Institute of Architecture of Application Systems

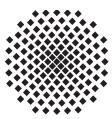
Automated Capturing and Systematic Usage of DevOps Knowledge for Cloud Applications

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     = {Automated Capturing and Systematic Usage of DevOps Knowledge
               for Cloud Applications},
  booktitle = {Proceedings of the IEEE International Conference on Cloud
               Engineering (IC2E)},
  year      = {2015},
  pages     = {60--65},
  publisher = {IEEE Computer Society}
}
```

Universität Stuttgart
Germany

Automated Capturing and Systematic Usage of DevOps Knowledge for Cloud Applications

Johannes Wettinger, Vasilios Andrikopoulos, Frank Leymann

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

***Abstract*—DevOps is an emerging paradigm to actively foster the collaboration between system developers and operations in order to enable efficient end-to-end automation of software deployment and management processes. DevOps is typically combined with Cloud computing, which enables rapid, on-demand provisioning of underlying resources such as virtual servers, storage, or database instances using APIs in a self-service manner. Today, an ever-growing amount of DevOps tools, reusable artifacts such as scripts, and Cloud services are available to implement DevOps automation. Thus, informed decision making on the appropriate approach(es) for the needs of an application is hard. In this work we present a collaborative and holistic approach to capture DevOps knowledge in a knowledgebase. Beside the ability to capture expert knowledge and utilize crowdsourcing approaches, we implemented a crawling framework to automatically discover and capture DevOps knowledge. Moreover, we show how this knowledge is utilized to deploy and operate Cloud applications.**

***Keywords*—DevOps, Crawling, Knowledge Management, Knowledge Model, Cloud Computing**

I. Introduction and Problem Statement

Today, it is vital for many software vendors and providers, especially in the field of Web applications and software-as-a-service (SaaS) applications to frequently and continuously roll out updates of an application. This is because users and customers expect new features and bug fixes to be available as quickly as possible. Consequently, a critical competitive advantage can be achieved by meeting these expectations and implementing mechanisms to drastically shorten application release cycles. Thorough and holistic automation is required to enable frequent and continuous software delivery [1]. DevOps is an emerging paradigm [2]–[4] to eliminate the split and barrier between developers and operations personnel:

"DevOps is a mix of patterns intended to improve collaboration between development and operations. DevOps addresses shared goals and incentives as well as shared processes and tools. Because of the natural conflicts among different groups, shared goals and incentives may not always be achievable. However, they should at least be aligned with one another." [5]

Tight collaboration between developers and operations (DevOps) as well as holistic end-to-end automation (e.g., in the form of a fully automated deployment pipeline [1]) is required to enable continuous delivery. A huge and confusing variety of tools, reusable artifacts, and services are available today. Prominent examples are the Chef configuration management framework [6], the Jenkins[1] continuous integration server, and Docker[2] as an efficient container virtualization approach. The open-source communities affiliated with these tools are publicly sharing reusable artifacts to package, deploy, and operate middleware and application components. For instance, Chef's Ruby-based domain-specific language [7] can be used to create and maintain cookbooks, which are basically scripts to automate the deployment and wiring of different components of an application stack. Alternatively or additionally, Docker container images can be built to package certain components in an isolated and reusable manner. There are further DevOps tools and corresponding artifacts such as Juju[3] with Juju charms[4] and Puppet [8], [9] with Puppet modules[5] that can be used alternatively or complementary.

DevOps approaches are typically combined with Cloud computing [10] to enable on-demand provisioning of resources such as virtual servers and storage in a self-service manner. Different interfaces are offered by Cloud providers (e.g., Amazon[6]) such as graphical user interfaces, command line interfaces, and APIs to provision and manage these resources. Especially APIs and command line interfaces are an efficient means to integrate DevOps automation approaches with Cloud resource management programmatically. This can be done for public, private, and hybrid Cloud scenarios. As an example, Amazon's APIs can be utilized to provision virtual servers on which Chef cookbooks or Juju charms are executed to deploy a certain application stack. Moreover, some Cloud providers offer higher-level services such as middleware services (e.g., runtime-as-a-service, database-as-a-service, etc.) and operations automation services, abstracting from the underlying infrastructure. These services can be used alternatively or complementary to the lower-level infrastructure services.

Because DevOps automation approaches and Cloud services appear, change, and disappear rapidly, it is hard to choose the most appropriate solutions and combinations thereof to implement holistic DevOps automation for a specific application. Informed decision making is hard because of the huge variety of options. DevOps knowledge is practically available at large scale, but it is not systematically captured and managed.

[1] Jenkins: http://jenkins-ci.org
[2] Docker: http://www.docker.com
[3] Juju: https://juju.ubuntu.com
[4] Juju charms: https://jujucharms.com
[5] Puppet modules: https://forge.puppetlabs.com
[6] Amazon EC2 documentation: http://aws.amazon.com/documentation/ec2

However, providing solutions for this *DevOps knowledge management problem* is of the utmost importance to end up with an efficient DevOps automation and collaboration process to enable continuous delivery. The major contributions of this paper can therefore be summarized by:

- A holistic approach toward systematic DevOps knowledge management.
- The design and implementation of a DevOps knowledgebase and the tooling required to populate it.
- An automated crawling approach to discover and capture DevOps knowledge.

The remainder of this paper is structured as follows: Section II outlines a motivating scenario that is used as running example for this paper. We present a holistic methodology to enable DevOps knowledge management in Section III. Because the DevOps knowledgebase is the core of our methodology, Section IV discusses DevOps knowledge classification and a prototype implementation in detail. Section V discusses the utilization of DevOps knowledge based on specific DevOps requirements. Finally, Section VI concludes the paper.

II. Motivation

As a motivating scenario and running example for our work we consider the deployment and operations requirements of WordPress[7], which is a popular open-source blog application. WordPress requires: (i) a *MySQL database server* version 5.0 or greater, (ii) a *PHP runtime* version 5.2.4 or greater, and (iii) a *Web server* such as Apache HTTP Server[8] or Nginx[9]. In order to enable continuous delivery of new versions of WordPress (bug fixes, new features, etc.), these requirements have to be considered when implementing an end-to-end DevOps automation and collaboration process.

Figure 1 presents an overview of the middleware components required to run WordPress. Moreover, different deployment options are outlined to make these middleware components available to the application, satisfying WordPress' requirements. As an example, an existing Amazon Machine Image (AMI)[10] could be used to run a complete LAMP[11] stack on a single virtual machine (VM) to operate WordPress. However, in this case resources are limited to a single VM, and as such scalability options are limited. Thus, open-source deployment scripts such as the MySQL cookbook[12] and the PHP application cookbook[13] (Chef) could be used to split the middleware across two VMs, running in a Linux or Windows VM. In case the scalability of the MySQL database server needs to be further improved, the MySQL charm[14] may be used as a set of operations scripts to run a MySQL database cluster in a master/slave setup: data that are written to the master instance are consistently replicated to the slave instances, so reading requests can be load-balanced between slave instances. However,

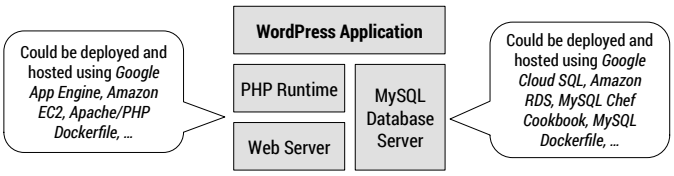


Figure 1. Middleware components of WordPress and deployment alternatives

there is a constraint: Juju charms can only be deployed on Ubuntu-based VMs.

To go one step further and provide elastic middleware for WordPress to scale in and out dynamically and automatically depending on the current workload, solutions such as database-as-a-service and runtime-as-a-service [11] offerings can be used instead of maintaining VMs. For instance, Amazon Elastic Beanstalk[15] and Google App Engine[16] provide PHP runtime environments transparently as a service without seeing or maintaining the underlying infrastructure such as VMs. Amazon Relational Database Service (RDS)[17] may be used as MySQL database-as-a-service to satisfy the MySQL requirement of WordPress. One downside of this elastic middleware-as-a-service approach is its limited configurability. For instance, a MySQL database hosted on a VM can be arbitrarily tuned and configured, whereas a database instance offered as a service only provides predefined configuration options[18].

Therefore, even for an application with a few deployment and operations requirements such as WordPress, there exists a great variety of alternatives. Because of the individual benefits and drawbacks of each alternative, it becomes difficult for application designers to decide on how to easily and efficiently deploy their application. Thus, the major challenge we tackle with our work is *how to systematically capture, link, and utilize DevOps knowledge as a foundation for informed decision making during application design and deployment.* For this purpose, a methodology and system to *collaboratively* maintain and use the captured knowledge is required because different parties such as developers and operations personnel may be involved. The following section presents a holistic approach toward this goal.

III. DevOps Knowledge Management Methodology

When developing and operating applications such as WordPress, a large variety of alternatives exist in choosing appropriate infrastructure and middleware solutions to fulfill DevOps requirements. Figure 2 outlines our proposal for a holistic DevOps knowledge management approach with the goal to provide the means to systematically resolve DevOps requirements. More specifically, DevOps knowledge is currently spread across different sources in the form of *unstructured* and *semi-structured data*. Public repositories, for instance, provide semi-structured data in the form of reusable artifacts such as scripts, templates, and images (e.g., preconfigured VM images) to operate middleware and application components. Prominent

[7]WordPress requirements: http://wordpress.org/about/requirements
[8]Apache HTTP Server: https://httpd.apache.org
[9]Nginx: http://nginx.org
[10]LAMP stack AMI: http://goo.gl/eVTzAY
[11]LAMP = Linux + Apache + MySQL + PHP
[12]MySQL cookbook: http://supermarket.chef.io/cookbooks/mysql
[13]PHP app. cookbook: http://supermarket.chef.io/cookbooks/application_php
[14]MySQL charm: https://jujucharms.com/precise/mysql-46
[15]Amazon Elastic Beanstalk: http://aws.amazon.com/elasticbeanstalk
[16]Google App Engine: https://cloud.google.com/products/app-engine
[17]Amazon Relational Database Service (RDS): http://aws.amazon.com/rds
[18]Creating a MySQL DB instance on Amazon RDS: http://goo.gl/JwA5Gr

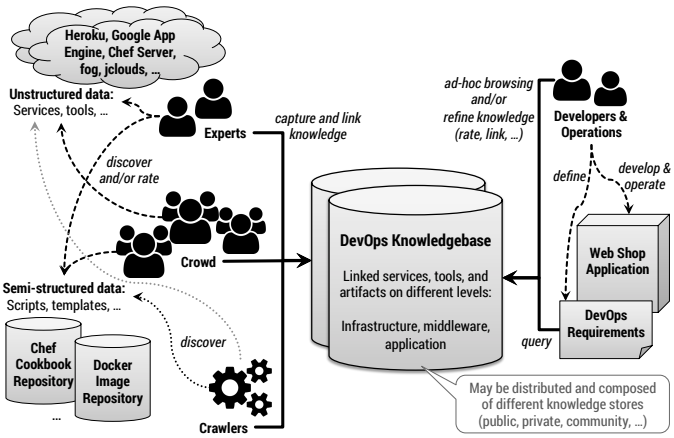


Figure 2. Holistic DevOps knowledge management (overview)

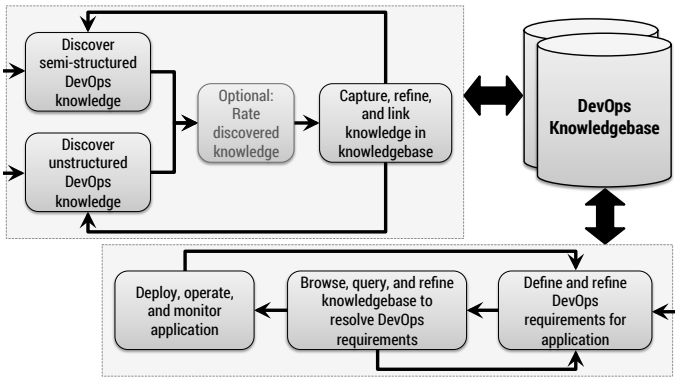


Figure 3. Methodology to manage and use DevOps knowledge

examples are the Chef cookbook repository[19], Puppet Forge[20], Docker Hub[21], and Amazon Web Services' Marketplace[22]. Crawlers can be used to discover such artifacts in an automated manner because the artifacts are annotated with meta data such as dependencies, categories, and input/output data specifications. Ratings associated with these artifacts may be discovered by the crawlers, too.

Discovering unstructured data in a fully automated manner is much more challenging because natural language processing techniques such as document classification [12] need to be utilized. These approaches do not always lead to correct results, so the resulting knowledgebase may become inconsistent quickly. Thus, additional sources of input can be used to discover and rate DevOps knowledge. For instance, experts are able to analyze the documentation and sources of DevOps tools (Chef [6], Puppet [8], etc.) to operate middleware and application components, libraries (fog[23], jclouds[24], etc.) to manage Cloud resources, and services (Heroku[25], Google App Engine, etc.) to utilize middleware-as-a-service offerings. However, discovery performed by humans is not limited to experts. Crowdsourcing approaches [13] may be used alternatively or in a complementary fashion. As an example, the growing open-source community centered around DevOps tools and artifacts such as Chef, Puppet, Juju, and Docker may follow such an approach to systematically consolidate DevOps knowledge. The knowledge discovered by crawlers, experts, and crowdsourcing efforts can then be captured, linked, and optionally refined in a *DevOps Knowledgebase (KB)* in a collaborative manner. This DevOps KB covers different levels of resources such as provisioning libraries on the level of infrastructure, deployment scripts on the level of middleware, and templates on the level of application stacks. As shown in Figure 3, the knowledge discovery and capturing is meant to be continuously repeated to refine and update the DevOps KB.

As shown in Figure 2 and Figure 3, developers and operations personnel define DevOps requirements for a particular application (e.g., WordPress requires MySQL and PHP, as discussed in Section II). These requirements can be used to query against the DevOps KB to find viable options to resolve the requirements. Alternatively, the KB may be browsed, for instance, to get an impression what options are available to host a MySQL database. Then, the DevOps requirements of a particular application may be refined accordingly. Moreover, the knowledgebase can be updated, e.g., by adding deployment scripts for new middleware components required by an application. All these tasks are performed to eventually resolve the DevOps requirements of an application in order to deploy and operate it. By end-to-end monitoring the application, occurring issues on different levels can be identified. For example, if there are problems with a certain middleware component, its DevOps requirements may have to be refined and adapted. As such, the proposed approach promotes the continuous and iterative accumulation, organization, and utilization of knowledge over the lifetime of applications.

IV. DevOps Knowledgebase

In the previous section we proposed a methodology to enable holistic DevOps knowledge management. The DevOps KB is the central component to implement a collaborative knowledge management system. Collaboration based on the KB is not limited to the efficient collaboration between developers and operations personnel. It further includes the collaborative discovery and capturing of DevOps knowledge by experts, crawlers, and crowdsourcing. In this section we focus on the conceptual structure of the DevOps KB to enable collaborative DevOps knowledge management as discussed in the previous section. Technically, the DevOps KB may be composed of multiple, possibly distributed knowledge stores. For instance, there could be public knowledge stores that are maintained by open-source communities. These stores may be focused on artifacts of certain kinds such as Chef cookbooks and Docker images. Private knowledge stores may be maintained by companies or departments for knowledge that is either very specific and/or is not meant to be available outside the scope of one organization. Consequently, the actual KB is a composition of multiple public and/or private knowledge stores.

A. Knowledge Classification

In order to classify existing DevOps tools, artifacts, and services in a systematic way, we propose a set of *base taxonomies* to categorize abstract entities and implementations.

[19]Chef cookbooks: http://supermarket.chef.io/cookbooks
[20]Puppet Forge: https://forge.puppetlabs.com
[21]Docker Hub: https://registry.hub.docker.com
[22]Amazon Web Services' Marketplace: https://aws.amazon.com/marketplace
[23]fog: http://fog.io
[24]jclouds: http://jclouds.apache.org
[25]Heroku: https://www.heroku.com

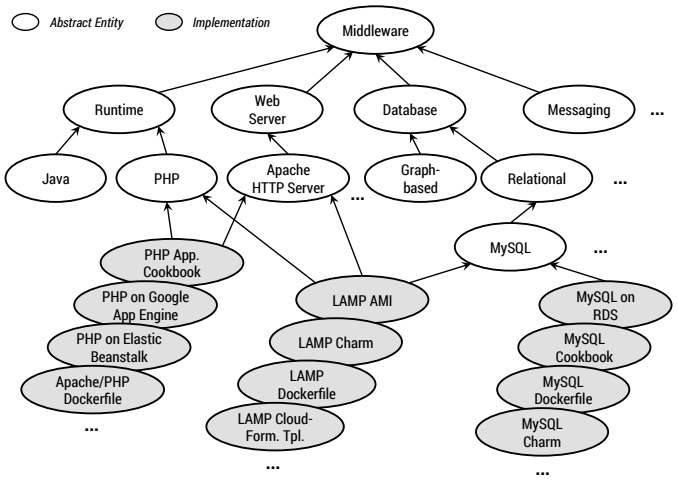


Figure 4. Middleware taxonomy

More specifically, knowledge is organized around middleware, infrastructure, providers, and DevOps automation tooling. Figure 4 presents an extract of our *middleware* taxonomy classifying different kinds of middleware such as *runtimes, Web servers,* and *databases.* These are captured as abstract entities, whereas implementations such as *PHP application cookbook, LAMP Dockerfile,* and *MySQL charm* can be instantiated to resolve specific DevOps requirements. This middleware taxonomy is based on the middleware categorizations of Cloud providers and DevOps tooling providers such as Heroku[26], Google[27], IBM Bluemix[28], and Chef[29].

In addition, an *infrastructure* taxonomy providing categorized infrastructure implementations is required to define dependencies for middleware implementations that are not offered as a service. For instance, the *MySQL cookbook* middleware implementation requires an *operating system* such as *Ubuntu* or *Amazon Linux* to be hosted on, whereas *MySQL on RDS* can be used as a service provided by Amazon RDS without having to resolve further infrastructure requirements. Middleware implementations may not only require infrastructure implementations. Additional tooling may be required by middleware implementations such as *operations* tools (e.g. Chef, Docker, Juju, etc.). Moreover, tooling to *build, test,* and *integrate* middleware and application components may be part of the DevOps requirements for a particular application. All these supporting tools covering both development and operations, as well as integration aspects are categorized using an additional *DevOpsware* taxonomy.

Some middleware implementations are offered as a service by certain providers such as *MySQL on RDS.* An additional provider taxonomy to classify provider implementations such as *RDS* offered by *Amazon.* These provider implementations are linked to the service implementations in the middleware taxonomy to define, for instance, that *MySQL on RDS* is hosted on *RDS*. To capture such relations between implementations (*hosted on, requires,* etc.) and further characteristics in the

[26]Heroku add-ons: https://addons.heroku.com
[27]App Engine features: https://developers.google.com/appengine/features
[28]IBM Bluemix services: http://bluemix.net
[29]Chef cookbooks: http://supermarket.chef.io/cookbooks

Implementation	Properties		
PHP Application Cookbook	*REQUIRES*	=	DevOpsware / Operations / Chef *AND* Middleware / Web Server / Apache Cookbook *AND* ...
	HOSTED_ON	=	Infrastructure / OS / Linux / Ubuntu *XOR* Infrastructure / OS / Linux / RHEL *XOR* ...
PHP on Google App Engine	*HOSTED_ON*	=	Provider / Google / App Engine
	ELASTIC	=	*TRUE*
Apache/PHP Dockerfile	*REQUIRES*	=	DevOpsware / Operations / Docker
	HOSTED_ON	=	Infrastructure / OS / Linux / Ubuntu *XOR* ...
MySQL on RDS	*HOSTED_ON*	=	Provider / Amazon / RDS
	SCALING	=	*MASTER-SLAVE*
MySQL Cookbook	*REQUIRES*	=	DevOpsware / Operations / Chef *AND* ...
	HOSTED_ON	=	Infrastructure / OS / Linux / Ubuntu *XOR* ...
MySQL Charm	*REQUIRES*	=	DevOpsware / Operations / Juju
	HOSTED_ON	=	Infrastructure / OS / Linux / Ubuntu
	SCALING	=	*MASTER-SLAVE*
LAMP AMI	*HOSTED_ON*	=	Provider / Amazon / EC2
LAMP Cloud-Formation Tpl.	*REQUIRES*	=	Provider / Amazon / CloudFormation
	HOSTED_ON	=	Provider / Amazon / EC2 *AND* Provider / Amazon / RDS
Ubuntu Server 14.04 64-bit AMI	*HOSTED_ON*	=	Provider / Amazon / EC2
OpsWorks	*REPLACES*	=	DevOpsware / Operations / Chef
	INT_WITH	=	Provider / Amazon / EC2 *AND* Provider / Amazon / RDS
...			

Figure 5. Properties of implementations

knowledgebase, properties are added as annotations to implementations. In the following we present an initial set of properties to annotate implementations:

- *HOSTED_ON* refers to one or more entities, either infrastructure or provider entities, on which this implementation can be hosted on. For instance, the MySQL cookbook may be hosted on Ubuntu, Amazon Linux, or Red Hat Enterprise Linux (RHEL).
- *REQUIRES* refers to one or more entities, most likely DevOpsware entities, that are needed to operate this implementation. For instance, the MySQL cookook requires Chef.
- *REPLACES* refers to one or more entities that can be replaced by this entity. For instance, Amazon OpsWorks can be used to replace a Chef server.
- *INT_WITH* refers to one or more entities that are integrated with this entity. For instance, Amazon OpsWorks is integrated with Amazon RDS.
- *VERSIONS* defines one or more tuples to express which versions of this implementation are supported. For instance, a MySQL implementation supporting two versions: *('MySQL', '5.5'), ('MySQL', '5.1').*
- *SCALING* defines the scaling mode (e.g., cluster, master-slave, etc.) of this implementation if scaling is supported at all.
- *ELASTIC* defines whether this implementation is elastic, meaning if it scales automatically and dynamically depending on the current load.

Figure 5 provides examples of properties for several implementations of different types. For instance, two of the four listed MySQL implementations support scaling in a master-slave manner, meaning additional slave instances can be added

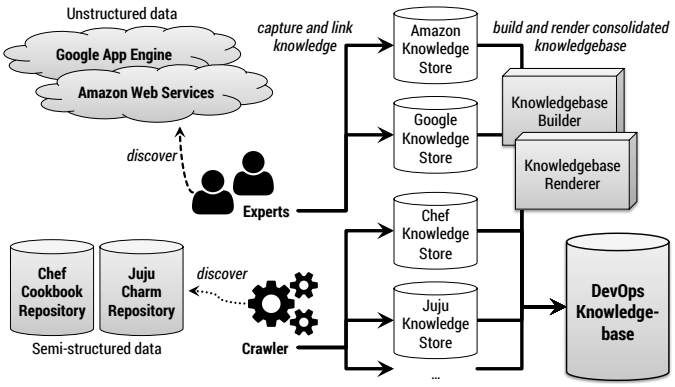


Figure 6. Overview of prototype implementation

as replicas to load-balance read requests between them; the other two do not support scaling. Considering implementations of a PHP runtime, some of them are elastic, whereas others do not have any scaling capabilities. Furthermore, some implementations are bound to certain providers such as *PHP on Google App Engine* or *LAMP AMI;* others only have to be hosted on certain operating systems that may run on arbitrary VMs at any provider or even on-premise. Additional properties may be defined to further characterize implementations. As an example, runtime-as-a-service offerings such as PHP on Google App Engine do not always provide unlimited filesystem access. Moreover, files stored in the filesystem may not be persistent. Such information can be captured using properties because application components may rely on corresponding features depending on their architecture and design.

B. DevOps KB Prototype Implementation

An overview of our prototype implementation of a DevOps KB is presented in Figure 6, following the methodology proposed in Section III and the classification discussed in the previous. More specifically, we have discovered and systematically captured unstructured data from documentations and feature descriptions of Google App Engine and Amazon Web Services in corresponding *knowledge stores*. Currently, each knowledge store is represented as a single YAML[30] file stored in a Git repository. Based on the provider taxonomy discussed in the previous Section IV-A, we captured infrastructure and middleware implementations offered by these two providers. We then categorized these implementations based on the infrastructure and middleware taxonomies outlined before. Furthermore, we implemented a *crawling framework* to automatically discover semi-structured data such as reusable scripts and configuration definitions such as Chef cookbooks from public repositories. Corresponding knowledge stores are automatically generated by the crawler as shown in Figure 6.

In order to eventually build and render a consolidated, interlinked DevOps knowledgebase we implemented a *knowledgebase builder* as a Node.js[31] module to merge individual knowledge stores. This is done by reading the contents of all knowledge stores and merging it into a hierarchically structured database. The technical foundation could be a single file rendered in JSON, XML, or YAML as it is implemented in our first prototype. But it could also be based on a database server such as a graph-based database server[32] or a relational database server to use and implement more efficient query mechanisms. The resulting DevOps KB currently holds roughly 4000 implementations, including 1430 Chef cookbooks, 2190 Puppet modules, and 278 Juju charms captured as middleware implementations by the crawling framework; the rest are additional implementations of types infrastructure, provider, and middleware derived from provider offerings as well as DevOpsware implementations. Finally, we implemented a *knowledgebase renderer* as a Node.js module to render the knowledgebase in different formats such as JSON, YAML, and XML. This eases the usage of the knowledgebase in very different contexts. Both the knowledgebase builder and the renderer can be used programmatically and through a command line interface.

V. DevOps Knowledge Utilization

In order to logically specify DevOps requirements for an application such as WordPress as outlined in Section II, predicates can be defined and combined in the form of Boolean expressions. These expressions can then be utilized as queries against the DevOps KB. Let's assume $\mathcal{E}$ is the domain of all entities captured in the taxonomies that are part of our DevOps KB. The predicate $P_{requires} : \mathcal{E} \rightarrow \{true, false\}$ then assigns each entity (abstract entity or implementation) a Boolean value. The $P_{requires}$ predicate returns $true$ if the given entity is an implementation or there is at least one implementation that inherits from the given entity; otherwise it returns $false$. In addition to $\mathcal{E}$, let's assume $\mathcal{P}$ is the domain of all properties and $\mathcal{V}$ is the domain of all property values; the predicate $P_{propertyEq} : \mathcal{E} \times \mathcal{P} \times \mathcal{V} \rightarrow \{true, false\}$ then assigns each combination of an entity, a property, and a property value a Boolean value. The $P_{propertyEq}$ predicate returns $true$ if the given entity owns the given property with the given value (in the DevOps KB), or there is at least one entity with the given property and value that inherits from the given entity; otherwise it returns $false$. The predicate $P_{propertyEqGr} : \mathcal{E} \times \mathcal{P} \times \mathcal{V} \rightarrow \{true, false\}$ is very similar to $P_{propertyEq}$ but it also returns $true$ if the actual property value is greater than the given value. For instance, this predicate can be used to express version dependencies in the sense of *a particular version or greater is required.* The predicates $P_{propertyEq}$ and $P_{propertyEqGr}$ implicitly cover the $P_{requires}$ predicate because if a certain property of an entity needs to be equal to or greater than a given value, the entity itself is obviously required. As an example, we could use the expression $P_{requires}$(*'Middleware/DB/.../MySQL'*) to specify the need for a MySQL database in our application stack. In case we require specific versions, we can go with a refined expression such as $P_{propertyEqGr}$(*'Middleware/DB/.../MySQL', 'versions', '5.0'*).

Beside expressing application-specific DevOps requirements an organization may enforce further constraints that are valid independent of a specific application. Such constraints, as shown by the following examples, can be expressed as additional requirements: (i) MySQL has to be hosted on an Ubuntu VM; (ii) operating any component on Amazon is forbidden; (iii) using Chef for deployment is forbidden.

[30]YAML Ain't Markup Language: http://www.yaml.org

[31]Node.js: http://nodejs.org

[32]Neo4j graph database: http://www.neo4j.org

Additional predicates may have to be defined to cover such requirements. For instance, we have to define the predicate $P_{excludes} : \mathcal{E} \rightarrow \{true, false\}$ to express that we do not allow Amazon to be involved in any application stack that we operate: $P_{excludes}$(*'Provider/Amazon'*). Such expressions can be merged with application-specific requirements as outlined before, to get a consolidated Boolean expression to be used as a query against the DevOps KB. We use the standardized Web Services Policy Framework (WS-Policy) [14] to render expressions as policies and merge such expressions. Finally, merged expressions can be transformed to WS-Policy normal form (basically a disjunctive normal form) to ease the processing of corresponding expressions and their usage for query purposes. As an example for the WordPress application discussed in Section II we can express its minimum requirements as:

$WordPress_{minimum} =$

$P_{propertyEqGr}$('Middleware/DB/.../MySQL', 'versions', '5.0') $\wedge$

$P_{propertyEqGr}$('Middleware/Runtime/PHP', 'versions', '5.2.4') $\wedge$

$P_{requires}$('Middleware/Web Server')

The $WordPress_{minimum}$ expression can then be evaluated against the DevOps KB to see whether there are appropriate implementations to operate WordPress. Moreover, the expression is used to derive possibly multiple alternatives and combinations to operate the application based on the implementations captured in the knowledgebase. A few examples drawn from the prototype implementation discussed in Section IV-B are:

1) *LAMP AMI* (middleware) hosted on *Amazon EC2* (provider).
2) *LAMP charm* (middleware) hosted on *Ubuntu* (infrastructure) hosted on *Amazon EC2* (provider).
3) *PHP application cookbook* (middleware) hosted on *Ubuntu* (infrastructure) hosted on *Amazon EC2* (provider) & *MySQL charm* (middleware) hosted on *Ubuntu* (infrastructure) hosted on *Amazon EC2* (provider).
4) *PHP on Elastic Beanstalk* (middleware) hosted on *Amazon Elastic Beanstalk* (provider) & *MySQL on RDS* (middleware) hosted on *Amazon RDS* (provider).
5) *PHP on Google App Engine* (middleware) hosted on *Google App Engine* (provider) & *MySQL charm* (middleware) hosted on *Ubuntu* (infrastructure) hosted on *Amazon EC2* (provider).

These are just several selected examples how the WordPress application can be deployed and operated. The requirements can be refined to limit the alternatives by additional constraints, e.g., by requiring a scalable MySQL implementation:

$WordPress_{scalableDB} =$

$P_{propertyEqGr}$('Middleware/DB/.../MySQL', 'versions', '5.0') $\wedge$

($P_{propertyEq}$(**'Middleware/.../MySQL', 'scaling', 'master-slave'**) $\oplus$

$P_{propertyEq}$(**'Middleware/.../MySQL', 'scaling', 'cluster'**)) $\wedge$ ...

Similarly, expressions can be further refined by adding constraints such as $P_{propertyEq}$(*'Middleware/Runtime/PHP', 'elastic', true*) and $P_{excludes}$(*'Provider/Amazon'*). These expressions are then rendered, merged, and normalized using WS-Policy to be used as queries against the DevOps KB.

VI. Conclusion

The DevOps paradigm is rising in prominence in contemporary information systems as the means for efficient, seamless end-to-end automated software management. The combination of DevOps approaches with Cloud computing solutions enables the rapid provisioning of infrastructure resources on demand. An ever expanding wealth of existing reusable DevOps artifacts and related Cloud services means that application designers and developers have ample opportunities for putting tried and tested DevOps knowledge into practice. However, deciding how to use this knowledge is hindered by the multitude of existing approaches, the scattering of knowledge between different communities and experts, and the width of the available design space in application design. In order to address the need for informed decision making, in the previous sections we presented our proposal for a holistic DevOps knowledge management methodology. More specifically, we identified a set of knowledge sources that can be (publicly) accessed, and the means to harvest the knowledge that is contained in these sources. This can be done in an automated manner based on our crawling approach. Finally, we discussed how to reflect, organize, store, and utilize the knowledge using a DevOps knowledgebase, predicate logic, and WS-Policy. Future work includes the improvement of the knowledge management approach by populating the DevOps KB with more offerings from Cloud providers, the evaluation of crawling techniques for the automatic capturing of such offerings, e.g., based on the providers' API documentations, and finally, we aim to enable the automated generation of alternative application topologies based on our previous research [15].

Acknowledgment

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

References

[1] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[2] J. Humble and J. Molesky, "Why Enterprises Must Adopt Devops to Enable Continuous Delivery," *Cutter IT Journal*, vol. 24, 2011.

[3] M. Walls, *Building a DevOps Culture*. O'Reilly Media, Inc., 2013.

[4] J. Wettinger, U. Breitenbücher, and F. Leymann, "DevOpSlang - Bridging the Gap Between Development and Operations," in *Proceedings of the 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC)*, 2014.

[5] M. Hüttermann, *DevOps for Developers*. Apress, 2012.

[6] S. Nelson-Smith, *Test-Driven Infrastructure with Chef*. O'Reilly Media, Inc., 2013.

[7] S. Günther, M. Haupt, and M. Splieth, "Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures," Very Large Business Applications Lab Magdeburg, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, Tech. Rep., 2010.

[8] J. Turnbull and J. McCune, *Pro Puppet*. Apress, 2011.

[9] T. Uphill, *Mastering Puppet*. Packt Publishing Ltd, 2014.

[10] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *National Institute of Standards and Technology*, 2011.

[11] S. Kächele, F. J. Hauck, C. Spann, and J. Domaschka, "Beyond IaaS and PaaS: An Extended Cloud Taxonomy for Computation, Storage and Networking," 2013.

[12] P. Trinkle, "An Introduction to Unsupervised Document Classification," 2009.

[13] S. Dustdar and K. Bhattacharya, "The Social Compute Unit," *Internet Computing, IEEE*, vol. 15, no. 3, pp. 64–69, 2011.

[14] W3C, "Web Services Policy Framework (WS-Policy), Version 1.5," 2007.

[15] V. Andrikopoulos, S. G. Sáez, F. Leymann, and J. Wettinger, "Optimal Distribution of Applications in the Cloud," in *Proceedings of the 26th Conference on Advanced Information Systems Engineering (CAiSE)*, ser. LNCS. Springer, 2014, pp. 75–90.