Institute of Architecture of Application Systems

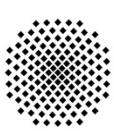
Enabling DevOps Collaboration and Continuous Delivery Using Diverse Application Environments

Johannes Wettinger, Vasilios Andrikopoulos, Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{wettinger, andrikopoulos, leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{Wettinger2015,
  author     = {Johannes Wettinger and Vasilios Andrikopoulos and
                Frank Leymann},
  title      = {Enabling DevOps Collaboration and Continuous Delivery Using
                Diverse Application Environments},
  booktitle  = {Proceedings of the 23rd International Conference on Cooperative
                Information Systems (CoopIS 2015)},
  year       = {2015},
  pages      = {348--358},
  series     = {Lecture Notes in Computer Science (LNCS)},
  publisher  = {Springer-Verlag}
}
```

University of Stuttgart
Germany

Enabling DevOps Collaboration and Continuous Delivery Using Diverse Application Environments

Johannes Wettinger, Vasilios Andrikopoulos, and Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart
Universitätsstr. 38, Stuttgart, Germany
{wettinger,andrikopoulos,leymann}@iaas.uni-stuttgart.de

Abstract. Aiming to provide the means for efficient collaboration between development and operations personnel, the DevOps paradigm is backed by an increasingly growing collection of tools and reusable artifacts for application management. Continuous delivery pipelines are established based on these building blocks by implementing fully automated, end-to-end application delivery processes, which significantly shorten release cycles to reduce risks and costs as well as gaining a critical competitive advantage. Diverse application environments need to be managed along the pipeline such as development, build, test, and production environments. In this work we address the need for systematically specifying and maintaining diverse application environment topologies enriched with environment-specific requirements in order to implement continuous delivery pipelines. Beside the representation of such requirements, we focus on their systematic and collaborative resolution with respect to the individual needs of the involved application environments.

Keywords: Continuous Delivery, Pipeline, Requirements, Topology, DevOps

1 Introduction

Continuous delivery [3] as an emerging paradigm aims to significantly shorten software release cycles by bridging existing gaps between developers, operations personnel (system administrators), and other parties involved in the delivery process. DevOps [4] is often considered in this context as an approach to improve the collaboration between development ('dev') and operations ('ops'). As a result of improved collaboration, new software releases can be made available much faster. Especially users, customers, and other stakeholders in the fields of Cloud services, Web & mobile applications, and the Internet of Things expect quick responses to changing demands and occurring issues. Consequently, shortening the time to make new releases available becomes a critical competitive advantage. In addition, tight feedback loops involving users and customers based on continuous delivery ensure building the 'right' software, which eventually improves customer satisfaction, shortens time to market, and reduces costs. Typically, cultural and organizational gaps between developers, operations personnel, and further groups appear, so these separated groups follow different goals such as 'push changes to production quickly' on the development side versus 'keep production stable' on the operations side. This often results in incompatible or even opposing processes and mindsets. By implementing continuous delivery, these goals

and processes are aligned. Independent of the chosen approach to establish continuous delivery by tackling cultural and organizational issues, a high degree of technical automation is required. This is typically achieved by implementing an automated *continuous delivery pipeline* (also known as deployment pipeline) [3], covering all required steps such as retrieving code from a repository, building packaged binaries, running tests, and deployment to production. Such an automated and integrated delivery pipeline improves software quality, e.g., by avoiding the deployment of changes that did not pass all tests. Moreover, the high degree of automation typically leads to significant cost reduction because the automated delivery process replaces most of the manual, time-consuming, and error-prone steps. Establishing a continuous delivery pipeline means implementing an individually tailored automation system, which considers the entire delivery process. Furthermore, a separate pipeline has to be established for each independently deployable unit, e.g., an application or microservice [8]. As a result, a potentially large and growing number of individual pipelines has to be maintained. Along each pipeline suitable application environments (development, test, production, etc.) must be established as their key building blocks. Toward this goal, our research focuses on *dynamically and systematically establishing corresponding application environments as the building blocks of continuous delivery pipelines to improve DevOps collaboration.*

The constantly growing DevOps community supports this notion by providing a huge variety of individual approaches such as tools and reusable artifacts to implement holistic delivery automation. Prominent examples are the Chef configuration management framework[1], the Jenkins[2] continuous integration server, and Docker[3] as an efficient container virtualization approach. The open-source communities affiliated with these tools publicly share reusable artifacts to package, deploy, and operate middleware and application components. For instance, Chef's Ruby-based domain-specific language [2] can be used to create and maintain cookbooks — basically scripts to automate the deployment and wiring of different components of an application stack. These approaches are typically combined with Cloud computing [7] to enable on-demand provisioning of resources such as virtual servers and storage in a self-service manner. This is not limited to the infrastructure level, but may also include database-as-a-service and other middleware-centric offerings.

The goal of our work is to systematically handle and resolve such requirements to establish suitable application environments (development, test, production, etc.) as the key building blocks of continuous delivery pipelines. For this purpose, we provide the means to collaboratively maintain such application environments, allowing developers and operations personnel to share and utilize a common meta-model. As part of this effort we reuse concepts from the fields of requirements engineering and software configuration management. The major contributions of this paper can therefore be summarized by the *representation* of (i) *environment requirements* imposed by applications across different dimensions and of (ii) *application environment topologies* to interlink corresponding requirements, and (iii) the systematic *resolution* of these requirements.

[1] Chef: `http://www.chef.io`
[2] Jenkins: `http://jenkins-ci.org`
[3] Docker: `http://www.docker.com`

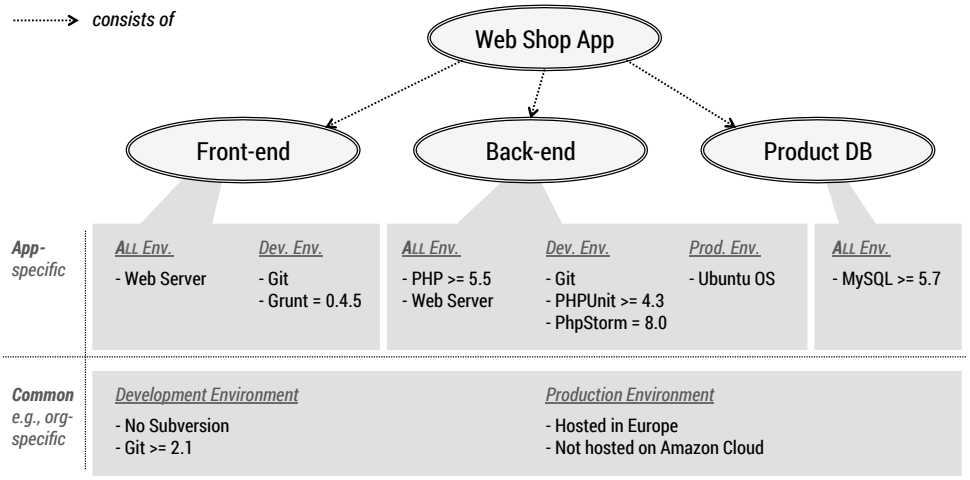


Fig. 1. Web shop application with its environment-specific requirements

The remainder of this paper is structured as follows: Section 2 presents a motivating scenario that is used throughout the rest of this work. Section 3 introduces the fundamental concepts of our proposal, together with their formalization as environment-specific requirements. Section 4 discusses how these requirements can be resolved into concrete application environments. Finally, Section 5 concludes the paper and outlines future work.

2 Motivating Scenario

In this section we introduce a Web shop application, which is used as motivating scenario and running example for this work. Figure 1 outlines the three-tier architecture of the Web application, consisting of a front-end, back-end, and product database. The front-end is implemented in HTML and JavaScript to provide the Web shop's user interface. It communicates with the PHP-based back-end using HTTP messages containing JSON data. The back-end itself provides a RESTful API to enable the HTTP-based communication with arbitrary clients such as the Web shop front-end or a mobile shopping application. Product information (inventory, etc.) are stored in the product database, which is based on a MySQL database server.

As shown in Fig. 1, diverse requirements regarding middleware, infrastructure, and tooling are attached to the application stack with respect to the different application environments. Some of them are required for all kinds of environments such as a Web server and a PHP runtime. Build tools and version control mechanisms such as Git[4] and Grunt[5] are only required for development & build environments. On the other hand, some requirements may only be relevant for operating the application in

[4] Git: `http://git-scm.com`
[5] Grunt: `http://gruntjs.com`

production such as using a specific operating system. Beside the distinction between environments, these requirements can be classified as *application-specific* or *common* requirements: the former ones are essential to develop and operate instances of a specific application, whereas the latter ones are application-agnostic and may be derived from organization- or domain-specific policies. As an example, such a policy could say that all data must be stored in Europe, so the whole application stack must be hosted in Europe when instantiated. In the following sections we discuss how such requirements can be represented, systematically handled, and resolved to collaboratively establish continuous delivery pipelines.

3 Fundamentals

In this section we focus on introducing a set of fundamental concepts that allow us to address the challenges identified in the previous sections. More specifically, we propose for this purpose the concept of *application environment requirements (AERs)* and discuss their relation to other existing kinds of requirements (Section 3.1). We then show how *application environment topologies (AETs)* can be used to describe application environments by interlinking AERs (Section 3.2). Finally, we outline how to collaboratively establish continuous delivery pipelines based on AERs and AETs (Section 3.3).

3.1 Application Environment Requirements

The Web shop application we introduced in Section 2 outlines several requirements attached to different parts of the application stack. These are all non-functional requirements — such as infrastructure and middleware requirements — that need to be satisfied to establish different kinds of environments for a particular application. For this purpose, we propose *application environment requirements (AERs)* as the *subset of non-functional requirements imposed by an application on its underlying environment for different stages of the application lifecycle.* We do not consider functional requirements of the application related to the application's user interface and other application features. However, we do consider the relation of AERs to other non-functional requirements. Referring to the Web shop application, for example, the requirement that the application is hosted in Europe may have been derived from a specific compliance requirement, saying that all data must stay in Europe to conform to the EU Data Protection Directive [6]. As already outlined by the Web shop example (Section 2) environment-specific AERs need to be considered and satisfied when building a corresponding environment such as a development environment or production environment. Some AERs are relevant for multiple environments such as the PHP runtime required by the back-end of the Web application: this does not only affect production and test environments, but also development environments, e.g., to allow developers to analyze the impact of their code changes immediately and locally on their developer environment.

Definition 1 (AER Predicate). *An* AER predicate *is a representation of an AER in predicate logic. Assuming that* $\mathcal{M} = \{im, ev\}$ *is the set of modes (*im *= 'immediately required',* ev *= 'eventually required') for AERs,* $\mathcal{E}$ *is the domain of all entities that*

can be potentially required (middleware, infrastructure, etc.), $\mathcal{P}$ *is the domain of all possible properties the entities may own, and* $\mathcal{V}$ *is the domain of all potential property values, there are two valid forms of AER predicates:* $P^{m,<name>}_{<label>} : \mathcal{E} \rightarrow \{true, false\}$ *and* $P^{m,<name>}_{<label>} : \mathcal{E} \times \mathcal{P} \times \mathcal{V} \rightarrow \{true, false\}$, $m \in \mathcal{M}$.

The following AER predicates can be identified based on this definition to properly express AERs for applications such as the Web shop application described previously:

- $P^{im,include}_{<label>} : \mathcal{E} \rightarrow \{true, false\}$ implies that a solution for a particular entity $e \in \mathcal{E}$ must *immediately* exist in the application stack (e.g., the runtime in which a component is executed); otherwise the predicate evaluates to $false$.
- $P^{ev,include}_{<label>} : \mathcal{E} \rightarrow \{true, false\}$ implies that a solution for a particular entity $e \in \mathcal{E}$ must *eventually* exist in the application stack (e.g., the underlying operating system); otherwise the predicate evaluates to $false$.

Further AER predicates can be defined in this fashion such as $P^{im|ev,exclude}_{<label>} : \mathcal{E} \rightarrow \{true, false\}$ as the inversion of $P^{im|ev,include}_{<label>}$. The '|' symbol is used to indicate variants of the defined predicates, e.g., $im|ev$ to *immediately* or *eventually* exclude solutions. Moreover, $P^{im|ev,equals|eqGr}_{<label>} : \mathcal{E} \times \mathcal{P} \times \mathcal{V} \rightarrow \{true, false\}$ implies that $P^{im|ev,include}_{<label>}(e) = true$ for a particular entity $e \in \mathcal{E}$ and the given solution owns a property $p \in \mathcal{P}$ and its value equals to (or is greater than) $v \in \mathcal{V}$; otherwise the predicate evaluates to $false$. As an example referring to the Web shop application, we may define the following AER predicate to express the requirement that the back-end must be *immediately* hosted on a PHP runtime, version 5.5 or better: $P^{im,eqGr}_{PHP}$('PHP', 'version', '5.5'). Such predicates can be bundled as logical expressions using logical operators such as:

$$P^{im,eqGr}_{PHP}(\text{'PHP', 'version', '5.5'}) \wedge P^{ev,include}_{Ubuntu}(\text{'Ubuntu OS'}) \wedge ...$$

3.2 Application Environment Topologies

For the purpose of representing both the application and the environment for which AERs are defined we use the concept of an *application environment topology (AET).* AETs can be expressed as *typed graphs* following [1], where all nodes and edges are of the form $< name : type >$ and $< type >$, respectively. Nodes represent components of the application stack, including *aaS solutions such as infrastructure-as-a-service (IaaS) and database-as-a-service (DBaaS), while edges represent the different types of relations between them, e.g., *HostedOn, DependsOn,* etc. Examples for nodes are *ApacheHTTPServer:WebServer* (named node) and *WebServer* (unnamed node). As discussed in [1], many existing works such as the TOSCA specification[6], the Cloud Blueprinting approach [9], and the CloudML language[7], as well as solutions

[6] TOSCA: `http://www.oasis-open.org/committees/tosca`
[7] CloudML: `http://cloudml.org`

like Amazon CloudFormation[8], the OpenNebula initiative[9], or OpenStack Heat[10], essentially build on this typed topology graph model in various forms.

In order to connect an AER with the actual application that they express requirements for, AERs are allowed to be attached to different parts of an AET. AERs for example can be attached to nodes, denoting the requirements expressed by the component on the environment, or to subgraphs in the AET, denoting requirements that need to be satisfied for all nodes in the subgraph, e.g., for the front-end of the application. An attachment map is used for this purpose:

Definition 2 (AER Attachment Map). *An* AER attachment map *is a mapping function f_{map} associated with a particular AET to assign AER predicates to specific parts of the topology. Assuming that N is the set of nodes in the topology T, $\mathcal{X}$ is the domain of logical expressions consisting of AER predicates, and $\mathcal{E} = \{development, test, production, ...\}$ representing the environments to which AERs are bound, then the mapping function is formally defined as $f_{map} : N \cup \{T\} \times \mathcal{E} \rightarrow \mathcal{X}$.*

The mapping function f_{map} is used to attach application-specific AERs to nodes in the AET and to the topology as a whole, without the need of modifying the topology definition. In case not all AERs are reflected by solutions in a topology, we refer to such a topology with an associated AER attachment map as an *unresolved topology:*

Definition 3 (Unresolved AET). *An* unresolved AET (unresolved topology) *is an application environment topology containing at least one AER that is not satisfied by an attached solution.*

Unresolved requirements can be satisfied in different ways, depending on which kind of application environment (development, production, etc.) should be described by a resulting *resolved topology:*

Definition 4 (Resolved AET). *A* resolved AET (resolved topology) *is an application environment topology containing at least one solution for each AER attached to the topology.*

As an example, a PHP runtime environment may be satisfied by a corresponding Docker container[11] with minimum overhead for a development environment. However, for a production environment, an elastic platform-as-a-service solution such as Google App Engine[12] could be more appropriate. In the following, we outline how AERs and AETs are utilized to establish continuous delivery pipelines.

3.3 Continuous Delivery Pipelines (CDPs)

The eventual purpose of AERs and AETs is to systematically and collaboratively establish continuous delivery. Technically, continuous delivery pipelines have to be built and maintained for this reason. In this context, we refer to a *continuous*

[8] Amazon CloudFormation: `http://aws.amazon.com/cloudformation`
[9] OpenNebula: `http://opennebula.org`
[10] OpenStack Heat: `https://wiki.openstack.org/wiki/Heat`
[11] PHP Docker container: `https://registry.hub.docker.com/_/php`
[12] Google App Engine: `https://cloud.google.com/appengine`

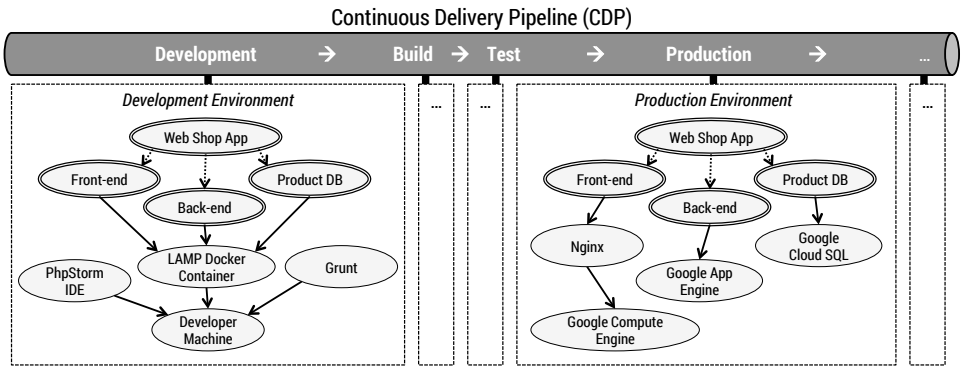


Fig. 2. Example for continuous delivery pipeline (CDP) for the Web shop application

delivery pipeline (CDP) as a delivery automation system, individually tailored and maintained per independently deployable unit such as an application or a microservice. Thus, a CDP implements an application-specific delivery plan. Consequently, a potentially large and growing number of individual CDPs have to be established and maintained, especially when following the emerging microservice architecture style [8]. In addition, this has to be done in a collaborative manner to enable aligned and automated delivery processes, considering diverse parties that are involved such as developers and operations personnel. By resolving AERs appropriately, diverse application environments (development environment, production environment, etc.) are established as key building blocks of a CDP. Figure 2 shows an example for a CDP, considering the requirements of the Web shop application described in Section 2. Each phase of the CDP has a dedicated resolved AET attached: a development environment typically aims to be lightweight, e.g., by running the entire application stack in a single container. Moreover, development tools are required to run on the developer machine. These are not necessary in a production environment. On the other hand, a production environment would be preferably based on scalable and elastic Cloud offerings to keep the application responsive even in case a lot of load appears. In the following, we present the usage of a knowledge base and a supporting process to find suitable solutions in order to resolve AERs. In this manner, resolved topologies are created in order to instantiate specific application environments, which can then be tied together to establish a continuous delivery pipeline.

4 Requirements Resolution

In order to create an instance of a particular application environment (e.g., a development environment for the Web shop application outlined in Section 2), all AERs attached to an unresolved AET need to be satisfied. In Section 4.1 we outline the usage of a *knowledge base (KB)* for the purpose of resolving AERs. Based on the existence of such a KB, Section 4.2 discusses how diverse application environments (development, production, etc.) can be built by deriving resolved topologies.

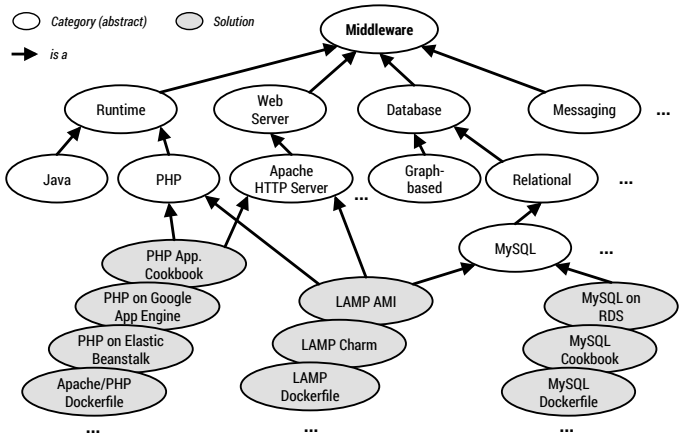


Fig. 3. Middleware solutions stored in the knowledge base and organized in a taxonomy

4.1 Application Environment Knowledge Base (KB)

In order to enable the resolution of *unresolved topologies* with attached AERs we propose the use of a *knowledge base (KB)* that enables informed decision making when satisfying AERs to create resolved topologies that can be used to instantiate application environments. The KB [10] contains linked solutions such as reusable artifacts, tools, and services to manage development, infrastructure, and middleware aspects of an application environment. Technically, the KB is distributed and composed, so it is not a monolithic system. This enables the KB to be collaboratively maintained, e.g., considering experts from different domains such as developers and system administrators.

Figure 3 outlines a small selection of middleware solutions stored in the KB. These are captured in a taxonomy that consists of abstract entities (for categorization purposes) and the actual solutions. Similarly, infrastructure solutions (e.g., virtual server images), development solutions (e.g., version control and build tools), and other supplementary solutions (e.g., monitoring tools) are captured in further taxonomies stored in the KB. Links that potentially cross taxonomy boundaries are established between solutions to express dependencies such as a middleware component that relies on a particular operating system to run[13].

4.2 Building Diverse Application Environments

The ultimate goal of our work is to allow the automated building of diverse application environments as key building blocks of a continuous delivery pipeline for a particular application such as the Web shop introduced in Section 2. Therefore, we need to provide the means to derive a suitable, resolved topology that satisfies all associated AERs. The following process can be used for this purpose:

[13] More information on the KB is available at: `http://github.com/jojow/aer-paper`

1. Define constraints to express *environment preferences* for the target environment, e.g., *minimum resource usage* for development environments vs. *resilient & elastic* for production environments.
2. Use the unresolved topology with its AER attachment map in conjunction with application-agnostic, common AERs to reason about the given AERs and constraints by querying the KB for appropriate solutions.
3. Evaluate environment preferences to filter (and potentially rank) the resulting resolved topologies.
4. Pick a suitable or, if a ranking is available, the most suitable resolved topology.

The concept of expressing additional preference requirements (environment-specific constraints in our case) is well-known, e.g., from goal-oriented approaches established in the field of requirements engineering [5]; beside the mandatory requirements (AERs in our case) that must be fulfilled by any proposed solution, (potentially prioritized) preference requirements ('nice-to-have') such as solution details and complex temporal properties are expressed and considered. This is enabled by not only searching the optimal solution based on the mandatory requirements, but also considering alternative solutions when satisfying AERs in unresolved topologies.

Building on previous work [1], the original, unresolved AET is represented as an application-specific α-*topology*; using the α-topology, a reusable γ-*topology* is derived from the knowledge base, considering all potential topology alternatives. Both α- and γ-topologies are typed graphs with inheritance (essentially, class diagrams using only association and inheritance relations, expressed as graphs), from the combination of which a set of *viable* topologies can be derived. Viable topologies in the context of [1] refer only to the fact that the topology graph is typed over the elements of the graph resulting from the union of the α- and γ-topologies, also called the μ-topology. A potentially large set of viable topologies can be created by the morphism from the type graph of the μ-topology to the typed graphs of application topologies. For this purpose, a *filter function* σ is used to prune down the number of potentially viable topologies $\mathcal{T}$. In principle, a set of constraints $\mathcal{C}$ is defined based on environment preferences for each viable topology T:

$$\sigma(T, c) = \begin{cases} T & \text{if } \textit{condition}(c)\text{=}\textit{true}, \\ \emptyset & \text{otherwise.} \end{cases}, \text{ where } T \in \mathcal{T}, c \in \mathcal{C}.$$

Filter functions can be chained to allow for multiple such constraints to be applied. Following [1], we also use *utility functions* to rank the filtered, viable topologies (e.g., minimum number of nodes) in order to find the most suitable topology that satisfies all AERs and the corresponding environment preferences. Any kind of function can be used as a utility function, as long as it allows for the mapping from the space of viable topologies to that of real numbers. For example, a utility function could return the number of nodes in the topology graph, aiming for minimizing the number of components required for the deployment of the application. Multiple dimensions can be combined, e.g., number of nodes with costs of operating the application under a given load, as discussed in [1]. We finally performed an evaluation using a case study based on the Web shop application[14].

[14] More information on the evaluation is available at: `http://github.com/jojow/aer-paper`

5 Conclusions

Continuous delivery and DevOps have emerged with the goal to bring together developers and operations personnel by enabling their efficient collaboration. This is technically supported by establishing automated continuous delivery pipelines to significantly shorten release cycles without quality degradation. Diverse application environments (development, test, production, etc.) form the key building blocks of a continuous delivery pipeline. In the previous sections we proposed the concept of application environment requirements as a particular kind of non-functional requirements and formalized their representation using predicate logic, which allowed us to define mappings between application topologies and application environment requirements. For the resolution of these requirements into concrete environments we proposed the usage of a knowledge base in combination with a resolution process with distinct tasks. With respect to the latter, in this work we focused on deriving suitable application environments driven by diverse environment preferences. We plan however to extend the resolution and selection process in future work by considering costs and QoS aspects, using [1] as the basis. Moreover, we aim to use AERs and the knowledge base to support the migration of existing applications, e.g., to consider the available options for partially or fully migrating an application to the Cloud.

Acknowledgment. This work is partially funded by the FP7 EU-FET project 600792 ALLOW Ensembles.

References

1. Andrikopoulos, V., Sáez, S.G., Leymann, F., Wettinger, J.: Optimal Distribution of Applications in the Cloud. In: Proceedings of the 26th Conference on Advanced Information Systems Engineering (CAiSE). LNCS, Springer (2014)
2. Günther, S., Haupt, M., Splieth, M.: Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures. Tech. rep., Very Large Business Applications Lab Magdeburg, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg (2010)
3. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional (2010)
4. Hüttermann, M.: DevOps for Developers. Apress (2012)
5. Liaskos, S., McIlraith, S.A., Sohrabi, S., Mylopoulos, J.: Representing and Reasoning About Preferences in Requirements Engineering. Requirements Engineering 16(3), 227–249 (2011)
6. Louridas, P.: Up in the Air: Moving Your Applications to the Cloud. Software, IEEE 27(4), 6–11 (2010)
7. Mell, P., Grance, T.: The NIST Definition of Cloud Computing. National Institute of Standards and Technology (2011)
8. Newman, S.: Building Microservices. O'Reilly Media (2015)
9. Papazoglou, M., van den Heuvel, W.: Blueprinting the Cloud. Internet Computing, IEEE 15(6), 74–79 (2011)
10. Wettinger, J., Andrikopoulos, V., Leymann, F.: Automated Capturing and Systematic Usage of DevOps Knowledge for Cloud Applications. In: Proceedings of the International Conference on Cloud Engineering (IC2E). IEEE Computer Society (2015)