNING, if at least one instance is RUNNING. STOPPED, CRASHED, or IDLE states only apply if all instances are in this same state. An update of an application follows the same cycle with a new iteration. All other application versions will be stopped before

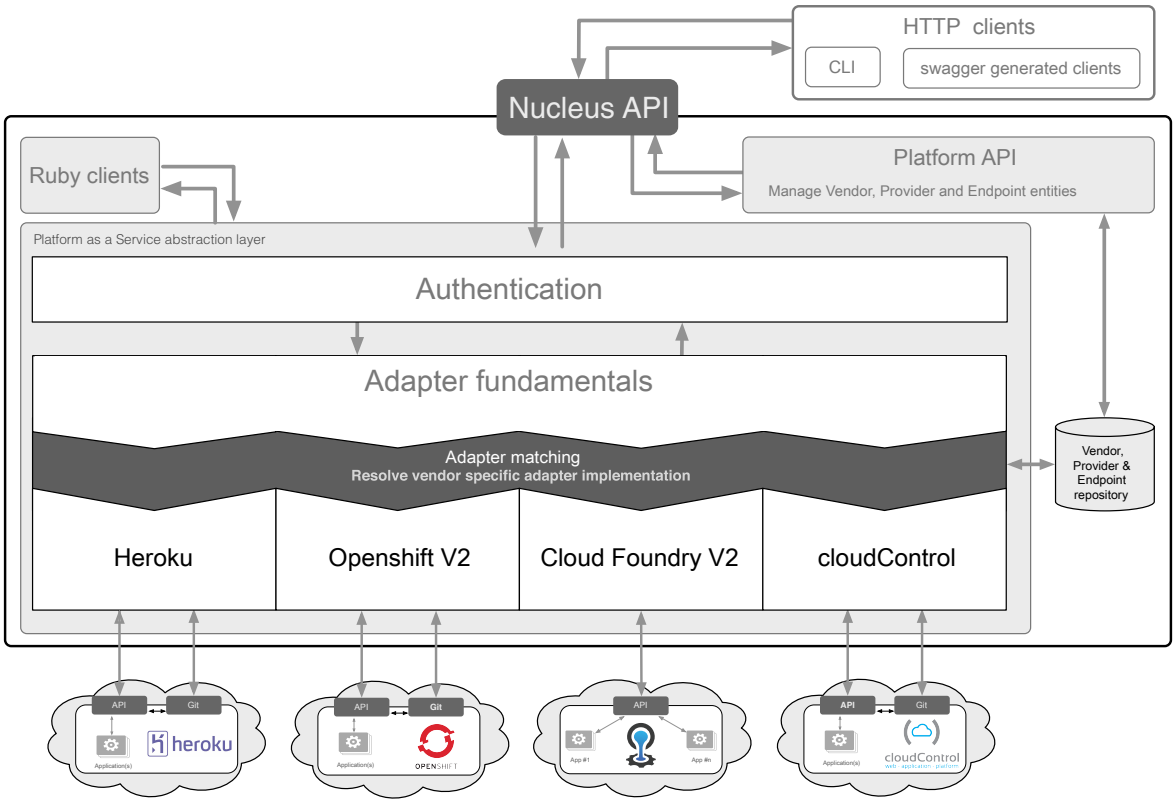


Figure 7.7.: Nucleus Architecture

or after (zero downtime deployment) the new version has RUNNING instances and can serve user traffic. Finally, old application images can be removed from the storage.

For further discussion and technical details of the presented artifacts, we refer to the technical report [312]. Next, the feasibility of the approach is validated by providing a reference implementation of the defined interface and by evaluating the prototype against several use cases.

7.3. Reference Implementation

In this chapter, the design of our reference implementation *Nucleus*[173] is discussed. We evaluate the details of the concrete implementation as well as the challenges that must be targeted to integrate and map existing offerings to the unified management interface presented in Section 7.2. To begin with, Figure 7.7 shows the overall architecture of the reference implementation.

At the heart of the implementation, the unified Nucleus API is provided to clients. The API itself is implemented in Ruby via Grape[174], a framework for creating RESTful APIs. One of the requirements for the API is to provide a platform and programming language-independent abstraction layer. Here, we

[173]The source code is available at `https://github.com/stefan-kolb/nucleus`.
[174]`https://github.com/ruby-grape/grape`

decided to create a RESTful API with the use of JSON over HTTP [305]. By using these technologies, the API can be directly queried by any client that is able to issue HTTP requests. Also, the technology choice makes it easy to extend the system with command line clients and other tools. Nevertheless, the absence of a standardized service contract, i.e., interface, for RESTful web services compared to classical SOAP/WSDL Web Services [78, 366], is a drawback that needs to be resolved [280]. This limitation often leads to documentation written in natural language that most likely implicates several deficiencies, such as informal and heterogeneous documentation, imprecise resource type descriptions, redundancy, and lack of visual support [65]. To reduce these problems, the developers need a precise API specification. Currently, neither the Web Service Description Language (WSDL) [70, 370] nor the Web Application Description Language (WADL) [137] that are per se capable of providing a contract definition for any web service implementation are widely accepted or used for RESTful web services [366]. Instead, JSON-based approaches like the OpenAPI Specification (OAS)[175] (formerly known as the Swagger specification) have gained traction. Those approaches are considered to be easy to write and lightweight compared to WADL. The OAS defines a standard, programming language-agnostic interface description for RESTful APIs[176], which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code or additional documentation. The interface schema can be automatically created by annotations or a DSL inside the source code. We use grape-swagger[177] to add OAS v2.0[178] compliant documentation to our API. Moreover, the standardized JSON schema can be displayed as human-readable API documentation including descriptions of the commands, data structures, errors, and try-out-functionality via Swagger UI[179] (see Figure 7.8). All these facts contribute to an up-to-date and well synchronized API description which minimizes any inconsistencies between code and documentation.

Nucleus' extendability is provided by a modular structure with dedicated adapters for each supported platform. A set of providers and endpoints for all adapters is available by default. This set can also be altered through the API which allows the addition of providers and endpoints at runtime. Moreover, Nucleus supports to host multiple versions of the API in parallel. Thus, new iterations of the API including breaking changes can be issued while keeping older ones functional side-by-side. If a new provider should be added or a provider changes its API, only the adapter implementation must be adjusted. In this way, Nucleus can maintain long-term stability for tools and automation scripts and backward compatibility across different API versions

[175] https://www.openapis.org
[176] https://github.com/OAI/OpenAPI-Specification
[177] https://github.com/ruby-grape/grape-swagger
[178] https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md
[179] https://swagger.io/swagger-ui

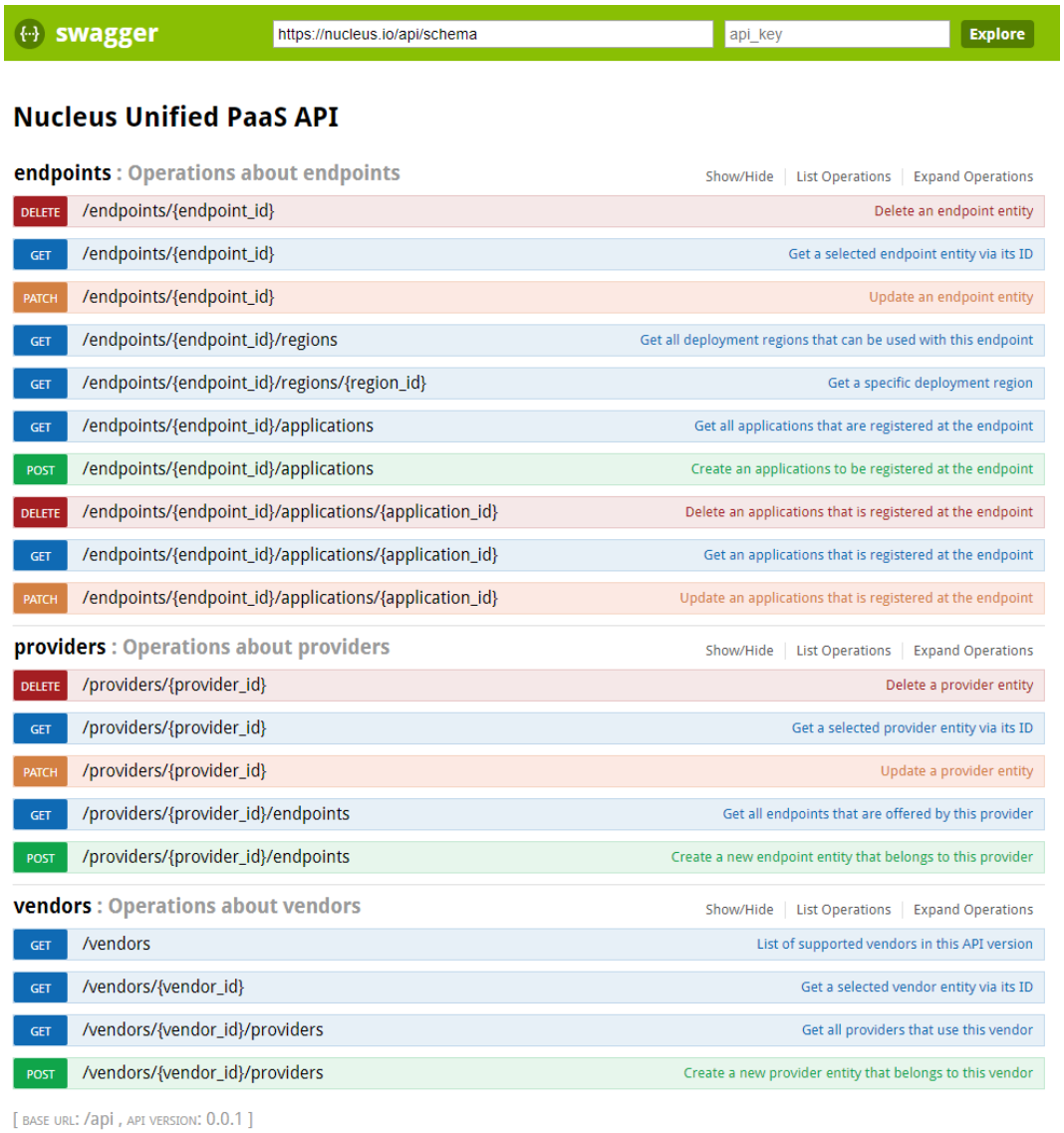


Figure 7.8.: Open API Documentation with Swagger

for the available vendors. Consequently, changes to the unified API must be versioned appropriately in the future, e.g., via semantic versioning[180]. Between the API and the adapters, an independent authentication layer capable of different authentication mechanisms, e.g., OAuth [141] or HTTP Basic [304], is provided that can be reused inside the adapter implementations. The entire Nucleus server implementation may be hosted as local instance as well as a public instance with multi-user support. For an easy installation on different OS platforms, we packaged Nucleus as a Docker image which is available via Docker Hub[181]. Due to the popularity of language-specific wrappers for APIs, the Nucleus API is also available as a Ruby Gem[182]. Hence, it can be integrated and directly called from source code for integrations if needed.

[180] `https://semver.org`
[181] `https://hub.docker.com/r/stfnklb/nucleus`
[182] `https://rubygems.org/gems/nucleus`

Table 7.2.: Selected Vendors for Adapter Implementations

	cloudControl	**Cloud Foundry**	**Heroku**	**OpenShift**
Type	Proprietary	Open Source	Proprietary	Open Source
Hosting	public, private	public, private	public, virtual private	public, private
Providers	(4)	9	1	5

The initial implementation of Nucleus included adapters for four leading cloud platforms. These were Cloud Foundry, Heroku, OpenShift, and cloudControl (see Table 7.2)[183]. Including their providers, Nucleus was able to support more than 12 public cloud platforms as well as any private deployment of these systems. Over time, with more providers adopting the popular platforms CF and OpenShift, this support is increasing even more. Support for the vendor cloudControl was removed after their bankruptcy end of February 2016. The decision on possible candidates for the adapter implementations was assisted by our knowledge base and cloud brokering tool *PaaSfinder*. One aspect was to cover a variety of technological implementations, such as Git- and HTTP-based deployment and different authentication protocols, to support new vendors in future releases without the need for major modifications to the prototype. Likewise, the vendor's impact and popularity within the market played an important role. In fact, OpenShift, Heroku, and Cloud Foundry were the top three vendors queried on *PaaSfinder*. All of them are available as public and (virtual) private cloud deployments[184]. Our approach does not require any additional documents, e.g., application descriptors, but can be used without adaption for existing applications. Those facts contribute to making the abstraction layer applicable for a wide variety of offerings, both hybrid and multi-cloud, empowering its practical utility.

One of our main points of criticism of the literature is that most of the proposed approaches are missing an evaluation through an existing and published implementation of their specification. In our opinion, a unified interface can only be validated and improved by a working reference implementation. The importance of reference implementations cannot be stressed enough as conceptual problems can only be reasonably discovered in practical use cases and evaluations. Although our implementation generally proves that it is possible to implement the suggested unified interface and make different existing vendors conform to the defined operations and resources, we experienced various issues while creating and evaluating the implementation.

A main challenge to harmonize the functionality of the four platforms lies in the mapping of proprietary operations and resources to the unified operations and resource schemata. A major problem is incomplete or insufficient up-to-

[183]Provider values were taken from PaaSfinder on 2018/3/15.

[184]Heroku Private Spaces offers isolated cloud deployments but no hosting on private data centers referred to as virtual private deployment.

Table 7.3.: Native API Requests per Unified Operation

	cloudControl	Cloud Foundry	Heroku	OpenShift
Operations	31	35	36	30
Total API requests	47	63	49	39
Max. API requests/operation	4	8	4	3
Avg. API requests/operation	1.52	1.8	1.36	1.3

dateness of the vendor's API documentation which is also a general problem when relying on the proprietary API. In response to this, we add the previously described OAS-compatible API description to our implementation. Although Table 7.1 shows broad support for the majority of the defined operations, these transformations are not solely one-to-one syntactical mappings but require a series of requests on native API resources to gather required properties and initiate all necessary operation steps. Whereas we have to omit a detailed consideration of the mappings, Table 7.3 shows an aggregated view of the mapping overhead. The table points out how many API requests have to be sent to achieve the semantically equivalent result on all platforms. Thereby, we only count operations that are supported by the respective vendor and actually conduct API requests. The request count for a unified operation lies between one and eight native API calls. Overall, compared with the average requests per operation values of the vendors, the unified API tends to take a more concise approach for the selected management operations.

Besides the operations, the resource entities need appropriate mapping rules as well. Thereby, the required information for a unified resource entity often has to be aggregated from different native entities. To illustrate this process, we present the mapping of the application object of Heroku in Table 7.4.

We choose the application object, as it is not only a core part of the abstraction layer, but the mapping has to consider more aspects than most other objects. The rules demonstrate how the API objects of the abstraction layer can be populated with data from the platforms' native API objects. Often, the object values can be applied without modifications, but some fields require processing of the original value or even state-dependent solutions. The attributes of the unified application entity are depicted in the first column of the table. The second and third column show the mapping from Heroku's native API objects. Thereby, the *object* column contains the name of Heroku's API entity, from which the information can be gathered from. The *field* column includes the specific field name or detection rules for deriving the value. Most of Heroku's application mappings can be obtained from the native application object. Mapping rules must be applied to the `instances`, `release_version`, and `runtimes` attributes. The number of application `instances` can be determined by counting all instances of the type *web* that are listed in the `formation` object. All `runtimes` can be gathered by inspecting the `buildpack-installations` object

Table 7.4.: Application Object Mapping Example Heroku

Attribute	Object	Field
id	app	✓
created_at	app	✓
updated_at	app	✓
name	app	✓
active_runtime	app	buildpack_provided_description
runtimes	buildpack-installations	array(buildpack/url)
region	app/region	name
autoscaled	-	*false*
instances	formation	quantity (type: web)
web_url	app	✓
release_version	releases	id[max(version)]
	dyno	release/id[max(release/version)]
state	Application state detection rules described in [312, pp. 38–40]	

and presenting an array of the individual buildpack URLs. A region is already included in the application object of Heroku, but as only an identifier and not an embedded object is desired, the name of the region object must be remapped. There are two approaches to diagnose the `release_version`. First, if no instance is assigned to the application, the ID of the latest version that is included in the `releases` object can be used. The second approach, if an instance is assigned, is to use the ID of the latest version that is assigned to one of the instances. For the `state` attribute, which contains the current state of the application, complex state detection rules (decision trees) need to be applied among the vendors. The unification of the application life cycle (see Figure 7.6) proved to be a difficult task due to the diversity among vendors. For example, within cloudControl, applications are directly started on deployment and cannot be stopped or put into maintenance. In general, we believe that the application life cycle is one of the properties that could benefit the most from a standardization. For a more extensive discussion of all technical details, see the current documentation or the technical report [312].

A main source for evaluating the utility of the prototype are the designed adapter tests that do not only test all the available operations but simulate the complete life cycle of cloud applications (see Figure 7.1 and 7.2) [108]. The overall test coverage is above 80 % of the code lines. The use cases directly interact with the vendor APIs and record all HTTP interactions that are then matched with the expected results and unified resource structures, validating the correct functioning of the implementation. For every vendor, a complete cycle is traversed. This includes the creation of an application, provisioning and configuring routes to required services, deploying the application data, and starting the application. Moreover, it covers the monitoring of application logs, scaling and recovering instances caused by load and instance failures, up to an application update (see Figure 7.2). Furthermore, the use cases do not alone cover positive paths but also failing actions during the application life cycle.

The feasibility of the operations defined by the interface are thereby not only validated in isolation but in relation with each other.

At the beginning of this chapter, it was claimed that portability and interoperability among PaaS can be improved with the creation of a unified interface. In summary, we improve the current state by the common operation and object model of our unified interface that applies to all supported platforms. Differences could be successfully harmonized among the four platforms. With its unified deployment and management capabilities, Nucleus allows to manage applications on different platforms. Most important, even though a fully automated vendor change is not yet viable, the effort that is needed when migrating an application to another vendor is reduced (see Chapter 6). Whereas most of the existing differentiations could be semantically unified, some problems can only be reasonably solved with the help of the vendors unifying important core aspects of their systems. In case of switching the provider, the effort to adapt the application's surrounding, for instance to enable Continuous Delivery and DevOps, can be minimized. Hybrid and multi-cloud deployments are facilitated due to the fact that the management operations can be handled consistently, if all vendors support a unified interface.

7.4. Related Work

In the past, several drafts for unified cloud interfaces were published as independent work or as part of a broader brokering approach. As stated, we argue that a majority of them are focused on the infrastructure provisioning model, missing out on cloud platforms [152, 285]. Furthermore, we show that existing approaches for PaaS do not adequately consider core functionalities of modern cloud platforms. Often, the approaches are focused on supporting a unified deployment of applications but do not apply a more holistic view of the fundamental management capabilities. The following paragraphs provide an overview of related work and give distinction of how our work differs and contributes to the existing approaches.

Standards

A number of standards that are often still in the process of making are proposed by several standardization organizations. In many respects, the use of standards is counterproductive to the vendors' aim to achieve a strong market position [60]. Therefore, most standard proposals suffer from the lack of acceptance and participation by industry leaders that prevents adoption.

Originally initiated to create a remote management API for cloud infrastructures, the OCCI claims to be a generic protocol and API to serve other models besides IaaS. Based on the core specification [268], extensions for infrastructures [269] and platforms are available [270]. Whereas the specification for

infrastructures possesses several practical implementations, the one for platforms can only be considered to be in an early state. The specification includes entity types for application, components, and links between them. A database mix-in is an example for a specific component instance that a provider should offer. Apart from these types, templates shall exist to apply predefined configurations to existing types. The specification provides application templates to define which underlying framework the application uses, e.g., the programming language. Moreover, it includes resource templates for referring to a preset resource configuration, such as a small or large container instance. Additionally, an application state model is defined. The model is a subset of the application instance states depicted in Figure 7.6. Overall, the specification stays on an abstract level, includes little details, and heavily relies on the capabilities of the OCCI core specification.

The Cloud Infrastructure Management Interface (CIMI) specification [89] explicitly states to be solely focused on infrastructure management. It consists of a model for cloud infrastructure resources as well as a standardized REST over HTTP interface to manage the defined resources.

In contrast, CAMP [261] focuses on providing a management API for cloud platforms. The recent proposal for a standard describes generic operations and artifacts that a PaaS cloud should ideally offer. Similar to our approach, a resource model for representing applications and their components is defined. CAMP also uses a REST-like protocol for manipulating the specified resources. The committee specification is awaiting the approval as an official OASIS standard, requiring the evidence of interoperable implementations which is still missing to date. Development for a fully CAMP-compliant API was started inside the OpenStack Solum project[185]. CAMP is designed to be language, framework, and platform-neutral with the goal of covering a variety of PaaS systems. The operations that CAMP specifies include building, life-cycle management, administration, and monitoring tasks. Nevertheless, we argue that the current scope and definition of operations and resources is too generic and it would require a huge customization effort to integrate it with today's cloud platforms. Despite the decision to not use CAMP, it should be feasible to integrate CAMP into our abstraction layer once it reaches a higher maturity level and a reference implementation becomes available.

In 2017, the DIN SPEC 91337 [88] "Unified Application Management Interface for Cloud Application Platforms" was published. The majority of the specification's contents are extracted from the European research project *PaaSport* which is discussed in the next paragraph. Additionally, an XML Schema Definition (XSD) for the resource entities of the interface and an application life cycle are added. The life cycle includes created, deployed, and running states together with temporary intermediate states for every transition between

[185] `https://wiki.openstack.org/wiki/Solum`

them. However, besides states for good cases, no error state for defective traces is included.

TOSCA [259] specifies a generic meta model for defining topologies of cloud applications. This includes both the structure of an IT service and how to instantiate and manage it. TOSCA defines a framework to specify management operations, whereas the concrete implementation of the management operations is left to the user. Our interface is a concrete instance for PaaS management which narrows down the generic definitions to a concrete set of operations and resources.

Further proposals to improve portability and interoperability in the cloud are worked on under the umbrella of the IEEE standards association. These are the Cloud Portability and Interoperability Profiles (P2301)[186] and a Standard for Intercloud Interoperability and Federation (P2302)[187]. Whereas these initiatives already exist for a longer period and are still marked as active projects, the working groups have not released any normative results so far.

Abstraction Layers

Abstraction layers are a solution to harmonize different systems which all share a common principle behind one interface. In contrast to standards, this approach often comes in conjunction with adapters for mapping the abstraction layer interface to the native APIs without the need for firsthand vendor support. A variety of projects are available in the area of cloud computing which follow a concept that is related to our approach.

A technique applied by some projects is to duplicate popular APIs from leading vendors and supply a different implementation. Two examples for such interface clones are Eucalyptus [258] that is based on Amazon's EC2 API and AppScale [69] that mirrors parts of the Google App Engine API. This approach is reasonable for open source clones of proprietary offerings that try to supply exactly the same functionalities as the base vendor. However, it is less feasible to unify a variety of technically differing offerings.

Apache Deltacloud[188] is an abandoned attempt to improve the interoperability among various infrastructure providers. It provides a set of RESTful APIs that allow to interface with a multitude of providers. The supported APIs include a custom Deltacloud API, the standardized CIMI API, and a clone of Amazon's EC2 API. Besides Deltacloud, there exist a multitude of language-specific abstraction layers for IaaS. Munteanu [249] evaluated and implemented an abstraction layer to several IaaS providers for use in their multi-cloud PaaS mOSAIC. Goonasekera et al. [3, 125] present CloudBridge[189], an academic project in the field of bioinformatics, that implements an IaaS wrapper in

[186] https://standards.ieee.org/develop/project/2301.html
[187] https://standards.ieee.org/develop/project/2302.html
[188] http://deltacloud.apache.org
[189] https://github.com/gvlproject/cloudbridge

Python. Moreover, several other nonacademic projects have established themselves in the field, such as jclouds[190], Fog[191], pkgcloud[192], or Libcloud[193]. Our approach combines both of the described styles, but for PaaS, as it serves as a programming language-independent RESTful API and a Ruby wrapper library.

Cunha et al. [76, 77] propose PaaS Manager, an approach similar to ours that defines a set of common operations abstracting the differences of application deployment and life cycle management of multiple providers. PaaS Manager declares to support adapters to three platforms, i.e., CloudBees, Cloud Foundry, and Heroku. However, to our knowledge, a proof-of-concept implementation has never been publicly released. From what can be deduced from the papers [76, 77], their API definition is lacking important environment configuration operations, e.g., domains and multi-region support.

Zwattendorfer et al. [399] present a small RESTful API for PaaS management. They only implement a limited set of operations focused on the deployment of an application, omitting all operations for the ecosystem of PaaS. Their solution only supports JVM-based applications and relies on a WAR deployment via file upload. The presented prototype supports the three providers Heroku, CloudBees, and Cloud Foundry.

Similarly, Sellami et al. [324] introduce the Compatible One Application and Platform Service (COAPS) API. Their implementation includes connectors to Cloud Foundry, Openshift, and Google App Engine [153, 324]. Compared to our findings, COAPS is missing various essential operations of typical cloud platforms, such as domains, monitoring of applications, or the management of services and deployment regions [324, 357].

Another EU-funded project from which a unified managing approach for PaaS evolved was Cloud4SOA [82]. The functionality of Cloud4SOA was partially transferred to subsequent projects CloudPier and most recently SeaClouds [56]. Cloud4SOA provides four core capabilities, of which one is the unified management and deployment of applications to cloud platforms. According to the publications, adapters were offered for AWS Elastic Beanstalk, Cloud Foundry, Openshift, and CloudBees [82, 175]. Later, this was extended with Heroku and cloudControl. Similar to the previously described approaches, the presented unified interface does not consider important standard functionalities. For Cloud4SOA, this especially concerns application management, e.g., domains, logging, scalability, deployment regions, and also a dedicated ability to manage application services.

The EU project PaaSport also includes a unified PaaS API as one of their deliverables [174]. According to their documents, they use the Cloud4SOA API [82] as foundation for their proposal. From the available related work, they

[190] `http://jclouds.apache.org`
[191] `http://fog.io`
[192] `https://github.com/pkgcloud/pkgcloud`
[193] `https://libcloud.apache.org`

most closely match with our proposed API (see Table 7.1). Besides minor differences, they do not include logging capabilities and propose a fundamentally different approach for binding services to applications. Instead of using environment variables, they suggest to use Spring Cloud Connectors[194] to connect applications to services. Yet, this technology is only applicable to JVM-based applications. Spring Cloud Connectors provide a simple abstraction to discover information about the cloud environment on which they are running, connect to services, and register discovered services as Spring beans. Currently, it provides support for discovering common services on Heroku and Cloud Foundry platforms and it supports custom service definitions through Java Service Provider Interfaces (SPIs). However, to make the technique applicable for different applications, ports for other programming languages and vendors are needed. Nevertheless, as it turned out, the usage of environment variables is generally accepted as de facto standard for cloud applications today and more portable than the concept of cloud connectors. The documents state that the PaaSport prototype supports the PaaS systems OpenShift, Heroku, Cloud Foundry, AWS Beanstalk, and cloudControl. In general, their prototype seems to be restricted to Java-based applications.[195] The results and source code of their prototype are not released to the public. Eventually, together with small enhancements, the PaaSport API description was published as DIN SPEC 91337 [88].

7.5. Limitations and Future Work

Even though the presented unified interface already copes with the core aspects of today's cloud platforms, there are still open challenges remaining which can be worked on in future projects.

Conceptually, PaaS is still evolving, and so do the management interfaces. Although we integrated flexibility to cover this case via the native loop-through functionality, also the unified interface will need revisions and upgrades over time to provide an appropriate abstraction of the state of the art. Application environments are an example of an upcoming feature. Such environments allow managing different versions of an application for, e.g., staging and production. With agile development, multiple environments are becoming a first class feature for developers that they also want to use in the cloud. As such features are not yet supported by a wide range of platforms, we cannot reasonably include them in the current version of the interface.

Although our contributions allow for the deployment of applications through a unified interface, we also have to take into account that the implementation artifacts needed to deploy onto the PaaS may be different, as shown in Chapter 6. Possible solutions for this have to be investigated independently.

[194] `https://cloud.spring.io/spring-cloud-connectors`
[195] `https://www.youtube.com/watch?v=zZIRJAfp6ug`

For extending our reference implementation, several starting points can be identified. Naturally, the range of applicability of the tool could be enhanced by additional adapters. Furthermore, we could provide multiple APIs in front of our implementation. Especially, if standard efforts like CAMP mature further, we could integrate these into our prototype and provide a standard-compliant interface to multiple providers straightaway. Another beneficial project would be to create a console client for our unified API. This would considerably enhance the value for end users, as it is the preferred way for users and developers to interact with cloud platforms.

Another idea is to evaluate if the interface can be generalized to a generic application deployment interface. During our work, we observed that our interface definition largely matches the application life cycle of cloud applications in general, independently of the cloud model or the underlying infrastructure. However, we need further structured evaluations and studies to validate this assumption.

7.6. Summary

The inherent need for a unified interface to manage applications in the cloud is stressed in multiple research papers [72, 77, 82, 217, 283, 324]. Whereas standards and approaches for the infrastructure provisioning model have already gained traction, proposals for cloud platforms are still premature [152, 285]. The focus on a high level of DevOps automation in PaaS further stresses the need for a homogeneous approach among vendors [381]. Therefore, we presented a unified interface to manage applications in cloud platforms. The results both build upon extending former approaches and an extensive evaluation of the state of the art. We validated our proposal with a reference implementation that supports four leading PaaS systems. Hence, we make the abstraction layer applicable for a wide variety of offerings, both hybrid and multi-cloud. Also, the ongoing consolidation in the PaaS market shows signs that such an approach is more likely to succeed as in the early years of the PaaS technology [314]. Today, we see that vendor count and technology stacks stabilize. The presented unified management interface for cloud platforms in combination with our reference implementation *Nucleus* increases the portability and interoperability of PaaS applications and thus helps to avoid critical vendor lock-in effects.

Part IV.

Conclusion and Outlook

8. Competing Approaches

In each of the previous chapters, we outlined and discussed related work that targets the specific topic presented in the chapter. As the different parts of the thesis are not only relevant as sole contributions but follow a logical connection which forms a cloud broker for PaaS systems, we want to outline important related work that follows a similar comprehensive approach. Therefore, in the next sections, we present the major competing approaches that tackle the heterogeneity between PaaS systems with a cloud broker aiming at application portability. All projects were developed during a similar time span as this work.

8.1. Cloud4SOA

Cloud4SOA was an EU project funded by the 7th Framework program[196] from September 2010 until August 2013 [82, 175, 216, 218]. Cloud4SOA focuses on interoperability and portability issues between PaaS clouds. The approach tries to integrate heterogeneous PaaS offerings across different providers that share the same technological ecosystem analogous to our approach. The Cloud4SOA platform consists of a number of components to enable matchmaking, management, monitoring, and migration of applications among multiple cloud offerings.

For a unification of the concepts and capabilities between different cloud systems, they define the Cloud4SOA Semantic Model [218]. The model consists of five tiers. Each tier describes a partial aspect of a PaaS offering and its participants. The tiers are the infrastructure tier, platform tier, application tier, user tier, and the enterprise tier [175]. Figure 8.1 shows how the relevant parts of the model map to our PaaS taxonomy presented in Section 4.4. Although the Cloud4SOA model covers roughly half of our fundamental entities, especially on the second level of the taxonomy, it lacks details compared to our work. The ontology does not distinguish between different software components such as middleware, frameworks, or add-ons but only explicitly models runtimes and services. Also, neither hosting capabilities, extensibility, nor status properties are described. In contrast, their ontology includes many additional properties targeting network, storage, and processing resources. Among others, they include specific QoS such as latency, bandwidth, response time, and thresholds next to capabilities like CPU frequency and disk capacity. However, as we

[196] https://ec.europa.eu/research/fp7

Figure 8.1.: Taxonomy Comparison Cloud4SOA and PaaSfinder

showed in the previous chapters, these properties are rarely provided by PaaS offerings. Additionally, an attribute to support user rankings for providers is included. Cloud4SOA uses OWL ontologies for defining and serializing the PaaS model. Furthermore, the model is implemented in a way that requires different instances per programming language for a single PaaS. Unlike our approach, the knowledge base is limited to 13 providers modeled by taxonomy instances.

Cloud4SOA includes a GUI for faceted browsing and search in the provider directory [82]. The matchmaking algorithm allows the user to distinguish between requirements and preferences. Cloud4SOA uses SPARQL queries for retrieving matching PaaS providers from the knowledge base. All required capabilities must be fulfilled by the providers in the candidate set of the selection. Afterwards, a ranking of the optional requirements is resolved into an ordered result on a percentage basis. This procedure is similar to our two-step selection approach outlined in Section 5.4 but without additional semantic changes to allow partial matching.

Cloud4SOA proposes a homogenized API for deploying and migrating applications between different PaaS systems. The Cloud4SOA API only covers the core set of application deployment functionalities of our proposed PaaS API. Table 8.1 visualizes the differences between our API and those of competing approaches described in this chapter. Cloud4SOA's unified interface does not consider important standard functionalities of application management, e.g.,

Table 8.1.: Unified Interface Operations of Competing Approaches

		Functionality	Description	Cloud4SOA	PaaSport	COAPS
App operations	App	GET	Get app details	✓	✓	✓
		UPDATE	Update the app	✓	✓	✓
		DELETE	Delete the app	✓	✓	✓
	Life Cycle	DEPLOY	Upload app package	✓	✓	✓
		REBUILD	Rebuild app package	✗	✗	✗
		DOWNLOAD	Download app package	✗	✓	✗
		START	Start the app	✓	✓	✓
		STOP	Stop the app	✓	✓	✓
		RESTART	Restart the app	✗	✓	✓
	Scaling	ADD INSTANCE	Scale-out	✗	✓	✓
		DELETE INSTANCE	Scale-in	✗	✓	✓
		SCALE INSTANCE	Scale-up	✗	✗	✓
	Domains	LIST	List all app domains	✗	✓	✗
		GET	Get domain entity	✗	✓	✗
		ADD	Assign domain to the app	✗	✓	✗
		DELETE	Remove domain	✗	✓	✗
		UPDATE	Update domain settings	✗	✓	✗
	Variables	LIST	List all env. variables	✗	✗	✓
		GET	Get an environment variable	✗	✗	✓
		CREATE	Create and set variable	✗	✗	✓
		UPDATE	Update variable value	✗	✗	✓
		DELETE	Remove a variable	✗	✗	✓
	Logging	LIST	Collect all app log files	✗	✗	✗
		GET	Get a specific log file	✗	✗	✗
		DOWNLOAD	Download all logs as archive	✗	✗	✗
	Services	LIST	List all installed services	✗	✓	✗
		GET	Get bound service entity	✗	✓	✗
		ADD	Install and bind to the app	✓	✓	✗
		UPDATE	Update bound service settings	✗	✓	✗
		REMOVE	Remove bound service	✓	✓	✗
General	App	LIST	List all apps	✓	✓	✓
		CREATE	Create the app	✓	✓	✓
	Service	LIST SERVICES	List all available services	✓	✓	✗
		GET SERVICE	Get available service entity	✗	✓	✗
		LIST PLANS	List available service plans	✗	✗	✗
		GET PLAN	Get service plan entity	✗	✗	✗
	Region	LIST	List all available regions	✗	✗	✗
		GET	Get available region entity	✗	✗	✗
			Overall feature coverage	29 %	63 %	45 %

domains, logging, scalability, deployment regions, and a dedicated ability to manage application services. Nonetheless, the proposed API acts as a foundation for many of the following equally motivated research projects. According to the publications, adapters were offered for AWS Elastic Beanstalk, Cloud Foundry, OpenShift, and CloudBees [82, 175], which were later extended with Heroku and cloudControl. The Cloud4SOA platform is implemented in Java and released as open source.[197]

[197] `https://github.com/Cloud4SOA/Cloud4SOA`

8.2. PaaSport

The PaaSport project[198] was an EU project between January 2013 and January 2016 funded by the European Commission's 7th Framework Programme [6, 27–29, 174]. The project focuses on resolving data and application portability issues that exist in the PaaS market. To that end, they define a cloud broker based on a PaaS ontology [29], a recommendation algorithm [28], and an API [174] for application migration between PaaS providers. Their idea of semantically interoperable PaaS solutions is based on a similar notion as our ecosystem portability approach, requiring PaaS offerings that are using the same technological background but different data models and APIs. The cloud broker shall assist customers, especially European Small and Medium-Sized Enterprises (SMEs), to deploy their business applications to the best matching provider for their demands and to be able to migrate their applications avoiding vendor lock-in.

PaaSport defines three semantic models as common vocabulary for their PaaS marketplace and recommendation algorithms. First, an offering model for describing the available PaaS offerings in terms of functionalities, resources, and business characteristics. Second, a corresponding application model for the requirements of an application and third, an SLA model for explicitly modeling the SLAs warranted by providers. Their models are based on and influenced by the semantic layer of Cloud4SOA. PaaSport uses OWL ontologies for building the PaaSport semantic models. Furthermore, the conceptual model of DOLCE and D&S [109] is used as the basic ontology design pattern. Generic PaaS models for systems such as CF can be defined and refined for provider instances. Figure 8.2 shows the coverage of their ontology compared to our proposed PaaS ontology. Besides some detailed second level properties, it is missing the hosting capability, extensibility, status, and a distinction for middleware technologies. In contrast, their ontology includes the same additional properties as described for Cloud4SOA targeting specific QoS and capabilities of network, storage, and processing resources. As argued before, these properties are rarely provided by PaaS offerings. When comparing the PaaSport ontology to the related Cloud4SOA ontology, the latter is broader in scope and targets the entire life cycle of a cloud application. PaaSport on the other hand focuses more on the matchmaking aspect between PaaS offerings. Consequently, the platform layer is more detailed and refined compared to Cloud4SOA (see Figure 8.1). In contrast, Cloud4SOA has its strength within the infrastructure, user, and enterprise tiers [29]. Regarding the knowledge base's sample size, PaaSport only lists between three [28] and eleven providers [29] which is far less than listed in our data set.

The matchmaking algorithms which rely on the defined OWL ontology are using SPARQL queries for retrieving relevant data from the provider repository.

[198] `http://paasport-project.eu`

Figure 8.2.: Taxonomy Comparison PaaSport and PaaSfinder

PaaSport uses Apache Jena[199] for parsing ontologies, processing data, and executing SPARQL queries. Jena is an open source Java framework for building semantic web and linked data applications. Users can query the knowledge base via a GUI. The algorithm uses a two-step process similar to our approach. As first step, a matchmaking via functional parameters is performed which filters potential candidates. Afterwards, the shortlist is ordered by ranking the providers by the user query's nonfunctional parameters [27]. Still, no semantic enhancements are performed on the query to account for possible data or query biases.

Again, according to their documents, the PaaSport API is based on the Cloud4SOA API [82]. Overall, from the available related work, the PaaSport API most closely matches our unified API (see Table 8.1). Except for minor differences, they do not include logging capabilities and propose a fundamentally different approach for binding services to applications. As described in Section 7.4, instead of utilizing environment variables, they suggest using Spring Cloud Connectors to connect applications to services, which is currently only viable for JVM-based applications. According to their deliverables, PaaSport supports OpenShift, Heroku, Cloud Foundry, AWS Beanstalk, and cloudControl. Nevertheless, their prototype seems to be restricted to Java-based applications.[200] The results as well as the source code of their prototype are not

[199] https://jena.apache.org
[200] https://www.youtube.com/watch?v=zZIRJAfp6ug

released to the public. Together with small enhancements, the PaaSport API description was published as DIN SPEC 91337 [88].

8.3. SeaClouds

SeaClouds was an EU 7th Framework project between October 2013 and March 2016 [56, 57, 257]. The goals of SeaClouds are very similar to those of the PaaSport project. The project intents to provide an open source platform to enable application developers to run and manage applications across multiple heterogeneous IaaS and PaaS clouds [56]. Whereas the focus lies on the operational aspects, one of the project's contributions is also the discovery of available offerings [57].

In contrast to the previous approaches, the models of SeaClouds are defined using the TOSCA standard instead of OWL. In particular, every cloud offering is represented as a `node_template` as defined by the TOSCA Simple Profile [260], serialized in YAML syntax. A `node_template` is an abstract concept for specifying software component nodes as part of a topology template. The allowed semantics of the node, e.g., properties and capabilities, are specified by a node type that is defined separately for reuse purposes. The discovery and matchmaking is implemented in the tool DrACO [57]. DrACO is also responsible for dynamically retrieving information about cloud offerings from the Internet and then converting it into a TOSCA representation. Next to CloudHarmony[201], the main source for the listed PaaS offerings is our knowledge base *PaaSfinder* [57]. Consequently, the amount of data included in the provider repository matches our data set. Also, the model closely follows our proposed PaaS taxonomy (see Figure 8.3). Except for the concepts status, extensibility, and frameworks, there are only little differences in the covered information. Again, the approach does not distinguish between native services and add-ons which leads to problems filling the required version attributes that cannot be given for all SaaS or BaaS offerings.

SeaClouds' provider repository can be accessed via a RESTful API or via a web-based UI. The recommendation algorithm of DrACO corresponds to our exact matching implementation, i.e., it only returns offerings that match all the provided capabilities [57]. The algorithm is implemented in Java and provides no partial matching or query adaptations.

SeaClouds does not propose a self-defined or self-developed PaaS API. According to the authors, the unified API for realizing multi-cloud capabilities relies upon the Cloud4SOA API [82] for PaaS providers and jclouds[202] for IaaS providers [56]. However, the SeaClouds implementation shows that the native Java client libraries of CF and OpenShift were used to bridge between

[201] `https://cloudharmony.com`
[202] `https://jclouds.apache.org`

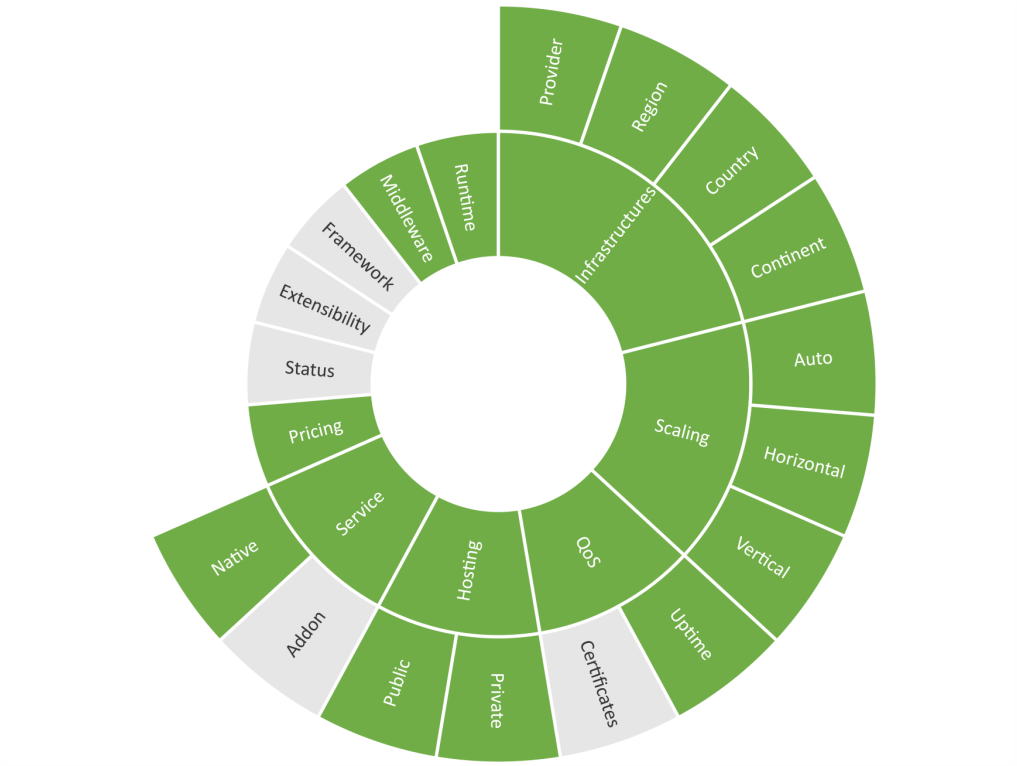


Figure 8.3.: Taxonomy Comparison SeaClouds and PaaSfinder

the PaaS systems. The implementation is not independent of the application's runtime language and needs separate module implementations for different programming languages. The deliverables of SeaClouds are released as open source.[203]

8.4. CompatibleOne

CompatibleOne was a project of Telecom SudParis[204] between November 2010 and October 2012 [324, 357, 387]. CompatibleOne proposes a cloud broker that helps customers with the selection of a suitable provider and multi-cloud deployments. At the core, CompatibleOne is a model and an execution platform. In between, the platform negotiates between user and application requirements and capabilities of the available execution platforms [387].

The CompatibleOne Resource Description System (CORDS) is an object-based description of cloud applications, services, and resources [387]. The CORDS model is based on the OCCI standard [268, 269]. A CORDS manifest is serialized as an XML document [387]. For modeling PaaS offerings, they propose an extension to the OCCI specification [386]. In particular, they define the three types container, database, and router as well as relations among

[203] https://github.com/SeaCloudsEU
[204] https://www.telecom-sudparis.eu

them. With these types they may describe all software components (runtime, framework, middleware, and service) of our model but none of the remaining capabilities. Nevertheless, the flexibility of the OCCI specification gives them the possibility to specify more properties. However, in their experiments they do not give any concrete examples. Therefore, we omit a detailed visual comparison with our model. The size of the knowledge base of the project is unknown and potentially limited to the amount of adapter implementations. Parts of their proposal made it into the final OCCI PaaS extension [270].

The execution platform called Advanced Capabilities for CORDS (ACCORDS), is a cloud application provisioning and deployment system. The ACCORDS parser processes and evaluates a given CORDS manifest, finally producing a resource provisioning plan if a valid provider configuration can be found [387]. The recommendation process of ACCORDS is slightly different from the other competing approaches. Instead of generating a set of suitable providers based on the submitted requirements and then presenting that set to the user, the broker only communicates if a valid combination is found. It optimizes the requirements and picks the ideal combination for the user that can afterwards be deployed by the deployment engine.

The provisioning plan can be used to deploy the application to the cloud providers. To that end, the ACCORDS platform communicates with the deployment abstraction layer, the COAPS API [324]. The COAPS API is a unified interface to PaaS management interfaces. Compared to our findings, COAPS is missing various essential operations of cloud platforms such as domains, monitoring of applications, or the management of services and deployment regions [324, 357] (see Table 8.1). Their implementation includes connectors for Cloud Foundry, OpenShift, and Google App Engine [153, 324]. All the described components communicate via RESTful HTTP communication. The source code of the project is available as open source.[205]

[205] `https://github.com/compatibleone/accords-platform`

9. Summary and Conclusion

Although PaaS offers many benefits to its users, several deficiencies exist that hinder the usability and create lock-in effects [124, 217, 283, 332]. Therefore, in this thesis, we analyzed and addressed various challenges of application portability in PaaS. We observed and tackled obstacles along the life cycle of an application, from the selection of an appropriate cloud provider up to the deployment and its operation. We used this workflow as a framework to identify and analyze portability problems in the application life cycle. Hereby, we presented approaches for solving important subproblems rather than creating a holistic middleware approach for the entirety of the portability problem. The initial situation already showed that such an approach was never successfully tackled by standardization organizations and therefore also is not feasible for a small research group [230, 262, 271]. Hence, at first, we identified decisive factors that hinder the portability. Afterwards, we developed proposals to reduce or prevent these problems in the future. Overall, the thesis advances the state of knowledge of PaaS and gives recommendations for suitable standards and best practices to avoid vendor lock-in and retain application portability.

The main contributions of the thesis include a new classification and model of PaaS. Next, a knowledge base with enhanced provider selection and application portability matching. Furthermore, we give evidence for application portability issues together with a suitable measurement framework for cloud migrations. Finally, a unified PaaS application management interface is presented to improve the interoperability among PaaS.

Despite the prominence of the term PaaS, the market is not as homogeneous as existing definitions suggest [18, 134]. For that reason, we first studied the literature and the state of the art, and extracted a more precise classification and model for PaaS that serves as a basis for all investigations and following steps of the thesis. We propose three distinct PaaS clusters in between the boundaries of IaaS and SaaS for refining the differentiation between PaaS offerings. Furthermore, we present a conceptual model of PaaS that resembles the state of the art and capture model instances in a comprehensive PaaS knowledge base. The model also achieves a semantic harmonization of typical features among offerings. By means of this new conceptualization, the breadth and differences in the market can be communicated more clearly among stakeholders. This advances the state of knowledge and provides an overview of the current state of the art in this area. Moreover, it enables a better distinction between PaaS

and other cloud offerings, and consequently allows us to study and overcome existing portability issues among them.

Due to the size of the market and the provider's differences, provider selection is a complex task. Moreover, before the initiation of our knowledge base, no structured knowledge repository was available [349, 394]. Based on our formalized model, we present an approach for PaaS provider selection. Besides typical multi-criteria selection [360], we propose and validate the possibility of application portability matching based on the technological ecosystems of the providers in the absence of standards in this field [262, 271]. We show that the software ecosystem of a PaaS matched with the requirements of an application can be used as a heuristic for evaluating the application portability between PaaS providers. Furthermore, we suggest several enhancements to typical but often neglected problems in technological knowledge bases, such as policies for data governance and semantic enhancements for partial matching of user queries. In that regard, we show that the quality of the knowledge bases must not be taken for granted and that appropriate precautions and improvements are necessary. Such measures contribute to the quality of selection results and improve the satisfaction of users. The approach is validated by *PaaSfinder*[206], an open source tool for supporting PaaS provider selection. Usage statistics and user feedback show that the decision support and the approach is accepted by users and considered useful.

As a follow-up study, we validate the practicability of our application portability matching with a real-world application migration among PaaS providers. The study provides a new perspective on migration between clouds which is lacking in related works [165]. Especially the use of a real-world application supports the generalizability of the study results. It uncovers existing migration efforts and identifies areas of functionality that are particularly problematic with respect to portability. This provides hints for assessing and avoiding such issues before designing applications or selecting providers. To make migration efforts measurable and comparable, we adapt the measurement framework by Lenhard, Harrer, and Wirtz [207] which is based on the ISO/IEC SQuaRE quality model [159] to PaaS environments. Hence, we suggest a generalized framework and toolkit for measuring and comparing the effort of cloud migrations. The results show that despite trying to design applications as vendor-neutral as possible, the unification of runtime environments and management interfaces between PaaS vendors remains an important topic.

Consequently, we propose a unified PaaS application management interface as our last major contribution. As identified in the case study, the interoperability of the management interfaces has a substantial impact on the portability of applications [152, 283]. Although management tasks are semantically equivalent for the deployment and operation of applications, very different and nonstandardized interfaces are used by the providers. Therefore, we develop a

[206] `https://paasfinder.org`

proposal for a unified management interface to enable a consistent management of applications across several providers [175, 231, 262]. Our approach reveals differences and shortcomings of existing initiatives [77, 82, 174, 324, 399] and provides valuable hints for standardization efforts. The approach further distinguishes itself from related work and the scarce standardization efforts by a working reference implementation[207].

Overall, the thesis provides several implications for theory and practice. Our model design shows that greater attention must be paid to the model-reality fit in research. Otherwise, developments may lead to artificial models that cannot be mapped to reality. Next, our observations regarding decision support and knowledge bases reveal that more emphasis must be given to data and query quality assurance. The findings further aggravate that similar scoped research should always include a practical evaluation of the proposed concepts. With our migration metrics, we provide a new way and framework for quantifying migration effort of application migrations in the cloud. Thus, researchers can compare the migration efforts that have occurred using reproducible and objective metrics. Finally, the unified application management interface presents a way to manage applications uniformly between different providers. The results and the improved portability and interoperability can be used as a starting point to evaluate multi-cloud scenarios in more detail [1].

In total, the view and comparability of PaaS providers could be improved in practice. This results in less biased customer information and more chances for portability and interoperability. Literature also reveals that the problems targeted in the thesis are not limited to PaaS but often repeat themselves for new emerging technologies. As an example, serverless and FaaS currently face similar problems including a lack of a clear terminology and scattered vision about the field [228, 364]. This makes benefits for users obscure and misinformation and confusion is inevitable. The thesis shows ways to counteract this.

Certainly, there exist several limitations and areas for future work. Many of them have already been discussed throughout the respective chapters. Although our portability heuristic proves to be a solid indicator for portability between PaaS systems, it does not deliver complete portability. Most importantly, the implementation perspective must be further investigated to minimize code changes due to low-level details. Whereas the case study showed that there are attempts that try to improve on that end, there is still work to be done in that area. We did not evaluate more in this direction, as it is hard to influence anything on this level without defining requirements for the vendors that are difficult to enforce. Also, we experienced that a model and knowledge base with selection capabilities alone does not provide great user experience and accurate

[207] `https://github.com/stefan-kolb/nucleus`

selection results due to data and query biases. Our enhancements with semantic algorithms mitigate these shortcomings. However, they further weaken the perceived portability guarantees and possibly lead to more migration effort. As we only conducted a migration of a particular cloud-native application, this affects the validity and generalizability of the approach. If the portability guarantees hold in the same way for other applications in other programming languages as for our cloud-native application is not safe to assume.

We see several points for further research that are either not targeted in this work or are facilitated by its contributions. In addition to further studies on the implementation perspective, the case study showed that cost-performance ratios are hardly perceivable for users at the moment. During the study, we realized that there are substantial differences between the providers. First, compared to the explicit VM configurations of IaaS offerings, the instance powers are unclear and incomparable between PaaS providers. Second, the applications may influence each other due to the physical co-location inside the virtualized environment [327]. Even though such values are difficult to measure, they provide important information to customers. Especially when application portability between platforms improves further, the differentiation via performance and pricing becomes more important.

We would like to conclude with a more general lookout on the applicability of PaaS. During the collaboration with Blinks Labs, the developers constantly reevaluated the suitability of the technology stack for the requirements of the application. In the beginning, the PaaS technology proved to be the perfect fit in terms of time-to-market, abstraction level, costs, and performance. However, with the growth of the company and user base, they hit several limitations of PaaS in terms of availability, network performance, and costs. Ultimately, this led to a gradual migration to more low-level technologies that allow more control for the developers and better scalability. Overall, it is interesting to investigate how PaaS can cope with the growth of an application over time. Or, to put it differently: To what extent do the advantages of the abstraction and the opinionated framework outweigh the benefits of lower-level technologies, such as IaaS?

To sum up, the thesis illustrates that application portability in PaaS systems is a challenging and relevant topic. We show that a common understanding of the conceptual foundations is a prerequisite for targeting the portability of applications. Next, we point out that standards alone do not guarantee portability and that portability can also be achieved without restricting standards. The contributions of our research show that portability is influenced by multiple aspects along the application life cycle. Overall, we think that the contributions of the thesis improve multiple aspects of portability in PaaS and can provide foundations for related scenarios in other research areas.

Bibliography

[1] L. Acquaviva, P. Bellavista, F. Bosi, A. Corradi, L. Foschini, S. Monti, and A. Sabbioni, "NoMISHAP: A Novel Middleware Support for High Availability in Multicloud PaaS," *IEEE Cloud Computing*, vol. 4, no. 4, pp. 60–72, Jul. 2017. (Cited on page 207.)

[2] L. A. Adamic and B. A. Huberman, "Zipf's law and the Internet," *Glottometrics*, vol. 3, no. 1, pp. 143–150, 2002. (Cited on page 98.)

[3] E. Afgan, A. Lonie, J. Taylor, and N. Goonasekera, "CloudLaunch: Discover and Deploy Cloud Applications," *Preprint*, 2018. (Cited on page 190.)

[4] Y. M. Afify, N. L. Badr, I. F. Moawad, and M. F. Tolba, "Evaluation of cloud service ontologies," in *Proceedings of the Conference on Intelligent Computing and Information Systems*. IEEE, Dec. 2017. (Cited on pages 83, 89, 90, 97, and 110.)

[5] D. Agrawal, A. E. Abbadi, F. Emekci, and A. Metwally, "Database Management as a Service: Challenges and Opportunities," in *Proceedings of the Conference on Data Engineering*. IEEE, Mar. 2009. (Cited on page 67.)

[6] F. AhmadiZeleti, I. Hassan, S. Abbas, A. Ojo, and L. Porwol, "Architecting the Recommendation Layer of a Platform-as-a-Service e-Marketplace," in *Proceedings of the Joint Conference on Software Technologies*. SCITEPRESS, 2016. (Cited on pages 97, 109, and 200.)

[7] M. Al-Roomi, S. Al-Ebrahim, S. Buqrais, and I. Ahmad, "Cloud Computing Pricing Models: A Survey," *International Journal of Grid and Distributed Computing*, vol. 6, no. 5, pp. 93–106, Oct. 2013. (Cited on page 93.)

[8] A. J. Albrecht and J. E. Gaffney, "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 6, 1983. (Cited on page 166.)

[9] L. F. J. Albuquerque, F. S. Ferraz, R. F. A. P. Oliveira, and S. M. L. Galdino, "Function-as-a-Service X Platform-as-a-Service: Towards a Comparative Study on FaaS and PaaS," in *Proceedings of the Conference on Software Engineering Advances*, 2017. (Cited on page 68.)

[10] A. Ali, S. M. Shamsuddin, and F. E. Eassa, "Ontology-based Cloud Services Representation," *Research Journal of Applied Sciences, Engineering and Technology*, vol. 8, no. 1, pp. 83–94, Jul. 2014. (Cited on pages 83, 90, and 110.)

[11] M. Almorsy, J. Grundy, and I. Müller, "An Analysis of the Cloud Computing Security Problem," 2016. (Cited on pages 44 and 45.)

[12] H. Amarasinghe and A. Karmouch, "SDN-Based Framework for Infrastructure as a Service Clouds," in *Proceedings of the Conference on Cloud Computing*. IEEE, Jun. 2016. (Cited on page 23.)

[13] A. Amato, B. D. Martino, and S. Venticinque, "Cloud Brokering as a Service," in *Proceedings of the Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. IEEE, Oct. 2013. (Cited on page 132.)

[14] G. F. Anastasi, E. Carlini, M. Coppola, and P. Dazzi, "QoS-aware genetic Cloud Brokering," *Future Generation Computer Systems*, vol. 75, pp. 1–13, Oct. 2017. (Cited on pages 35 and 132.)

[15] V. Andrikopoulos, Z. Song, and F. Leymann, "Supporting the Migration of Applications to the Cloud through a Decision Support System," in *Proceedings of the Conference on Cloud Computing*, 2013. (Cited on pages 28, 133, 137, and 166.)

[16] D. Androcec, N. Vrcek, and J. Seva, "Cloud Computing Ontologies: A Systematic Review," in *Proceedings of the Conference on Models and Ontology-Based Design of Protocols, Architectures and Services*. IARIA, 2012. (Cited on pages 83 and 109.)

[17] A. Apostolico and C. Guerra, "The Longest Common Subsequence Problem Revisited," *Algorithmica*, vol. 2, no. 1-4, pp. 315–336, 1987. (Cited on page 127.)

[18] 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. (Cited on pages 3, 4, 18, 19, 20, 44, 45, 64, 69, and 205.)

[19] D. Arnott and G. Pervan, "A critical analysis of decision support systems research," *Journal of Information Technology*, vol. 20, no. 2, pp. 67–87, 2005. (Cited on page 30.)

[20] M. Ayad, M. Taher, and A. Salem, "Real-Time Mobile Cloud Computing: A Case Study in Face Recognition," in *Proceedings of the Conference on Advanced Information Networking and Applications Workshops*. IEEE, May 2014. (Cited on pages 44 and 45.)

[21] L. Badger, T. Grance, R. Patt-Corner, and J. Voas, "Cloud Computing Synopsis and Recommendations," NIST, Special Publication 800-146, 2012. (Cited on pages 6, 7, 9, 41, 69, 73, 83, 109, 137, and 164.)

[22] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, "Serverless Computing: Current Trends and Open Problems," in *Research Advances in Cloud Computing*. Springer Singapore, 2017, vol. abs/1706.03178, pp. 1–20. (Cited on page 68.)

[23] G. Baranwal and D. P. Vidyarthi, "A cloud service selection model using improved ranked voting method," *Concurrency and Computation: Practice*

and Experience, vol. 28, no. 13, pp. 3540–3567, Jan. 2016. (Cited on page 132.)

[24] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, p. 164, Dec. 2003. (Cited on pages 19, 22, and 65.)

[25] C. Bartolini, D. El Kateb, Y. Le Traon, and D. Hagen, "Cloud providers viability," *Electronic Markets*, Jan. 2018. (Cited on pages 4, 19, 26, 53, and 169.)

[26] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? An analysis of topics and trends in Stack Overflow," *Empirical Software Engineering*, vol. 19, no. 3, pp. 619–654, Nov. 2012. (Cited on page 59.)

[27] N. Bassiliades, E. Kontopoulos, G. Meditskos, and M. Symeonidis, "PaaSport Recommendation Algorithm and Model – Deliverable 2.1," PaaSport Project, Tech. Rep., 2014. (Cited on pages 134, 200, and 201.)

[28] N. Bassiliades, M. Symeonidis, G. Meditskos, E. Kontopoulos, P. Gouvas, and I. Vlahavas, "A Semantic Recommendation Algorithm for the PaaSport Platform-as-a-Service Marketplace," *Expert Systems with Applications*, Sep. 2016. (Cited on pages 97, 109, 134, and 200.)

[29] N. Bassiliades, M. Symeonidis, P. Gouvas, E. Kontopoulos, G. Meditskos, and I. Vlahavas, "PaaSport Semantic Model: An Ontology for a Platform-as-a-Service Semantically Interoperable Marketplace," *Data & Knowledge Engineering*, vol. 113, pp. 81–115, Jan. 2018. (Cited on pages 83, 90, 110, and 200.)

[30] D. Beimborn, T. Miletzki, and S. Wenzel, "Platform as a Service (PaaS)," *Business & Information Systems Engineering*, vol. 3, no. 6, pp. 381–384, 2011. (Cited on pages 41, 43, 44, 45, and 83.)

[31] V. Belton and T. Gear, "On a Short-coming of Saaty's Method of Analytic Hierarchies," *Omega*, vol. 11, no. 3, pp. 228–230, 1983. (Cited on pages 33 and 34.)

[32] M. Benedict, "Coming to (your) Terms with Platform-as-a-Service (PaaS)," Progress Software Corporation, Tech. Rep., 2013. (Cited on pages 45, 46, 48, and 86.)

[33] A. Benlian, T. Hess, and P. Buxmann, "Drivers of Saas-Adoption – An Empirical Study of Different Application Types," *Business & Information Systems Engineering*, vol. 1, no. 5, p. 357, 2009. (Cited on pages 20, 24, and 26.)

[34] P. V. Beserra, A. Camara, R. Ximenes, A. B. Albuquerque, and N. C. Mendonca, "Cloudstep: A Step-by-Step Decision Process to Support Legacy Application Migration to the Cloud," in *Proceedings of the Workshop Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, 2012. (Cited on pages 28, 137, and 165.)

[35] A. Beslic, R. Bendraou, J. Sopena, and J.-Y. Rigolet, "Towards a solution avoiding Vendor Lock-in to enable Migration Between Cloud Platforms," in *Proceedings of the Workshop Model-Driven Engineering for High Performance and Cloud Computing*, 2013. (Cited on page 165.)

[36] C.-P. Bezemer and A. Zaidman, "Multi-Tenant Saas Applications: Maintenance Dream or Nightmare?" in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and Workshop on Principles of Software Evolution (IWPSE)*. ACM, 2010, pp. 88–92. (Cited on page 25.)

[37] S. Bhardwaj, L. Jain, and S. Jain, "Cloud Computing: A Study of Infrastructure as a Service (IaaS)," *International Journal of Engineering and Information Technology*, vol. 2, no. 1, pp. 60–63, 2010. (Cited on pages 20, 22, 44, and 45.)

[38] F. Biscotti, Y. V. Natis, M. Pezzini, P. Malinverno, J. Thompson, M. Cantara, and J. Murphy, "Market Trends: Platform as a Service, Worldwide, 2013–2018, 2Q14 Update," Gartner, Tech. Rep., 2014. (Cited on pages 3, 4, and 169.)

[39] J. Bitzer, "Commercial versus open source software: the role of product heterogeneity in competition," *Economic Systems*, vol. 28, no. 4, pp. 369–381, 2004. (Cited on page 73.)

[40] A. Bockmayr and J. N. Hooker, "Constraint Programming," *Handbooks in Operations Research and Management Science*, vol. 12, pp. 559–600, 2005. (Cited on page 132.)

[41] B. W. Boehm, B. K. Clark, E. Horowitz, A. W. Brown, D. Reifer, S. Chulani, R. Madachy, and B. Steece, *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000, ISBN: 9780130266927. (Cited on page 165.)

[42] M. Böhm, S. Leimeister, C. Riedl, and H. Krcmar, *Cloud Computing – Outsourcing 2.0 or a new Business Model for IT Provisioning?* Wiesbaden: Gabler, 2011, pp. 31–56, ISBN: 9783834964922. (Cited on pages 3, 18, and 19.)

[43] P. Boldi, F. Bonchi, C. Castillo, and S. Vigna, "From "Dango" to "Japanese Cakes": Query Reformulation Models and Patterns," in *Proceedings of the Joint Conference on Web Intelligence and Intelligent Agent Technology*. IEEE, 2009. (Cited on page 121.)

[44] Borst, Willem Nico, "Construction of Engineering Ontologies for Knowledge Sharing and Reuse," Ph.D. dissertation, University of Twente, 1997. (Cited on page 87.)

[45] S. Borzsony, D. Kossmann, and K. Stocker, "The Skyline Operator," in *Proceedings of the Conference on Data Engineering*. IEEE, 2001, pp. 421–430. (Cited on page 33.)

[46] J. Bosch, "Software Product Families in Nokia," in *Proceedings of the Conference on Software Product Lines*, Springer. Springer Berlin Heidelberg, 2005, pp. 2–6. (Cited on pages 40 and 79.)

[47] ——, "From Software Product Lines to Software Ecosystems," in *Proceedings of the Software Product Line Conference*. Carnegie Mellon University, 2009, pp. 111–119. (Cited on pages 40, 79, and 80.)

[48] J. Bosch and P. Bosch-Sijtsema, "From integration to composition: On the impact of software product lines, global development and ecosystems," *Journal of Systems and Software*, vol. 83, no. 1, pp. 67–76, Jan. 2010. (Cited on page 79.)

[49] J. Bosch and P. M. Bosch-Sijtsema, "Softwares Product Lines, Global Development and Ecosystems: Collaboration in Software Engineering," in *Collaborative Software Engineering*. Springer, 2010, pp. 77–92. (Cited on page 79.)

[50] 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," Jul. 2012, http://ec.europa.eu/information_society/activities/cloudcomputing/docs/quantitative_estimates.pdf. (Cited on page 76.)

[51] J. Bragge, P. Korhonen, H. Wallenius, and J. Wallenius, "Bibliometric Analysis of Multiple Criteria Decision Making/Multiattribute Utility Theory," in *Multiple Criteria Decision Making for Sustainable Energy and Transportation Systems*. Springer, 2010, pp. 259–268. (Cited on page 33.)

[52] M. A. Bramer, "A survey and critical review of expert systems research," *Introductory Readings in Expert Systems*, pp. 3–29, 1982. (Cited on page 31.)

[53] L. Braubach, K. Jander, and A. Pokahr, "A Middleware for Managing Non-Functional Requirements in Cloud PaaS," in *Proceedings of the Conference on Cloud and Autonomic Computing*. IEEE, Sep. 2014. (Cited on pages 44 and 45.)

[54] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *Journal of Systems and Software*, vol. 80, no. 4, pp. 571–583, Apr. 2007. (Cited on pages 42 and 43.)

[55] L. C. Briand, S. Morasca, and V. R. Basily, "Property-based software engineering measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, 1996. (Cited on page 150.)

[56] A. Brogi, J. Carrasco, J. Cubo, F. D'Andria, A. Ibrahim, E. Pimentel, and J. Soldani, "EU Project SeaClouds: Adaptive Management of Service-based Applications Across Multiple Clouds," in *Proceedings of the Conference on Cloud Computing and Services Science*, 2014. (Cited on pages 35, 109, 170, 191, and 202.)

[57] A. Brogi, P. Cifariello, and J. Soldani, "DRACO: Discovering available cloud offerings," *Computer Science-Research and Development*, vol. 32, no. 3-4, pp. 269–279, 2017. (Cited on page 202.)

[58] D. M. Buede and D. T. Maxwell, "Rank disagreement: A comparison of multi-criteria methodologies," *Journal of Multi-Criteria Decision Analysis*, vol. 4, no. 1, pp. 1–21, 1995. (Cited on page 33.)

[59] P. Buxmann, T. Hess, and S. Lehmann, "Software as a Service," *Wirtschaftsinformatik*, vol. 50, no. 6, pp. 500–503, 2008. (Cited on page 25.)

[60] C. F. Cargill, "Why Standardization Efforts Fail," *Journal of Electronic Publishing*, vol. 14, no. 1, Aug. 2011. (Cited on pages 4, 28, 76, and 188.)

[61] N. G. Carr, *Does It Matter?: Information Technology and the Corrosion of Competitive Advantage*. Harvard Business Review Press, 2004, ISBN: 9781591394440. (Cited on pages 3 and 39.)

[62] J. Carrasco, F. Durán, and E. Pimentel, "Trans-Cloud: CAMP/TOSCA-based Bidimensional Cross-Cloud," *Computer Standards & Interfaces*, vol. 58, pp. 167–179, May 2018. (Cited on pages 44 and 45.)

[63] B. Carterette and R. Jones, "Evaluating Search Engines by Modeling the Relationship between Relevance and Clicks," in *Proceedings of the Advances in Neural Information Processing Systems*, 2008, pp. 217–224. (Cited on page 98.)

[64] L. Carvalho, M. Fleming, A. Hilwa, R. P. Mahowald, and B. McGrath, "Worldwide Competitive Public Cloud Platform as a Service 2014–2018 Forecast and 2013 Vendor Shares," IDC, Tech. Rep., 2014. (Cited on page 3.)

[65] S. Challita, F. Zalila, C. Gourdin, and P. Merle, "A Precise Model for Google Cloud Platform," in *Proceedings of the Conference on Cloud Engineering*, Orlando, Florida, United States, Apr. 2018. (Cited on page 183.)

[66] D. Chappell, "What Is an Application Platform?" Chappell & Associates, Tech. Rep., 2011. (Cited on pages 57 and 58.)

[67] M. A. Chauhan and M. A. Babar, "Migrating Service-Oriented System to Cloud Computing: An Experience Report," in *Proceedings of the Conference on Cloud Computing*, 2011. (Cited on page 165.)

[68] ——, "Towards Process Support for Migrating Applications to Cloud Computing," in *Proceedings of the Conference on Cloud and Service Computing*, 2012. (Cited on page 165.)

[69] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski, "Appscale: Scalable and Open AppEngine Application Development and Deployment," in *Cloud Computing*. Springer, 2010, pp. 57–70. (Cited on page 190.)

[70] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) 1.1," *W3C Note*, 2001. (Cited on page 183.)

[71] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002, ISBN: 9780201703320. (Cited on pages 40 and 79.)

[72] Cloud Computing Use Case Discussion Group, "Cloud Computing Use Cases White Paper – Version 4.0," Jul. 2010, http://opencloudmanifesto.org/Cloud_Computing_Use_Cases_Whitepaper-4_0.pdf. (Cited on pages 10, 26, 170, and 193.)

[73] Cloud Standards Consumer Council, "Interoperability and Portability for Cloud Computing: A Guide – Version 2.0," http://www.cloud-council.org/deliverables/CSCC-Interoperability-and-Portability-for-Cloud-Computing-A-Guide.pdf, Dec. 2017. (Cited on pages 26, 28, 29, 30, and 35.)

[74] K. Coyle, "Open Source, Open Standards," *Information Technology and Libraries*, vol. 21, no. 1, pp. 33–36, 2002. (Cited on pages 27 and 140.)

[75] CSMIC, "Service Measurement Framework Version 2.1," Carnegie Mellon University Silicon Valley, Tech. Rep., Jul. 2014. (Cited on pages 56, 84, and 93.)

[76] D. Cunha, P. Neves, and P. Sousa, "A Platform-as-a-Service API Aggregator," in *Advances in Information Systems and Technologies*. Springer, 2013. (Cited on page 191.)

[77] ——, "PaaS Manager: A Platform-as-a-service Aggregation Framework," *Computer Science and Information Systems*, vol. 11, no. 4, 2014. (Cited on pages 10, 109, 170, 174, 176, 180, 191, 193, and 207.)

[78] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing*, vol. 6, no. 2, pp. 86–93, 2002. (Cited on page 183.)

[79] M. A. Cusumano and D. B. Yoffie, "Software Development on Internet Time," *IEEE Computer*, vol. 32, no. 10, pp. 60–69, 1999. (Cited on page 93.)

[80] V. G. da Silva, M. Kirikova, and G. Alksnis, "Containers for Virtualization: An Overview," *Applied Computer Systems*, vol. 23, no. 1, pp. 21–27, May 2018. (Cited on page 19.)

[81] F. J. Damerau, "A Technique for Computer Detection and Correction of Spelling Errors," *Communications of the ACM*, vol. 7, no. 3, pp. 171–176, Mar. 1964. (Cited on page 127.)

[82] F. D'Andria, S. Bocconi, J. G. Cruz, J. Ahtes, and D. Zeginis, "Cloud4SOA: Multi-cloud Application Management Across PaaS Offerings," in *Proceedings of the Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012, pp. 407–414. (Cited on pages 10, 109, 134, 170, 174, 176, 180, 191, 193, 197, 198, 199, 201, 202, and 207.)

[83] A. V. Dastjerdi, S. G. H. Tabatabaei, and R. Buyya, "An Effective Architecture for Automated Appliance Management System Applying Ontology-Based Cloud Discovery," in *Proceedings of the Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010. (Cited on page 133.)

[84] M. Denne, "Pricing Utility Computing Services," *International Journal of Web Services Research*, vol. 4, no. 2, pp. 114–127, Apr. 2007. (Cited on page 93.)

[85] B. Di Martino, "Applications Portability and Services Interoperability among Multiple Clouds," *IEEE Cloud Computing*, vol. 1, no. 1, 2014. (Cited on pages 8, 9, 113, 137, and 164.)

[86] G. Di Modica and O. Tomarchio, "Matching the business perspectives of providers and customers in future cloud markets," *Cluster Computing*, vol. 18, no. 1, pp. 457–475, 2015. (Cited on pages 83, 90, and 110.)

[87] T. Dillon, C. Wu, and E. Chang, "Cloud Computing: Issues and Challenges," in *Proceedings of the Conference on Advanced Information Networking and Applications*. Ieee, 2010, pp. 27–33. (Cited on pages 4, 44, and 45.)

[88] *DIN SPEC 91337 – Unified Application Management Interface for Cloud Application Platforms*, DIN Std., Mar. 2017. (Cited on pages 109, 189, 192, and 202.)

[89] DMTF, "Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol – An Interface for Managing Cloud Infrastructure," 2012. (Cited on page 189.)

[90] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu, "Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends," in *Proceedings of the Conference on Cloud Computing*. IEEE, Jun. 2015. (Cited on page 65.)

[91] A. Dubey, J. Mohiuddin, and A. Baijal, "Emerging Platform Wars in Enterprise Software," McKinsey, Tech. Rep., 2008. (Cited on pages 44 and 45.)

[92] D. Durkee, "Why Cloud Computing Will Never Be Free," *Communications of the ACM*, vol. 53, no. 5, pp. 62–69, 2010. (Cited on pages 53 and 73.)

[93] DZone Research, "The 2014 Cloud Platform Research Report," Dzone, Inc., Tech. Rep., 2014. (Cited on page 96.)

[94] M. Eisa, M. Younas, K. Basu, and H. Zhu, "Trends and Directions in Cloud Service Selection," in *Proceedings of the Symposium on Service-Oriented System Engineering*. IEEE, Mar. 2016. (Cited on pages 10, 34, 35, 83, and 97.)

[95] A. Elhabbash, F. Samreen, J. Hadley, and Y. Elkhatib, "Cloud Brokerage: A Systematic Survey," 2018. (Cited on pages 5, 34, 35, 36, 113, and 131.)

[96] Engine Yard, "PaaS Is Dead: The Digital Evolution of a "Dying" Platform," https://www.engineyard.com/paas-is-dead, Mar. 2018. (Cited on pages 44 and 45.)

[97] T. Erl, *SOA Principles of Service Design*, M. L. Taub, Ed. Pearson Education, 2008, ISBN: 9788131723098. (Cited on page 68.)

[98] M. Eurich, A. Giessmann, T. Mettler, and K. Stanoevska-Slabeva, "Revenue Streams of Cloud-based Platforms: Current State and Future Directions," in *Proceedings of Americas Conference on Information Systems*, 2011. (Cited on pages 44 and 45.)

[99] 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, Nov. 1995. (Cited on pages 55 and 85.)

[100] A. J. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. K. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forgó, T. Sharif, and C. Sheridan, "OPTIMIS: A holistic approach to cloud service provisioning," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 66–77, Jan. 2012. (Cited on page 35.)

[101] S. Few, "Dashboard Confusion," *Intelligent Enterprise*, 2004. (Cited on page 99.)

[102] G. Fick and R. H. Sprague, Eds., *Decision Support Systems: Issues and Challenges*, vol. 11. Elsevier, 1980. (Cited on page 30.)

[103] P. N. Finlay, "Decision Support Systems and Expert Systems: A Comparison of Their Components and Design Methodologies," *Computers & Operations Research*, vol. 17, no. 6, pp. 535–543, Jan. 1990. (Cited on pages 30, 31, and 32.)

[104] F. Firozbakht, W. J. Obidallah, and B. Raahemi, "Cloud Computing Service Discovery Framework for IaaS and PaaS Models," in *Proceedings of the Second Conference on Internet of Things, Data and Cloud Computing*. ACM Press, 2017. (Cited on pages 44 and 45.)

[105] P. C. Fishburn, "Additive Utilities with Incomplete Product Sets: Application to Priorities and Assignments," *Operations Research*, vol. 15, no. 3, pp. 537–542, 1967. (Cited on pages 33 and 122.)

[106] F. N. Ford, "Decision Support Systems and Expert Systems: A Comparison," *Information & Management*, vol. 8, no. 1, pp. 21–26, Jan. 1985. (Cited on pages 30, 31, and 32.)

[107] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud Computing and Grid Computing 360-Degree Compared," in *Proceedings of the Grid Computing Environments Workshop*. IEEE, Nov. 2008. (Cited on pages 3, 17, and 18.)

[108] F. D. Frate, P. Garg, A. P. Mathur, and A. Pasquini, "On the Correlation between Code Coverage and Software Reliability," in *Proceedings of the Symposium on Software Reliability Engineering*. IEEE Comput. Soc. Press, 1995. (Cited on page 187.)

[109] A. Gangemi and P. Mika, "Understanding the Semantic Web through Descriptions and Situations," in *Proceedings of the OTM Confederated Conferences "On the Move to Meaningful Internet Systems"*. Springer, 2003, pp. 689–706. (Cited on page 200.)

[110] J. García-Galán *et al.*, "Migrating to the Cloud: a Software Product Line based analysis," in *Proceedings of the Conference on Cloud Computing and Services Science*, 2013. (Cited on page 133.)

[111] S. K. Garg, S. Versteeg, and R. Buyya, "A framework for ranking of cloud computing services," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1012–1023, Jun. 2013. (Cited on page 132.)

[112] Gartner Inc., "Gartner Says Cloud Consumers Need Brokerages to Unlock the Potential of Cloud Services," https://www.gartner.com/newsroom/id/1064712, Jul. 2009. (Cited on page 34.)

[113] S. Gebus and K. Leiviskä, "Knowledge acquisition for decision support systems on an electronic assembly line," *Expert Systems with Applications*, vol. 36, no. 1, pp. 93–101, Jan. 2009. (Cited on pages 31, 118, 121, and 124.)

[114] M. Geiger, S. Harrer, J. Lenhard, and G. Wirtz, "On the Evolution of BPMN 2.0 Support and Implementation," in *Proceedings of the Symposium on Service-Oriented System Engineering*. Institute of Electrical and Electronics Engineers (IEEE), Mar. 2016. (Cited on pages 4, 28, and 76.)

[115] M. R. Genesereth and N. J. Nilsson, "Logical Foundations of Artificial Intelligence," *Morgan Kaufmann*, vol. 58, 1987. (Cited on page 88.)

[116] F. Gens, "Worldwide and Regional Public IT Cloud Services 2014–2018 Forecast," IDC, Tech. Rep., 2014. (Cited on pages 4 and 169.)

[117] J. C. Giarratano and G. Riley, *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co., 1989, ISBN: 9780878353354. (Cited on page 31.)

[118] A. Giessmann and K. Stanoevska, "Platform as a Service – A Conjoint Study on Consumers' Preferences," in *Proceedings of the Conference on Information Systems*, 2012. (Cited on pages 43, 44, and 45.)

[119] A. Giessmann and K. Stanoevska-Slabeva, "Business Models of Platform as a Service (PaaS) Providers: Current State and Future Directions," *Journal of Information Technology Theory and Application*, vol. 13, no. 4, p. 4, 2012. (Cited on page 41.)

[120] M. Godse and S. Mulik, "An Approach for Selecting Software-as-a-Service (SaaS) Product," in *Proceedings of the Conference on Cloud Computing*, 2009. (Cited on page 135.)

[121] A. Gomez-Perez and O. Corcho, "Ontology Languages for the Semantic Web," *IEEE Intelligent Systems*, vol. 17, no. 1, pp. 54–60, Jan. 2002. (Cited on page 91.)

[122] C. Goncalves, D. Cunha, P. Neves, P. Sousa, J. P. Barraca, and D. Gomes, "Towards a Cloud Service Broker for the Meta-Cloud," in *Proceedings of*

the Conferencia sobre Redes de Computadores, Nov. 2012, Aveiro, Portugal. (Cited on pages 109 and 135.)

[123] S. Gong and K. M. Sim, "CB-Cloudle: A Centroid-based Cloud Service Search Engine," in *Proceedings of the MultiConference of Engineers and Computer Scientists*, 2014. (Cited on page 133.)

[124] F. Gonidis, I. Paraskakis, and D. Kourtesis, "Addressing the Challenge of Application Portability in Cloud Platforms," in *Proceedings of the South-East European Doctoral Student Conference*, 2012, pp. 565–576. (Cited on pages 4, 28, 107, and 205.)

[125] N. Goonasekera, A. Lonie, J. Taylor, and E. Afgan, "CloudBridge: a Simple Cross-Cloud Python Library," in *Proceedings of the XSEDE on Diversity, Big Data, and Science at Scale*. ACM Press, 2016. (Cited on page 190.)

[126] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The Cost of a Cloud: Research Problems in Data Center Networks," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, p. 68, Dec. 2008. (Cited on page 19.)

[127] I. Grigorik, *High Performance Browser Networking*. O'Reilly UK Ltd., 2016, ISBN: 9781449344764. (Cited on pages 56 and 84.)

[128] W. Grohmann, *Von der Software zum Service: ASP, Software on Demand, Software-as-a-Service – Neue Formen der Software-Nutzung [From Software to Service: ASP, Software on Demand, Software as a Service – New Ways of Software Usage]*. H. K. P. Consulting, 2007, ISBN: 9783939968078. (Cited on pages 44 and 45.)

[129] F. Gropengießer and K.-U. Sattler, "Database Backend as a Service: Automatic Generation, Deployment, and Management of Database Backends for Mobile Applications," *Datenbank-Spektrum*, vol. 14, no. 2, pp. 85–95, Jun. 2014. (Cited on page 67.)

[130] T. R. Gruber, "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199–220, Jun. 1993. (Cited on page 87.)

[131] ——, "Toward Principles for the Design of Ontologies Used for Knowledge Sharing," *International Journal of Human-Computer Studies*, vol. 43, no. 5-6, pp. 907–928, Nov. 1995. (Cited on page 88.)

[132] S. Gudenkauf, M. Josefiok, A. Göring, and O. Norkus, "A Reference Architecture for Cloud Service Offers," in *Proceedings of the IEEE Enterprise Distributed Object Computing Conference (EDOC)*, Sep. 2013, Vancouver, Canada. (Cited on pages 6, 41, 69, 73, 83, 90, and 110.)

[133] S. Gupta, P. Matthews, V. Muntés-Mulero, and J. Dominiak, "Cloud Service Offer Selection," in *Model-Driven Development and Operation of Multi-Cloud Applications*. Springer International Publishing, Dec. 2016, pp. 13–22. (Cited on page 119.)

[134] J. D. Haan, "Categorizing the Cloud Landscape: A New Model," in *The 2014 Cloud Platform Research Report*. DZone, 2014. (Cited on pages 42, 69, and 205.)

[135] V. Haarslev, K. Hidde, R. Möller, and M. Wessel, "The RacerPro Knowledge Representation and Reasoning System," *Semantic Web*, vol. 3, no. 3, pp. 267–277, 2012. (Cited on page 91.)

[136] H. Hacigumus, B. Iyer, and S. Mehrotra, "Providing Database as a Service," in *Proceedings of the Conference on Data Engineering*. IEEE, 2002, pp. 29–38. (Cited on page 67.)

[137] M. Hadley, "Web Application Description Language," *W3C Member Submission*, Aug. 2009. (Cited on page 183.)

[138] 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," *Computer Communication Review*, vol. 40, no. 4, pp. 243–254, 2010. (Cited on pages 28, 73, 137, and 165.)

[139] T. Han and K. M. Sim, "An Ontology-enhanced Cloud Service Discovery System," in *Proceedings of the MultiConference of Engineers and Computer Scientists*, vol. 1, no. 2010, 2010, pp. 17–19. (Cited on pages 83, 90, and 110.)

[140] G. K. Hanssen, "A longitudinal case study of an emerging software ecosystem: Implications for practice and theory," *Journal of Systems and Software*, vol. 85, no. 7, pp. 1455–1466, Jul. 2012. (Cited on pages 79 and 80.)

[141] D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749, Oct. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6749.txt (Cited on page 184.)

[142] R. Harms and M. Yamartino, "The Economics of the Cloud," Microsoft Corporation, Tech. Rep., 2010. (Cited on pages 3 and 21.)

[143] A. Harris and K. Haase, *Sinatra: Up and Running*. O'Reilly Media, Inc, USA, 2011, ISBN: 9781449304232. (Cited on page 101.)

[144] R. Harris, "Introduction to Decision Making," https://www.virtualsalt.com/crebook5.htm, Jun. 2012. (Cited on page 32.)

[145] P. Harsh, F. Dudouet, R. G. Cascella, Y. Jegou, and C. Morin, "Using Open Standards for Interoperability: Issues, Solutions, and Challenges facing Cloud Computing," in *Proceedings of the Conference on Network and Service Management and Workshop on Systems Virtualiztion Management*, Oct. 2012, pp. 435–440. (Cited on pages 27 and 140.)

[146] J. Heflin, J. Hendler, and S. Luke, "SHOE: A Knowledge Representation Language for Internet Applications," University of Maryland at College Park, Dept. of Computer Science, Tech. Rep., 1999. (Cited on page 91.)

[147] D. Hilkert, C. M. Wolf, A. Benlian, and T. Hess, "The "As-a-Service"-Paradigm and Its Implications for the Software Industry – Insights from

a Comparative Case Study in CRM Software Ecosystems," in *Proceedings of the Conference of Software Business*. Springer, 2010, pp. 125–137. (Cited on pages 3 and 80.)

[148] D. Hilley, "Cloud Computing: A Taxonomy of Platform and Infrastructure-level Offerings," Georgia Institute of Technology, Tech. Rep., Apr. 2009. (Cited on pages 44, 45, 83, 90, and 110.)

[149] S. Hoare, "A study of the state-of-the-art of PaaS interoperability," in *Proceedings of the Conference on Evaluation and Assessment in Software Engineering*, 2016. (Cited on page 4.)

[150] F. Hoch, M. Kerr, A. Griffith *et al.*, "Software as a Service: Strategic Backgrounder," Software & Information Industry Association (SIIA), Tech. Rep., Feb. 2001. (Cited on pages 20, 24, and 25.)

[151] C. N. Höfer and G. Karagiannis, "Cloud computing services: taxonomy and comparison," *Journal of Internet Services and Applications*, vol. 2, no. 2, pp. 81–94, 2011. (Cited on pages 83, 90, and 110.)

[152] M. Hogan, F. Liu, A. Sokol, and J. Tong, "NIST Cloud Computing Standards Roadmap," NIST, Special Publication 500-291, 2011. (Cited on pages 9, 10, 13, 27, 28, 29, 76, 77, 83, 88, 131, 137, 138, 140, 164, 169, 170, 176, 188, 193, and 206.)

[153] E. Hossny, S. Khattab, F. Omara, and H. Hassan, "A Case Study for Deploying Applications on Heterogeneous PaaS Platforms," in *Proceedings of the Conference on Cloud Computing and Big Data*, 2013. (Cited on pages 191 and 204.)

[154] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010, ISBN: 9780321601919. (Cited on page 171.)

[155] J. Humble and J. Molesky, "Why Enterprises Must Adopt DevOps to Enable Continuous Delivery," *Cutter IT Journal*, vol. 24, no. 8, p. 6, 2011. (Cited on pages 4 and 170.)

[156] *Codes for the representation of names of countries and their subdivisions – Part 1: Country codes*, ISO Std. 3166-1, 2006. (Cited on page 94.)

[157] *Software engineering – Product quality – Part 1: Quality models*, ISO/IEC Std. 9126-2, 2003. (Cited on page 78.)

[158] ISO/IEC, "Information technology – Open Virtualization Format (OVF) specification," 2011. (Cited on pages 30, 76, and 78.)

[159] *Systems and software engineering – System and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*, ISO/IEC Std. 25 010, 2011. (Cited on pages 10, 13, 27, 81, 126, 149, and 206.)

[160] *Information technology – Cloud computing – Interoperability and portability*, ISO/IEC Std. 19 941, 2017. (Cited on pages 26, 27, and 29.)

[161] *Information technology – Cloud computing – Overview and vocabulary*, ISO/IEC Std. 17 788, 2014. (Cited on pages 27, 34, 44, 45, and 63.)

[162] *Systems and software engineering – Vocabulary*, ISO/IEC/IEEE Std. 24 765, 2017. (Cited on pages 26, 27, 29, 40, 75, and 171.)

[163] R. Jabbari, N. bin Ali, K. Petersen, and B. Tanveer, "What is DevOps? A Systematic Mapping Study on Definitions and Practices," in *Proceedings of the Scientific Workshop of XP*. ACM Press, 2016. (Cited on pages 4 and 170.)

[164] Y. Jadeja and K. Modi, "Cloud Computing-concepts, Architecture and Challenges," in *Proceedings of the Conference on Computing, Electronics and Electrical Technologies*. IEEE, 2012, pp. 877–880. (Cited on pages 44 and 45.)

[165] P. Jamshidi, A. Ahmad, and C. Pahl, "Cloud Migration Research: A Systematic Review," *IEEE Transactions on Cloud Computing*, vol. 1, no. 2, 2013. (Cited on pages 9, 30, 139, 164, and 206.)

[166] S. Jansen, A. Finkelstein, and S. Brinkkemper, "A Sense Of Community: A Research Agenda for Software Ecosystems," in *Proceedings of the Conference on Software Engineering*. IEEE, 2009, pp. 187–190. (Cited on page 79.)

[167] W. Jansen and T. Grance, "Guidelines on Security and Privacy in Public Cloud Computing," NIST, Special Publication 800-144, 2011. (Cited on pages 19 and 85.)

[168] B. Javed, P. Bloodsworth, R. U. Rasool, K. Munir, and O. Rana, "Cloud Market Maker: An automated dynamic pricing marketplace for cloud users," *Future Generation Computer Systems*, vol. 54, pp. 52–67, Jan. 2016. (Cited on page 133.)

[169] T. Joachims, L. Granka, B. Pan, H. Hembrooke, and G. Gay, "Accurately Interpreting Clickthrough Data As Implicit Feedback," in *Proceedings of the ACM SIGIR Forum*, vol. 51, no. 1. Acm, 2017, pp. 4–11. (Cited on page 98.)

[170] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," *Preprint*, vol. abs/1702.04024, 2017. (Cited on page 68.)

[171] A. Juan-Verdejo and B. Surajbali, "XaaS Multi-Cloud Marketplace Architecture Enacting the Industry 4.0 Concepts," in *Proceedings of the Technological Innovation for Cyber-Physical Systems*, L. M. Camarinha-Matos, A. J. Falcão, N. Vafaei, and S. Najdi, Eds. Cham: Springer International Publishing, 2016, pp. 11–23. (Cited on page 133.)

[172] G. Jung, T. Mukherjee, S. Kunde, H. Kim, N. Sharma, and F. Goetz, "CloudAdvisor: A Recommendation-as-a-Service Platform for Cloud Configuration and Pricing," in *Proceedings of the World Congress on Services*, Jun. 2013. (Cited on page 132.)

[173] S. Kachele, C. Spann, F. J. Hauck, and J. Domaschka, "Beyond IaaS and PaaS: An Extended Cloud Taxonomy for Computation, Storage

and Networking," in *Proceedings of the Conference on Utility and Cloud Computing*. IEEE, Dec. 2013. (Cited on pages 44 and 45.)

[174] K. Kalaboukas, G. Ledakis, P. Gouvas, N. Drosos, N. Bassiliades, and A. Papadopoulos, "Unified Cloud Platforms Interface Model and API – Deliverable 4.1," PaaSport Project, Tech. Rep., 2014. (Cited on pages 109, 174, 176, 191, 200, and 207.)

[175] E. Kamateri, N. Loutas, D. Zeginis, J. Ahtes, F. D'Andria, S. Bocconi, P. Gouvas, G. Ledakis, F. Ravagli, O. Lobunets, and others, "Cloud4SOA: A Semantic-Interoperability PaaS Solution for Multi-cloud Platform Management and Portability," in *Service-Oriented and Cloud Computing*. Springer, 2013. (Cited on pages 10, 13, 109, 169, 191, 197, 199, and 207.)

[176] A. Kanso and A. Youssef, "Serverless: Beyond the Cloud," in *Proceedings of the Workshop on Serverless Computing*. ACM Press, 2017. (Cited on page 67.)

[177] P. D. Karp, V. K. Chaudhri, and J. Thomere, "XOL: An XML-Based Ontology Exchange Language," 1999. (Cited on page 91.)

[178] L. M. Kaufman, "Data Security in the World of Cloud Computing," *IEEE Security & Privacy*, vol. 7, no. 4, pp. 61–64, Jul. 2009. (Cited on pages 55 and 82.)

[179] P. G. W. Keen, *Decision Support Systems: An Organizational Perspective*. Addison-Wesley, 1978, ISBN: 9780201036671. (Cited on page 30.)

[180] R. E. Kent, "Conceptual Knowledge Markup Language: An introduction," *Netnomics*, vol. 2, no. 2, pp. 139–169, 2000. (Cited on page 91.)

[181] T. Kern, M. C. Lacity, and L. Willcocks, *Netsourcing: Renting Business Applications and Services Over a Network*. FT Press, 2002, ISBN: 9780130923554. (Cited on page 24.)

[182] A. Khajeh-Hosseini, D. Greenwood, and I. Sommerville, "Cloud Migration: A Case Study of Migrating an Enterprise IT System to IaaS," in *Proceedings of the Conference on Cloud Computing*, 2010. (Cited on pages 28, 137, and 166.)

[183] A. Khajeh-Hosseini, I. Sommerville, J. Bogaerts, and P. Teregowda, "Decision Support Tools for Cloud Migration in the Enterprise," in *Proceedings of the Conference on Cloud Computing*, 2011. (Cited on page 166.)

[184] Y. Khalidi, "Building a Cloud Computing Platform for New Possibilities," *Computer*, vol. 44, no. 3, pp. 29–34, Mar. 2011. (Cited on pages 44 and 45.)

[185] A. Kidd, *Knowledge Acquisition for Expert Systems: A Practical Handbook*. Springer Science & Business Media, 1987, ISBN: 9781461318231. (Cited on page 32.)

[186] M. Kifer, G. Lausen, and J. Wu, "Logical Foundations of Object-oriented and Frame-Based Languages," *Journal of the ACM (JACM)*, vol. 42, no. 4, pp. 741–843, Jul. 1995. (Cited on page 91.)

[187] W. Kim, "Cloud Architecture: A Preliminary Look," in *Proceedings of the Conference on Information Integration and Web-Based Applications and Services*. ACM Press, 2011. (Cited on pages 44 and 45.)

[188] B. Kitchenham *et al.*, "Guidelines for performing Systematic Literature Reviews in Software Engineering," Keele University, Keele,UK and University of Durham, UK, Tech. Rep., 2007. (Cited on pages 42 and 43.)

[189] R. Kohavi, "The Power of Decision Tables," in *Proceedings of the European Conference on Machine Learning*. Springer, 1995, pp. 174–189. (Cited on page 32.)

[190] S. Kolb and C. Röck, "Unified Cloud Application Management," in *Proceedings of the World Congress on Services*, 2016. (Cited on pages 3, 14, and 169.)

[191] S. Kolb and G. Wirtz, "Towards Application Portability in Platform as a Service," in *Proceedings of the Symposium Service-Oriented System Engineering*, 2014. (Cited on pages 14, 39, 73, 113, 114, and 139.)

[192] ——, "Data Governance and Semantic Recommendation Algorithms for Cloud Platform Selection," in *Proceedings of the Conference on Cloud Computing*, 2017. (Cited on pages 14, 73, and 113.)

[193] S. Kolb, J. Lenhard, and G. Wirtz, "Application Migration Effort in the Cloud – The Case of Cloud Platforms," in *Proceedings of the Conference Cloud Computing*, 2015. (Cited on pages 10, 14, 137, and 170.)

[194] ——, "Application Migration Effort in the Cloud," *Services Transactions on Cloud Computing*, vol. 3, no. 4, pp. 1–15, 2015. (Cited on pages 3, 14, 73, and 137.)

[195] Y. Kondratenko, G. Kondratenko, and I. Sidenko, "Multi-Criteria Decision Making for Selecting a Rational IoT Platform," in *Proceedings of the Conference on Dependable Systems, Services and Technologies*. IEEE, May 2018. (Cited on page 33.)

[196] T. Koonen, "Fiber to the Home/Fiber to the Premises: What, Where, and When?" *Proceedings of the IEEE*, vol. 94, no. 5, pp. 911–934, May 2006. (Cited on page 19.)

[197] M. Kostoska, M. Gusev, and S. Ristov, "A New Cloud Services Portability Platform," *Procedia Engineering*, vol. 69, pp. 1268–1275, 2014. (Cited on page 107.)

[198] N. Kratzke and P.-C. Quint, "Understanding Cloud-native Applications after 10 Years of Cloud Computing – A Systematic Mapping Study," *Journal of Systems and Software*, vol. 126, pp. 1–16, Apr. 2017. (Cited on pages 137 and 138.)

[199] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015. (Cited on page 23.)

[200] C. Krintz, "Platform-as-a-Service (PaaS)," in *Encyclopedia of Database Systems*. Springer New York, 2017, pp. 1–2. (Cited on pages 44 and 45.)

[201] V. S. Lai, R. P. Trueblood, and B. K. Wong, "Software selection: a case study of the application of the analytical hierarchical process to the selection of a multimedia authoring system," *Information & Management*, vol. 36, no. 4, pp. 221–232, 1999. (Cited on pages 33 and 34.)

[202] V. S. Lai, B. K. Wong, and W. Cheung, "Group decision making in a multiple criteria environment: A case using the AHP in software selection," *European Journal of Operational Research*, vol. 137, no. 1, pp. 134–144, 2002. (Cited on page 34.)

[203] G. Lawton, "LAMP Lights Enterprise Development Efforts," *Computer*, vol. 38, no. 9, pp. 18–20, 2005. (Cited on page 64.)

[204] ——, "Developing Software Online with Platform-as-a-Service Technology," *Computer*, vol. 41, no. 6, 2008. (Cited on pages 44 and 45.)

[205] D. B. Lenat and R. V. Guha, "Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project," *Artificial Intelligence*, vol. 61, no. 1, pp. 81–94, May 1993. (Cited on page 91.)

[206] J. Lenhard and G. Wirtz, "Measuring the Portability of Executable Service-Oriented Processes," in *Proceedings of the Enterprise Distributed Object Computing Conference*, Vancouver, Canada, Sep. 2013. (Cited on page 27.)

[207] J. Lenhard, S. Harrer, and G. Wirtz, "Measuring the Installability of Service Orchestrations Using the SQuaRE Method," in *Proceedings of the Conference on Service-Oriented Computing and Applications*, 2013. (Cited on pages 10, 13, 149, 151, 152, 153, and 206.)

[208] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm, "What's Inside the Cloud? An Architectural Map of the Cloud Landscape," in *Proceedings of the Workshop on Software Engineering Challenges of Cloud Computing*, 2009. (Cited on pages 20 and 22.)

[209] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals," Ph.D. dissertation, Soviet physics doklady, 1966. (Cited on pages 127 and 129.)

[210] G. Lewis, "The Role of Standards in Cloud-Computing Interoperability," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, United States, Tech. Rep., Oct. 2012. (Cited on pages 26, 81, 107, and 140.)

[211] J. Lewis and M. Fowler, "Microservices," 2014, http://martinfowler.com/articles/microservices.html. (Cited on pages 68 and 172.)

[212] F. Leymann, C. Fehling, S. Wagner, and J. Wettinger, "Native Cloud Applications: Why Virtual Machines, Images and Containers Miss the Point!" in *Proceedings of the Conference on Cloud Computing and Service Science*, 2016, pp. 7–15. (Cited on page 138.)

[213] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: Comparing Public Cloud Providers," in *Proceedings of the Conference on Internet Measurement*. ACM Press, 2010. (Cited on page 175.)

[214] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, and D. Leaf, "NIST Cloud Computing Reference Architecture," NIST, Special Publication 500-292, 2011. (Cited on pages 5, 34, and 35.)

[215] P. Lorenc and M. Woda, "IaaS vs. Traditional Hosting for Web Applications – Cost Effectiveness Analysis for a Local Market," in *Advances in Dependability Engineering of Complex Systems*. Springer International Publishing, May 2017, pp. 233–243. (Cited on page 64.)

[216] N. Loutas, V. Peristeras, T. Bouras, E. Kamateri, D. Zeginis, and K. Tarabanis, "Towards a Reference Architecture for Semantically Interoperable Clouds," in *Proceedings of the Conference on Cloud Computing Technology and Science*, 2010. (Cited on pages 172 and 197.)

[217] N. Loutas, E. Kamateri, F. Bosi, and K. Tarabanis, "Cloud computing interoperability: the state of play," in *Proceedings of the Conference on Cloud Computing Technology and Science*. IEEE, 2011, pp. 752–757, Athens, Greece. (Cited on pages 4, 6, 10, 28, 34, 41, 69, 83, 109, 170, 193, and 205.)

[218] N. Loutas, E. Kamateri, and K. Tarabanis, "A Semantic Interoperability Framework for Cloud Platform as a Service," in *Proceedings of the Conference on Cloud Computing Technology and Science*. IEEE, 2011, pp. 280–287, Athens, Greece. (Cited on pages 83, 90, 110, 134, and 197.)

[219] J. Lu and D. Ruan, *Multi-Objective Group Decision Making: Methods, Software and Applications with Fuzzy Set Techniques*. Imperial College Press, 2007, vol. 6, ISBN: 9781860947933. (Cited on page 114.)

[220] J. L. Lucas-Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Cost optimization of virtual infrastructures in dynamic multi-cloud scenarios," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 9, pp. 2260–2277, Dec. 2012. (Cited on page 35.)

[221] S. Luke, L. Spector, and D. Rager, "Ontology-Based Knowledge Discovery on the World-Wide Web," in *Working Notes of the Workshop on Internet-Based Information Systems at the National Conference on Artificial Intelligence*. AAAI Press, 1996, pp. 96–102. (Cited on page 91.)

[222] C. Lv, Q. Li, Z. Lei, J. Peng, W. Zhang, and T. Wang, "PaaS: A Revolution for Information Technology Platforms," in *Proceedings of the 2010 Conference on Educational and Network Technology*, Jun. 2010. (Cited on page 171.)

[223] Y. B. Ma, S. H. Jang, and J. S. Lee, "Ontology-Based Resource Management for Cloud Computing," in *Intelligent Information and Database Systems*. Springer Berlin Heidelberg, 2011, pp. 343–352. (Cited on pages 83 and 90.)

[224] G. S. Machado, D. Hausheer, and B. Stiller, "Considerations on the Interoperability of and between Cloud Computing Standards," in *Proceedings of the Workshop From Grid to Cloud Networks*, 2009, Banff, Canada. (Cited on page 7.)

[225] P.-J. Maenhaut, H. Moens, V. Ongenae, and F. De Turck, "Migrating legacy software to the cloud: approach and verification by means of two medical software use cases," *Software: Practice and Experience*, 2015. (Cited on page 166.)

[226] K. Manikas and K. M. Hansen, "Software ecosystems – A systematic literature review," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1294–1306, May 2013. (Cited on pages 79 and 80.)

[227] H. B. Mann and D. R. Whitney, "On a Test of Whether One of Two Random Variables Is Stochastically Larger Than the Other," *Annals of Mathematical Statistics*, vol. 18, no. 1, 1947. (Cited on page 161.)

[228] J. Manner, S. Kolb, and G. Wirtz, "Troubleshooting Serverless Functions: A Combined Monitoring and Debugging Approach," in *Proceedings of the Advanced Summer School on Service Oriented Computing*, 2018. (Cited on pages 67 and 207.)

[229] M. Mao and M. Humphrey, "A Performance Study on the VM Startup Time in the Cloud," in *Proceedings of the Conference on Cloud Computing*, 2012. (Cited on page 160.)

[230] E. Markoska, I. Chorbev, S. Ristov, and M. Gusev, "Cloud Portability Standardization Overview," in *Proceedings of the Convention on Information and Communication Technology, Electronics and Microelectronics*, May 2015. (Cited on pages 5, 28, 53, 170, and 205.)

[231] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud computing – The business perspective," *Decision Support Systems*, vol. 51, no. 1, 2011. (Cited on pages 4, 10, 13, 26, 44, 45, 169, and 207.)

[232] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 2002, ISBN: 9780135974445. (Cited on pages 140 and 171.)

[233] B. D. Martino, G. Cretella, and A. Esposito, "Towards a Unified OWL Ontology of Cloud Vendors' Appliances and Services at PaaS and SaaS level," in *Proceedings of the Conference on Complex, Intelligent and Software Intensive Systems*. IEEE, Jul. 2014. (Cited on pages 83, 90, and 110.)

[234] B. D. Martino, G. Cretella, A. Esposito, and G. Carta, "An OWL ontology to support cloud portability and interoperability," *International Journal of Web and Grid Services*, vol. 11, no. 3, p. 303, 2015. (Cited on pages 83, 90, and 110.)

[235] M. Masse, *REST API Design Rulebook*. O'Reilly Media, 2011, ISBN: 9781449310509. (Cited on page 178.)

[236] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008. (Cited on page 23.)

[237] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," NIST, Special Publication 800-145, Sep. 2011. (Cited on pages 3, 18, 19, 20, 21, 22, 24, 25, 41, 44, 45, 53, 65, 84, 94, and 175.)

[238] S. Meng and L. Liu, "Enhanced Monitoring-as-a-Service for Effective Cloud Management," *IEEE Transactions on Computers*, vol. 62, no. 9, pp. 1705–1720, Sep. 2013. (Cited on page 67.)

[239] M. Menzel and R. Ranjan, "CloudGenius: Decision Support for Web Server Cloud Migration," in *Proceedings of the Conference on World Wide Web*, 2012. (Cited on page 165.)

[240] M. Menzel, M. Schönherr, and S. Tai, "$(MC^2)^2$: criteria, requirements and a software prototype for Cloud infrastructure decisions," *Software: Practice and Experience*, vol. 43, no. 11, pp. 1283–1297, Aug. 2013. (Cited on page 133.)

[241] D. G. Messerschmitt and C. Szyperski, "Software Ecosystem: Understanding an Indispensable Technology and Industry," *MIT Press Books*, 2005. (Cited on page 79.)

[242] H. Mezni and T. Abdeljaoued, "A Cloud Services Recommendation System Based on Fuzzy Formal Concept Analysis," *Data & Knowledge Engineering*, Jun. 2018. (Cited on page 132.)

[243] É. Michon, J. Gossa, S. Genaud, L. Unbekandt, and V. Kherbache, "Schlouder: a broker for IaaS clouds," *Future Generation Computer Systems*, vol. 69, pp. 11–23, Apr. 2017. (Cited on page 35.)

[244] D. W. Miller and M. K. Starr, "Executive Decisions and Operations Research," *Journal of Marketing*, vol. 34, no. 3, p. 100, Jul. 1963. (Cited on page 33.)

[245] J. Miranda, J. Guillén, J. M. Murillo, and C. Canal, "Assisting Cloud Service Migration Using Software Adaptation Techniques," in *Proceedings of the Conference on Cloud Computing*, 2013. (Cited on page 166.)

[246] D. Mitchell, "Defining Platform-As-A-Service, or PaaS," https://bungeeconnect.wordpress.com/2008/02/18/defining-platform-as-a-service-or-paas, Feb. 2008. (Cited on pages 43, 44, and 45.)

[247] F. Moscato, R. Aversa, B. Di Martino, T.-F. Fortiş, and V. Munteanu, "An Analysis of mOSAIC ontology for Cloud Resources annotation," in *Proceedings of the Federated Conference on Computer Science and Information Systems*. IEEE, 2011, pp. 973–980. (Cited on pages 83, 90, 97, 108, and 110.)

[248] H. R. Motahari-Nezhad, B. Stephenson, and S. Singhal, "Outsourcing Business to Cloud Computing Services: Opportunities and Challenges,"

IEEE Internet Computing, vol. 10, no. 4, pp. 1–17, 2009. (Cited on pages 44 and 45.)

[249] V. Munteanu, C. Şandru, and D. Petcu, "Multi-cloud resource management: cloud service interfacing," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 3, no. 1, p. 3, 2014. (Cited on page 190.)

[250] A. Muschalle, F. Stahl, A. Löser, and G. Vossen, "Pricing Approaches for Data Markets," in *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg, 2013, pp. 129–144. (Cited on page 93.)

[251] S. Nag, C. Eschinger, F. Ng, D. E. Ackerman, C. Graham, F. Biscotti, E. Anderson, M. Dorosh, and C. Roth, "Market Share Analysis: Public Cloud Services, Worldwide, 2016," Gartner Inc., Tech. Rep., Sep. 2017. (Cited on page 23.)

[252] 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., Jun. 2013, http://www.gartner.com/id=2515316. (Cited on pages 47, 49, and 96.)

[253] G. Navarro, "A Guided Tour to Approximate String Matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, Mar. 2001. (Cited on page 127.)

[254] S. B. Needleman and C. D. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970. (Cited on page 127.)

[255] J. Nielsen, "Usability Inspection Methods," in *Proceedings of the Conference on Human Factors in Computing Systems*. ACM Press, 1994. (Cited on page 151.)

[256] N. Nikiforakis, W. Joosen, and M. Johns, "Abusing Locality in Shared Web Hosting," in *Proceedings of the European Workshop on System Security*. ACM Press, 2011. (Cited on page 64.)

[257] E. D. Nitto, R. Sosa, M. Oriol, S. Zenzaro, J. Cubo, A. Turli, D. Pérez, D. Athanasopoulos, and J. Carrasco, "SeaClouds Project D5.1.2 – Integrated Platform," SeaClouds, Tech. Rep., 2015. (Cited on page 202.)

[258] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus Open-Source Cloud-Computing System," in *Proceedings of the Symposium on Cluster Computing and the Grid*, 2009. (Cited on page 190.)

[259] *Topology and Orchestration Specification for Cloud Applications Version 1.0*, OASIS Std., Nov. 2013. (Cited on pages 107 and 190.)

[260] *TOSCA Simple Profile in YAML Version 1.0*, OASIS Std., Dec. 2016. (Cited on page 202.)

[261] *Cloud Application Management for Platforms Version 1.2*, OASIS Committee Specification, May 2018. (Cited on pages 109 and 189.)

[262] 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. (Cited on pages 5, 10, 13, 76, 169, 205, 206, and 207.)

[263] A. Ojala and N. Helander, "Value creation and evolution of a value network: A longitudinal case study on a Platform-as-a-Service provider," in *Proceedings of the Hawaii International Conference on System Sciences*. IEEE, Jan. 2014. (Cited on pages 55 and 61.)

[264] J. Opara-Martins, "Taxonomy of Cloud Lock-in Challenges," in *Mobile Computing - Technology and Applications*. InTech, May 2018. (Cited on pages 19, 26, and 76.)

[265] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical Review of Vendor Lock-in and Its Impact on Adoption of Cloud Computing," in *Proceedings of the Conference on Information Society*. IEEE, 2014, pp. 92–97. (Cited on page 26.)

[266] *Open Container Initiative – Runtime Specification*, Open Container Initiative Std., Rev. 1.0.1, Nov. 2017. (Cited on pages 78 and 107.)

[267] *Open Container Initiative – Image Format Specification*, Open Container Initiative Std., Rev. 1.0.1, Nov. 2017. (Cited on pages 78 and 107.)

[268] Open Grid Forum, "Open Cloud Computing Interface – Core," 2016, gFD-R-P.221. (Cited on pages 109, 188, and 203.)

[269] ——, "Open Cloud Computing Interface – Infrastructure," 2016, gFD-R-P.224. (Cited on pages 83, 109, 188, and 203.)

[270] ——, "Open Cloud Computing Interface – Platform," 2016, gFD-R-P.227. (Cited on pages 109, 180, 188, and 204.)

[271] S. Ortiz, "The Problem with Cloud-Computing Standardization," *Computer*, vol. 44, no. 7, pp. 13–16, 2011. (Cited on pages 5, 28, 53, 205, and 206.)

[272] A. Osterwalder, "The Business Model Ontology: A Proposition in a Design Science Approach," Ph.D. dissertation, University of Lausanne, 2004. (Cited on page 93.)

[273] C. Pahl and H. Xiong, "Migration to PaaS Clouds – Migration Process and Architectural Concerns," in *Proceedings of the Symposium on Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, 2013. (Cited on page 165.)

[274] C. Pahl, H. Xiong, and R. Walshe, "A Comparison of On-Premise to Cloud Migration Approaches," in *Service-Oriented and Cloud Computing*. Springer Berlin Heidelberg, 2013, pp. 212–226. (Cited on pages 28 and 137.)

[275] G. Pallis, "Cloud Computing: The New Frontier of Internet Computing," *IEEE Internet Computing*, vol. 14, no. 5, pp. 70–73, 2010. (Cited on pages 17, 19, and 20.)

[276] F. Pardee, T. Kirkwood, K. MacCrimmon, J. Miller, C. Phillips, J. Ranftl, K. Smith, and D. Whitcomb, "Measurement and evaluation of transportation system effectiveness," RAND Corporation, Tech. Rep., 1969. (Cited on page 120.)

[277] D. F. Parkhill, "The Challenge of the Computer Utility," *Operational Research Quarterly*, vol. 18, no. 3, p. 324, Sep. 1967. (Cited on page 17.)

[278] I. Patiniotakis, Y. Verginadis, and G. Mentzas, "Preference-based Cloud Service Recommendation as a Brokerage Service," in *Proceedings of the Workshop on CrossCloud Systems*. ACM Press, 2014. (Cited on page 132.)

[279] C. Pautasso, "RESTful Web Services: Principles, Patterns, Emerging Technologies," in *Web Services Foundations*. Springer, Sep. 2013, pp. 31–51. (Cited on page 178.)

[280] C. Pautasso, O. Zimmermann, and F. Leymann, "RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision," in *Proceedings of the Conference on World Wide Web*. ACM Press, 2008. (Cited on page 183.)

[281] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski, "Introducing STRATOS: A Cloud Broker Service," in *Proceedings of the Conference on Cloud Computing*, Jun. 2012. (Cited on page 133.)

[282] S. Pearson and A. Benameur, "Privacy, Security and Trust Issues Arising from Cloud Computing," in *Proceedings of the Conference on Cloud Computing Technology and Science*. IEEE, Nov. 2010. (Cited on pages 19, 20, 55, and 175.)

[283] D. Petcu, "Portability and Interoperability between Clouds: Challenges and Case Study," in *Towards a Service-Based Internet*. Springer, 2011, pp. 62–74. (Cited on pages 4, 10, 13, 27, 28, 29, 138, 140, 169, 170, 193, 205, and 206.)

[284] D. Petcu and A. V. Vasilakos, "Portability in Clouds: Approaches and Research Opportunities," *Scalable Computing: Practice and Experience*, vol. 15, no. 3, 2014. (Cited on pages 7, 137, and 139.)

[285] D. Petcu, C. Craciun, and M. Rak, "Towards a Cross Platform Cloud API," in *Proceedings of the Conference on Cloud Computing and Services Science*, 2011. (Cited on pages 10, 35, 170, 176, 188, and 193.)

[286] D. Petcu, G. Macariu, S. Panica, and C. Crăcium, "Portable Cloud applications – From theory to practice," *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1417–1430, 2013. (Cited on pages 8, 9, 76, 108, 113, 137, and 164.)

[287] R. D. Pietro and F. Lombardi, "Virtualization Technologies and Cloud Security: advantages, issues, and perspectives," *Preprint*, 2018. (Cited on page 61.)

[288] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "A Framework and Algorithm for Energy Efficient Container Consolidation in Cloud Data Centers," in *Proceedings of the Conference on Data Science and Data Intensive Systems*. IEEE, 2015, pp. 368–375. (Cited on page 68.)

[289] D. J. Power, "Web-Based and Model-Driven Decision Support Systems: Concepts and Issues," *Proceedings of Americas Conference on Information Systems*, p. 387, 2000. (Cited on page 31.)

[290] D. J. Power and S. Kaparthi, "The Changing Technological Context of Decision Support Systems," in *Context-Sensitive Decision Support Systems*. Springer, 1998, pp. 41–54. (Cited on page 31.)

[291] R. Prodan and S. Ostermann, "A Survey and Taxonomy of Infrastructure as a Service and Web Hosting Cloud Providers," in *Proceedings of the Conference on Grid Computing*. IEEE, 2009, pp. 17–25, Banff, Alberta. (Cited on pages 44, 45, 83, 90, 96, and 110.)

[292] L. H. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 4, 1978. (Cited on page 165.)

[293] L. Qian, Z. Luo, Y. Du, and L. Guo, "Cloud Computing: An Overview," in *Proceedings of the Conference on Cloud Computing*. Springer, 2009, pp. 626–631. (Cited on pages 17, 18, 19, 20, and 22.)

[294] A. Quarati, A. Clematis, and D. D'Agostino, "Delivering Cloud Services with QoS Requirements: Business Opportunities, Architectural Solutions and Energy-saving Aspects," *Future Generation Computer Systems*, vol. 55, pp. 403–427, Feb. 2016. (Cited on page 132.)

[295] J. R. Quinlan, "Induction of Decision Trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986. (Cited on page 32.)

[296] C. Quinton, D. Romero, and L. Duchien, "Automated Selection and Configuration of Cloud Environments Using Software Product Lines Principles," in *Proceedings of the Conference on Cloud Computing*, Jun. 2014. (Cited on pages 83, 90, 109, 110, and 134.)

[297] ——, "SALOON: A Platform for Selecting and Configuring Cloud Environments," *Software: Practice and Experience*, vol. 46, no. 1, pp. 55–78, 2016. (Cited on page 134.)

[298] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2015. [Online]. Available: https://www.R-project.org (Cited on page 159.)

[299] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource Management for Isolation Enhanced Cloud Services," in *Proceedings of the Workshop on Cloud Computing Security*. ACM, 2009, pp. 77–84. (Cited on page 64.)

[300] Z. Rehman, F. K. Hussain, and O. K. Hussain, "Towards Multi-Criteria Cloud Service Selection," in *Proceedings of the Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, Jun. 2011. (Cited on page 131.)

[301] Z. Rehman, O. K. Hussain, and F. K. Hussain, "IaaS Cloud Selection using MCDM Methods," in *Proceedings of the Conference on E-Business Engineering*, Sep. 2012. (Cited on pages 33 and 133.)

[302] M. Rekik, K. Boukadi, W. Gaaloul, and H. Ben-Abdallah, "Anti-Pattern Specification and Correction Recommendations for Semantic Cloud Services," in *Proceedings of the Hawaii International Conference on System Sciences*, 2017. (Cited on pages 6, 41, 69, 83, 109, 119, 120, and 124.)

[303] J. Repschlaeger, S. Wind, R. Zarnekow, and K. Turowski, "A Reference Guide to Cloud Computing Dimensions: Infrastructure as a Service Classification Framework," in *Proceedings of the Hawaii International Conference on System Sciences*. IEEE, Jan. 2012. (Cited on pages 41, 73, and 110.)

[304] J. Reschke, "The 'Basic' HTTP Authentication Scheme," RFC 7617, Sep. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7617.txt (Cited on page 184.)

[305] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, 2008, ISBN: 9780596529260. (Cited on page 183.)

[306] M. Richardson, E. Dominowska, and R. Ragno, "Predicting Clicks: Estimating the Click-Through Rate for New Ads," in *Proceedings of the Conference on World Wide Web*. ACM Press, 2007. (Cited on page 98.)

[307] S. Ried, "Multiple PaaS Flavors Hit The Enterprise," Forrester, Tech. Rep., Aug. 2012, http://www.forrester.com/Multiple+PaaS+Flavors+Hit+The+Enterprise/fulltext/-/E-RES78101. (Cited on page 41.)

[308] 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. (Cited on pages 49 and 96.)

[309] S. Y. Rieh and H. I. Xie, "Analysis of Multiple Query Reformulations on the Web: The Interactive Information Retrieval Context," *Information Processing & Management*, vol. 42, no. 3, pp. 751–768, May 2006. (Cited on page 121.)

[310] B. P. Rimal, E. Choi, and I. Lumb, "A Taxonomy and Survey of Cloud Computing Systems," in *Proceedings of the Joint Conference on INC, IMS and IDC*. IEEE, 2009. (Cited on pages 44, 45, and 96.)

[311] L. Rising and N. S. Janoff, "The Scrum Software Development Process for Small Teams," *IEEE Software*, vol. 17, no. 4, pp. 26–32, 2000. (Cited on page 99.)

[312] C. Röck and S. Kolb, "Nucleus – Unified Deployment and Management for Platform as a Service," University of Bamberg, Tech. Rep., 2016. (Cited on pages 14, 169, 182, and 187.)

[313] L. Rodero-Merino, L. M. Vaquero, E. Caron, A. Muresan, and F. Desprez, "Building safe PaaS clouds: A survey on security in multitenant software platforms," *Computers & Security*, vol. 31, no. 1, pp. 96–108, 2012. (Cited on pages 44 and 45.)

[314] D. Rowley, "The Business of Application Portability," *StandardView*, vol. 4, no. 2, pp. 80–87, Jun. 1996. (Cited on pages 4, 5, 26, 27, 28, 53, 76, and 193.)

[315] B. Roy, "Classement et choix en présence de points de vue multiples [Ranking and choice in the presence of multiple points of view]," *Revue Française d'Automatique, d'Informatique et de Recherche Opérationnelle*, vol. 2, no. 1, pp. 57–75, 1968. (Cited on pages 33 and 122.)

[316] ——, "The Outranking Approach and the Foundations of ELECTRE Methods," in *Readings in Multiple Criteria Decision Aid*. Springer, 1990, pp. 155–183. (Cited on page 33.)

[317] J. R. Rymer and J. Staten, "The Forrester Wave: Enterprise Public Cloud Platforms, Q2 2013," Forrester, Tech. Rep., Jun. 2013, http://www.forrester.com/The+Forrester+Wave+Enterprise+Public+Cloud+Platforms+Q2+2013/fulltext/-/E-RES86441. (Cited on pages 47, 49, and 96.)

[318] T. L. Saaty, "How to make a decision: The Analytic Hierarchy Process," *European Journal of Operational Research*, vol. 48, no. 1, pp. 9–26, Sep. 1990. (Cited on pages 33, 34, and 122.)

[319] ——, *Theory and Applications of the Analytic Network Process: Decision Making with Benefits, Opportunities, Costs, and Risks*. RWS publications, 2005, ISBN: 9781888603064. (Cited on page 33.)

[320] D. Sankoff, "Time Warps, String Edits, and Macromolecules," *The Theory and Practice of Sequence Comparison, Reading*, 1983. (Cited on page 127.)

[321] J. Savolainen, J. Kuusela, and A. Vilavaara, "Transition to Agile Development: Rediscovery of Important Requirements Engineering Practices," in *Proceedings of the Requirements Engineering Conference*, IEEE. IEEE, Sep. 2010, pp. 289–294. (Cited on page 99.)

[322] S. Schlauderer and S. Overhage, "Selecting Cloud Service Providers – Towards a Framework of Assessment Criteria and Requirements," in *Proceedings of the Wirtschaftsinformatik*, 2015. (Cited on pages 82 and 135.)

[323] A. Schönberger, J. Schwalb, and G. Wirtz, "Has WS-I's Work Resulted in WS-* Interoperability?" in *Proceedings of the Conference on Web Services*. IEEE, Jul. 2011. (Cited on pages 4, 28, and 76.)

[324] M. Sellami, S. Yangui, M. Mohamed, and S. Tata, "PaaS-independent Provisioning and Management of Applications in the Cloud," in *Proceed-*

ings of the Conference on Cloud Computing, 2013. (Cited on pages 10, 35, 109, 170, 174, 176, 180, 191, 193, 203, 204, and 207.)

[325] J. Shao and Q. Wang, "A Model-Driven Monitoring Approach for Internetware on Platform-as-a-Service (PaaS)," in *Proceedings of the Asia-Pacific Symposium on Internetware*. ACM Press, 2012. (Cited on pages 44 and 45.)

[326] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3–4, 1965. (Cited on page 161.)

[327] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and Virtual Machines at Scale: A Comparative Study," in *Proceedings of the Conference on Middleware*. ACM Press, 2016. (Cited on page 208.)

[328] V. S. Sharma, S. Sengupta, and S. Nagasamudram, "MAT: A Migration Assessment Toolkit for PaaS Clouds," in *Proceedings of the Conference on Cloud Computing*, 2013. (Cited on page 165.)

[329] A. Sheth and A. Ranabahu, "Semantic Modeling for Cloud Computing, Part 2," *IEEE Internet Computing*, vol. 14, no. 4, pp. 81–84, 2010. (Cited on pages 76 and 172.)

[330] J. P. Shim, M. Warkentin, J. F. Courtney, D. J. Power, R. Sharda, and C. Carlsson, "Past, present, and future of decision support technology," *Decision Support Systems*, vol. 33, no. 2, pp. 111–126, 2002. (Cited on pages 30 and 32.)

[331] J. Siegel and J. Perdue, "Cloud Services Measures for Global Use: The Service Measurement Index (SMI)," in *Proceedings of the Annual SRII Global Conference*. San Jose, CA, USA: IEEE, 2012, pp. 411–415. (Cited on pages 56, 84, 93, and 132.)

[332] G. C. Silva, L. M. Rose, and R. Calinescu, "A Systematic Review of Cloud Lock-In Solutions," in *Proceedings of the Conference on Cloud Computing Technology and Science*, 2013. (Cited on pages 4, 8, 9, 28, 34, 35, 113, 137, 164, and 205.)

[333] ——, "Cloud DSL: A Language for Supporting Cloud Portability by Describing Cloud Entities," in *Proceedings of the Conference on Model Driven Engineering Languages and Systems*, 2014. (Cited on pages 83, 90, and 108.)

[334] J. L. L. Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Dynamic Placement of Virtual Machines for Cost Optimization in Multi-cloud Environments," in *Proceedings of the Conference on High Performance Computing & Simulation*. IEEE, Jul. 2011. (Cited on page 35.)

[335] M. Slawik, B. İ. Zilci, F. Knaack, and A. Küpper, "The Open Service Compendium. Business-pertinent Cloud Service Discovery, Assessment, and Selection," in *Economics of Grids, Clouds, Systems, and Services*. Springer

International Publishing, 2016, pp. 115–129. (Cited on pages 75, 91, and 132.)

[336] M. Slawik, B. İ. Zilci, and A. Küpper, "Establishing User-centric Cloud Service Registries," *Future Generation Computer Systems*, Mar. 2018. (Cited on pages 91, 99, 114, 115, 120, and 132.)

[337] J. Smeds, K. Nybom, and I. Porres, "DevOps: A Definition and Perceived Adoption Impediments," in *Lecture Notes in Business Information Processing*. Springer International Publishing, 2015, pp. 166–177. (Cited on pages 4 and 170.)

[338] D. M. Smith, "Hype Cycle for Cloud Computing," Gartner Inc., Tech. Rep., 2011. (Cited on page 52.)

[339] 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. (Cited on pages 19, 78, and 107.)

[340] J. Spillner and A. Schill, "A Versatile and Scalable Everything-as-a-Service Registry and Discovery," in *Proceedings of the Conference on Cloud Computing and Services Science*, 2013. (Cited on page 131.)

[341] R. H. Sprague Jr, "A Framework for the Development of Decision Support Systems," *MIS Quarterly*, pp. 1–26, 1980. (Cited on pages 30 and 32.)

[342] I. Sriram and A. Khajeh-Hosseini, "Research Agenda in Cloud Technologies," 2010. (Cited on pages 44 and 45.)

[343] S. Staab and R. Studer, *Handbook on Ontologies*. Springer, 2009, ISBN: 9783540709992. (Cited on page 87.)

[344] H. Stachowiak, *Allgemeine Modelltheorie [General Model Theory]*. Springer, 1973, ISBN: 9783211811061. (Cited on pages 7, 74, 75, 88, 119, and 127.)

[345] R. Studer, V. R. Benjamins, and D. Fensel, "Knowledge Engineering: Principles and Methods," *Data & Knowledge Engineering*, vol. 25, no. 1-2, pp. 161–197, Mar. 1998. (Cited on page 87.)

[346] X. Su and L. Ilebrekke, "A Comparative Study of Ontology Languages and Tools," in *Advanced Information Systems Engineering*. Springer Berlin Heidelberg, 2002, pp. 761–765. (Cited on page 91.)

[347] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of Network and Computer Applications*, vol. 34, no. 1, pp. 1–11, Jan. 2011. (Cited on pages 19, 20, 21, 44, 45, and 55.)

[348] K. Sun and Y. Li, "Effort Estimation in Cloud Migration Process," in *Proceedings of the Symposium on Service Oriented System Engineering*, 2013, pp. 84–91, San Francisco, United States. (Cited on pages 73, 149, and 166.)

[349] L. Sun, H. Dong, F. K. Hussain, O. K. Hussain, and E. Chang, "Cloud service selection: State-of-the-art and future research directions," *Journal of Network and Computer Applications*, vol. 45, pp. 134–150, Oct. 2014. (Cited on pages 6, 7, 34, 35, 36, 41, 69, 74, 83, 87, 109, 113, 115, 131, and 206.)

[350] S. Sundareswaran, A. Squicciarini, and D. Lin, "A Brokerage-Based Approach for Cloud Service Selection," in *Proceedings of the Conference on Cloud Computing*, Jun. 2012. (Cited on page 132.)

[351] B. Surajbali and A. Juan-Verdejo, "A Marketplace Broker for Platform-as-a-Service Portability," in *Advances in Service-Oriented and Cloud Computing*, ser. Communications in Computer and Information Science. Springer, 2015, vol. 508. (Cited on pages 97, 109, 135, and 170.)

[352] A. Tahamtan, S. A. Beheshti, A. Anjomshoaa, and A. M. Tjoa, "A Cloud Repository and Discovery Framework Based on a Unified Business and Cloud Service Ontology," in *Proceedings of the World Congress on Services*. IEEE, Jun. 2012. (Cited on pages 83, 90, and 110.)

[353] B. C. Tak, B. Urgaonkar, and A. Sivasubramaniam, "To Move or Not to Move: The Economics of Cloud Computing," in *Proceedings of the Conference on Hot Topics in Cloud Computing*, 2011. (Cited on page 166.)

[354] A. S. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 2007, ISBN: 9781530281756. (Cited on page 3.)

[355] L. Tao, "Shifting Paradigms with the Application Service Provider Model," *Computer*, vol. 34, no. 10, pp. 32–39, 2001. (Cited on page 25.)

[356] G. D. Tecuci, "Automating Knowledge Acquisition As Extending, Updating, and Improving a Knowledge Base," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no. 6, pp. 1444–1460, 1992. (Cited on page 31.)

[357] Telecom SudParis, Computer Science Department, "The Compatible One Application and Platform Service (COAPS) API specification – Version 1.5.3," 2013. (Cited on pages 191, 203, and 204.)

[358] V. T. K. Tran, J. Keung, A. Liu, and A. Fekete, "Application Migration to Cloud: A Taxonomy of Critical Factors," in *Proceedings of the Workshop Software Engineering for Cloud Computing*, 2011. (Cited on page 166.)

[359] V. T. K. Tran, K. Lee, A. Fekete, A. Liu, and J. Keung, "Size Estimation of Cloud Migration Projects with Cloud Migration Point (CMP)," in *Proceedings of the Symposium on Empirical Software Engineering and Measurement*, 2011. (Cited on page 166.)

[360] E. Triantaphyllou, *Multi-Criteria Decision Making Methods*. Boston, MA: Springer, 2000, pp. 5–21, ISBN: 9781475731576. (Cited on pages 33, 114, 119, 121, and 206.)

[361] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource Overbooking and Application Profiling in Shared Hosting Platforms," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, p. 239, Dec. 2002. (Cited on page 64.)

[362] M. Uschold and M. Gruninger, "Ontologies and Semantics for Seamless Connectivity," *ACM SIGMOD Record*, vol. 33, no. 4, p. 58, Dec. 2004. (Cited on page 87.)

[363] O. S. Vaidya and S. Kumar, "Analytic hierarchy process: An overview of applications," *European Journal of Operational Research*, vol. 169, no. 1, pp. 1–29, 2006. (Cited on pages 33 and 34.)

[364] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures," in *Proceedings of the of the World Organisation of Systems and Cybernetics*, 2017. (Cited on pages 68 and 207.)

[365] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A Break in the Clouds: Towards a Cloud Definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2009. (Cited on pages 44 and 45.)

[366] R. Verborgh, A. Harth, M. Maleshkova, S. Stadtmüller, T. Steiner, M. Taheriyan, and R. Van de Walle, *Survey of Semantic Description of REST APIs*. New York, NY: Springer New York, 2014, pp. 69–89, ISBN: 9781461492986. (Cited on page 183.)

[367] W. Voorsluys, J. Broberg, and R. Buyya, "Introduction to Cloud Computing," *Cloud Computing: Principles and Paradigms*, pp. 1–41, 2011. (Cited on page 18.)

[368] Q. H. Vu and R. Asal, "Legacy Application Migration to the Cloud: Practicability and Methodology," in *Proceedings of the World Congress Services*, 2012. (Cited on pages 28, 137, and 165.)

[369] W3C, "OWL Web Ontology Language," *W3C Recommendation*, 2004. (Cited on page 91.)

[370] ——, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," *W3C Recommendation*, 2007. (Cited on page 183.)

[371] ——, "OWL 2 Web Ontology Language," *W3C Recommendation*, 2012. (Cited on page 91.)

[372] ——, "SPARQL 1.1 Query Language," *W3C Recommendation*, 2013. (Cited on page 91.)

[373] ——, "RDF 1.1 Primer," *W3C Working Group Note*, 2014. (Cited on page 91.)

[374] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, p. 181, Dec. 2002. (Cited on pages 22 and 65.)

[375] L. Wang, J. Tao, M. Kunze, A. C. Castellanos, D. Kramer, and W. Karl, "Scientific Cloud Computing: Early Definition and Experience," in *Proceedings of the Conference on High Performance Computing and Commu-*

nications. Institute of Electrical and Electronics Engineers (IEEE), Sep. 2008. (Cited on pages 44 and 45.)

[376] C. Ward, N. Aravamudan, K. Bhattacharya, K. Cheng, R. Filepp, R. Kearney, B. Peterson, L. Shwartz, and C. C. Young, "Workload Migration into Clouds – Challenges, Experiences, Opportunities," in *Proceedings of the Conference on Cloud Computing*, 2010. (Cited on page 166.)

[377] S. Wardley, "Zimki," https://de.slideshare.net/swardley/zimki-2006, 2006, presentation. (Cited on page 40.)

[378] C.-C. Wei, C.-F. Chien, and M.-J. J. Wang, "An AHP-based approach to ERP system selection," *International Journal of Production Economics*, vol. 96, no. 1, pp. 47–62, 2005. (Cited on pages 33 and 34.)

[379] C. D. Weissman and S. Bobrowski, "The Design of the Force.com Multitenant Internet Application Development Platform," in *Proceedings of the Conference on Management of Data*. ACM, 2009, pp. 889–896. (Cited on pages 43, 44, and 45.)

[380] J. West, "How Open is Open Enough? Melding Proprietary and Open Source Platform Strategies," *Research Policy*, vol. 32, no. 7, pp. 1259–1285, Jul. 2003. (Cited on page 140.)

[381] J. Wettinger, U. Breitenbücher, O. Kopp, and F. Leymann, "Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel," *Future Generation Computer Systems*, vol. 56, pp. 317–332, Mar. 2016. (Cited on pages 170 and 193.)

[382] W. E. Winkler, "String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage," in *Proceedings of the Section on Survey Research*. ERIC, 1990. (Cited on page 127.)

[383] E. Wittern, J. Kuhlenkamp, and M. Menzel, "Cloud Service Selection Based on Variability Modeling," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science. Springer, 2012, vol. 7636, pp. 127–141. (Cited on page 132.)

[384] T. Wood, P. J. Shenoy, A. Gerber, J. E. van der Merwe, and K. K. Ramakrishnan, "The Case for Enterprise-Ready Virtual Private Clouds," in *Proceedings of the Conference on Hot Topics in Cloud Computing*, 2009. (Cited on pages 21 and 94.)

[385] X. Xu, "From cloud computing to cloud manufacturing," *Robotics and Computer-Integrated Manufacturing*, vol. 28, no. 1, pp. 75–86, Feb. 2012. (Cited on pages 44 and 45.)

[386] S. Yangui and S. Tata, "CloudServ: PaaS resources provisioning for service-based applications," in *Proceedings of the Conference on Advanced Information Networking and Applications*. IEEE, 2013, pp. 522–529. (Cited on page 203.)

[387] S. Yangui, I.-J. Marshall, J.-P. Laisne, and S. Tata, "CompatibleOne: The Open Source Cloud Broker," *Journal of Grid Computing*, vol. 12, no. 1, pp. 93–109, 2014. (Cited on pages 203 and 204.)

[388] E. Yaqub, R. Yahyapour, P. Wieder, A. I. Jehangiri, K. Lu, and C. Kotsokalis, "Metaheuristics-based Planning and Optimization for SLA-aware Resource Management in PaaS Clouds," in *Proceedings of the Cnference on Utility and Cloud Computing*. IEEE, Dec. 2014. (Cited on pages 44 and 45.)

[389] R. Yasrab, "Platform-as-a-Service (PaaS): The Next Hype of Cloud Computing," *Preprint*, 2018. (Cited on page 61.)

[390] O. M. Yigitbasioglu and O. Velcu, "A review of dashboards in performance management: Implications for design and research," *International Journal of Accounting Information Systems*, vol. 13, no. 1, pp. 41–59, Mar. 2012. (Cited on page 97.)

[391] M. Younas, D. N. A. Jawawi, I. Ghani, T. Fries, and R. Kazmi, "Agile development in the cloud computing environment: A systematic review," *Information and Software Technology*, Jun. 2018. (Cited on pages 4, 113, and 171.)

[392] L. Youseff, M. Butrico, and D. Da Silva, "Toward a Unified Ontology of Cloud Computing," in *Proceedings of the Grid Computing Environments Workshop*. IEEE, 2008, pp. 1–10. (Cited on pages 17 and 20.)

[393] E. K. Zavadskas, A. Mardani, Z. Turskis, A. Jusoh, and K. M. D. Nor, "Development of TOPSIS Method to Solve Complicated Decision-Making Problems: An Overview on Developments from 2000 to 2015," *International Journal of Information Technology & Decision Making*, vol. 15, no. 03, pp. 645–682, May 2016. (Cited on page 33.)

[394] M. Zhang, R. Ranjan, S. Nepal, M. Menzel, and A. Haller, "A Declarative Recommender System for Cloud Infrastructure Services Selection," *Economics of Grids, Clouds, Systems, and Services*, vol. 7714, pp. 102–113, 2012. (Cited on pages 7, 35, 74, 113, 119, 133, and 206.)

[395] S. Zickau, M. Slawik, D. Thatmann, S. Uhlig, I. Denisow, and A. Küpper, "TRESOR – Towards the Realization of a Trusted Cloud Ecosystem," in *Trusted Cloud Computing*. Springer International Publishing, 2014, pp. 141–157. (Cited on page 135.)

[396] B. İ. Zilci, M. Slawik, and A. Küpper, "Cloud Service Matchmaking Approaches: A Systematic Literature Survey," in *Proceedings of the Workshop on Database and Expert Systems Applications*. IEEE, Sep. 2015. (Cited on page 132.)

[397] ——, "Cloud Service Matchmaking using Constraint Programming," in *Proceedings of the Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE, Jun. 2015. (Cited on page 132.)

[398] H.-J. Zimmermann and L. Gutsche, *Multi-Criteria Analyse: Einführung in Die Theorie Der Entscheidungen Bei Mehrfachzielsetzungen [Multi-Criteria Analysis: Introduction to the Theory of Multiple Objective Decisions]*. Springer-Verlag, 2013, ISBN: 9783540544838. (Cited on page 33.)

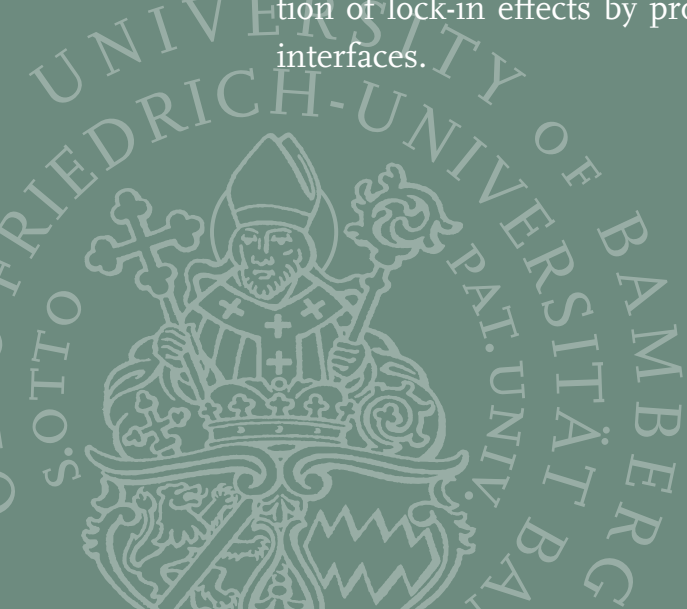
[399] B. Zwattendorfer, B. Suzic, and G. Schanner, “PaaSPort – A Unified PaaS-Cloud Management Application Avoiding Vendor Lock-in,” in *Proceedings of the Conference on E-Society*, 2015. (Cited on pages 109, 174, 191, and 207.)

University of Bamberg Press

In recent years, the cloud hype has led to a multitude of different offerings across the entire cloud market, from Infrastructure as a Service (IaaS) to Platform as a Service (PaaS) to Software as a Service (SaaS). Despite the high popularity, there are still several problems and deficiencies. Especially for PaaS, the heterogeneous provider landscape is an obstacle for the assessment and feasibility of application portability.

Thus, the thesis deals with the analysis and improvement of application portability in PaaS environments. In the course of this, obstacles over the typical life cycle of an application - from the selection of a suitable cloud provider, through the deployment of the application, to the operation of the application - are considered. To that end, the thesis presents a decision support system for the selection of cloud platforms based on an improved delimitation and conceptualization of PaaS. With this system, users can identify offerings that enable application portability. For validation, a case study with a real-world application is conducted that is migrated to different cloud platforms. In this context, an assessment framework for measuring migration efforts is developed, which allows making the differences between compatible providers quantifiable. Despite semantically identical use cases, the application management interface of the providers is identified as a central effort factor of the migration. To reduce the effort in this area, the thesis presents a unified interface for application deployment and management.

In summary, the work provides evidence of application portability problems in PaaS environments and presents a framework for early detection and avoidance. In addition, the results of the work contribute to a reduction of lock-in effects by proposing a suitable standard for management interfaces.



eISBN: 978-3-86309-632-8

9 783863 096328

www.uni-bamberg.de/ubp