IJCC Editorial Board

Editors-in-Chief
Hemant Jain, University of Wisconsin–Milwaukee, USA
Rong Chang, IBM T.J. Watson Research Center, USA

Associate Editor-in-Chief
Bing Li, Wuhan University, China

Editorial Board
Danilo Ardagna, Politecnico di Milano, Italy
Janaka Balasooriya, Arizona State University, USA
Roger Barga, Microsoft Research, USA
Viraj Bhat, Yahoo, USA
Rajdeep Bhowmik, Cisco Systems, Inc., USA
Jiannong Cao, Hong Kong Polytechnic University, Hong Kong
Buqing Cao, Hunan University of Science and Technology, China
Keke Chen, Wright State University, USA
Haopeng Chen, Shanghai Jiao Tong University, China
Malolan Chetlur, IBM India, India
Alfredo Cuzzocrea, ICAR-CNR & University of Calabria, Italy
Ernesto Damiani, University of Milan, Italy
De Palma, University Joseph Fourier, France
Claude Godart, Nancy University and INRIA, France
Nils Gruschka, University of Applied Sciences, Germany
Paul Hofmann, Saffron Technology, USA
Ching-Hsien Hsu, Chung Hua University, Taiwan
Patrick Hung, University of Ontario Institute of Technology, Canada
Hai Jin, HUST, China
Li Kuang, Central South University, China
Grace Lin, Institute for Information Industry, Taiwan
Xumin Liu, Rochester Institute of Technology, USA
Shiyong Lu, Wayne State University, USA
J.P. Martin-Flatin, EPFL, Switzerland
Vijay Naik, IBM T.J. Watson Research Center, USA
Surya Nepal, Commonwealth Scientific and Industrial Research Organisation, Australia
Norbert Ritter, University of Hamburg, Germany
Josef Schiefer, Vienna University of Technology, Austria
Jun Shen, University of Wollongong, Australia
Weidong Shi, University of Houston, USA
Liuba Shrirra, Brandeis University, USA
Kwang Mong Sim, University of Kent, UK
Wei Tan, IBM T.J. Watson Research Center, USA
Tao Tao, IBM T. J. Watson Research Center, USA
Kunal Verma, Accenture Technology Labs, USA
Raymond Wong, University of New South Wales & NICTA, Australia
Qi Yu, Rochester Institute of Technology, USA
Jia Zhang, Carnegie Mellon University – Silicon Valley, USA
Gong Zhang, Oracle Corporation, USA
Call for Articles
International Journal of Cloud Computing

Mission
Cloud Computing has become the de facto computing paradigm for Internet-scale service development, delivery, brokerage, and consumption in the era of Services Computing, fueling innovative business transformation and connected human society. 15 billion smart devices would be communicating dynamically over inter-connected clouds by 2015 as integral components of various industrial service ecosystems. The technical foundations of this trend include Service-Oriented Architecture (SOA), business & IT process automation, software-defined computing resources, elastic programming model & framework, and big data management and analytics. In terms of the delivered service capabilities, a cloud service could be, among other as-a-service types, an infrastructure service (managing compute, storage, and network resources), a platform service (provisioning generic or industry-specific programming API & runtime), a software application service (offering email-like ready-to-use application capabilities), a business process service (providing a managed process for, e.g., card payment), a mobile backend service (facilitating the integration between mobile apps and backend cloud storage and capabilities) or an Internet-of-things service (connecting smart machines with enablement capabilities for industrial clouds).

The International Journal of Cloud Computing (IJCC) aims to be a reputable resource providing leading technologies, development, ideas, and trends to an international readership of researchers and engineers in the field of Cloud Computing. IJCC only considers extended versions of conference papers published at reputable conferences such as IEEE International Conference of Cloud Computing.

Topics
The International Journal of Cloud Computing (IJCC) covers state-of-the-art technologies and best practices of Cloud Computing, as well as emerging standards and research topics which would define the future of Cloud Computing. Topics of interest include, but are not limited to, the following:

- ROI Model for Infrastructure, Platform, Application, Business, Social, Mobile, and IoT Clouds
- Cloud Computing Architectures and Cloud Solution Design Patterns
- Self-service Cloud Portal, Business Dashboard, and Operations Management Dashboard
- Autonomic Process and Workflow Management in Clouds
- Cloud Service Registration, Composition, Federation, Bridging, and Bursting
- Cloud Orchestration, Scheduling, Autoprovisioning, and Autoscaling
- Cloud Enablement in Storage, Data, Messaging, Streaming, Search, Analytics, and Visualization
- Software-Defined Resource Virtualization, Composition, and Management for Cloud
- Security, Privacy, Compliance, SLA, and Risk Management for Public, Private, and Hybrid Clouds
- Cloud Quality Monitoring, Service Level Management, and Business Service Management
- Cloud Reliability, Availability, Serviceability, Performance, and Disaster Recovery Management
- Cloud Asset, Configuration, Software Patch, License, and Capacity Management
- Cloud DevOps, Image Lifecycle Management, and Migration
- Cloud Solution Benchmarking, Modeling, and Analytics
- High Performance Computing and Scientific Computing in Cloud
- Cloudlet, Cloud Edge Server, Cloud Gateway, and IoT Cloud Devices
- Cloud Programming Model, Paradigm, and Framework
- Cloud Metering, Rating, and Accounting
- Innovative Cloud Applications and Experiences
- Green Cloud Computing and Cloud Data Center Modularization
- Economic Model and Business Consulting for Cloud Computing
Table of Contents

EDITOR-IN-CHIEF PREFACE
iv
Hemant Jain, University of Wisconsin–Milwaukee, USA
Rong Chang, IBM T.J. Watson Research Center, USA

RESEARCH ARTICLES
1 Application Migration Effort in the Cloud
Stefan Kolb, University of Bamberg
Jörg Lenhard, University of Bamberg
Guido Wirtz, University of Bamberg

16 A Rest Service Framework for RaaS Clouds
Li Li, Huawei
Wu Chou, Huawei
Min Luo, Huawei

32 Call for Papers: IEEE CLOUD/ICWS/SCC/MS/BigData/SERVICES 2014
Call for Articles: International Journal of Services Computing (IJSC)
Call for Articles: International Journal of Big Data (IJBD)
Editor-in-Chief Preface:
Cloud Management and Assessment

Hemant Jain               Rong Chang
University of Wisconsin–Milwaukee, USA IBM T.J. Watson Research, USA

Welcome to the inaugural issue of International Journal of Cloud Computing (IJCC), the first open access on-line journal on cloud computing. The increasing importance of cloud computing is evidenced from the rapid adoption of this technology in businesses around the globe. The cloud computing is redefining the business model of various industries from video rental (Netflix is enabled by cloud) to small start-up companies (companies can be started with very little investment using cloud infrastructure). The potential of cloud computing is even more promising. The cloud computing combined with developments like internet of things can significantly change the life as we know today. However, to deliver on these promises and to prevent cloud computing from becoming a passing fad significant technical, economic, and business issues need to be addressed. IJCC is designed to be an important platform for disseminating high quality research on above issues in a timely manner and provide an ongoing platform for continuous discussion on research published in this journal. We aim to publish high quality research that addresses important technical challenges, economics of sustaining this environment, and business issues related to use of this technology including privacy and security concerns, legal protection, etc. We seek to publish original research articles, expanded version of papers presented at high quality conferences, key survey articles that summarizes the research done so far and identify important research issues, and some visionary articles. We will make every effort to publish articles in a timely manner.

This inaugural issue collects the extended version of five IEEE CLOUD 2013 articles in the general area of managing Cloud computing environment.

The first article titled, *QOS-Based Resource Allocation Framework for Multi-Domain SLA Management in Clouds* by Lu, Yahyapour, Wieder, Kotsokalis, Yaqub, and Jehangiri tackles the issue of downtime and service unavailability due to live migration. They present an OpenStack based implementation of a cloud resource allocation framework, named Generic SLA Manager, that supports downtime-aware VM selection and allocation during live migration of VMs. A simulation based evaluation of the proposed framework is reported as well.

The second article titled, “Rapidly Alternating Bottlenecks: A Study of Two Cases in N-Tier Applications” by Wang, Kanemasa, Li, Shimizu, Matsubara, Kawaba, and Pu reveals the importance of identifying the location of performance bottlenecks when scaling n-tier applications in computing clouds. They propose a bottleneck detection method that could be used to rapidly detect alternating bottlenecks. Experimental evaluation results for the proposed method are reported via two use cases.

The third article titled, “Cross Cloud MapReduce: A Result Integrity Check Framework on Hybrid Clouds” by Wang, Wei, and Srivatsa tackles the trust issue in adopting large-scale MapReduce on public clouds. They present a framework, named Cross Cloud MapReduce (CCMR), which overlays the MapReduce computation on a hybrid cloud where a master ensures correct result. A result integrity check scheme is also presented for accuracy and performance. Both theoretical and experimental analyses are reported.

The fourth article titled, “Implementation and Empirical Assessment of A Web Application Cloud Deployment Tool” by Sampaio, Costa, Mendonça, and Filho tackles the time-consuming issue in migrating applications to an IaaS cloud via application-specific VM images. They present an automated application deployment approach that requires less cataloged VM images. The approach can be supported via a tool, called TREXCLOUD, and an empirical evaluation of the tool is reported.
The fifth article titled, "A Queuing Model to Achieve Proper Elasticity for Cloud Cluster Jobs" by Salah tackles the issue of achieving proper elasticity for parallelized jobs running on cloud clusters. Based on finite queuing systems, the article presents an analytical model that can be used to determine the minimal number of cloud resources needed to satisfy the SLO requirements with constraints. Discrete Event Simulation is reported to verify the correctness of the proposed model.

We would like to thank the authors for their effort in delivering those five quality articles. We would also like to thank the reviewers as well as the Program Committee of IEEE CLOUD 2013 for their help with the review process. Finally, we are grateful for the effort Jia Zhang and Liang-Jie Zhang made in giving birth to this inaugural issue of International Journal of Cloud Computing (IJCC).

About the Editors-in-Chief

Dr. Hemant Jain is the Interim Director of Biomedical and Health Informatics Research Institute, Roger L. Fitzsimonds Distinguished Scholar and Professor of Information Technology Management at University of Wisconsin–Milwaukee. Dr. Jain specializes in information system agility through web services, service oriented architecture and component based development. His current interests include development of systems to support real time enterprises which have situational awareness, can quickly sense-and-respond to opportunities and threats, and can track-and-trace important items. He is also working on issues related to providing quick access to relevant knowledge for cancer treatment and to providing medical services through a virtual world. Dr. Jain is an expert in architecture design, database management and data warehousing. He teaches courses in database management, IT infrastructure design and management, and process management using SAP. Dr. Jain was the Associate Editor-in-Chief of IEEE Transactions on Services Computing and is Associate Editor of Journal of AIS, the flagship journal of the Association of Information Systems.

Dr. Rong N. Chang is Manager & Research Staff Member at the IBM T.J. Watson Research Center. He received his Ph.D. degree in computer science & engineering from the University of Michigan at Ann Arbor in 1990 and his B.S. degree in computer engineering with honors from the National Chiao Tung University in Taiwan in 1982. Before joining IBM in 1993, he was with Bellcore researching on B-ISDN realization. He is a holder of the ITIL Foundation Certificate in IT Services Management. His accomplishments at IBM include the completion of a Micro MBA Program, one IEEE Best Paper Award, and many IBM awards, including four corporate-level Outstanding Technical Achievement Awards and six division-level accomplishments. He is an Associate Editor of the IEEE Transactions on Services Computing and the International Journal of Services Computing. He has chaired many conferences & workshops in cloud computing and Internet-enabled distributed services and applications. He is an ACM Distinguished Member/Engineer, a Senior Member of IEEE, and a member of Eta Kappa Nu and Tau Beta Pi honor societies.
APPLICATION MIGRATION EFFORT IN THE CLOUD

Stefan Kolb, Jörg Lenhard, and Guido Wirtz
University of Bamberg
{stefan.kolb, joerg.lenhard, guido.wirtz}@uni-bamberg.de

Abstract

Over the last years, the utilization of cloud resources has been steadily rising and an increasing number of enterprises are moving applications to the cloud. A leading trend is the adoption of Platform as a Service to support rapid application deployment. By providing a managed environment, cloud platforms take away a lot of complex configuration effort required to build scalable applications. However, application migrations to and between clouds cost development effort and open up new risks of vendor lock-in. This is problematic because frequent migrations may be necessary in the dynamic and fast changing cloud market. So far, the effort of application migration in PaaS environments and typical issues experienced in this task are hardly understood. To improve this situation, we present a cloud-to-cloud migration of a real-world application to seven representative cloud platforms. In this case study, we analyze the feasibility of the migrations in terms of portability and the effort of the migrations. We present a Docker-based deployment system that provides the ability of isolated and reproducible measurements of deployments to platform vendors, thus enabling the comparison of platforms for a particular application. Using this system, the study identifies key problems during migrations and quantifies these differences by distinctive metrics.

Keywords: Cloud Computing, Platform as a Service, Migration, Case Study, Portability, Metrics

1 INTRODUCTION

Throughout the last years, cloud computing is making its way to mainstream adoption. After the rise of Infrastructure as a Service (IaaS), also the higher-level cloud model Platform as a Service (PaaS) is finding its way into enterprise systems (Biscotti et al., 2014; Carvalho, Fleming, Hilwa, Mahowald, & McGrath, 2014). PaaS systems provide a managed application platform, taking away most configuration effort required to build scalable applications. 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 (Petcu & Vasilakos, 2014). 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 (Hogan, Liu, Sokol, & Tong, 2011; Badger, Grance, Patt-Corner, & Voas, 2012; Petcu, Macariu, Panica, & Craciun, 2013; Di Martino, 2014; Silva, Rose, & Calinescu, 2013).

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, 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. In contrast to an on-premises application, this kind of software is already built to run in the cloud. Therefore, we primarily investigate application portability between cloud vendors, rather than changes that are caused by adjusting an 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 (Hogan, Liu, Sokol, & Tong, 2011; Petcu, 2011). 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 (Kolb & Wirtz, 2014):

RQ 1: Is it possible to move a real-world application between different cloud platforms?

RQ 2: What is the development effort involved in porting a cloud-native application between cloud platforms?

1 See the twelve-factor methodology at http://12factor.net.
The utilized application, Blinkist, is a Ruby on Rails web application developed by Blinkist Labs GmbH. The set of selected PaaS vendors includes IBM Bluemix, cloudControl, AWS Elastic Beanstalk, EngineYard, Heroku, OpenShift Online, and Pivotal Web Services. 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 measurements of deployments to platform vendors, thus enabling the comparison of platforms for a particular application. Using this system, the study identifies key problems during migrations and quantifies differences between the platforms by distinctive metrics. In this study, we target implementation portability (Kolb & Wirtz, 2014; Petcu & Vasilakos, 2014) of the 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, e.g., NoSQL databases, impose substantial lock-in problems. With our results, we are able to compare migration effort between different cloud platforms and to identify existing portability problems.

This article is an extended version of our earlier work (Kolb, Lenhard, & Wirtz, 2015), in which we introduced the case study and deployability framework. We extend (Kolb, Lenhard, & Wirtz, 2015) in multiple directions. Firstly, we provide a more detailed description and illustration of the case study application, the vendor selection process, and the deployment tooling, as well as a more in-depth discussion of related work. Secondly, we update the pricing data for the platforms to contemporary levels and enhance the presentation of the data with several figures. Third, we perform a more sophisticated statistical analysis of the data resulting from the case study, including significance tests on the difference between container-based and VM-based platforms. This analysis reinforces the results from (Kolb, Lenhard, & Wirtz, 2015), hardens our interpretation, and confirms significant differences between container-based and VM-based platforms in deployment times and reliability.

The remainder of the paper is structured as follows. In Section 2, we describe our research methodology including details of the application used, the process of vendor selection, the automation of deployment, and the measurement of deployment effort. Section 3 presents the results of our measurements and describes problems that occurred during the execution of the migration. In Section 4, we review related work. Section 5 discusses limitations and future work that can be derived from the results. Finally, Section 6 summarizes the contributions of the paper.

The primary focus of this study is on the migration execution and evaluation, as the initial planning step can be largely assisted by our cloud brokering tool from (Kolb & Wirtz, 2014) that covers the details of provider brokering and application requirements matching.

2 Methodology

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 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, these steps and modifications are automated to enable a reproducible and comparable deployment among them. To be able 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 includes the functional portability of applications, but also the portability of service management interfaces between vendors (Hogan, Liu, Sokol, & Tong, 2011; Petcu, 2011). In our case, due to the use of open technologies and a cloud-native application, this effort is mainly associated with application deployment. Hence, 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. Therefore, our deployment workflow includes a redeployment of a newer 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 functional and integration tests. As concluding step, we evaluate our findings in the migration evaluation, including measured results and a discussion about problems and differences between providers.

2.1 Migrated Application

The application Blinkist is built by a Berlin-based mobile learning company launched in January 2013 and distills key insights from nonfiction books into fifteen-minute reads and audio casts. Currently, 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. We target the web application\(^2\), which is built in Ruby on Rails. The high-level architecture relevant for this study can be seen in Figure 2.

The user facing front end is a Rails 4 application with access to decoupled business logic written in Ruby. The application uses a MongoDB database for persistence of user data and book summaries. Moreover, page caching and distribution of static application assets, e.g., images, is implemented via Redis and Amazon's CloudFront content delivery network (CDN). The web interface is run with at least two application instances in parallel, hosted by a Puma 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. The application part totals for about 60,000 Lines of Code (LOC).

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 independent external service vendors and are configured via environment variables (see Fig. 2).

The decision on possible candidates for the application can be assisted by the knowledge base and cloud brokering tool\(^3\) presented in (Kolb & Wirtz, 2014). The knowledge base is founded on a taxonomy describing essential components and capabilities of PaaS vendors. The classification was extracted from an extensive study of the vendor landscape and literature reviews. To enable matching and filtering of PaaS offerings, the taxonomy is transformed into a standardized machine-readable profile. The underlying assumption of the matching strategy is that an application can be ported among vendors that support the same application dependencies natively. Figure 3 exemplifies the approach for a set of three requirements, including two application dependencies and one platform capability. The overlapping sections of the requirements include sets of vendors that can be divided into partially compatible, and compatible. Compatible vendors support all listed demands. Therefore, the application is portable to their system. Partially compatible vendors support a subset of requirements and might only be candidates if some of the application requirements can be relaxed or manually upgraded by the customer. In contrast, incompatible vendors (all vendors outside the subsets) do not support any of the requested requirements.

This approach compensates the lack of commonly accepted portability standards in the cloud context. By following the dimensions of our taxonomy, we are also able to solve semantic conflicts between PaaS by providing a restricted common set of capabilities. The feasibility of the approach was validated in (Kolb & Wirtz, 2014). To enhance the accuracy and up-to-dateness of the knowledge base, all data is collaboratively maintained by vendors and customers as an open source project. To the best of our knowledge, this is the most recent, comprehensive, and publicly available collection of PaaS vendors. For more details on the specification and taxonomy see (Kolb & Wirtz, 2014). Listing 1 shows the desired PaaS profile, as defined in (Kolb & Wirtz, 2014), for the application requirements of the case study application.

The broker tool allows us to filter 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 request from Listing 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 vendors have already 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 duplicate control pair (Pivotal and Bluemix). The final selection of the seven vendors, presented in Table 1, was based on a concluding relevance assessment of the remaining offerings.

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

Listing 1. PaaS Profile for the Application

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 we can see in Table 1, there are substantial pricing differences between the vendors. Pricing is based on equivalent production grade configurations dependent on the technology descriptions and specifications of the vendors. Nevertheless, first results reveal performance differences, which are not included in this consideration. Currently, a price-performance value can hardly be investigated by a customer upfront. In general, container-based PaaS are cheaper to start with than VM-based ones. Still, instance performance is lower with respect to the technology setup. When looking at instance prices of container-based PaaS per hour, the most expensive vendor charges over two and a half times more than the cheapest one. However, it is common among PaaS vendors 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 container instances. For example, the differences between Bluemix and cloudControl, caused by a higher free hour quota of Bluemix, will level up with increasing instance count. Pricing among VM-based offerings is even 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 vendors.

### 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 associated with application deployment. To be able to measure and compare this effort, we automate the deployment workflows by using the provider’s client 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 will be denoted explicitly. The automation of the workflows helps to better understand, measure, and reproduce the presented results. We implemented an automatic deployment system, called Yard, that works similar for every provider and prevents errors due to repeatable deployment workflows. This enables a direct comparison of deployment among providers.

Yard 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 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. This typically involves authenticating with the provider platform, creating a new application space, setting necessary environment variables, deploying the

---

4 Pricing is based on selected RAM usage, resp. instance type. 720 h/month 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.

5 See https://github.com/stefan-kolb/paasyard.
application code, and finally verifying the availability of the remote application. 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 with the particular provider. Any necessary provider-specific artifacts, like 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 Yard before any module script execution. The deployments are automated via Bash scripts. User input is inserted automatically via `Here Documents` or `Expect` scripts. This guarantees that user input is supplied consistently for every deployment. As an example, Listing 2 shows the `deploy` script for Heroku.

```bash
#!/bin/bash
echo "-----> Initializing application space..."
# authentication
heroku login <<END
$HEROKU_USERNAME  ...

# create app space
heroku create $APPNAME
# environment variables
heroku config:set MONGO_URL=$MONGO_URL

# assets_url
heroku config:set ASSET_URL=$ASSET_URL

# deployment scripts
deploy

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

Figure 2. Deployment 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 to 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 requests 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 an application installation in a clean platform environment and reset it to default settings by running the `delete` script. The set of scripts must ensure that subsequent deployments are not influenced by settings made to the remote environment through previous runs. As the different build steps and deployment tools will 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 through an abstraction layer of operating-system-level virtualization features⁶.

---

Table 1. PaaS Vendors and Selected Configurations

<table>
<thead>
<tr>
<th>Type</th>
<th>Heroku</th>
<th>cloudControl</th>
<th>Pivotal Web Services</th>
<th>Bluemix</th>
<th>OpenShift</th>
<th>Elastic Beanstalk</th>
<th>EngineYard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isolation</td>
<td>Proprietary</td>
<td>Container</td>
<td>Open Source</td>
<td>Open Source</td>
<td>Open Source</td>
<td>Proprietary</td>
<td>Proprietary</td>
</tr>
<tr>
<td>RAM (instance)</td>
<td>512 MB</td>
<td>512 MB</td>
<td>512 MB</td>
<td>512 MB</td>
<td>512 MB</td>
<td>3.75 GB</td>
<td>3.75 GB</td>
</tr>
<tr>
<td>Pricing</td>
<td>$ 0.035/h</td>
<td>$ 0.04/h</td>
<td>$ 0.015/h</td>
<td>$ 0.035/h</td>
<td>$ 0.025/h</td>
<td>$ 0.067/h</td>
<td>$ 0.12/h</td>
</tr>
<tr>
<td>$ (2 instances/month)</td>
<td>$ 50</td>
<td>$ 50.10</td>
<td>$ 21.60</td>
<td>$ 24.15</td>
<td>$36</td>
<td>$ 48.24</td>
<td>$ 86.40</td>
</tr>
</tbody>
</table>

⁶ See https://www.docker.com/whatisdocker for more details.
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.

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. In (Lenhard, Harrer, & Wirtz, 2013), we proposed and validated a measurement framework for evaluating these characteristics for service orchestrations and orchestration engines, based on the ISO/IEC SQuaRE quality model (ISO/IEC, 2011). Despite the difference 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 for instance the expertise of the workers involved. However, it is possible to introduce such factors by adding weighting factors to the metrics computation, as for instance done in (Sun & Li, 2013).

![Deployment Metrics Framework](Image)

**Figure 5. Deployment Metrics Framework**

As cloud platforms are preconfigured and managed environments, there is no need to consider the installability of the environment itself, as in (Lenhard, Harrer, & Wirtz, 2013). Instead, the focus lies on the deployability of an application to a cloud platform. Figure 5 outlines the adapted framework for deployability. We capture this quality attribute with the direct metrics *average deployment time* (ADT), *deployment reliability* (DR), *deployment flexibility* (DF), *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, DR, and DF are classic size metrics in the sense of (Briand, Morasca, & Basily, 1996). This means, they are non-negative, 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 on the interval scale of [0,1]. The following paragraphs briefly introduce the metrics.

**Average deployment time (ADT).** 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 repeating this process a suitable number of times. Here, we use the median as measure of central tendency.

**Deployment reliability (DR).** Deployment reliability captures the reliability of an application deployment to a particular vendor. It is computed by repeating the deployment a suitable amount of times and dividing the number of successful deployments of an application \(a(N_{\text{suc}})\) with the total number of attempted deployments \(N_{\text{total}}\): \[\text{DR}(a) = \frac{N_{\text{suc}}}{N_{\text{total}}}.\] DR(a) will be equal to one, if all deployments succeed.

**Deployment flexibility (DF).** (Lenhard, Harrer, & Wirtz, 2013) defines deployment flexibility as the amount of alternative ways that exist to achieve the deployment of an application. In our case, available deployment techniques are, e.g., CLI-based deployment, web-based deployment or IDE plug-ins. The more of these options a platform supports, the more flexible it is. As we are concentrating on deployment via command line tools in this study, hereafter, we omit a more detailed consideration of this metric.

**Number of deployment steps (NDS).** 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 Sect. 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. Usually, there are different ways to deploy an application. Here, we tried to find 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 vendors. As an example, the value of NDS for the deployment script in Listing 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 (NDSP).**
The number of steps for a deployment are 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 deployment script in Listing 2 uses six different variables and requires no additional command line parameters, resulting in NDSP(heroku) = 6.

**Effort of deployment steps (EDS).**
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. Given an application a: EDS(a) = NDS(a) + NDSP(a). For our example, this amounts to EDS(heroku) = 10.

**Number of configuration & code changes (NCC).**
The deployment of an application to a particular vendor may require the construction of different vendor-specific configuration artifacts. This includes platform configuration files and files that adjust the execution of the application, e.g., a Procfile. Again, the construction of these files results in effort related to their size (Lenhard, Harrer, & Wirtz, 2013). 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 LOC function. Furthermore, it might be necessary to modify source files to mitigate incompatibilities between different platforms. This can be due to unsupported dependencies that must be adjusted, e.g., native libraries or middleware versions. Any of those changes will be measured via a LOC function. The sum of the size of all configuration files and the amount of code changes corresponds to the configuration & code changes metric.

For an application a that consists of the configuration files file1, ..., fileN_conf and the code files file1', ..., fileN_code' along with their platform-adjusted versions file1'', ..., fileN_code'' NCC can be computed as:

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

**Number of build steps (NBS).** Another effort driver in traditional application deployment is the number of build steps, i.e., source compilation and the packaging of artifacts into an archive (Lenhard, Harrer, & Wirtz, 2013). 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 the application artifacts is possible (NBS(a) = 0), shifting the responsibility of package construction to the platform. For some platforms it is still necessary, which is why we capture it in the same fashion as the number of deployment steps.

**Deployment effort (DE).** To provide a comprehensive indicator for effort associated with deployment, we provide 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.

## 3 Results

In this section, we first describe the execution of the measurements, followed by a presentation, discussion, and interpretation of the results in Section 3.2 and a summary in Section 3.3.

### 3.1 Execution of Measurements

As part of our migration experiment, we need to compute values for the deployment metrics from the preceding Section 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, 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 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, where excluded from the calculation. EngineYard forms an exception in the measurement.

---

7 See https://devcenter.heroku.com/articles/procfile.

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

9 See http://linux.die.net/man/1/time.
management interfaces. With the system should be homogenized by, e.g., unified container environments, major parts of the interaction are to be expected and can only be prevented by unified servers, were needed. Whereas some of these problems compromises with certain technology setups, e.g., web getting the application to run was difficult and based PaaS. Even with this common kind of application, instance access for debugging, especially with container-documentation of the vendors and missing direct additional important obstacles are incomplete work was required. Besides the captured effort values, that can show differences between vendors. Such differences will be separately identified with the help of significance tests. All deployments were measured with a single instance deployment at first, i.e., no scaling included. The values for each metric were evaluated and validated by an in-group peer review. The gathered metrics can be seen in Tables 2 and 3.

Table 2. Deployment Efforts

<table>
<thead>
<tr>
<th>Effort of deployment steps (EDS)</th>
<th>Heroku</th>
<th>cloudControl</th>
<th>OpenShift</th>
<th>Pivotal</th>
<th>Bluemix</th>
<th>Elastic Beanstalk</th>
<th>EngineYard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>15</td>
<td>24</td>
<td>17</td>
<td>17</td>
<td>12</td>
<td>23</td>
</tr>
<tr>
<td>Number of deployment steps (NDS)</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Automated</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Manual</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of deployment step parameters (NDSP)</td>
<td>6</td>
<td>10</td>
<td>18</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>Number of configuration &amp; code changes (NCC)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>40</td>
<td>7</td>
</tr>
<tr>
<td>Deployment artifacts</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>40</td>
<td>7</td>
</tr>
<tr>
<td>Application code</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of build steps (NBS)</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Deployment reliability (DR)</td>
<td>0.96</td>
<td>0.72</td>
<td>0.78</td>
<td>1</td>
<td>0.89</td>
<td>0.99</td>
<td>1</td>
</tr>
<tr>
<td>Average deployment time (ADT)</td>
<td>6.75 min</td>
<td>9.13 min</td>
<td>8.42 min</td>
<td>5.83 min</td>
<td>7.03 min</td>
<td>15.94 min</td>
<td>28.44 min</td>
</tr>
</tbody>
</table>

Table 3. Redeployment Efforts

<table>
<thead>
<tr>
<th>Effort of deployment steps (EDS)</th>
<th>Heroku</th>
<th>cloudControl</th>
<th>OpenShift</th>
<th>Pivotal</th>
<th>Bluemix</th>
<th>Elastic Beanstalk</th>
<th>EngineYard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Number of deployment steps (NDS)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Number of deployment step parameters (NDSP)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Deployment reliability (DR)</td>
<td>0.96</td>
<td>0.97</td>
<td>0.97</td>
<td>1</td>
<td>0.93</td>
<td>0.98</td>
<td>0.96</td>
</tr>
<tr>
<td>Average deployment time (ADT)</td>
<td>6.69 min</td>
<td>5.71 min</td>
<td>7.41 min</td>
<td>5.73 min</td>
<td>6.61 min</td>
<td>8.71 min</td>
<td>8.25 min</td>
</tr>
</tbody>
</table>

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 vendors in a timely manner. As a downside, one vendor 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.

3.2 Effort Analysis

The following section describes the results of our case study in detail. We discuss the metric values and their implications and give insights into the problems that did occur during the migrations.

**Effort of deployment steps (EDS).** 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 tooling for every vendor and to adapt existing automation. Figure 6 depicts the effort of deployment steps of all vendors. On average, deployment takes 17 steps with a maximum spread of 14 and a standard deviation of 5. Some vendors require more steps, whereas others require less steps but more parameters.  Heroku, cloudControl, Pivotal, and Bluemix are driven by
a similar concise deployment workflow. In contrast, OpenShift requires a cumbersome configuration of the initial code repository. Only the deployment for EngineYard could not be automated entirely. The creation of VM instances must be initiated via a web interface, whereas the application deployment can be triggered by the client tools.

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. In comparison, this must be explicitly configured up-front with 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.

Additionally, in contrast to other vendors, a custom recipe repository must be cloned to use environment variables and 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 required 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 vendor, 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.

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 the effect of having a cloud-native application based on open technologies. Furthermore, all vendors 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 (code packages) cause portability problems between PaaS offerings. These Gems may depend on special system libraries that are not available in every PaaS offering and cannot always be installed afterwards. Buildpacks can help to prevent such problems by unifying the environment bootstrapping, making it easier to support special dependencies that would otherwise be hard to maintain.
Number of build steps (NBS). The NBS for deployment is similar between vendors. As sole packaging requirement, most vendors mandate that the source code is organized in a Git repository, either locally or remotely (NBS = [3,4]). This is often naturally the case but must be counted as build effort.

Deployment reliability (DR). For some vendors, 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 uploaded SSH keys or other platform configuration problems. From the descriptive data in Table 2, 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\(^\text{10}\). Since deployment success is coded in a binary fashion, i.e., either success or failure, it is possible to apply a binomial test. We 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 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 section, 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 we mentioned at the beginning of this section and shows room for improving the maturity of the platforms.

Average deployment time (ADT). Figure 8 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. The authors of (Mao & Humphrey, 2012) 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.

A majority of the deployment time (≈ 46\%) is spent for installing necessary application dependencies with Bundler. Another considerably large part is the asset precompilation\(^\text{11}\) of CSS files, JavaScript files, and static assets (≈ 18\%). The remaining time (≈ 35\%) is consumed by other tasks of the build process and the platform configuration.

---

\(^{10}\) All statistical tests in this paper were executed using the R software (R Core Team, 2015).

\(^{11}\) See http://guides.rubyonrails.org/asset_pipeline.html.
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.

Measured time values are also interesting for the case of redeployment. To that end, we take a newer version from a typical code sprint of Blinkist's release cycle. Besides code changes, it includes new and updated versions of dependencies as well as asset changes. In general, the redeployment times are less than for the initial deployment, which can be mainly attributed to dependency caching. In total, the installation of updated or new dependencies takes \(\approx 50\%\) less time than on the initial deployment. In our redeployment, there are more assets to process, resulting in a slightly longer precompilation time than for the initial deployment. For redeployment, all timings of the vendors are in a close range. Here, VM-based offerings catch up with container-based PaaS due to the absence of environmental changes. The average redeployment time for all offerings is 7.02 min and only deviates by 65 seconds. Some vendors still benefit from a better deployment configuration, e.g., parallelized Bundler runs. Vendors 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 still are 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 non-normal. 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 vendors 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 vendor, or if it is sufficient to decide between VM-based and container-based vendors. Put differently, we checked if there are significant differences among the container-based vendors as well. We omit a detailed presentation of the results here due to the amount of comparisons necessary (each pair of vendors 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 vendors, 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 Cloud Foundry 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 vendor. Whereas this observation was only validated for application deployment in this study, it can be expected that this also holds for application response times, which should be investigated separately.

**Deployment effort (DE).** The values for total deployment effort are substantially different between the platforms, with a maximum spread of 41 and a standard deviation of 13. Most 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.

![Figure 9. Overall Deployment Effort](image_url)

### 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 vendors (RQ 1). Only one vendor 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 vendors. We had to make trade-offs and changes to the technology setup, especially the server startup. With the automation...
of the migration, together with the presented toolkit and deployment metrics, we could quantify the effort of the migration (RQ 2). Our results show that there are considerable differences between the vendors, especially between VM-based and container-based offerings. Our measurements provide insights into migration effort, both quantifying the developer effort caused by deployment steps and code changes, as well as effort created by deployment and redeployment times of the application.

4 RELATED WORK

Jamshidi et al. (2013) 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 (Hogan, Liu, Sokol, & Tong, 2011; Petcu, Macariu, Panica, & Craciun, 2013; Di Martino, 2014; Silva, Rose, & Calinescu, 2013; Badger, Grance, Patt-Corner, & Voas, 2012). This study is a first step towards filling the identified gaps.

In (Kolb & Wirtz, 2014), we already ported a small application between five PaaS vendors 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., (Chauhan & Babar, 2011; Chauhan & Babar, 2012; Vu & Asal, 2012). These studies typically describe a migration case study including basic considerations of provider selection and application requirements and afterwards 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 (2013) 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. (2010) and Bessera et al. (2012), focus on minimizing cost aspects in their migration decision processes. A broader set of target variables is presented by Menzel and Ranjan (2012) 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, also several studies exist to assist automatic application inspection and transformation for migration execution. Sharma et al. (2013) 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. Beslic et al. (2013) discuss an approach for an application migration among PaaS vendors related to our study. 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 quantify the effort of the described translations.

When it comes to the measurement of the 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 (Boehm, et al., 2000) or Putnam (Putnam, 1978). However, such traditional algorithmic models were developed in the context of software development projects, not for on-premises or cloud migration (Sun & Li, 2013). Based on the accepted estimation model function points (Albrecht & Gaffney, 1983), Tran et al. (2011) define a metric, called cloud migration point (CMP), for effort estimation of cloud migrations. Another study by Sun and Li (2013) estimates expected effort in terms of man hours for an infrastructure-level migration. Similar, Miranda et al. (2013) 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 unveiling occurred effort, the focus is often on operational cost comparisons (Khajeh-Hosseini, Greenwood, & Sommerville, 2010; Khajeh-Hosseini, Sommerville, Bogaerts, & Teregowda, 2011; Tak, Uragaonkar, &
Sivasubramaniam, 2011; Andrikopoulos, Song, & Leymann, 2013), e.g., infrastructure costs, support, and maintenance or migration effort in man hours (Tran, Keung, Liu, & Fekete, 2011; Maenhaut, Moens, Ongenae, & De Turck, 2015). Solely, Ward et al. (2010) mention migration metrics related to the effort to create build automation or the server provisioning time comparable to our deployment metrics.

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, indicated by this paper, is the unification of management interfaces for application deployment and management of cloud platforms. Despite semantically equivalent workloads, the current solutions are invariably proprietary at the expense of recurring developer effort when moving between vendors. To overcome this issue, we are currently developing Nucleus, a RESTful abstraction layer and Ruby gem that unifies core management functions of PaaS systems. It forms a next step in our ongoing efforts towards a unified management interface for Platform as a Service. As revealed by our study, further work is needed regarding the unification of runtime environments between cloud vendors and also on-premises 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 vendors 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 CONCLUSION

In this paper, we carried out 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 between the platforms. We were able to move the application to a majority of vendors, 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 Yard, a Docker-based deployment system that is able to deploy source code to different platform vendors via isolated containers. Yard also includes a small abstraction layer for unified creation, deployment, and deletion of applications throughout the vendors. 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 vendors, aggregated to different formal effort metrics. The results show that there are major differences between the vendors and the associated effort of the migration. 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. As part of the study, we identified problems that prevented the portability of the application among vendors 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 ACKNOWLEDGMENT

We would like to thank the people at Blinks Labs GmbH for their generosity by providing their assets for carrying out this research. Special thanks go to Sebastian Schleicher as primary contact for fruitful discussions and assistance. We also appreciate the support given by the selected vendors.

8 REFERENCES


---

12 See https://github.com/stefan-kolb/nucleus.


Authors

Stefan Kolb, M.Sc., is a PhD student at the Distributed Systems Group of the University of Bamberg. In his research, he focuses on Cloud Computing, especially on application migration in Platform-as-a-Service environments. He is the author of http://www.paasify.it and several other open source tools that help in cloud application migration.

Jörg Lenhard, M.Sc., is a PhD student at the Distributed Systems Group of the University of Bamberg. His research interests include portability assessment of service-oriented software, process engine benchmarking, software metrics definition and evaluation, and the quality comparison of open source and proprietary software. He is a regular contributor to several open source projects.

Prof. Dr. Guido Wirtz is a full professor of computer science, heads the Distributed Systems Group of the University of Bamberg and is Vice President of Technology and Innovation of the University of Bamberg. He received his PhD from the University of Bonn and his habilitation from the University of Siegen. His research interests are in the field of software development for complex, esp. distributed, systems on all levels. This includes design methods, visual languages and tools for distributed systems development as well as middleware, SOA and cloud computing. Current interests are on the seamless transition from business processes to their implementation in a SOA and cloud context as well as in correctness, conformance, compatibility and portability issues of process languages like BPMN.
A REST SERVICE FRAMEWORK FOR RaaS CLOUDS
Li Li, Wu Chou, and Min Luo
Huawei
{li.nj.li, wu.chou, min.ch.luo}@huawei.com

Abstract
RaaS (Resource-as-a-Service) clouds represent a new market and technology driven paradigm shift from renting prebuilt virtual machines towards renting resources to compose elastic machines. By offering cloud users fine-grained resources, including CPU, memory, disk and network based on their demands, RaaS clouds have the potential to address several inherent problems in IaaS clouds. RaaS architecture requires a new service layer to abstract and compose heterogeneous and dynamic fine-grained cloud resources at high frequency. This paper proposes a REST service framework consisting of Resource-Oriented Network (RON) and monadic service composition. RON abstracts heterogeneous cloud resources with uniform REST resources and represents their dynamic relations with hypertext. Monadic service composition provides a concise functional programming language with monad to concurrently control large scale RON through dynamically generated workflows. A prototype system has been implemented based on Docker containers and Linux control groups. Our experimental results show that the approach is feasible and the performance is consistent with the client and server workloads.

Keywords: RaaS, REST, Resource-oriented Network, elastic machine, monad, service composition, Docker, control group

1. INTRODUCTION
Conventional IaaS clouds are based on Virtual Machine (VM) technologies that emulate physical machines. An IaaS cloud provider offers its users a dozen or so VM types. Once the type of a VM is chosen, it will have fixed amount of CPU, memory, and I/O capacities that cannot be changed easily by the user. A user typically rents a VM for hours and days, and the cloud providers price VM and charge the users by minutes or hours. IaaS clouds based on fixed VM and long rental intervals are neither optimal for cloud providers nor flexible enough for cloud users, because in most cases the workloads and resource demands of a VM are unpredictable.

Resource-as-a-Service (RaaS) (Ben-Yehuda 2014) is a cloud computing paradigm to capture the current trends from renting virtual machines with prebuilt resources towards renting resources to compose elastic machines with variable capacities. In RaaS Cloud, fine-grained resources, like memory, CPU, and storage, will become the basic units that a cloud user can rent from a cloud provider at short time intervals. Users can dynamically mix and match different amount of fine-grained resources and change them on demand for an elastic machine. Several public cloud providers, including Amazon, Google, CloudSigma, Gridspot, and ProfitBricks, are moving towards this direction, by letting the users customize the capacities of their VMs, while pricing and charging the resources by minutes. As this trend continues, the future RaaS clouds may allow users to sell and buy fine-grained and fine-timed resources, such as 1GB memory for 10 seconds at $0.05, through automated economic agents. RaaS clouds can significantly increase resource utilization for the providers, and at the same time, decrease the rental cost for cloud users, without compromising the performance of cloud applications.

Besides the economic and market forces mentioned in (Ben-Yehuda 2014), several technological forces are driving cloud computing towards RaaS. The first one is the container-based virtualization, accelerated by the advance of Docker (Docker 2015) technologies that make the containers easy to build, deploy, control, and manage. The second one is the advances in network architectures, including SDN and NFV, which provide flexible control over fine-grained network resources. The third one is the disaggregated datacenters where fine-grained resources can be shared and composed from resource pools.

Docker container is much more efficient than VM, because it eliminates the hypervisor and the guest OS layers in VM, by having the container and the host share the Linux kernel while keeping them separated using Linux namespaces. A recent study (Russell 2015) shows that Docker containers consume only 6% of CPU and 16% of memory comparing to a KVM based VM, and moreover, its boot time is only 60% and its reboot time is only 2% of the corresponding VMs. Docker image size is typically in the MB range, whereas VM images are typically in the GB range. In addition, workload tests show that processes running inside Docker containers have almost the same performance as running on host machines, making containerization more efficient than virtualization for cloud computing.

The significant improvement of container over VM is not limited in performance but also in the deployment density. The small footprint of container leads to a deployment density that can be 6-12 times of the deployment density of VMs. High deployment density in cloud computing means that fewer physical machines are needed to support the same amount of clients, thereby
reducing the capital (e.g. purchase) and operation (e.g. energy) costs for cloud providers. For these reasons, many companies and open source communities, including Google, IBM, Microsoft, VMware, and OpenStack, are investing in containerization technologies for cloud computing, and the advance of Docker makes cloud computing at the Internet scale a potential reality.

Recent advances in network architectures can be regarded as following a trend towards RaaS. In particular, the combination of SDN and NFV are opening datacenter networks to the cloud users as a programmable platform.

Software-Defined Networking (SDN) (ONF 2015) decouples the data plane and control plane that were tightly coupled and distributed in traditional network architecture. This separation enables a logically centralized SDN controller to control an entire network of devices through some southbound APIs and to provide a northbound REST API for users to create, configure, and control a virtual network over the HTTP protocol.

Network Functions Virtualization (NFV) (NFV 2015) moves various network functions, such as NAT and FW, from proprietary hardware boxes to commodity servers, to reduce cost and increase flexibility. Virtual network functions from different vendors can run on a common datacenter network platform. Users can rent network functions and compose them to process network flows according to their business needs.

Disaggregated datacenters can radically change cloud computing, by allowing a virtual machine to access fine-grained resources drawn from some large resource pools instead of from the servers. In addition, while storage has been disaggregated from CPU and memory by technologies such as NAS (Network Attached Storage) and SAN (Storage Area Network), recent work (Lim 2009, Han 2013, Weiss 2014, Li, C.S. 2015, Yoshikawa 2015) also explored the disaggregated CPU and memory, such that a CPU core in a pool can be composed with memory frames in another pool within the limited delay.

Disaggregated datacenter aims to eliminate resource fragmentations inherent in server-based datacenters and RaaS is a natural fit for that, because they share the same goal that machines are not prebuilt but dynamically composed on demand. We think the central idea of RaaS can be embodied by Elastic Machine (EM) described in this paper, which can take the form of VM, Docker container or other types of machines, and it has the following properties:

- An EM can be composed from cloud resources (distributed or collocated) on demand.
- The cloud resources of an EM can change over time at high frequency.
- An EM is autonomous and can trade cloud resources with the provider or another EM.
- An EM can be composed from other EMs.

To realize RaaS based on the concept of EM requires a new service layer to abstract and compose fine-grained and fine-timed cloud resources, and the service layer need to address the following challenges:

1. Heterogeneity: how to deal with variations in space, i.e. abstract the wide range of fine-grained resources (e.g. CPU, GPU, FPGA and ASIC) that differ in many dimensions, including types, functions, sizes, access methods, and communication protocols, into some uniform services for the clients.
2. Dynamism: how to deal with variations in time, i.e. hide from clients the changes to fine-grained resources due to hardware, firmware, and software updates.
3. Composability: how to maintain and represent various and dynamic organizations of cloud resources in an EM by a uniform, extensible, and easy to understand way for the clients (for example, an EM connects several hybrid ARM64 cores with the same instruction set but different microarchitectures to remote memory).
4. Overhead: how to minimize the potential overhead introduced by the service layer to the communications between the fine-grained resources, when they have strict latency requirements (for example, the 20ms latency requirement for CPU-Memory communication).
5. Scalability: how to use small programs to concurrently and efficiently control 10⁵ to 10⁷ fine-grained resources in a datacenter, which has 10⁵ or more EMs, and each EM can have 10-100 fine-grained resources.

Heterogeneity of cloud resources can come from two sources: intrinsic and configuration variations. Fine-grained cloud resources differ in types, functions, interfaces, performances, and costs. For example, computing resources include CPU, GPU, FPGA and ASIC, etc., and memory resources can be volatile or nonvolatile, physical or virtual, and organized in UMA or NUMA architectures. The physical servers in a datacenter may be configured to run different operating systems or the same operating system with different services.

Dynamism of cloud resources can come from two sources: evolutions by design and changes by configurations. As the Linux operating system undergoes updates and redesigns, the APIs to access the fine-grained resources can also change over time. The recent redesign of the cgroup hierarchy (Heo 2014), should it be widely adopted, will alter the current Linux cgroup structure. As fine-grained resources are added or removed by operations from the clients, the topology of the RON can change in some significant ways as well.

To address these challenges, this paper proposes a REST service framework that consists of Resource-Oriented Network (RON) and monadic service composition. In particular, RON addresses the first four challenges, while monadic service composition addresses the last challenge. RON abstracts heterogeneous and dynamic cloud resources...
and their relationships as connected and uniform REST services. Monadic composition provides a concise functional programming language with monad to concurrently and efficiently discover and invoke the REST services in RONs through dynamically generated workflows.

This framework has been combined with Docker container technology, where Docker and its REST API provide a basic construct of computing unit, and our REST framework provides an efficient and extensible architecture to dynamically manage the resources allocated to Docker containers.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the proposed RON framework and its architectural properties, and Section 4 describes some concrete RON REST services and the solutions to address the above challenges. Section 5 describes monadic service composition language and its runtime architecture. Section 6 describes the implementation and experimental results, and we conclude the findings of this paper with Section 7.

2. RELATED WORK

2.1 FINE-GRAINED RESOURCE CONTROL

Figure 1 is a high-level architecture of Docker, where a Docker daemon provides REST APIs to launch and manage Docker containers from Docker images that package the additional libraries and files needed by the applications. Docker containers can run on any Linux flavored platforms as long as they have the required Linux kernel. A Docker container has its own Linux file system and process namespace isolated from the host Linux OS and other containers. To users, a Docker container is like a VM. For example, we can run an Ubuntu container on a CentOS host. Docker uses Union File System to organize its images so that a Docker image can be updated by just downloading the added layer to bridge the difference.

<table>
<thead>
<tr>
<th>Applications (Web server, database, etc.)</th>
<th>Docker Containers</th>
<th>Docker images</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docker Daemon (REST API)</td>
<td>Linux OS (Redhat, CentOS, Ubuntu, Fedora)</td>
<td>Hardware (CPU, memory, storage, network)</td>
</tr>
</tbody>
</table>

*Figure 1. Docker architecture*

Docker uses Linux control groups (cgroups) (Cgroup 2015) to allocate hardware resources among the containers, such that the capacity of each container can be adjusted dynamically using the cgroup API. For example, we can change the memory limit or the number of CPU cores of a running container at any time. Since cgroup is a Linux kernel feature that works for any processes, it is possible to resize the capacity dynamically to meet the application needs. However, as far as we know, cgroup has only a command-line interface and there is no REST API to cgroups, making it difficult to manage the fine-grained resources as services.

Most hypervisors, such as KVM, vSphere, Xen, and Hyper-V, provide a control interface similar to the one in (VMWare 2015) for dynamically adjusting the capacity of a VM, including its CPU time, CPU cores, and memory size. But not all guest OS can take advantage of the increased capacity in the same way as containers, and some tests (Rao 2009, 2011) show that a VM takes minutes to become stable after its capacity is changed. Docker avoids this issue because it is container based. A container does not involve any hypervisor or guest OS in its operation, and its running resource state can be changed almost instantly.

There are several computer cluster management systems based on Docker technologies. Kubernetes (Kubernetes 2015) is an open-source project from Google that provides REST API to deploy and manage tightly integrated containers (called Pod) in clusters. But it can only control a Pod as the smallest unit. Apache Mesos (Mesos 2015) is an open source cluster scheduler that collects fine-grained computing resources, e.g. CPU and memory, in a cluster and offers them to the registered frameworks which in turn will schedule their tasks based on the offers. However, Mesos does not provide a REST API to control those fine-grained resources while the tasks are running.

Docker has been integrated into OpenStack (OpenStack 2015), an open-source IaaS cloud platform, where Docker containers can run on top of VMs. However, OpenStack currently does not provide REST API to dynamically reconfigure container’s capacity. CoreOS (CoreOS 2015) is a Linux OS customized for Docker containers and it provides API and job scheduler to deploy and manage containers on the CoreOS hosts. However, the smallest control unit in CoreOS is a container, and it does not control fine-grained resources within a container.

2.2 SERVICE COMPOSITION

WS-BPEL (Jordan 2007) is a standard XML based service composition language. Its primitives include basic activities for service interactions, such as <invoke> that sends a message to a service, <reply> that receives a message from a service, and structural activities that control these interactions. Structural activities include <sequence> that executes the child activities sequentially, <flow> that executes the child activities concurrently, and <pick> that waits for exactly one from the child activities. However, it falls short to support runtime generation of workflows:

- The structural activities in WS-BPEL are static because its child activities must be fully specified at the design time. This restriction makes it difficult to
handle situations where the resource identifications and operations are unknown at design time.

- WS-BPEL depends on WSDL, but most REST services do not use WSDL, because WSDL groups operations by centralized interfaces, whereas REST services group operations by distributed hypertext.
- WS-BPEL uses complicated correlation set to identify process sessions, but this is incompatible with REST services that use URI to identify sessions.
- From the performance and portability perspectives, using a WS-BPEL engine this way is an over-kill as most of its features are not directly applicable in the paradigm of REST services.

On the other hand, WS-BPEL can define abstract process, which is a template with opaque or omitted activities. These opaque and omitted activities in WS-BPEL can be instantiated and extended by different executable processes. Although this is a very useful feature to reuse an abstract process for different purposes, it lacks a way to compose abstract and executable processes. In fact, abstract processes cannot be embedded in executable processes by definition, making it difficult to use in practice.

BPEL for REST (Pautasso 2008) is an effort to adapt WS-BPEL to REST service composition by adding a set of REST service related activities to WS-BPEL, including <Get>, <Put>, <Post>, <Delete>, and <onGet> that are modeled after HTTP 1.1 operations. Bite (Rosenberg 2008) is another REST service composition language modeled after WS-BPEL, but it is much more lightweight than WS-BPEL. Bite encodes a service composition as a flat graph that consists of activities and links between them. The activities include HTTP communications, utilities, and control primitives, while the links are URI templates that can change dynamically. Communication activities can be executed concurrently if there is no dependency between them. Bite treats a composition process itself as a REST resource that can be created, inspected, and managed with them. Bite treats a composition process itself as a REST resource that can be created, inspected, and managed with them. Bite treats a composition process itself as a REST resource that can be created, inspected, and managed with them. Bite treats a composition process itself as a REST resource that can be created, inspected, and managed with them. Bite treats a composition process itself as a REST resource that can be created, inspected, and managed with them.

JOpera (Pautasso 2009) provides a GUI for creating a service composition with control and data flow graphs, where nodes in a control flow represent tasks that invoke REST services or process internal data. A task can contain variable parameters, such as URI, method, and body, to be substituted at runtime. The tasks in JOpera are dynamically bound to runtime programs by various adaptors, such that a task can include Unix shell, XSLT, and Java. JOpera also treats a workflow as a regular REST resource like Bite.

Petri-Net models have also been applied to create and analyze service compositions in several research work (Hamadi 2003, Decker 2008, Xiong 2010, Alarcon 2011). In a service composition, Petri-Net transitions represent activities that operate on tokens (messages and data) stored in the places, while the entire workflow is defined by a Petri-Net structure. Petri-Net is a powerful mathematical model that can represent concurrency, synchronization, and choices, but it lacks primitives to interact with REST services, or rules that govern the compositions of Petri-Nets.

Although some of these languages provide templates to facilitate dynamic service compositions, they lack a concise and formal method to compose the templates, or abstract a workflow for different applications.

FP System (Backus 1977) is a functional programming language proposed by John Backus in his Turing Award Lecture. The FP language makes extensive use of lists and algebraic rules to compose functions, such that a short program can represent a complex process. Monad (Wadler 1992) is a functional programming construct to encapsulate computations with side-effects, such as I/O and state, and allow them to be manipulated with functions. A monad $M x$ is defined as an operator $M$ on type $x$, together with three functions:

1. $\text{map}: (x \rightarrow y) \rightarrow (M x \rightarrow M y)$: transform monad $M x$ to monad $M y$ with a function $x \rightarrow y$.
2. $\text{unit}: x \rightarrow M x$: construct monad $M x$ from type $x$.
3. $\text{join}: M M x \rightarrow M x$: flatten a nested monad.

These functions must satisfy the following rules, where $id$ denotes the identity function that maps anything to itself, $f, g$ are functions, and operator $\circ$ composes functions:

- $\text{map}(id) = id$;
- $\text{map}(g \circ f) = \text{map}(g) \circ \text{map}(f)$;
- $\text{map}(f) \circ \text{unit} = \text{unit} \circ f$;
- $\text{map}(f) \circ \text{join} = \text{join} \circ \text{map}(\text{map}(f))$;
- $\text{join} \circ \text{unit} = id$;
- $\text{join} \circ \text{map}(\text{unit}) = id$;
- $\text{join} \circ \text{join} = \text{join} \circ \text{map}(\text{join})$.

The diagram in Figure 2 illustrates how these functions are connected by these rules. For brevity, the $id$ functions that loop back on each box are not shown. By these rules, we can transform the typed data inside monads using functions.

![Figure 2. Illustration of Monad Laws](image-url)

Taking rule c) as an example: it says that if $M x$ is a monad constructed from $x$ by $\text{unit}$, and $f$ is a function that maps $x$ to $y$, then we can apply $f$ to $M x$ to produce a monad $M y$, by first applying $f$ to $x$ to produce $y$, then constructing the monad $M y$ from $y$.

The $\text{map}$ function is often used to apply a function to a list of items, which is a basic monad. Given a list monad: $\text{List int} = [1, 2, 3]$, $f(x) = 2x$ and $g(x) = x - 1$, we have:
map(g◦f)([1,2,3])=map(g)([2,4,6])=[1,3,5].

The \textit{join} function can flatten the nested list monads, for example:

\text{join}([[1,2,3],[4,5]])=[1,2,3,4,5].

A monad can also be defined by a \textit{bind} function: \text{bind}: M \ x → (x→M \ y)→M \ y. Function \text{bind} applies a function \( x→M \ y \) to \( M \ x \), such as function \( h \) in Figure 2, to produce monad \( M \ y \). The \textit{bind} and \textit{map} functions are related by the equations:

\( h=\text{unit}f \) and \text{bind}(M \ x, h)=\text{map}(f)(M \ x).

Looking at the diagram in Figure 2 from left to right, we can regard monad \( M \) as a container that carries a work item (e.g. \( x \)) through a pipeline. The \textit{unit} function loads the work items into the containers, and the \textit{map} and \textit{join} functions acts as a controller that calls a worker function (e.g. \( f \)) to process the work item to produce a new container.

By separating the container, controller, and workers, the monadic design offers the following advantages, in addition to the benefits of the functional programming: 1) modularity: pipelines can be constructed from containers, controllers, and workers; 2) flexibility: a pipeline can be changed easily by replacing its components; and 3) isolation: impure functions whose output depending on hidden states can be encapsulated in monads and safely incorporated into a pure functional programming language.

\section{3. Resource-Oriented Network Model}

Instead of replacing IaaS by RaaS, we take an incremental approach and insert RaaS as a new service layer between IaaS and the operating systems, as illustrated in Figure 3, where each layer is depicted as a rectangle. In this new cloud stack, RaaS exposes fine-grained resources as RON (Resource-Oriented Network) and EM (Elastic Machine). A RON defines the fine-grained resources of an EM and provides REST API to control and monitor the states and connections of these resources. An EM is also treated as a REST resource and can be part of a RON. At a higher level, a RON can be regarded as a dynamic graph of REST resources connected by hyperlinks, such that a REST client can reach all the REST resources from a distinct REST resource called \textit{entry point}. With this abstraction, a RaaS cloud can be regarded as a dynamic graph of RONs connected by hyperlinks to the entry points.

Inspired by SDN that decouples a network into control plane and the data plane, RON decouples RaaS into three planes as shown in Figure 4:

- \textbf{Data}: It consists of heterogeneous cloud hardware and software components, such as CPU, memory, storage, network, switches, routers, etc., that communicate in a variety of protocols.
- \textbf{Control}: It consists of connected REST resources implemented in any programming languages that expose uniform interfaces to clients and interact with the cloud resources through their APIs.
- \textbf{Representation}: It consists of hypertexts exchanged between clients and RON through protocols (e.g. HTTP) that represent the states and connections of the REST resources in the control plane.

The separation of control and data planes can be logical if the REST resources are part of the components. The control and representation planes constitute the REST API of a RON. A RON is designed to have the following properties:

- \textbf{Access Transparency}: The REST resources at the control plane provide uniform interfaces (e.g. HTTP operations \textit{GET}, \textit{PUT}, \textit{POST} and \textit{DELETE} with hypertext encoded in XML or JSON) to access heterogeneous cloud resources at the data plane.
- \textbf{Location Transparency}: The REST resources at the control plane are identified by URI, which is independent of the physical locations or network addresses (MAC or IP) of the cloud resources at the data plane.
- \textbf{Connection Transparency}: The topology of a RON can change at runtime in an unconstrained way and a client can use hypertext-driven navigation to discover the connections from an entry point to the RON.
- \textbf{Control Transparency}: The communications between the cloud resources at the data plane can be
determined without going through the control plane, and the communications between the REST resources at the control plane can be determined without going through the representation plane (i.e., the REST clients);

These properties lay the foundation to address the challenges listed in the Introduction section, namely Heterogeneity, Dynamism, Composability, Overhead, and Scalability.

In particular, the Overhead issue is addressed by Control Transparency. Because an EM and its fine-grained resources are in the data plane, the EM does not need to go through the control plane in order to access those resources. As a consequence, the overhead of RON can be minimized.

The Composability issue is addressed by Connection Transparency. When a CPU core is connected to some remote memory board at the data plane, the physical connection results in a logical composition between the CPU resource and the memory resource at the control plane. In the representation plane, the logical composition is represented by a hyperlink to the memory resource included in the hypertext retrieved from the CPU resource. Such compositions between the REST resources can be formed automatically using the Categorial Link mechanism described in (Li 2014). The hyperlink can be annotated with attributes such as bandwidth, delay, and rental duration that are useful for the clients to decide what workload can be assigned to the CPU and memory.

The Heterogeneity issue is addressed based on Access Transparency and Location Transparency, which will be discussed in sections 4.1 and 4.2. The Dynamism issue is addressed by a combination of several techniques based on the RON model, which will be discussed in section 4.3. The Scalability issue is addressed by partitions of cloud resources into small RONs, which will be discussed in sections 4.1 and 4.2, and the monad service composition, which will be discussed in section 5.

To induce the above properties of RON, we use REST Chart (Li 2011), a Petri-Net based modeling framework, to describe the REST API for a RON. The REST Chart models a REST API as a Colored Petri Net where the places define the possible resource representations of the REST API and a transition binds a hyperlink template between two places to a network protocol. The interactions between a REST Client and a REST API are modeled by tokens moving in the Petri-Net defined by the REST Chart. A key distinction between REST Chart and the previous approaches is that REST Chart follows a bottom-up design to REST API. REST Chart only specifies the resource representations and interactions at the design time, and the clients can discover the resource identifications and connections at the runtime through hypertext-driven navigation. Consequently, a REST API can change its identifications and connections without breaking the clients.

To speed up RON development, we developed a Java toolkit RC-Java to automatically generate the representation and control planes of a RON from a REST Chart, such that a developer only needs to implement the code to connect the control plane with the data plane to complete the RON. The workflow of this tool is depicted in Figure 5, where the planes are separated by the dashed lines, and the solid arrows indicate automated processes, whereas the dashed arrows indicate manual processes. The tool loads a REST Chart and generates Java Message Classes using JAXB from the XML schemas and the JAX-RS compliant Java Resource Classes from the REST Chart. A developer then implements the Java methods in the Resource Classes to connect them with the data plane APIs. Finally, the developer uses the tool to compile, package, and deploy the artifacts into Apache Tomcat server.

4. RON ORGANIZATIONS

The described RON framework is used to design and implement REST APIs to dynamically control the fine-grained resources of containers based on Linux Control Groups (cgroups). Due to the large number of fine-grained resources in a datacenter, it is impossible to organize them into a single RON. Moreover, a single RON is also difficult to isolate the users and providers who have different logical views over the shared cloud resources.

For these reasons, we develop 4 types of RONs described below:

- Task RON (T-RON): A Task EM consists of a user’s container and it exposes a REST API to dynamically control the capacity of the Task EM.
- Job RON (J-RON): A Job EM consists of a group of related Task EMs which may be collocated or distributed, and it exposes a REST API to control the capacities of the Job EM and the Task EMs.
- Server RON (S-RON): A Server EM consists of aggregated or disaggregated hardware resources, and it exposes a REST API to control the capacity of the Server EM (notice that an aggregated server is not truly elastic, but a disaggregated server is).
• Cluster RON (C-RON): A Cluster EM consists of a group of Server EMs whose resources are shared as a pool, and it exposes a REST API to control the capacity of the Cluster EM and the Server EMs.

S-RON is modeled closely after cgroup hierarchy to provide full access to its functionalities, whereas the other RONs adopt different models from the cgroup hierarchy to hide the underlying cgroup variations and changes from the clients.

This RON organization has several advantages. First, it allows the EMs to autonomously manage and trade their cloud resources without involving the provider. Second, it allows a client to manage the fine-grained resources at different scales using a single RON. For example, a client can manage the resource pool of a container using a T-RON, and manage the resource pools of all the containers using the service composition at a J-RON. The same is true for the S-RON and C-RON. Third, it can provide crash fault-tolerance against C-RON crashes. The cost of composition on the direct links. Similarly, it can provide crash fault-tolerance by eliminating single point of failures. Since a J-RON is connected to all its T-RON through hyperlinks, and a client can discover these hyperlinks at runtime following hypertext-driven navigation, from which a client has two control paths to each T-RON: direct and indirect through J-RON. If the J-RON fails, the client can still control the resource pools of the J-RON by running a service composition on the direct links. Similarly, it can provide fault-tolerance against C-RON crashes. The cost of hyperlink discovery can be reduced by using page-based link-based differential caches (Zhou 2014) at proxies or clients that are fully compatible with HTTP.

Figure 6. Mapping S-RON and T-RON to cgroups.

4.1 Server RON and Task RON

Each physical server can have one S-RON and several T-RONs. Figure 6 illustrates how S-RON and T-RON are mapped to the cgroups in a server A, where the cgroup hierarchy is depicted as nested boxes.

S-RON provides control to the fine-grained resources of Linux processes on a server with the REST API described in Table I. For example, the REST API of S-RON can be used to control the CPU shares, memory limit, and I/O rate of the processes in container C1 with the following URIs:

.../cpu/docker/C1/cpu.shares;
.../memory/docker/C1/memory.limit_in_bytes;
.../blkio/docker/C1/blkio.throttle.read_iops_device.

Table I. Namespaces of Server RON

<table>
<thead>
<tr>
<th>URI template</th>
<th>method</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>/servers/{sid}/em/ron</td>
<td>GET</td>
<td>retrieve cgroups</td>
</tr>
<tr>
<td>.../{cgroup}</td>
<td>GET</td>
<td>retrieve a cgroup</td>
</tr>
<tr>
<td>.../{cgroup}/</td>
<td>PUT</td>
<td>update all cgroup parameters</td>
</tr>
<tr>
<td>.../{cgroup}/{parameter}</td>
<td>GET</td>
<td>retrieve a cgroup parameter</td>
</tr>
<tr>
<td>.../{cgroup}/{parameter}</td>
<td>PUT</td>
<td>update a cgroup parameter</td>
</tr>
</tbody>
</table>

Table II. Namespaces of Task RON

<table>
<thead>
<tr>
<th>URI template</th>
<th>method</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>/jobs/{jid}/tasks/{tid}/em/ron</td>
<td>GET</td>
<td>retrieve root cgroup</td>
</tr>
<tr>
<td>.../{control}</td>
<td>GET</td>
<td>retrieve all control parameters</td>
</tr>
<tr>
<td>.../{control}/{parameter}</td>
<td>PUT</td>
<td>update all control parameters</td>
</tr>
<tr>
<td>.../{control}/{parameter}</td>
<td>GET</td>
<td>retrieve a control parameter</td>
</tr>
<tr>
<td>.../{control}/{parameter}</td>
<td>PUT</td>
<td>update a control parameter</td>
</tr>
</tbody>
</table>

Unlike process-centric S-RON, T-RON provides a task-centric view on resource management, where the fine-grained resources (CPU, memory, IO, etc.) of a task is represented as a network REST resources. The REST API of T-RON is summarized in Table II, where {control} represents a type of fine-grained resource allocated to the task.

A T-RON automatically discovers all the tasks from the cgroup hierarchy on the server and builds a control model that maps the URIs in Table II to the cgroup nodes as illustrated in Figure 6. Upon receiving a request, its REST API calls the appropriate Linux cgroup API based on the control model. For example, to access the CPU shares of task 2 of job 1, it can use URI: /jobs/1/tasks/2/cpu/cpu.shares, which is mapped to a cgroup node by the control model. As the control model may differ on different Linux systems, a T-RON hides the cgroup hierarchies from the clients.

4.2 Job RON and Cluster RON

An architecture that layers J-RON on T-RON and C-RON on S-RON are illustrated in Figure 7, where 3 servers (A through D) run two jobs (1 and 2) and 1 cluster that consists of servers B and C. J-RON 1 for job 1 consists of task 11 on server B and task 12 on server C, while J-RON 2 for job 2 consists of task 21 on server B and task 22 on server C. For security reasons, J-RON 1 can only access containers 11 and 12, while J-RON 2 can only access containers 21 and 22. However, C-RON 1 is permitted to access the S-RONs on server B and server C with the administrator privilege on those servers.
Services Transactions on Cloud Computing (ISSN 2326-7550) Vol. 3, No. 4, October-December 2015

Figure 7. Mapping J-RON to T-RON and C-RON to S-RON.

Table III. Namespaces of Job RON

<table>
<thead>
<tr>
<th>URI template</th>
<th>method</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>/jobs/{jid}/em/ron</td>
<td>GET</td>
<td>retrieve hyperlinks to task RONs and aggregated controls</td>
</tr>
<tr>
<td>Table II row 1, column 1</td>
<td>GET</td>
<td>Table II row 1, column 3</td>
</tr>
<tr>
<td>.../[control]</td>
<td>GET</td>
<td>retrieve a particular control</td>
</tr>
<tr>
<td>.../[control]/[parameter]</td>
<td>GET</td>
<td>retrieve a control parameter</td>
</tr>
</tbody>
</table>

Table IV. Namespaces of Cluster RON

<table>
<thead>
<tr>
<th>URI template</th>
<th>method</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>/clusters/{cid}/em/ron</td>
<td>GET</td>
<td>retrieve hyperlinks to server RONs and aggregated controls</td>
</tr>
<tr>
<td>Table I row 1, column 1</td>
<td>GET</td>
<td>Table I row 1, column 3</td>
</tr>
<tr>
<td>.../[control]</td>
<td>GET</td>
<td>retrieve a particular control</td>
</tr>
<tr>
<td>.../[control]/[parameter]</td>
<td>GET</td>
<td>retrieve a control parameter</td>
</tr>
</tbody>
</table>

The REST APIs of J-RON and C-RON are summarized in Table III and Table IV respectively, where the two REST APIs are very similar to each other, and they are also similar to the T-RON REST API in Table II, except the entry URIs. This similarity gives the REST APIs a familiar look-and-feel for users on one hand, and makes the REST APIs easy to create and maintain for developers on the other hand. These benefits are the results of adopting a common RON model for REST API design.

Each cgroup parameter $p$ identified by URI template $.../[parameter]$ at J-RON (C-RON) aggregates the values of the corresponding parameters in T-RON (S-RON) into a range $p=(\text{max}, \text{min})$, such that the clients knows the range of possible resources in this J-RON (C-RON). A J-RON (C-RON) can also run workflows to control multiple fine-grained resources, e.g., simultaneously increase the CPU usage shares of containers 11 and 12.

4.3 RON DYNAMICS

We employ three techniques to minimize the impact of RON changes to the REST clients: 1) dynamic mapping; 2) hypertext-driven navigation; and 3) event-driven synchronization.

Dynamic mapping can hide the underlying cgroup changes from RON, by using a control model that maps the REST resources in a RON to a given cgroup hierarchy. When the cgroup hierarchy changes, we can replace the mappings without changing the RON. This idea is illustrated in Figure 8, where a hierarchical RON for a container at the middle chooses one of the two mappings: one for unified cgroup hierarchy and one for multi cgroup hierarchy. A unified cgroup hierarchy is shown on the upper left and the mappings from the RON to the cgroup are shown by the dashed arrows from the RON nodes to the left cgroup nodes. A multi cgroup hierarchy is shown on the upper right and the mappings from the RON to the cgroup are shown by the dashed arrows from the RON nodes to the right cgroup nodes.

Despite dynamic mappings, changes to RON connections are sometimes inevitable and a client needs to use hypertext-driven navigation to invoke the same REST services despite these changes. An example of this approach is illustrated in Figure 9, which shows two different Server RONs at multi cgroup hierarchy (a) and unified cgroup hierarchy (b).

The REST interfaces and hyperlinks of the nodes in these two RONs remain the same, while the connections between some nodes are changed. For example, the path to node $A$ in the first RON is $\text{entry} \rightarrow \text{cgroup} \rightarrow \text{cpu} \rightarrow A$, while it becomes $\text{entry} \rightarrow \text{root} \rightarrow A$ in the second RON. By using hypertext-driven navigation, a client that can find $A$ in the first RON can start from the entry URI to the second RON. A more detailed example can be found in (Li 2016).

Hypertext-driven navigation is flexible but has certain cost associated with it. Since each discovery requires two messages, traversing a discovery path of $N$ resources requires $2N$ messages. To address this issue, event-driven synchronization is used to proactively push changes at the Task RONs to a Job RON and from the Server RONs to a
Cluster RON. Since each event notification requires one message, detecting changes in a path of N resources requires only $N-1$ (event messages) + 2 (discovery messages) = $N+1$ messages. In the reverse direction, the data plane can notify changes to the control and representation planes based on the event architectures and techniques discussed in (Li 2010a, 2010b).

![a Server RON based on multi cgroup hierarchy](image1)

**Figure 9: RON Changes with cgroup**

### 4.4 RON Security Architecture

A RON permits two kinds of actions as defined below:
- invoke: invoke the REST API as a client; and
- manage: control the lifecycle of the RON, including deploy, start, stop, pause, and resume.

Different subjects in the cloud are granted with different action permissions to a RON, and these permissions are summarized in Table V.

<table>
<thead>
<tr>
<th>Access</th>
<th>J-RON</th>
<th>T-RON</th>
<th>S-RON</th>
<th>C-RON</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>invoke</td>
<td>invoke</td>
<td>invoke</td>
<td>invoke</td>
</tr>
<tr>
<td>Provider</td>
<td>manage</td>
<td>manage</td>
<td>invoke</td>
<td>manage</td>
</tr>
</tbody>
</table>

**Table V. Access Policy for RON**

![a Server RON based on unified cgroup hierarchy](image2)

![Figure 10. RBAC for T-RON and J-RON](image3)

This policy permits the cloud users to use the J-RON and T-RON REST APIs, but not to manage them. The management action is left to the cloud providers. The S-RON and C-RON are infrastructure resources which are not accessible to anyone but the cloud provider. Under this general policy, we ensure that each user can only access the J-RON associated with his jobs and T-RON associated with his tasks. If user $u1$ owns job $j1$ and its tasks, and user $u2$ owns job $j2$ and its tasks, then each user can only access his own namespaces shown in Table VI, according to the namespace templates in Table II and Table III.

There are different access control mechanisms to enforce the namespace policy. Since Tomcat supports RBAC (Role-Based Access Control) model (Sandhu 2000), we adopt this model for access control. The basic idea of RBAC model is to decide which users (cloud users and cloud providers) can access what namespaces based on the roles of the user. More formally, given a set of users $U$, a set of roles $R$, and a set of namespaces $N$, a RBAC model determines the mapping $U \rightarrow N$ by composing the two independent mappings: 1) $U \rightarrow R$ maps the users ($U$) to the roles ($R$); and 2) $R \rightarrow N$ maps the roles ($R$) to the namespaces ($N$).

**Figure 10 illustrates an architecture where a container factory dynamically configures the RBAC databases used to authorize users. The RBAC databases are set up by the solid arrows as follows:**

1. The factory accepts commands from user $u$ to create task containers for job $j$ from the task images.
2. The factory authenticates $u$, creates a role $r(u,j)$ for $u$, and saves $u \rightarrow r(u,j)$ in database $U \rightarrow R$.
3. The factory asks Docker to launch the containers $C(j)$ for job $j$.
4. The factory assigns the namespaces $N(C(j))$ to $r(u,j)$ and save $r(u,j) \rightarrow N(C(j))$ in the $R \rightarrow N$ database (see Table VI for examples of $N(C(j))$).

When user $u$ adds a new job $j$, a new role $r(u,j)$ is created for job $j$ and its tasks. When a job $j$ is removed, the corresponding role $r(u,j)$ is also removed. However, when user $u$ adds or removes tasks for job $j$, no change to roles is needed and we only need to change the namespaces $N(C(j))$ associated with job $j$. When a task container migrates to a new server and assumes a new identity, it is treated as removing the old task and adding a new one.

RBAC databases are accessed by the J-RON and T-RON to control client access as shown by the dashed arrows in Figure 10:
5. RON Service Composition

RON REST APIs normalize the access to individual cloud resources through a set of primitive HTTP operations, but they do not provide a mechanism for a client to concurrently control a large number of RON resources with a few operations. Programming distributed REST resources based on primitive HTTP operations is quite difficult for developers and a high-level service composition language and runtime engine is needed. For this purpose, we choose functional programming with monad, because it has the following advantages over the alternative approaches:

- **Concurrency**: a functional program can be executed concurrently as it either has no side-effects or can isolate the side-effects with monads (Abelson 1996).

- **Efficiency**: a functional program can be executed efficiently as it removes the data bottleneck of imperative programming (Backus 1977).

- **Dynamism**: RON resource identifications and operations unknown at design time can be dynamically computed at runtime by monads and functions according to control criteria, such as task priority, affinity, deadline, network conditions and server capacities.

- **Composability**: universal compositions of functions and monads can be defined and carried out by a few generic algebraic rules (Wadler 1992).

- **Transmission**: functional programs tend to be small in size and therefore easy to transmit.

In our composition framework, REST resources are treated as distributed objects with local states (Abelson 1996) that can be accessed by the GET, PUT, POST, and DELETE operations. Since the output of these methods depends on the resource states, they cannot be modeled as pure functions whose output only depends on the input. For this reason, our composition framework uses monads to encapsulate dynamic resource identifications, operations and controls on the REST resources. A collection of monads arranged in a certain way is called a composition program, and it can be evaluated at runtime to a workflow that actually interacts with the REST resources. The overall architecture of our framework is depicted in Figure 11, where the Composition Engine accepts a composition program \( p \) through a REST API, evaluates \( p \) to a workflow \( a \), and executes \( a \) to access the REST resources through the REST clients implemented in certain programming language, e.g. Java.

More formally, the framework can be represented by a 6-tuple \( (T, P, W, A, \text{eval}, \text{exec}) \):

- **\( T \)**: a set of primitive and composite types defined by a type system.
- **\( P \)**: a set of programs whose types are defined by \( T \).
- **\( W \)**: the REST resources, proxies, gateways, and caches that programs in \( P \) can access.
- **\( A \)**: a set of workflows that access \( W \) through network protocols, such as HTTP 1.1.
- **\( \text{eval} : P \rightarrow A \)**: the procedure that evaluates a program in \( P \) to a workflow in \( A \).
- **\( \text{exec} : A \rightarrow T \)**: the procedure that executes a workflow in \( A \) to produce a result in \( T \), which can be success or faults.

![Figure 11. Architecture of Composition Framework](image-url)

The separation of procedures \( \text{eval} \) and \( \text{exec} \) allows the framework to change the execution environment, for example from Java to Python, without changing the evaluation procedure or the REST API.

A program in \( P \) is composed from three kinds of primitive expressions:

- **value**: an expression that evaluates to itself, such that \( \text{eval}(\text{value}) = \text{value} \);

- **function**: an expression that evaluates to a mapping between types, such that \( \text{eval}(\text{function}) = t_i \rightarrow t_j \), where \( t_i, t_j \in T \);

- **monad**: an expression that evaluates to a workflow in \( A \), such that \( \text{eval}(\text{monad}) = a \in A \).

For example, expression \( v = \text{http://h/a/b} \) represents a value whose type is URI.

Expression \( u(x,y) = \text{http://h/[x]/[y]} \) represents a URI template as a function of type \( \text{string} \rightarrow \text{string} \rightarrow \text{URI} \).

Expression \( u(a)(b) = u(a,b) \) represents a program composed from function \( u \) and strings \( a \) and \( b \), and we have \( \text{eval}(u(a,b)) = v \), the URI expression defined above.

The composition of two primitive expressions \( x \) and \( y \) is denoted by a polymorphic binary operator \( x \Delta y \), read as \( x \) meet \( y \). Table VII enumerates compositions permitted by the core language.

To allow order-free compositions, operator \( \Delta \) must be associative such that \( (x \Delta y) \Delta z = x \Delta (y \Delta z) \) for all the cases. If \( f \) is a function, \( u \) and \( v \) are values, then the associativity requires that \( (f \Delta u) \Delta v = f \Delta (u \Delta v) \). The associativity always
holds because \( f(u,v) \) and \( f([u,v]) \) are semantically equivalent expressions:

1. \( (f\Delta u)\Delta v = f(u)\Delta (v) = f(u,v) \);
2. \( f\Delta (u\Delta v) = f\Delta [u,v] = f([u,v]) = f(u,v) \).

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( x\Delta y )</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>value</td>
<td>([x,y])</td>
<td>list ( x ) and ( y )</td>
</tr>
<tr>
<td>value</td>
<td>function</td>
<td>( y(x) )</td>
<td>function application</td>
</tr>
<tr>
<td>function</td>
<td>value</td>
<td>( x(y) )</td>
<td>function application</td>
</tr>
<tr>
<td>function</td>
<td>function</td>
<td>( x\Delta y )</td>
<td>function composition</td>
</tr>
<tr>
<td>monad</td>
<td>function</td>
<td>( \text{map}(y)(x) )</td>
<td>apply function ( y ) to monad ( x )</td>
</tr>
<tr>
<td>function</td>
<td>monad</td>
<td>( \text{map}(x)(y) )</td>
<td>apply function ( x ) to monad ( y )</td>
</tr>
<tr>
<td>monad</td>
<td>monad</td>
<td>( \text{join}(x,y) )</td>
<td>join monads ( x ) and ( y )</td>
</tr>
<tr>
<td>value</td>
<td>monad</td>
<td>( \text{join}([\text{unit}(x),y]) )</td>
<td>join the monad from ( x ) with ( y )</td>
</tr>
<tr>
<td>monad</td>
<td>value</td>
<td>( \text{join}([x,\text{unit}(y)]) )</td>
<td>join ( x ) with the monad from ( y )</td>
</tr>
</tbody>
</table>

To allow order-free compositions, operator \( \Delta \) must be associative such that \((x\Delta y)\Delta z = x\Delta (y\Delta z)\) for all the cases. If \( f \) is a function, \( u \) and \( v \) are values, then the associativity requires that \((f\Delta u)\Delta v = f\Delta (u\Delta v)\). The associativity always holds because \( f(u,v) \) and \( f([u,v]) \) are semantically equivalent expressions:

3. \( (f\Delta u)\Delta v = f(u)\Delta (v) = f(u,v) \);
4. \( f\Delta (u\Delta v) = f\Delta [u,v] = f([u,v]) = f(u,v) \).

For other cases, the associativity also holds from the Monad rules. Notice that the meet operator is commutative because a list is not commutative.

The polymorphic meet operator between the primitives gives us the freedom to compose lists of primitives using additional polymorphic algebraic operators. A primitive \( x \) can be composed by dot-product * with a list of primitives \( y=[y_1,...,y_n] \), denoted by \( x^*y \), to produce a new list monad as follows:

\[
x^*y = [x^*y_1, ..., x^*y_n].
\]

This definition is recursive so that it works for nested lists. These recursions will eventually reduce to the meet operator on the primitives in Table VII.

Two list monads \( x=[x_1,...,x_m] \) and \( y=[y_1,...,y_n] \) of length \( n \) can be composed by \( x^*y \) to produce a new list monad of length \( n \) as follows:

\[
x^*y = [x_1^*y_1, ..., x_m^*y_n].
\]

Two list monads \( x=[x_1,...,x_m] \) and \( y=[y_1,...,y_n] \) can also be composed by cross-product \( x\times y \) to produce a list monad of mn items as follows:

\[
xx = [x_1^*y_1, ..., x_1^*y_n, ..., x_m^*y_1, ..., x_m^*y_n].
\]

Since the meet operator is associative, the dot-product and cross-product operators are also associative:

1. \( (x^*y)^*z = x^*(y^*z) \);
2. \( (x\times y)(z) = x\times(y\times z) \).

It is evident that \( x^*y=y^*x \) and \( x\times y=y\times x \) also hold for cases where the meet operators are commutative. The precedence of these operators is: \( <> \Delta > * > x \).

The four polymorphic operators, \( \Delta, *, \times \) and \( \circ \), allow us to create concise and dynamic REST service compositions based on three types of monads: 1) URI monad: a list of URI templates \([u(...)]\) evaluated to absolute URIs by functions at runtime; 2) operation monad: a list of operation templates \([o(...)]\) evaluated to HTTP operations by functions at runtime; and 3) control monad: a tree of parallel \( \text{par}[...] \) and sequential \( \text{seq}[...] \) control templates evaluated to workflows by functions at runtime.

Two examples are used to illustrate the composition programs and evaluation process, while the formal definitions of these monads and more examples can be found in (Li, L. 2015).

**Example 1:** Suppose we want to decrease the CPU and memory usage of some idle tasks and give them back to the tasks whose workloads are high, we can write a program as follows:

\[
\begin{align*}
  p1 &= \text{seq}[] \left[ \text{\texttt{[u[jid,tid]]}} -\text{\texttt{f1}} \cdot \text{\texttt{par[\texttt{o[\texttt{cpu=10,mem=10}, \texttt{cpu=20,mem=20]}]}]} \right] \\
  p2 &= \text{seq}[] \left[ \text{\texttt{[u[jid,tid]]}} -\text{\texttt{f2}} \cdot \text{\texttt{par[\texttt{o[\texttt{cpu=30,mem=30}, \texttt{cpu=40,mem=40]}]}]} \right]
\end{align*}
\]

Here functions \( f1 \) and \( f2 \) find the tasks, and functions \( g1 \) and \( g2 \) determine the proper capacity limits for the tasks, as shown in the previous examples. The program evaluates to:

\[
\begin{align*}
  f1 &= [[\texttt{[cpu=10,mem=10]}], [\texttt{cpu=20,mem=20}]], \\
  g1 &= [[\texttt{cpu=10,mem=10}]], [\texttt{cpu=20,mem=20}]] \\
  g2 &= [[\texttt{cpu=30,mem=30}]], [\texttt{cpu=40,mem=40}]]
\end{align*}
\]

Then:

\[
\begin{align*}
  \text{eval}(p1) &= \text{seq}[] \\
  &\text{\texttt{par[ \texttt{o[\texttt{...11,put,cpu=10,mem=10}, \texttt{cpu=20,mem=20}]}]}} \\
  &\text{\texttt{par[ \texttt{o[\texttt{...12,put,cpu=20,mem=20}]}]}}
\end{align*}
\]

This program will execute the first groups of operations in parallel to decrease the capacities of the idle tasks, and after they are successful, execute the second group of operations in parallel to increase the capacities of the busy tasks.

**Example 2:** If we want to move two sets of busy tasks in parallel to new machines and reset their CPU and memory capacities in the sequential order, we can use a program like this:

\[
\begin{align*}
  p2 &= \text{par}[] \\
  &\text{\texttt{h-move[\texttt{par[u[1]}\times\texttt{seq[\texttt{o[\texttt{cpu=1,put,g1}], \texttt{o[\texttt{cpu=2,put,g2}]}]}]}]} \\
  &\text{\texttt{h-move[\texttt{par[u[2]}\times\texttt{seq[\texttt{o[\texttt{cpu=1,put,g3}], \texttt{o[\texttt{cpu=2,put,g4}]}]}]}]}}
\end{align*}
\]

This program consists of the following functions:
• move: operation \( o(u, move, target) \rightarrow \text{hypertext} \) moves a task to the target machine;
• \( h \): function that finds control hyperlinks from a hypertext;
• \( f1, f2 \): functions that find the busy tasks;
• \( g1, g2, g3, g4 \): functions that determine the task capacities on the target machine;
• \( u1 \): function \( u1: u \rightarrow \{u/\text{cpu} \} \) expands URI \( u \) with path for CPU; and
• \( u2 \): function \( u2: u \rightarrow \{u/\text{mem} \} \) expands URI \( u \) with path for memory.

Function \( h \) extracts the control URI from the hypertext, and if it is not there, follows the hyperlinks to discover the control URI. Suppose operation \( o(t1, move, z2) \) returns this hypertext:

\[
200 \text{ OK HTTP 1.1} \\
\text{location: http://z2/jobs/j1/tasks/t1} \\
\text{control: http://z2/jobs/j1/tasks/t1/control}
\]

Applying \( h \) to the hypertext, we have:

\[
h \circ o(t1, move, z2) = \text{http://z2/jobs/j1/tasks/t1/control}. \\
\]

The above program evaluates the following control monad, after the successful move operations:

\[
\text{if } f1=\{[j1, t1], [j2, t2]\}, \quad f2=\{[j3, t3], [j4, t4]\}, \\
g1=\{\text{cpu}=10, \text{mem}=20\}, \quad g2=\{\text{mem}=40\}, \\
g3=\{\text{cpu}=30\}, \quad g4=\{\text{mem}=40\},
\]

then

\[
\text{eval}(p2) = \text{par} [ \\
\text{seq} [ \\
\text{o}(\ldots; t1/\text{control/cpu, put,}\{\text{cpu}=10\}), \\
\text{o}(\ldots; t1/\text{control/mem, put,}\{\text{mem}=20\}) ] \\
\text{seq} [ \\
\text{o}(\ldots; t2/\text{control/cpu, put,}\{\text{cpu}=10\}), \\
\text{o}(\ldots; t2/\text{control/mem, put,}\{\text{mem}=20\}) ] \\
\text{seq} [ \\
\text{o}(\ldots; t3/\text{control/cpu, put,}\{\text{cpu}=30\}), \\
\text{o}(\ldots; t3/\text{control/mem, put,}\{\text{mem}=40\}) ] \\
\text{seq} [ \\
\text{o}(\ldots; t4/\text{control/cpu, put,}\{\text{cpu}=30\}), \\
\text{o}(\ldots; t4/\text{control/mem, put,}\{\text{mem}=40\}) ] \\
].
\]

The program contains 4 sequential groups that execute in parallel.

The Composition Engine exposes a REST API to accept composition programs encoded in XML. A client can submit (POST) a XML program to the REST API, which will call the \text{eval} procedure and, if successful, return a distinct hyperlink to control the workflow execution. The client can run the workflow many times by sending POST messages to the control resource, which will call the exec procedure and return execution status in the response.

6. Prototype System and Experiments

The described RON REST APIs and the Monad service composition engine have been implemented in Java with the help of the RC-Java tool described in Section III. The experimental environment for both prototypes involved a Linux server machine and a Window 7 client machine connected by LAN as shown in Figure 12. The browser at the client machine was used to issue individual REST API requests to test the effect of the REST API to control and manage RON running on the server machine, while the JMeter is used to run the performance tests.

\[\text{Figure 12. Test environment.}\]

5.1 RON Prototype and Experiments

The RON REST APIs were deployed on Tomcat servers, and the prototype system was integrated with Docker running on Linux servers.

Figure 13 shows the effect of increasing the CPU cores from 1 to 3 for a container using the REST API, as observed by cAdvisor, an open-source container monitoring tool (cAdvisor 2015). The blue curve indicates that the container uses core 0 at the beginning. After we used the REST API to set the cores to 3, the container immediately begins to use three cores 0, 1 and 2, while the load on core 0 was decreased.

Figure 14 illustrates the effect of memory limit change for a container observed by cAdvisor. At beginning the memory limit of the container was kept at the maximum, and after we used the REST API to set it to 1.9GB, the total memory (blue curve) immediately dropped down to the new limit.

To test the performance of the REST API, the RON REST APIs and the Tomcat that hosted them were packaged into a Docker image and ran as a Docker container with all the default configurations. We used JMeter to simulate 5 and 10 concurrent REST clients, while each REST client sent 20 different GET and PUT requests to the REST APIs. The average client response time (millisecond) and the total RON based REST API system processing throughput (#request/s) were calculated. In each session, we also recorded the REST API processing time (millisecond) for each request at the server side. To test the task workload impact on the REST API performance, we ran 7, 22, and 32 Docker containers to simulate the task loads on the same Linux server, whereas the RON based REST API server was running inside the separate dedicated container.
Figure 15 shows the performance of 5 concurrent clients (time on the left y-axis and throughput on the right y-axis) when the number of active task containers increased (on x-axis). The results show that the average client response time to the RON container increased linearly from 56ms to 138ms and 178ms for 7, 22 and 32 task containers respectively. The gap between the average client response time and server time is 5ms, which accounts for the average network latency and Tomcat processing time.

Figure 13. Effect of CPU cores set by REST API (not all cores are in use).

Three REST service composition programs in cloud computing were used to test the performance of the workflow engine. The REST API runs inside the Tomcat on a Linux server and it accepts REST composition programs that adjust the CPU, memory, and I/O resources used by Docker containers (Docker 2015), where effects of resource management by these operations on the running containers were observed and monitored by cAdvisor (cAdvisor 2015).

Figure 14. Effect of memory limit set by REST API.

Figure 16 shows performance for 10 concurrent clients that follows the same trends as the 5-client case, i.e. the average client response time increased from 94ms to 216ms and 277ms respectively, for 7, 22, and 32 task containers. The gap between the average client response time and server time is 14ms, which accounts for average network latency and Tomcat processing time.

In this experiment, as the number of concurrent client increased from 5 to 10, the overall RON based REST API server system throughput increased by 31% on the average, while the average client response time increased by 58% and the average server processing time increased by 53%.

These experiments showed that the average client and server response time increased linearly in most cases with the number of concurrent clients, and it increased the task workloads under the default configurations for Tomcat and Linux. We believe that the performance can be further improved with tuning and optimization of the Linux and Tomcat configurations as well as REST API implementations.

5.2 COMPOSITION PROTOTYPE AND EXPERIMENTS
parameters for them, and then restart the containers in parallel.

The first experiment tested the time of the eval procedure, which converts XML programs to Java workflow objects. Each client in a test session repeated 6 requests 10 times to submit and delete the above 3 programs. As the result, the total requests for 1, 5, and 10 concurrent clients were 60, 300, and 600 respectively. The test results are depicted in Figure 17, Figure 18, and Figure 19.

The second set of experiments tested the time of the exec procedure, which involves computation inside the Java engine as well as external calls to the composed REST resources in the same machine. Each client in a test session repeated 3 requests to run the 3 workflows for 10 times. As the result, the total requests for 1, 5, and 10 clients were 30, 150, and 300 respectively. The test results are shown in Figure 20, Figure 21, and Figure 22.

In these tests, the client response time and server processing time decreased and the throughputs increased, as the number of concurrent clients increased. This could be due to the memory caches that save frequently used Java objects in memory as more requests repeat themselves in the experiments. Since the server has 12 physical CPU cores, increasing the number of concurrent clients did not slow down the server processing time as they were not overloaded.

In these tests, the client response time and server processing time increased linearly with the number of concurrent clients, except for workflow update_multiple, which only contains parallel operations that are implemented by Java Threads. With 1 client, the average client and server time for update_multiple was 15ms, and

![Figure 17. Creation time for update_cpuset](image1)

![Figure 18. Creation time for update_multiple](image2)

![Figure 19. Creation time for update_usehierarchy](image3)

![Figure 20. Execution time for update_cpuset](image4)

![Figure 21. Execution time for update_multiple](image5)

![Figure 22. Execution time of update_usehierarchy](image6)
7ms respectively. For update_cpuset, it was 287ms and 279ms, and for update_usehierarchy, it was 2401ms and 2391ms. The difference is partly due to the structural differences between the workflows and partly due to our current implementation strategy that uses Java threads for parallel operations. This also explains the execution time differences between the workflows as the number of concurrent client increases.

The overall test results indicate that the proposed approach is feasible and there is room to fine-tune and improve the implementations.

7. CONCLUSIONS
The main contributions of this paper are summarized below:

- A REST service framework based on Resource-Oriented Network (RON) for RaaS Cloud that decouples the representation plane, control plane, and data plane for efficient resource management.
- The RON based REST APIs to normalize the access to heterogeneous fine-grained resources, including CPU, memory and storage, at the levels of tasks, jobs, containers, servers, and clusters; and it has been implemented and tested with Docker containers.
- A dynamic RBAC mechanism to control the access to REST API namespaces where the roles and permissions are created when the containers are launched to enhance the security.
- A lightweight REST service composition framework to dynamically construct REST workflows using monads at runtime to concurrently control a large number of REST resources with a few operations.

The monadic composition language currently does not support more advanced features, such as conditions and exceptions, as our use cases do not require them. We anticipate such features will become necessary as we apply the framework to more complicated situations.

For future work, we plan to expand the functionalities of RON and integrate it with automated agents for resource allocation, scheduling, scaling, and trading into elastic machines in server-based and disaggregated RaaS clouds.

8. ACKNOWLEDGMENT
The authors would like to thank Tony Tang for implementing and testing the prototypes described in this paper.

9. REFERENCES


**Authors**

Dr. Li Li received his Ph.D. in computer sciences from University of Alabama at Birmingham, USA in 1995, and M.S. in computational linguistics from Huazhong University of Sciences and Technology, China in 1987. He joined Huawei Shannon IT Lab in 2012 and his current research interest includes web services, cloud computing and software-defined networking. He has published over 70 conference and journal papers and 1 book on Artificial Intelligence. He currently holds 12 US patents. Dr. Li is a member of IEEE and ACM, and he was the editor of 2 ISO/ECMA CSTA III standards and made significant contributions to W3C WS-RA standard suite.

Dr. Wu Chou currently is VP and Chief Technology Officer (CTO) of Network and Enterprise Communications product line of Huawei. He graduated from Stanford University with four advanced degrees in science and engineering. He joined AT&T Bell Labs after obtaining his Ph.D. degree in electrical engineering, and he continued his professional career from AT&T Bell Labs to Lucent Bell Labs and Avaya Labs before joining Huawei. He published over 200 journal and conference papers, and holds over 40 US and international patents with many additional patent applications pending. Dr. Chou is an IEEE Fellow. He served as editor and area expert for multiple international standards at W3C, ECMA, ISO, ETSI, etc. He was an editor for more than 11 years. He also worked as Senior Operations Research Analyst, Senior Manager and Director of Transportation Network Planning and Technologies for two Fortune 500 companies for 7 Years. He is a senior member of IEEE, he has been serving on the organizing committee for IEEE’s ICWS and SCS/CC Conferences, chaired sessions, presented several tutorials on SOA and Enterprise Architecture and their best practices and gave lectures at the Service University. He has served as adjunct professors in several US and Chinese universities since 1996. He obtained his Ph.D. in Electrical Engineering from Georgia Institute of Technology in 1992. He also held a M.S. (1987) and B.S. (1982) in Computer Science.
Call for Articles
International Journal of Services Computing

Mission
The International Journal of Services Computing (IJSC) aims to be a reputable resource providing leading technologies, development, ideas, and trends to an international readership of researchers and engineers in the field of Services Computing. To ensure quality, IJSC only considers extended versions of papers published at reputable international conferences such as IEEE ICWS.

From the technology foundation perspective, Services Computing covers the science and technology needed for bridging the gap between Business Services and IT Services, theory and development and deployment. All topics regarding Web-based services lifecycle study and management align with the theme of IJSC. Specially, we focus on: 1) Web-based services, featuring Web services modeling, development, publishing, discovery, composition, testing, adaptation, and delivery, and Web services technologies as well as standards; 2) services innovation lifecycle that includes enterprise modeling, business consulting, solution creation, services orchestration, services optimization, services management, services marketing, business process integration and management; 3) cloud services featuring modeling, developing, publishing, monitoring, managing, delivering XaaS (everything as a service) in the context of various types of cloud environments; and 4) mobile services featuring development, publication, discovery, orchestration, invocation, testing, delivery, and certification of mobile applications and services.

Topics
The International Journal of Services Computing (IJSC) covers state-of-the-art technologies and best practices of Services Computing, as well as emerging standards and research topics which would define the future of Services Computing. Topics of interest include, but are not limited to, the following:

- Services Engineering
- XaaS (everything as a service)
- Cloud Computing for Internet-based services
- Big Data services
- Internet of Things (IoT) services
- Pervasive and Mobile services
- Social Networks and Services
- Wearable services
- Web 2.0 and Web X.0 in Web services
- Service-Oriented Architecture (SOA)
- RESTful Web Services
- Service modeling and publishing
- Service discovery, composition, and recommendation
- Service operations, management, and governance
- Services validation and testing
- Service privacy, security, and trust
- Service deployment and evolution
- Semantic Web services
- Scientific workflows
- Business Process Integration and management
- Service applications and implementations
- Business intelligence, analytics and economics for Services
Mission

Big Data has become a valuable resource and mechanism for the practitioners and researchers to explore the value of data sets in all kinds of business scenarios and scientific investigations. New computing platforms such as cloud computing, mobile Internet, social network are driving the innovations of big data. From government initiative perspective, Obama Administration in United States launched "Big Data" initiative that announces $200 Million in new R&D investments on March 29, 2012. European Union also announced "Big Data at your service" on July 25, 2012. From industry perspective, IBM, SAP, Oracle, Google, Microsoft, Yahoo, and other leading software and internet service companies have also launched their own innovation initiatives around big data.

The International Journal of Big Data (IJBD) aims to provide the first Open Access publication channel for all authors working in the field of all aspects of Big Data. Big Data is a dynamic discipline. One of the objectives of IJBD is to promote research accomplishments and new directions. Therefore, IJBD welcomes special issues in any emerging areas of big data.

Topics

IJBD includes topics related to the advancements in the state of the art standards and practices of Big Data, as well as emerging research topics which are going to define the future of Big Data. Topics of interest to include, but are not limited to, the following:

**Big Data Models and Algorithms** (Foundational Models for Big Data, Algorithms and Programming Techniques for Big Data Processing, Big Data Analytics and Metrics, Representation Formats for Multimedia Big Data)

**Big Data Architectures** (Cloud Computing Techniques for Big Data, Big Data as a Service, Big Data Open Platforms, Big Data in Mobile and Pervasive Computing)

**Big Data Management** (Big Data Persistence and Preservation, Big Data Quality and Provenance Control, Management Issues of Social Network enabled Big Data)

**Big Data Protection, Integrity and Privacy** (Models and Languages for Big Data Protection, Privacy Preserving Big Data Analytics Big Data Encryption)

**Security Applications of Big Data** (Anomaly Detection in Very Large Scale Systems, Collaborative Threat Detection using Big Data Analytics)

**Big Data Search and Mining** (Algorithms and Systems for Big Data Search, Distributed, and Peer-to-peer Search, Machine learning based on Big Data, Visualization Analytics for Big Data)
