Application Migration Effort in the Cloud – The Case of Cloud Platforms

Stefan Kolb, Jörg Lenhard, and Guido Wirtz
Distributed Systems Group
University of Bamberg
Bamberg, Germany
{*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**

I. 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 [1], [2]. 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 [3]. 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 [4]–[8].

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 application is already built to run in the cloud[1]. Therefore, we primarily observe 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 [4], [9]. 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 [10]:

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?

The use case application, *Blinkist*, is a Ruby on Rails web application developed by Blinks Labs GmbH. The set of selected PaaS vendors includes IBM Bluemix, cloudControl, AWS Elastic Beanstalk, EngineYard, Heroku, OpenShift, 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 these differences by distinctive metrics. In this study, we target implementation portability [3], [10] 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 NoSQL technologies 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.

The remainder of the paper is structured as follows. In Section II, we describe our research methodology including

[1]See the twelve-factor methodology at http://12factor.net.

Fig. 1: Migration Evaluation Process [11]

details of the used application, vendor selection, deployment automation, and the measurement of deployment effort. Section III presents the results of our measurements and describes problems that occurred during migrations. In Section IV, we review related work. Section V discusses limitations and future work that can be derived from the results. Finally, Section VI summarizes the contributions of the paper.

II. Methodology

The goal of this study is to analyze cloud-based application migration 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 in the migration execution. 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 [4], [9]. 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.

The primary focus of this study is on the steps two and three, as the initial step can be largely assisted by our cloud brokering tool from [10] that covers the details of provider brokering and application requirements matching.

A. 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 650 books in its digital library. Blinkist has a user count of more than 300 000 registered customers worldwide. The product is created by a team of 18 full-time employees and is available for Android, iPhone, iPad, and web. We target

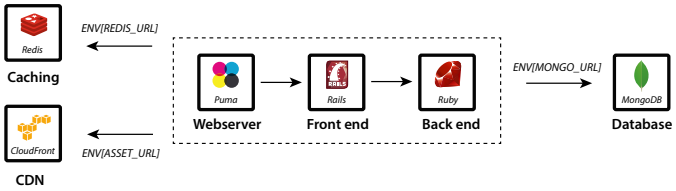

Fig. 2: Web Application Architecture

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 front end is a Rails 4 application with access to business logic written in Ruby. The application uses a MongoDB database for persistence. Moreover, caching 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 application part totals for about 60 000 *Lines of Code* (LOC).

B. 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 [10]. This tool allows us to filter from the multitude of available platform offerings based on ecosystem capabilities and requirements. 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 I, was based on a concluding relevance assessment.

For reasons of comparability, we selected equal instance configurations and geographical locations among the different vendors, grouped by virtualization technology. 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 I, there are substantial pricing differences between the vendors. Pricing is based on equivalent 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,

[2]The recent application version can be accessed at https://www.blinkist.com.

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

TABLE I: PaaS Vendors and Selected Configurations[4]

	Heroku	**cloudControl**	**Pivotal Web Services**	**Bluemix**	**OpenShift**		**Elastic Beanstalk**	**EngineYard**
Type	Proprietary	Proprietary	Open Source	Open Source	Open Source		Proprietary	Proprietary
Isolation	Container	Container	Container	Container	Container		Virtual Machine	Virtual Machine
RAM (instance)	512 MB	512 MB	512 MB	512 MB	512 MB		3.75 GB	3.75 GB
Geo location	Virgina, US	Dublin, IE	Virgina, US	Dallas, US	Virgina, US		Virgina, US	Virgina, US
Pricing	$ 0.05/h	$ 0.04/h	$ 0.015/h	$ 0.035/h	$ 0.025/h		$ 0.07/h	$ 0.123/h
Σ(2 instances/month)	$ 34.50	$ 54.14	$ 21.60	$ 24.15	$ 36	Σ (1 VM/month)	$ 50.40	$ 88.56

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 three 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 Heroku and cloudControl, caused by a higher free hour quota of Heroku, 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.

C. 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 most 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*[5], 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. 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 additional artifacts, like deployment manifests or configuration files, must be kept in a subfolder adjacent to the deployment scripts. 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 1 shows the `deploy` script for Heroku.

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

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

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

Listing 1: Deployment Script for Heroku

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 features[6].

A graphical overview of our deployment system Yard can be seen in Figure 3. For each container, a base image is used that only consists of a minimal Ubuntu installation, including Python and Ruby runtimes. From the base image, a deployment image is created that bootstraps the necessary provider tool dependencies. This is achieved by executing the `init` script of each provider module inside the base image, which results in

[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. Currency converted. Date: 04/17/2015.

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

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

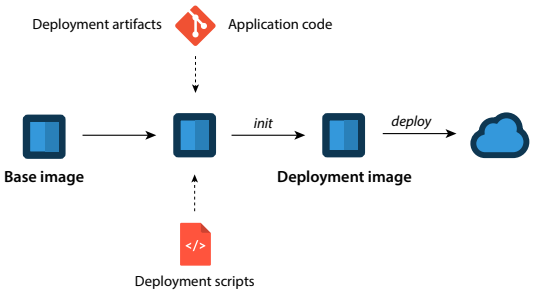

Fig. 3: Yard – Isolated Deployment with Docker

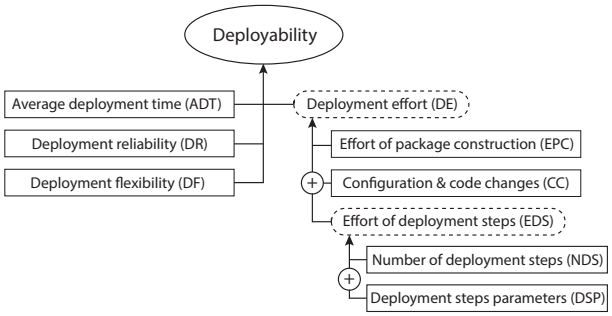

Fig. 4: Deployment Metrics Framework

a new container image. Additionally, the application code and the deployment artifacts are directly merged into a common repository. This is done to avoid additional bootstrapping before each deployment, which could influence the timing results of the deployment run. The resulting image can be used to deploy the code to different providers from every Docker-compatible environment via a console command.

D. 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 [12], 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 [13]. 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., man hours. Such aspects are hardly quantifiable without a larger empirical study and influenced by a lot of other factors, like the expertise of involved workers. However, it is possible to introduce human factors by adding weighting factors to the metrics computation, as for instance done in [28].

As cloud platforms are preconfigured and managed environments, there is no need to consider the installability of the environment itself, as in [12]. Instead, the focus lies on the deployability of an application to a cloud platform. Figure 4 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), *deployment steps parameters* (DSP), *configuration & code changes* (CC), and the *effort of package construction* (EPC). 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 [14]. The following paragraphs briefly introduce the metrics.

1) 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.

2) Deployment reliability (DR): DR 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^a_{succ}) with the total number of attempted deployments (N^a_{total}): $DR(a) = N^a_{succ}/N^a_{total}$. $DR(a)$ will be equal to one, if all deployments succeed.

3) Deployment flexibility (DF): [12] 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.

4) 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. II-C). 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 1 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`

5) Deployment steps parameters (DSP): 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 steps parameters, which counts all user input and command parameters that are necessary for deployment. The deployment script in Listing 1 uses six different variables and requires no additional command line parameters, resulting in $DSP(heroku) = 6$.

6) Effort of deployment steps (EDS): The two direct metrics NDS and DSP 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: $EDS(a) = NDS(a) + DSP(a)$. For our example, this amounts to $EDS(heroku) = 10$.

7) Configuration & code changes (CC): 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*[7]. Again, the construction of these files results in effort related to their size [12]. 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 difference 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 $file_i, ..., file_{N_{conf}}$ and the code files $file_j, ..., file_{N_{code}}$, along with their platform-adjusted versions $file'_j, ..., file'_{N_{code}}$, CC can be computed as:

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

8) Effort of package construction (EPC): Another effort driver in traditional application deployment is source compilation and the packaging of artifacts into an archive [12]. 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[8]. At best, a direct deployment of the application artifacts is possible ($EPC = 0$), 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.

9) 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: $DE(a) = EDS(a) + CC(a) + EPC(a)$. 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.

III. Results

In this section, we first describe the execution of the measurements, followed by a presentation, discussion, and interpretation of the results in Section III-B and a summary in Section III-C.

A. Execution of Measurements

As part of our migration experiment, we need to compute values for the deployment metrics from the preceding Section II-D. The timing for the ADT of an individual deployment run can be calculated by prefixing the script invocation with the Unix `time` 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. An exception to this is EngineYard with a total of 50 runs. The reason for this is that the deployment could not be fully automated and each run involved manual steps. The measurements were conducted at varying times during workdays, to simulate a normal deployment cycle inside a company. To minimize effects of external load-induced influences (e.g. *RubyGems* mirror) on the measurement, the deployments were run in parallel. 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 II and III.

Even though we could successfully migrate the application to all but one vendor, a substantial amount of work was required. Besides the captured effort values, additional important obstacles are incomplete documentation of the vendors and missing direct instance access for debugging, especially with container-based PaaS. Even with this common kind of application, getting the application to run was difficult and compromises with certain technology setups, e.g., web servers, were needed. Whereas some of these problems are to be expected and can only be prevented by unified container environments, major parts of the interaction with the system should be homogenized by, e.g., unified management interfaces.

During the case study, a number of bugs had to be fixed inside the cloud platforms. In total, we discovered four confirmed bugs on different vendors 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 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.

B. 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. On average, deployment takes an effort of 17 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

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

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

TABLE II: Deployment Efforts

	Heroku	**cloudControl**	**OpenShift**	**Pivotal**	**Bluemix**	**Elastic Beanstalk**	**EngineYard**
Effort of deployment steps (EDS)	10	15	24	17	17	12	23
Number of deployment steps (NDS)	4	5	6	6	6	2	8
Automated	4	5	6	6	6	2	4
Manual	0	0	0	0	0	0	4
Deployment steps parameters (DSP)	6	10	18	11	11	10	15
Configuration & code changes (CC)	1	1	0	1	1	40	7
Deployment artifacts	1	1	0	1	1	40	7
Application code	0	0	0	0	0	0	0
Effort of package construction (EPC)	3	3	3	0	0	3	4
Deployment reliability (DR)	0.96	0.72	0.78	1	0.89	0.99	1
Average deployment time (ADT) $\tilde{t}$	6.75 min	9.13 min	8.42 min	5.83 min	7.03 min	15.94 min	28.44 min
Deployment effort (DE)	14	19	27	18	18	55	34

TABLE III: Redeployment Efforts

	Heroku	**cloudControl**	**OpenShift**	**Pivotal**	**Bluemix**	**Elastic Beanstalk**	**EngineYard**
Effort of deployment steps (EDS)	1	2	1	2	2	1	2
Number of deployment steps (NDS)	1	1	1	1	1	1	1
Deployment steps parameters (DSP)	0	1	0	1	1	0	1
Deployment reliability (DR)	0.96	1	0.97	1	0.93	0.98	0.96
Average deployment time (ADT) $\tilde{t}$	6.69 min	5.71 min	7.41 min	5.73 min	6.61 min	8.71 min	8.25 min

instances must be initiated via a web interface, whereas the application deployment can be triggered by the client tools. As instance setup is normally performed once and not repetitively, this has less influence in practice than other steps would have. In the case of Elastic Beanstalk, the low EDS value of 12 is contrasted by a large configuration file. The majority of modern container-based PaaS reduce effort with respect to the EDS through an intelligent application type detection. 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.

Configuration & code changes (CC). Notably the container-based platforms can be used with only few deployment artifacts. Four out of five vendors support a Procfile-based deployment for specifying application startup commands ($CC = 1$). Whereas this compatibility helps to reproduce the application and server startup between those vendors, it is a major problem with the others. Especially custom server configuration inside the Procfile, i.e., the Puma web server, is a source of portability problems among platforms. Two platforms only support a preconfigured native system installation of Puma and one does not support the web server in conjunction with the necessary Ruby version at all. Moreover, the native installations can lead to dependency conflicts, if the provider uses another version than specified in the application's dependencies, resulting in compulsory code modifications. The only two vendors for which more configuration is needed are both VM-based offerings. In the case of EngineYard, the deployment descriptor can be kept small in a minimal configuration. Additionally, in contrast to other 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 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. 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.

Effort of package construction (EPC). The EPC 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 ($EPC = \{3, 4\}$).

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. In the case of redeploying existing applications, on average, we experienced less failures resulting in higher DR values.

Average deployment time (ADT). 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 [15] measured an average startup time for *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. 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 attributed to dependency caching. 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.

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 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.

C. 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.

IV. Related Work

Jamshidi et al. [11] 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 [4]–[8]. This study is a first step towards filling the identified gaps.

In [10], we already ported a small application between five PaaS vendors in a nonstructured 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. The majority of existing cloud migration studies are confined to feasibility and experience reports [16], [17] and omit a quantification and comparison of differences between migrations.

When measuring effort, the focus is often on operational cost comparisons [18]–[22], e.g., infrastructure costs, support, and maintenance or migration effort in man hours [23]. Beslic et al. [24] discuss an approach for an application migration among PaaS vendors similar to our study. Their migration scenario includes vendor discovery, application transformation, and deployment. However, the paper only outlines a high-level concept of the migration process and no concrete effort data of the steps. Pahl and Xiong [25] introduce a generic PaaS migration process for on-premises applications. Their framework is largely motivated by a view on different organizational and technological changes between the systems, but not focused on a detailed case study or measurement.

Miranda et al. [26] conduct a cloud-to-cloud migration between two IaaS offerings. In contrast to this work, the study uses software metrics to calculate the estimated migration costs in man hours rather than making migration efforts explicit. Tran et al. [27] define a metric, called cloud migration point (CMP), for effort estimation of cloud migrations derived from function points. Compared to our metrics, CMP estimates migration effort in an early phase of the development cycle, whereas we are evaluating factual changes after the implementation phase. Finally, another study that estimates effort in terms of man hours can be found in [28].

V. 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 workflows, the current solutions are invariably proprietary at the expense of recurring developer effort when moving between vendors. 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.

VI. 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 a 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.

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.

References

[1] F. Biscotti *et al.*, "Market Trends: Platform as a Service, Worldwide, 2013–2018, 2Q14 Update," Gartner, Tech. Rep., 2014.

[2] L. Carvalho *et al.*, "Worldwide Competitive Public Cloud Platform as a Service 2014–2018 Forecast and 2013 Vendor Shares," IDC, Tech. Rep., 2014.

[3] D. Petcu and A. V. Vasilakos, "Portability in Clouds: Approaches and Research Opportunities," *Scalable Computing: Practice and Experience*, vol. 15, no. 3, 2014.

[4] M. Hogan *et al.*, "NIST Cloud Computing Standards Roadmap," *NIST Special Publication 500-291*, 2011.

[5] L. Badger *et al.*, "Cloud Computing Synopsis and Recommendations," *NIST Special Publication 800-146*, 2012.

[6] D. Petcu *et al.*, "Portable Cloud applications – From theory to practice," *Future Generation Computer Systems*, vol. 29, no. 6, 2013.

[7] B. Di Martino, "Applications Portability and Services Interoperability among Multiple Clouds," *IEEE Cloud Computing*, vol. 1, no. 1, 2014.

[8] G. C. Silva *et al.*, "A Systematic Review of Cloud Lock-In Solutions," in *Proc. Conf. Cloud Computing Technology and Science*, 2013.

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

[10] S. Kolb and G. Wirtz, "Towards Application Portability in Platform as a Service," in *Proc. 8th Symp. Service-Oriented System Engineering*, 2014.

[11] P. Jamshidi *et al.*, "Cloud Migration Research: A Systematic Review," *IEEE Transactions on Cloud Computing*, vol. 1, no. 2, 2013.

[12] J. Lenhard *et al.*, "Measuring the Installability of Service Orchestrations Using the SQuaRE Method," in *Proc. 6th Conf. Service-Oriented Computing and Applications*, 2013.

[13] *Systems and software engineering – System and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*, ISO/IEC Std. 25 010, 2011.

[14] L. Briand, S. Morasca, and V. Basily, "Property-based software engineering measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, 1996.

[15] M. Mao and M. Humphrey, "A Performance Study on the VM Startup Time in the Cloud," in *Proc. 5th Conf. Cloud Computing*, 2012.

[16] M. A. Chauhan and M. A. Babar, "Migrating Service-Oriented System to Cloud Computing: An Experience Report," in *Proc. 4th Conf. Cloud Computing*, 2011.

[17] ——, "Towards Process Support for Migrating Applications to Cloud Computing," in *Proc. Conf. Cloud and Service Computing*, 2012.

[18] A. Khajeh-Hosseini *et al.*, "Cloud Migration: A Case Study of Migrating an Enterprise IT System to IaaS," in *Proc. 3rd Conf. Cloud Computing*, 2010.

[19] ——, "Decision Support Tools for Cloud Migration in the Enterprise," in *Proc. 4th Conf. Cloud Computing*, 2011.

[20] B. C. Tak *et al.*, "To Move or Not to Move: The Economics of Cloud Computing," in *Proc. 3rd Conf. Hot Topics in Cloud Computing*, 2011.

[21] M. Menzel and R. Ranjan, "CloudGenius: Decision Support for Web Server Cloud Migration," in *Proc. 21st Conf. World Wide Web*, 2012.

[22] M. Hajjat *et al.*, "Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud," *Computer Communication Review*, vol. 40, no. 4, 2010.

[23] V. Tran *et al.*, "Application Migration to Cloud: A Taxonomy of Critical Factors," in *Proc. 2nd Workshop Software Engineering for Cloud Computing*, 2011.

[24] A. Beslic *et al.*, "Towards a solution avoiding Vendor Lock-in to enable Migration Between Cloud Platforms," in *Proc. 2nd Workshop Model-Driven Engineering for High Performance and Cloud computing*, 2013.

[25] C. Pahl and H. Xiong, "Migration to PaaS Clouds – Migration Process and Architectural Concerns," in *Proc. 7th Symp. Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, 2013.

[26] J. Miranda *et al.*, "Assisting Cloud Service Migration Using Software Adaptation Techniques," in *Proc. 6th Conf. Cloud Computing*, 2013.

[27] V. Tran *et al.*, "Size Estimation of Cloud Migration Projects with Cloud Migration Point (CMP)," in *Proc. Symp. Empirical Software Engineering and Measurement*, 2011.

[28] K. Sun and Y. Li, "Effort Estimation in Cloud Migration Process," in *Proc. 7th Symp. Service Oriented System Engineering*, 2013.