Transactional Migration of Inhomogeneous Composite Cloud Applications

Josef Spillner and Manuel Ramírez López

Zurich University of Applied Sciences, School of Engineering
Service Prototyping Lab (blog.zhaw.ch/icclab/), 8401 Winterthur, Switzerland
{josef.spillner,ramz}@zhaw.ch

Abstract. For various motives such as routing around scheduled downtimes or escaping price surges, operations engineers of cloud applications are occasionally conducting zero-downtime live migrations. For monolithic virtual machine-based applications, this process has been studied extensively. In contrast, for composite microservice applications new challenges arise due to the need for a transactional migration of all constituent microservice implementations such as platform-specific lightweight containers and volumes. This paper outlines the challenges in the general heterogeneous case and solves them partially for a specialised inhomogeneous case based on the OpenShift and Kubernetes application models. Specifically, the paper describes our contributions in terms of tangible application models, tool designs, and migration evaluation. From the results, we reason about possible solutions for the general heterogeneous case.

1 Introduction

Cloud applications are complex software applications which require a cloud environment to operate and to become programmable and configurable through well-defined and uniform service interfaces. Typically, applications are deployed in the form of virtual machines, containers or runtime-specific archives into environments such as infrastructure or platform offered as a service (IaaS and PaaS, respectively). Recently, container platforms (CaaS) which combine infrastructure and higher-level platform elements such as on-demand volumes and scheduling policies have become popular especially for composite microservice-based applications [1].

The concern of continuous deployment in these environments is then to keep the applications up to date from the latest development activities [2]. Another concern is to maintain flexibility in where the applications are deployed and how quickly and easily they can be re-deployed into another environment. When a new deployment from the development environment is not desired or simply not possible due to the lack of prerequisites, a direct migration from a source to a target environment may be a solution despite hurdles to full automation [3].

Cloud application migration from this viewpoint can be divided into different categories: Homogeneous and heterogeneous migrations, referring to differences

in the source and target environment technologies, same-provider and cross-provider migrations, referring to the ability to migrate beyond the boundaries of a single hosting services provider, as well as offline and online/live migrations, referring to the continuity of application service provisioning while the migration goes on. On the spectrum between homogeneity and heterogeneity, inhomogeneous migrations are concerned with minor automatable differences. This paper is concerned with *live, heterogeneous/inhomogeneous, cross-provider migrations* as shown in Fig. 1.

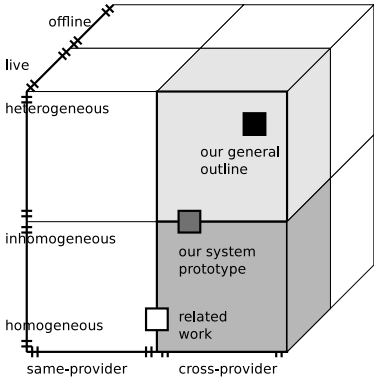

Fig. 1. Positioning within the multi-dimensional categories of cloud application migrations

An additional distinction is the representation of applications. Most of the literature covers monolithic applications which run as instances of virtual machine images where the main concern is pre-copy/post-copy main memory synchronisation [4]. Few emerging approaches exist for more lightweight compositions of stateless containers, where main memory is no longer a concern, and further platform-level components such as database services, volumes, secrets, routes and templates, some of which keep the actual state [5]. This paper is therefore concerned with migrating applications based on *container compositions* between diverse cloud platforms.

Consequently, the main contribution of the paper is a discussion of migration tool designs and prototypes for containerised Docker Compose, OpenShift and Kubernetes applications across providers. OpenShift is one of the most advanced open source PaaS stacks based on Kubernetes, a management and scheduling platform for containers, and in production use at several commercial cloud providers including RedHat's OpenShift Online, the APPUiO Swiss Container Platform, and numerous on-premise deployments [6]. Additional pure Kubernetes hosting is offered by the Google Cloud Platform, by Azure Container Services and by the overlay platform Tectonic for AWS and Azure, among other providers [7]. Both platforms orchestrate, place, schedule and scale ideally-

stateless Docker containers, while simpler compositions can also be achieved with Docker Compose.

The possibility to have the same containerised application deployed and running in different cloud providers and using different container platforms or orchestration tools is useful for both researcher and for companies. It facilitates the comparison of different cloud providers or different orchestration tools. For companies, it facilitates to run the applications in the most attractive hosting options by cost or other internal constraints. Key questions to which the use of our tools gives answers typically are: Is the migration feasible? Is it lossless? How fast is it? Does the order matter?

The paper is structured as follows. First, we analyse contemporary application compositions to derive requirements for the generalised live heterogeneous migration process (Sect. 2), followed by outlining the tool design principles (Sect. 3) and architecture (Sect. 4) for a simpler subset, inhomogeneous migration. The implemented tools are furthermore described (Sect. 5) and evaluated with real application examples (Sect. 6). The paper concludes with a summary of achievements (Sect. 7) and a discussion on filling the gap to truly heterogeneous live migration.

2 Analysis

In the definition given in a ten-year review of cloud-native applications [8], such applications are designed using self-contained deployment units. In currents applications the consensus is to use containers for reasonable isolation and almost native performance. Among the container technologies, Docker containers are the most common technology, although there are alternatives including Rkt, Containerd or CRI-O, as well as research-inspired prototypical engines such as SCONE [9]. In the following, we define a well-designed cloud application as a blueprint-described application, using containers to encapsulate the logic in microservices bound to the data confined in volumes. For deploying these applications in production into the cloud, just the container technology is not enough. Generally, a proper containerised application also uses an advanced container platform or an orchestration solution to add self-healing, auto-scaling, load balancing, service discovery and other properties which make it easier and faster to develop and deploy applications in the cloud. The platform also leverages more resilience, higher availability and scalability in the application itself. Among the most popular tools and platforms used to orchestrate containers are Docker Swarm, Docker Compose, Kubernetes, OpenShift, Rancher, and similar platforms. All of these can run in different cloud providers or on-premise. Moreover, usually each cloud provider has their own container platform. In Fig. 2 a diagram about the main container platforms and container orchestrators with their different associated composition blueprints is shown. The diagram also reveals relations and classifies the approaches by licencing (open source or proprietary) and by fitness for production. This complex technological landscape leads to different blueprints for the same containerised application depending which causes

practical difficulties for migrations. Despite fast ongoing consolidation, including the announced discontinuation of Docker Cloud in 2018, minor variations such as installed Kubernetes extensions continue to be a hurdle for seamless migration.

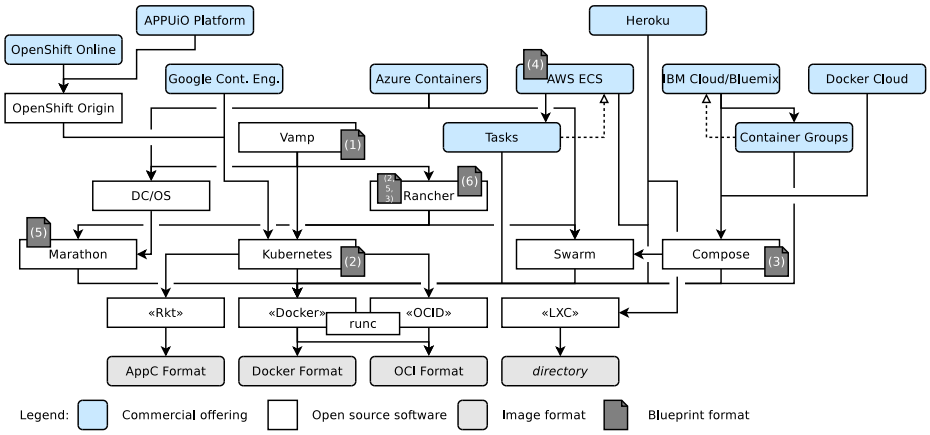

Fig. 2. Map of major container platforms and orchestration tools

The planning of the migration of a containerised application thus encompasses two key points which restrain the ability to automate the process:

- The blueprints: Even though containers encapsulate all the code in images which are meant to be portable and run everywhere, most of the real applications will need an orchestration tool to exploit all advantages that the cloud environment introduces: service discovery, definition of the number of replicas or persistence configuration. As most orchestration tools will introduce specific blueprints or deployment descriptors, the migration tool will need to convert between blueprint formats through transformation, perform minor modifications such as additions and removals of expressions, or rewrite limits and group associations (requirement R1).
- The data: Migration of the persisted data and other state information is non-trivial. In most container engines, the persistence of the data is confined to volumes. Depending of the cloud provider, the blueprints processed by the orchestration tools could reference volumes differently even for homogeneous orchestration tools, leading to slight differences and thus inhomogeneity (requirement R2).

To address these two points and increase the automation, the design of a suitable migration tool needs to account specifically for blueprint conversion and properly inlined data migration. We formalise a simplified composite application deployment as $D = \{b, c, v, \dots\}$, respectively, where: b: blueprint; c: set of associated containers; v: set of associated volumes. For example, a simplified OpenShift application is represented as $D_{openshift} = \{b, c, v, t, is, r, \dots\}$, where:

t: set of templates; is: set of image streams; r: set of routes. The goal of ideal heterogeneous migration m is to find migration paths from any arbitrary source deployment to any target deployment: $m = D \rightarrow D'$.

Fig. 3 summarises the different realistically resulting inhomogeneous migration paths between the three possible configurations $D_{kubernetes}$, $D_{openshift}$ and $D_{compose}$. Through various modifications applied to the orchestration descriptors, sources and targets can be largely different while mostly avoiding a loss of deployment information in fulfilment of R1.

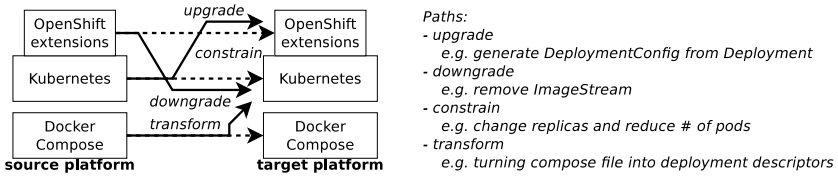

Fig. 3. Inhomogeneous application migration paths between three systems

3 General Application Migration Workflows

Requirement R1 calls for a dedicated blueprint extraction, conversion and redeployment process. We consider four steps in this process (see Fig. 4) which shall be implemented by a migration tool:

- Step 1. Downloading the blueprints of the composite application: The tool will connect to the source platform the application is running on, will identify all the components of the application and download the blueprints to a temporary location.
- Step 2. Converting the blueprints: A conversion from source to target format takes place. Even when homogeneous technologies are in place on both sides, re-sizing and re-grouping of components can be enforced according to the constraints on the target side (fulfilling R1).
- Step 3. Deploying the application: The tool will connect to the new orchestration platform and deploy the application there.
- Step 4. Deleting the application: Once the new application instance is running in the new place, the tool can delete the old application instance from the previous place. This step is optional and only executed under move semantics as opposed to copy semantics.

A major issue is the transactional guarantee of achieving a complex running and serving application on the target platform which in all regards equals the source. To make this process successful in all cases, the tool algorithm must further fulfil the following three requirements:

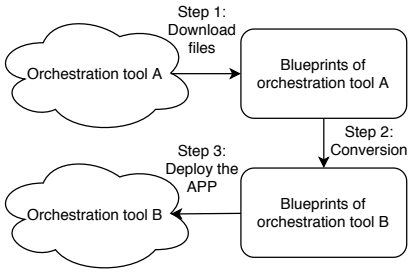

Fig. 4. Blueprints process diagram

- Connect to each of the different platforms in scope for heterogeneous migration.
- Convert between all the blueprints.
- Download and upload the application components from/to all the platforms, ensuring a re-deployment in the right order and a smooth hand-over by name service records which are typically external to both source and target platform.

With the previously described workflow, the tool can migrate stateless application or the stateless components of a stateful application. To complete the migration, the data in the containers needs to be migrated as well according to R2. In practice, this refers to volumes attached to containers, but also to databases and message queues which must be persisted in volume format beforehand. The process of the migration of a volume will be as follows:

- Step 1. Find the list of volumes linked to an application and for each one the path to the data.
- Step 2. Download the data to a temporary location. Due to the size, differential file transfer will be used.
- Step 3. Identify the same volume in the new deployment and pre-allocate the required storage space.
- Step 4. Upload the data to the new volume.

Now, we devise a fictive tool to express how the combined fulfilment of R1 and R2 in the context of heterogeneous application migration can be realised, expressed by Fig. 5 which highlights the separation into blueprints and data.

Although practicioners and researchers would benefit greatly from such a generic and all-encompassing tool, its conception and engineering would take many person months of software development work, needlessly delaying a prototype to answer the previously identified questions many companies in the field have right now. Instead, to focus on key research questions as outlined in the introduction follows a divide-and-conquer strategy. We subdivide the overall fictive tool into a set of smaller tools logically grouped into three categories, as shown in Fig. 6. Thus, we put our own prototypical work into context of a wider

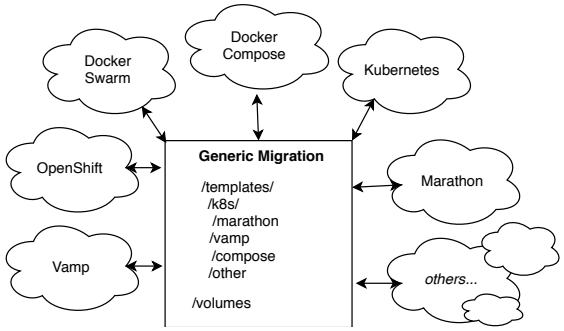

Fig. 5. Stateful application components diagram

ecosystem with some existing tools and further ongoing and future developments, making it possible to evaluate migration scenarios already now. The tools are:

- Homogeneously migrating containerised applications between multiple instances of the same orchestration tool: `os2os` (our work).
- Converting blueprints between the formats required by the platforms related to R1: Kompose (existing work).
- Rewriting Kubernetes blueprints to accomodate quotas: `descriptorrewriter` (our work).
- Migrating volumes related to R2: `volume2volume` (our work).
- Homogeneous transactional integration of volume and data migration for OpenShift as a service: `openshifter` (our early stage work).

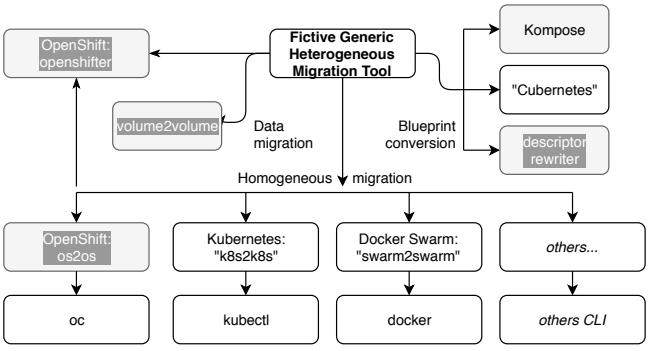

Fig. 6. Implementation strategy for fictive heterogeneous migration tool

We contribute in this paper the architectural design, implementation and combined evaluation of four tools referring to inhomogeneous OpenShift/Kubernetes/Docker Compose-to-OpenShift/Kubernetes migration. Use cases encompass intra-region replication and region switching within one provider, migration

from one provider to another, and developer-centric migration of local test applications into a cloud environment. All tools are publicly available for download and experimentation[1].

4 Migration Tools Design and Architecture

The general design of all tool ensures user-friendly abstraction over existing low-level tools such as `oc` and `kubectl`, the command line interfaces to OpenShift and Kubernetes, as well as auxiliary tools such as `rsync` for differential data transfer. Common migration and copy/replication workflows are available as powerful single commands. In Openshifter, these are complemented with full transaction support so that partial migrations can be gracefully interrupted or rolled back in case of ocurring issues.

As Fig. 7 shows on the left side, `os2os` uses `oc` to communicate with the source and target OpenShift clusters and temporarily stores all artefacts in local templates and volumes folders. This choice ensures that only a single provider configuration file needs to be maintained and that any features added to `oc` will be transparently available. On the right side of Fig. 7, the Openshifter tool is depicted which follows a service-oriented design. This choice ensures that the migration code itself runs as stateless, resilient and auto-scaled service. A further difference between the tools is that for Openshifter, we have explored a conceptual extension of packaged template and configuration data archives, called Helm charts, into *fat charts* which include a snapshot of the data, closing the gap to monolithic virtual machines.

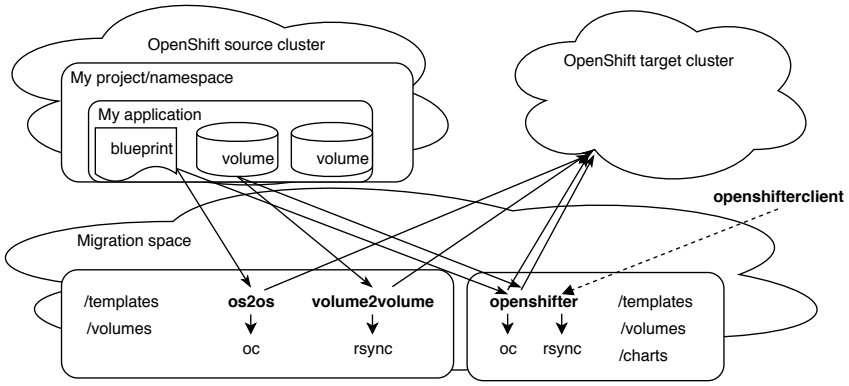

Fig. 7. OS2OS/Volume2Volume architectures (left); Openshifter architecture (right)

Exemplarily for all tools, `os2os` is composed of the following commands:

[1] Tools website: https://github.com/serviceprototypinglab/

- export: Connect to one cluster and export all the components (objects) of one application in one project, saved locally in a folder called `templates`.
- up: Connect to one cluster and upload all the components of one application in one project which are saved in `templates`.
- down: Connect to one cluster and delete all the components of one application in one project.
- migrate: Combine all the commands chronologically for a full migration in a single workflow.

The tools are implemented in different ways following the different designs. Both `os2os` and `volume2volume` are inspired by Kompose. They are implemented as command-line tools using Go with Cobra as library for handling the command-line parameters. Furthermore, the command names are derived from Kompose, making it easy to learn the tool for existing Kompose users. As usual in applications using Cobra, the configuration of the tool is stored in a YAML file. It contains the credentials to connect to the clusters, the cluster endpoints, the projects and the object types to migrate, overriding the default value of all object types. The `openshifter` prototype is implemented in Python using the AIO-HTTP web library to expose RESTful methods and works without any configuration file by receiving all parameters at invocation time.

5 Evaluation

When evaluating cloud migration tools, three important questions arise on whether the migration is lossless, performing and developer-acceptable. The measurable evaluation criteria are:

- C1 / Losslessness: The migration needs to avoid loss of critical application deployment information even after several roundtrips of migration between inhomogeneous systems. This is a challenge especially in the absence of features on some platforms. For instance, Kubernetes offers auto-scaling while Docker Swarm does not, leading to the question of how to preserve the information in case a migration from Kubernetes to Docker Swarm is followed by a reverse migration while the original source platform has vanished.
- C2 / Performance: A quantitative metric to express which time is needed both overall and for the individual migration steps. Further, can this time be pre-calculated or predicted in order to generate automated downtime messages, or can any downtime be alleviated.
- C3 / Acceptance: The migration needs to be easy to use for developers and operators as well as in modern DevOps environments.

A testbed with two local virtual machines running OpenShift 3.6 (setup S1) as well as a hosted OpenShift environment provided by the Swiss container platform APPUiO (S2) were set up to evaluate our tools experimentally according to the defined criteria C1 and C2. A synthetic scenario application consisting of three deployments and three services was prepared for that matter (A1), and the existing Snafu application (A2) was used for the comparison. The evaluation of C3 is left for future work.

5.1 Evaluation of Losslessness

For Kubernetes and OpenShift, the scenario service consists of shared Service and ConfigMap objects as well as platform-specific ones which are subject to loss; for Docker Compose, it consists of roughly equivalent directives. The deployed service was migrated from source to target and, with swapped roles between the platforms, back again from target to source. The following table reports on the loss of information depending on the system type. The Kompose tool incorrectly omits the lowercasing of object names and furthermore does not automatically complete the generated descriptors with information not already present in the Docker Compose files. To address the first issue, we have contributed a patch, whereas the second one would require a more extensive tool modification. The upgrade from Kubernetes to OpenShift works although OpenShift merely supports Deployment objects as a convenience whereas DeploymentConfig objects would be needed.

Table 1: Losslessness of blueprint transformations

Source	**Target**	**Loss**
OpenShift	OpenShift	none (assuming equal quotas)
OpenShift	Kubernetes (manual)	ImageStream,Route, DeploymentConfig
Kubernetes	OpenShift (manual)	(Deployment)
Docker Compose	Kubernetes (w/ Kompose)	none (yet incomplete & incorrect)

As a result, we have been able to automate all migrations except for the downgrade from OpenShift to Kubernetes using a combination of our tools which is invoked transparently when using Openshifter. The losslessness further refers to in-flight import and export of volume data. To avoid data corruption, applications need to perform modifications on the file level atomically, for instance by placing uploads into temporary files which are subsequently atomically renamed. Support for applications not adhering to this requirement is outside the scope of our work.

5.2 Evaluation of Performance

The synthetic scenario service A1 was exported from the source, re-deployed at the target, and torn down at the source 10 times with `os2os` in order to get information about the performance and its deviation in the local-to-local migration setup S1. Fig. 8 shows the results of the performance experiments. An evident characteristic is that exporting objects without changing them is more stable than running the down/up commands which modify the objects and cause changes to the scheduling of the remaining objects. A second observation is that, counter-intuitively, the `down` command consumes most of the time. A plausible explanation is that instead of simple deletions, objects are rather scheduled for deletion into a queue.

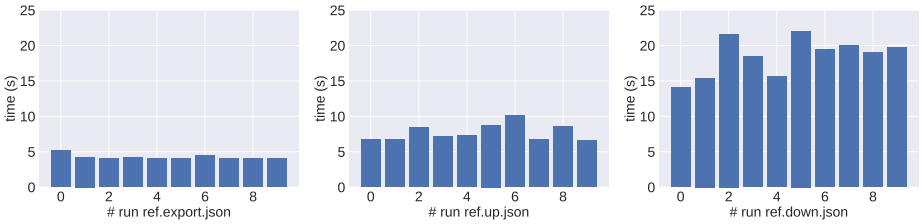

Fig. 8. Durations of the individual migration phases – export (left figure), up (middle), down (right) – between two local Kubernetes clusters

Service A2 was transformed automatically to measure the influence of the transformation logic on planned live migrations. The creation of Kubernetes descriptors with Kompose takes approximately 0.028 s. The adjustment of quotas and consolidation of pods, as performed by `descriptorrewriter`, takes approximately 0.064 s on the resulting Kubernetes descriptors. Both transformations are thus negligible which implies that apart from blueprint exports, the data transfer, which is primarily limited by the cluster connectivity, is the dominant influence on overall performance.

6 Conclusion

We have conducted a first analytical study on migrating cloud-native applications between inhomogeneous development and production platforms. The analysis was made possible through prototypical migration tools whose further development is in turn made possible by the results of the experiments. The derived findings from the experimental evaluation suggest that application portability is still an issue beyond the implementation (container) images. Future cloud platforms should include portability into the design requirements.

7 Future work

The current prototypes only support Kubernetes-based platforms. All functionality to convert other formats has been integrated into the experiments with external and existing tools. In the future, we want to integrate them in a unified way into `openshifter`. Further, we want to work on stricter requirements concerning a production-ready migration. They encompass improved user interfaces for easier inter-region/-zone migration within one provider, automatic identification of associated state and data formats, plugins for databases and message queues which keep non-volume state, data checksumming, and pre-copy statistics about both expected timing and resource requirements of the process and the subsequent deployment.

Acknowledgements

This research has been funded by Innosuisse - Swiss Innovation Agency in project MOSAIC/19333.1.

References

1. S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya. Efficient Virtual Machine Sizing for Hosting Containers as a Service (SERVICES 2015). In *2015 IEEE World Congress on Services*, pages 31–38, June 2015.
2. Pilar Rodríguez, Alireza Haghighatkhah, Lucy Ellen Lwakatare, Susanna Teppola, Tanja Suomalainen, Juho Eskeli, Teemu Karvonen, Pasi Kuvaja, June M. Verner, and Markku Oivo. Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, 123:263–291, 2017.
3. Massimo Ficco, Christian Esposito, Henry Chang, and Kim-Kwang Raymond Choo. Live Migration in Emerging Cloud Paradigms. *IEEE Cloud Computing*, 3(2):12–19, 2016.
4. Petronio Bezerra, Gustavo Martins, Reinaldo Gomes, Fellype Cavalcante, and Anderson F. B. F. da Costa. Evaluating live virtual machine migration overhead on client's application perspective. In *2017 International Conference on Information Networking, ICOIN 2017, Da Nang, Vietnam, January 11-13, 2017*, pages 503–508, 2017.
5. Jaemyoun Lee and Kyungtae Kang. Poster: A Lightweight Live Migration Platform with Container-based Virtualization for System Resilience. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'17, Niagara Falls, NY, USA, June 19-23, 2017*, page 158, 2017.
6. C. Pahl. Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 2(3):24–31, May 2015.
7. Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Commun. ACM*, 59(5):50–57, 2016.
8. Nane Kratzke and Peter-Christian Quint. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. *Journal of Systems and Software*, 126:1–16, 2017.
9. Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 689–703, 2016.