Institute of Architecture of Application Systems

Deployment Aggregates - A Generic Deployment Automation Approach for Applications Operated in the Cloud

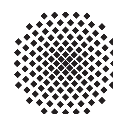
Johannes Wettinger, Katharina Görlach, Frank Leymann

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

BibTeX:

```
@inproceedings{Wettinger2014,
  author    = {Johannes Wettinger and Katharina G{\"o}rlach and Frank Leymann},
  title     = {Deployment Aggregates - A Generic Deployment Automation Approach
               for Applications Operated in the Cloud},
  booktitle = {Proceedings of the 18th International Enterprise Distributed
               Object Computing Conference Workshops and Demonstrations},
  year      = {2014},
  pages     = {173--180},
  publisher = {IEEE Computer Society}
}
```

© 2014 IEEE Computer Society. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.



Universität Stuttgart
Germany

Deployment Aggregates – A Generic Deployment Automation Approach for Applications Operated in the Cloud

Johannes Wettinger, Katharina Görlach, and Frank Leymann
Institute of Architecture of Application Systems (IAAS)
University of Stuttgart, Stuttgart, Germany
{wettinger, goerlach, leymann}@iaas.uni-stuttgart.de

Abstract

One of the most essential requirements to make use of the benefits of Cloud computing is fully automated provisioning and deployment of applications including all related resources. This leads to crucial cost reductions when deploying and operating applications in the Cloud because manual processes are slow, error-prone, and thus costly. Both Cloud providers and the open-source community provide a huge variety of tools, APIs, domain-specific languages, and reusable artifacts to implement deployment automation. However, the meta-models behind these approaches are diverse. This diversity makes it challenging to combine different approaches, avoiding vendor lock-in and tooling lock-in. In this work we propose deployment aggregates as a generic means to use and orchestrate different kinds of deployment approaches. We define a generic meta-model and show its relation to existing meta-models in the domain of deployment automation. Moreover, we discuss how existing artifacts can be used as deployment aggregates as a result of transformation and enrichment.

Keywords

Deployment; aggregate; topology; provisioning; operations; unification; orchestration; transformation; meta-model; Cloud computing; DevOps

1. Introduction

Cloud computing [1], [2] as an emerging paradigm is used by a growing number of enterprises today. New applications are developed as Cloud-native applications [3] and existing applications are migrated into the the Cloud [4], [5]. Not only public Cloud offerings [2] such as Amazon Web Services (AWS)[1] are used to benefit from the advantages of Cloud computing such as pay-per-use and on-demand self-service capabilities. A large number of enterprises and other organizations support open-source and standards-driven initiatives such as OpenStack [6] to establish both private and hybrid Cloud [2] environments.

Cloud providers such as Amazon and Cloud frameworks such as OpenStack provide cost-effective and fast ways to deploy and run applications. However, as of today, there is a large variety of deployment tools and techniques available. They differ in various dimensions, most importantly in the meta-models behind the different approaches. Some use application stacks (e.g., AWS OpsWorks[2] or Ubuntu Juju[3]) or infrastructure topologies (e.g., OpenStack Heat[4]), others use lists of scripts (e.g., Chef run lists[5]) or even PaaS-centric application package descriptions such as Cloud Foundry manifests[6]. This makes it challenging to combine different approaches and especially to orchestrate artifacts published by communities affiliated with the different tools, techniques, and providers. However, this is highly desirable because some communities share a lot of reusable artifacts such as portable scripts or container images as open-source software. Prominent examples are Chef cookbooks[7], Puppet modules[8], Juju charms[9], and Docker images[10].

In this work we analyze meta-models of existing deployment automation approaches to identify their commonalities and differences. Based on this analysis we propose a formalized meta-model for any kind of *deployment aggregates* and instances of them as a generic means to orchestrate different kinds of artifacts and approaches. The key contributions of our work can therefore be summarized as follows:

- We analyze and compare existing deployment automation approaches for applications operated in the Cloud.
- We propose a formalized and generic meta-model based on deployment aggregates to enable the orchestration of different kinds of artifacts and approaches. We show how this generic meta-model is related to existing meta-models, models, and instances.
- We further discuss how existing artifacts can be transformed and/or enriched to become deployment aggregates. Furthermore, we present an algorithm to instantiate deployment aggregates.

The remaining of this paper is structured as follows: Section 2 presents the problem statement that motivates the introduction of deployment aggregates. The meta-model for deployment aggregates and related entities is formally defined in Section 3. Based on that, Section 4 and 5 discuss how to

1. Amazon Web Services (AWS): http://aws.amazon.com
2. AWS OpsWorks: http://aws.amazon.com/opsworks
3. Ubuntu Juju: http://juju.ubuntu.com
4. OpenStack Heat: http://wiki.openstack.org/wiki/Heat
5. Chef run lists: http://goo.gl/cIyROr
6. Cloud Foundry manifests: http://goo.gl/4UlDJk
7. Chef cookbooks: http://community.opscode.com/cookbooks
8. Puppet modules: http://forge.puppetlabs.com
9. Juju charms: http://jujucharms.com
10. Docker images: http://index.docker.io

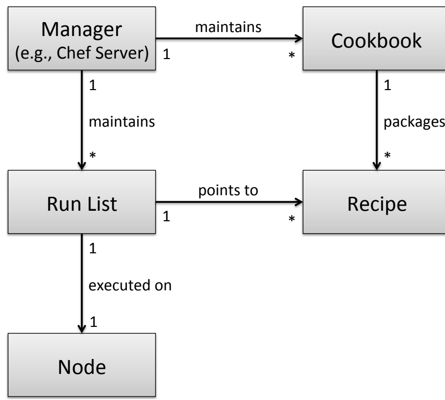


Figure 1. Chef meta-model

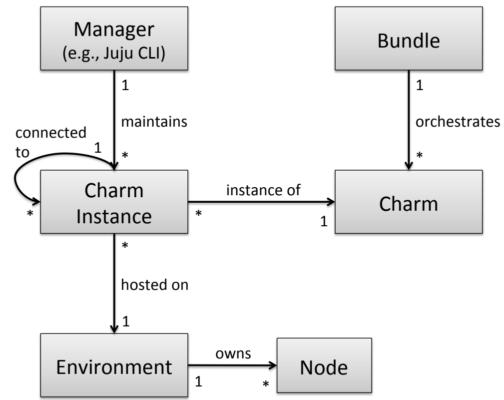


Figure 2. Juju meta-model

build deployment aggregates based on existing artifacts and how to create instances of these. Finally, Section 6 and 7 present related work, conclusions, and future work.

2. Problem Statement

In the introduction (Section 1) we already mentioned a major challenge: different communities publish a lot of reusable, portable artifacts to automate the deployment of applications operated in the Cloud. However, most of the approaches affiliated with the artifacts are not compatible among each other because their underlying meta-models differ. Thus, these artifacts cannot be used and handled in a unified manner because of different invocation mechanisms, state models, parameter passing mechanisms, etc.

As an example, Figure 1 presents a simplified meta-model for Chef [7]: recipes are deployment scripts packaged in cookbooks. A manager entity such as a Chef server maintains the cookbooks that are used. It further maintains a run list for each node, i.e., a physical or virtual machine. The run list points to an arbitrary number of recipes that have to be executed on a particular node. In contrast to Chef, Juju has a different meta-model that is shown in Figure 2 in a simplified form: charms are packages of scripts implementing the lifecycle (install, start, stop, uninstall) of a certain middleware or application component. Multiple charms can be orchestrated using a bundle, i.e., a bundle may be used to model a complete application topology. The manager entity such as the Juju command-line interface (CLI) maintains the instances created based on existing charms. In contrast to Chef, charms are not immediately hosted on single nodes. They are instead hosted on an environment consisting of an arbitrary number of nodes. Such an environment can be used for transparent scaling, i.e., adding additional nodes if required to scale out a particular component.

The differences in the underlying meta-models do not only make it hard to use single artifacts in a unified manner as discussed previously. It is especially challenging to combine multiple approaches seamlessly to orchestrate different kinds of artifacts such as Chef cookbooks, Juju charms, and Docker images. Moreover, there are artifacts that are not immediately deployable or cannot be used directly to automate the deployment of applications in the Cloud. The reasons may be diverse: (i) certain deployment logic is missing such as a script to deploy a particular application component in a given stack. (ii) Even if the artifact is complete and contains the whole deployment logic, in some cases additional interpretations or assumptions are required to deploy the artifact. This could, for instance, apply to a complete application topology without an overarching deployment plan attached. A major goal of our work is to introduce a generic meta-model to be used to tackle the issues discussed previously. The meta-model is based on *deployment aggregates*. Such an aggregate can be any kind of higher-level artifact (e.g., an overarching, orchestrating deployment plan) or lower-level artifact (e.g., a script to deploy a single application component on a VM).

Figure 3 outlines the relations between meta-models, models, and instances following an advanced meta-modeling approach introduced in [8] based on model-driven architectures. The horizontal hierarchy represents the *conventional* meta-modeling levels considering *syntactic instance-of dependencies* between

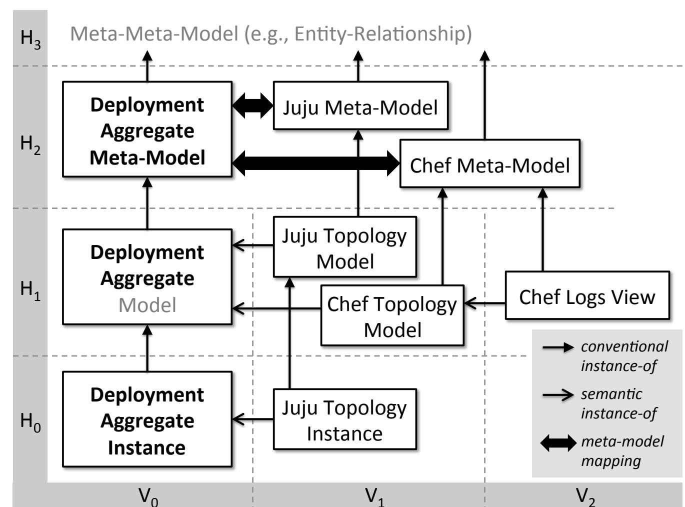


Figure 3. Horizontal and vertical levels of meta-modeling

meta-meta-models (H_3), meta-models (H_2), models (H_1), and instances (H_0). The vertical hierarchy covering the horizontal layers V_0 and V_1 represents an orthogonal *semantic* meta-modeling hierarchy considering *semantic instance-of dependencies.* A semantic instance-of dependency specifies which model content of a semantic meta-model is inherited in which way by a semantic instance. As shown in Figure 3 we do not introduce an overarching meta-meta-model for existing deployment automation meta-models such as the Juju meta-model and Chef meta-model discussed previously. Instead, we introduce a generic meta-model next to the existing meta-models enabling unified representation, aggregation, and orchestration of arbitrary artifacts of different existing meta-models.

The semantic instance-of dependency on the horizontal layer H_1 in Figure 3 illustrates the impact of such a generic meta-model. In particular, a generic deployment aggregate model that is a conventional instance of the introduced generic meta-model is suitable to specify required artifacts for deployment in general, i.e., by specifying deployment aggregates. The model content of the deployment aggregate model on the vertical layer V_0 can be inherited by a model on the adjacent vertical layer V_1. When inheriting the model content it is specialized. For instance, the Juju topology model on V_1 in Figure 3 inherits the model content of the deployment aggregate model on V_0 and specializes the model content considering the capabilities the corresponding meta-model such as the Juju meta-model provides. Consequently, the specialization enables the ability to process the particular model in an appropriate environment. For instance, a deployment aggregate specialized as a Juju topology model may be processed in a Juju runtime environment. If a meta-model does not provide appropriate capabilities to cover all properties of a generic deployment aggregate some of the model content may get lost or implicit while semantically instantiating the generic deployment aggregate model.

In summary, a model on V_1 is a view on the particular generic deployment model using a specific (existing) meta-model and inheriting the model content if possible, i.e., some model content may be lost or implicit, but no model content is added. Continuing this inheritance further vertical layers such as V_2 are possible holding views on the implemented deployment models allowing to abstract specific model content in order to focus on specific aspects of the model. For instance, a Chef logs view may be introduced to analyze all logs produced during a particular deployment.

Automated transformations from generic deployment models (V_0) to specialized models (V_1) are mainly driven by the meta-model mapping specified on the next higher horizontal layer. Additionally, the semantic instance-of dependency specifies which model content is inherited and which model content is lost or made implicit. However, the semantic instance-of dependency has low impact on the transformation from V_0 to V_1 because as much model content as possible should be transferred to a specialized model. In contrast, the automated transformation from an executable model to a view using the same overarching meta-model is mainly driven by the semantic instance-of dependency specifying which model content is inherited by the view and which model content is removed, e.g., for simplification or abstraction purposes. In the following section, we provide a number of definitions and examples to define our meta-model based on deployment aggregates.

3. Deployment Aggregate Fundamentals

This section defines four key entities that are used for further discussions in this paper. These entities are:

- Deployment Aggregate (Definition 1)
- Deployment Aggregate Configuration (Definition 2)
- Deployment Aggregate Instance (Definition 3)
- Pre-Deployment Aggregate (Definition 4)

Moreover, we show how these entities are interrelated and provide some concrete examples for each of them.

Definition 1 (Deployment Aggregate): A *deployment aggregate (DA)* is an immediately deployable artifact or immediately deployable composite of artifacts. We define it as a tuple $DA = (Impl, Deps)$ where $Impl$ is the actual implementation of this DA and $Deps$ is a set of deployment aggregates on which this DA depends.

If $Deps = \emptyset$ the DA does not have any dependencies and thus consists of its own implementation only. In this case we refer to such a DA as an *atomic deployment aggregate*. Definition 1 states that an artifact or an artifact composite has to be *immediately deployable* to be a DA: it means the DA's implementation, plus its dependencies, plus a proper configuration for the DA can be used immediately without any hidden conventions, assumptions, or additional dependencies to create instances of the DA.

The type of a DA and what it represents can be diverse, covering different Cloud service models [2] (IaaS, PaaS, database-as-a-service, etc.). In the following we list a few examples:

1) A script implemented in Ruby using the fog[11] library to provision and manage AWS EC2[12] machines. Beside the fog library this DA depends on another DA providing a Ruby runtime environment.
2) A workflow implemented in BPEL [9] to provision a cluster of EC2 machines. The Ruby script described in (1) can be reused as a dependency for this DA. However, its functionality need to be exposed as a Web service based on WSDL to be able to orchestrate different provisioning and management actions in BPEL.
3) A Chef cookbook to install and configure an Apache Web server on a VM. This DA depends on a Chef runtime environment such as Chef solo [7] and on an operating system that is compatible with the cookbook (e.g., a Linux-based operating system).
4) An AWS CloudFormation template to deploy the whole stack for running WordPress[13] including the PHP runtime

11. fog: http://fog.io
12. Amazon Web Services EC2: http://aws.amazon.com/ec2
13. WordPress: http://www.wordpress.org

environment, the Apache Web server, and the MySQL database server on AWS' infrastructure. This DA depends on another DA (e.g., a script) that uses the command-line interface for CloudFormation to deploy a stack described by a CloudFormation template.

5) A Cloud Foundry manifest to deploy a chat application implemented using Node.js with a chat log database in the background based on MongoDB. This DA depends on another DA providing a Cloud Foundry platform with support for Node.js and MongoDB such as IBM BlueMix[14]. Alternatively, a DA could instantiate a new instance of the platform using the open-source Cloud Foundry framework based on existing IaaS offerings (e.g., Amazon Web Services) to satisfy the dependency.
6) An SQL database dump containing data to be deployed to AWS RDS[15]. In terms of dependencies, two requirements need to be fulfilled by one or more further DAs: (i) the API of AWS RDS needs to be accessed to manage RDS instances; (ii) corresponding database drivers (e.g., PostgreSQL or MySQL) to push the data to a particular RDS instance.
7) A composite of DAs such as the ones described previously to deploy a Web application consisting of multiple components hosted in different environments. This is how different Cloud deployment models [2] such as private Cloud, public Cloud, or hybrid Cloud can be targeted and combined. For instance, some components of the Web application may run on AWS' public Cloud infrastructure, others may run on an OpenStack-based on-premise datacenter.

Technically, DA dependencies can either be packaged with the DA to make a DA truly self-contained. Alternatively, DA dependencies can be expressed using references to resources that can be retrieved from the Web or other sources.

Definition 2 (Deployment Aggregate Configuration): A *deployment aggregate configuration (DAC)* is an arbitrary data structure, e.g., rendered in JSON or XML. It is used in combination with a DA to create a deployment aggregate instance (Definition 3).

Definition 3 (Deployment Aggregate Instance): A *deployment aggregate instance (DAI)* is a concrete instance of a DA. We define it as a tuple $DAI = (DA, DAC, Host, InstanceDeps, Runs)$ where DA is the deployment aggregate itself, DAC is the deployment aggregate configuration, $Host$ is the place where this DAI is hosted on, $InstanceDeps$ is a set of DAIs on which this DAI depend, and $Runs$ is a set of invocations that have been triggered for this DAI.

The DAC, for instance, may define where exactly the DAI is hosted on. In case of an IaaS-based host such as AWS EC2 the number and types of VM instances may be defined (e.g., two 'm1.small' instances or one 'm1.large' instance). In case of a PaaS-based host such as Heroku[16] the number and types of units may be specified (e.g., two '1X dynos' or one '2X dyno'). However, a $Host$ cannot only be a VM (IaaS) or a platform (PaaS). It could also be a container managed by Docker[17], a database instance, or any other kind of environment that can host a DAI.

Each run of a DAI can be represented as a tuple $Run = (Conf, Res)$ where $Conf$ is the configuration and Res is the result of this run. The configuration is the input, whereas the result represents the output of a run. The run's configuration overrides the given DAC of a DAI. If $Conf = undefined$ then the DAC is used as configuration: $Conf := DAC$. The following listing outlines a simple example for a run of a provisioning script for EC2 machines, rendered in JSON:

```
{
 "config": {
  "action": "provisionNewVM",
  "awsAccessKeyId": "...",
  "awsSecretAccessKey": "...",
  "region": "us-east-1",
  "image": "ami-7000f019",
  "flavor": "m1.small"
 },
 "result": {
  "sshUser": "ubuntu",
  "sshKey": "...",
  "publicDns": "ec2-...-amazonaws.com"
 }
}
```

Definition 4 (Pre-Deployment Aggregate): A *pre-deployment aggregate (PDA)* is an artifact or composite of artifacts that needs to be *transformed* and/or *enriched* using corresponding functions to make it an immediately deployable DA: $\forall\ PDA\ \exists$ function $f : f(PDA) = DA$

Different approaches established in the state of the art may be used to create PDAs. The Topology and Orchestration Specification for Cloud Applications (TOSCA) [10] is an emerging standard to define the structure of a Cloud application as a graph-based topology model consisting of nodes (VMs, middleware, application components, etc.) and relationships between nodes ('hosted on' relations, dependencies, database connections, etc.). Other approaches to create PDAs are Enterprise Topology Graphs [11], Blueprints [12], UML deployment diagrams [13], or any viable, i.e., eventually deployable topology described using a graph-based topology notation [14].

Some PDAs need to be enriched to make DAs out of them. For instance, an arbitrary topology model with all or some plans or scripts missing for the deployment of its different parts needs to be enriched with corresponding artifacts to become a DA or a refined PDA. This may include executables to provision VMs, provision database instances, install and configure middleware as well as application components, etc.

14. IBM BlueMix: http://ace.ng.bluemix.net
15. Amazon Web Services RDS: http://aws.amazon.com/rds
16. Heroku: http://www.heroku.com
17. Docker: http://www.docker.io

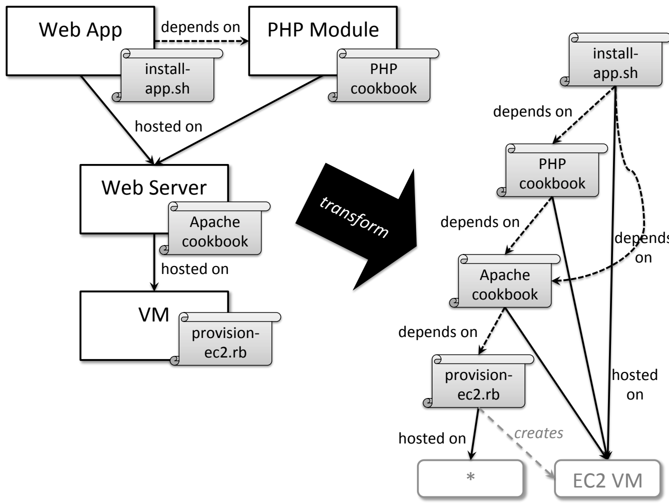


Figure 4. Transformation of application topology (PDA) into deployment topology (DA)

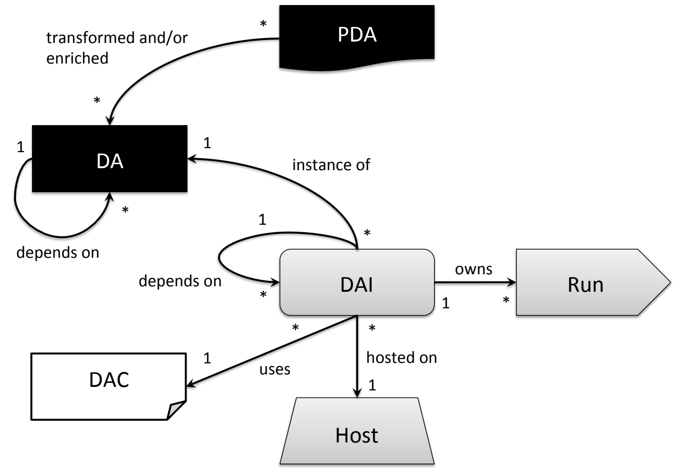


Figure 5. Proposed generic meta-model for deployment aggregates and related entities

An example for a PDA requiring a transformation is a topology model specifying the structure of a Cloud application stack with some abstractions in it. Figure 4 outlines such a transformation. The original application topology defines the topological structure of a simple Web application. It includes all artifacts required to provision and deploy all parts of the application stack. However, this topology model cannot be deployed immediately without some assumptions or interpretation. For instance, according to the original topology the Web application with its *install-app.sh* installation script is hosted on a Web server. However, this does not imply that the *install-app.sh* script is executed on the Web server. Actually, the script has to be executed on the underlying VM, but not before the Web server has been installed on it using the *Apache cookbook*. This is why we need to transform the topology to explicitly express these deployment facts, e.g., in the form of a deployment topology as shown in Figure 4.

Beside the dependencies shown in the deployment topology in Figure 4 there are additional dependencies. For instance, a Chef runtime environment is required to execute cookbooks. These dependencies may be satisfied by embedding or referencing additional DAs. However, transforming a PDA into a deployment topology as discussed previously is just one of many options to generate a DA. Another alternative would be to generate a monolithic DA such as a script or a workflow to perform the deployment.

Figure 5 summarizes the generic meta-model for DAs and related entities such as DAIs, DACs, and PDAs defined in this section. In particular, we saw that PDAs cannot be deployed immediately because transformation and/or enrichment is required to make them immediately deployable DAs. Based on that, in the following Section 4 we discuss how to process PDAs accordingly.

4. Processing Pre-Deployment Aggregates

The process of transforming and/or enriching PDAs to become immediately deployable DAs can be either (semi-)automatic or manual. In case of a semi-automatic or manual process there may be a decision support system involved. Alternatively, a DA can be generated or manually created from scratch without creating a PDA at first. However, in this section we focus on processing PDAs toward DAs.

In the previous section we discussed Figure 4 showing a sample transformation of an application topology (PDA) into a deployment topology (DA). Figure 6 shows another example consisting of two enrichment steps. The original PDA gets enriched by several artifacts such as scripts and cookbooks to provision and deploy all the parts involved in the application topology. Then, the resulting PDA gets further enriched by an overarching artifact to make it eventually a DA. Such an overarching artifact could be a deployment plan that orchestrates the scripts and cookbooks, or it could be a deployment engine that interprets the topology and triggers corresponding actions to provision and deploy all parts of the topology.

Let's assume $\mathcal{P}$ is the space of all PDAs and $\mathcal{D}$ is the space of all DAs: functions can be defined and implemented such as $enrich_1 : \mathcal{P} \rightarrow \mathcal{P}$ to enrich a TOSCA topology model with corresponding scripts or $transform_1 : \mathcal{P} \rightarrow \mathcal{D}$ to transform an application topology into a deployment topology. These functions are used to refine PDAs or to create DAs based on them using enrichment and/or transformation. Related works [15], [16] present approaches to automatically generate deployment plans for given application topology models. These approaches could be implemented as functions to enrich PDAs.

Such functions can be chained to consolidate multiple transformation and enrichment steps. For instance, the two functions mentioned before can be chained to create a function that directly transforms a TOSCA topology model into a deployment topology: $tosca2DeplTopology = enrich_1 \circ transform_1$.

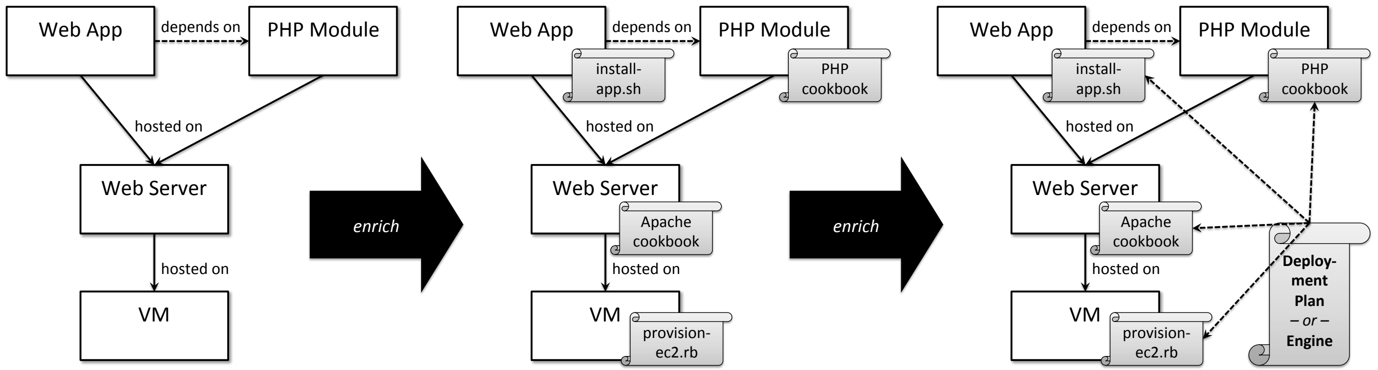


Figure 6. Example for multi-step processing of pre-deployment aggregates

5. Instantiating Deployment Aggregates

Based on the definitions given in Section 3, let's assume $\mathcal{D}$ is the space of all DAs, $\mathcal{C}$ is the space of all DACs, $\mathcal{H}$ is the space of all hosts, and $\mathcal{I}$ is the space of all DAIs: the function $instantiate : \mathcal{D} \times \mathcal{C} \times \mathcal{H} \rightarrow \mathcal{I}$ assigns each combination of DA, DAC, and host a concrete instance (DAI). As a precondition for an algorithm implementing the $instantiate$ function we assume that there are no cyclic dependencies between DAs.

Definition 5 (DA's Dependency Graph): A *deployment aggregate's dependency graph (DG)* is a directed graph representing all dependencies $Deps$ (Definition 1) recursively for a given DA. We define it as a graph $DG_{DA} = (N, E)$ where DA is the deployment aggregate itself, N is a set of nodes n_i representing DAs, and E is a set of directed edges. Each edge (n_i, n_j) represents a dependency between two DAs, meaning DA n_i depends on DA n_j.

Constraint 1 (No cyclic dependencies among DAs): $\forall\ DA : DG_{DA}$ must be acyclic.

A recursive algorithm to implement the $instantiate$ function is presented in Figure 7. It is a depth-first search (DFS) algorithm. As defined by Constraint 1, cyclic dependencies between DAs are strictly forbidden. Otherwise the algorithm may end up in infinite recursion.

Figure 8 shows an example for an acyclic dependency graph for the *Web application deployer* DA, which itself is implemented as Shell script. It depends on several other DAs that are implemented using a variety of deployment tools and scripting languages such as Chef, Juju, and Ruby. Figure 9 shows a sample DAI that results from executing the $instantiate$ algorithm for the *Web application deployer* DA.

Because the $instantiate$ algorithm shown in Figure 7 is a DFS algorithm, its worst-case time complexity is linear based on the total number of dependencies: $\mathcal{O}(n), n = |E_{DG_{DA}}|$. This is because we iterate over all dependencies recursively, instantiate them, and finally instantiate the DA itself.

```
 1: function INSTANTIATE(DA, DAC, Host)
 2:
 3:     DAI ← new Deployment Aggregate Instance
 4:
 5:     DAI.DA ← DA
 6:     DAI.DAC ← DAC
 7:     DAI.Host ← Host
 8:
 9:     DAI.InstanceDeps ← new Set
10:     DAI.Runs ← new Set
11:
12:     for all DepDA in DA.Deps do
13:         DepDAC ← DAC.getDepConf(DepDA)
14:         DepHost ← DAC.getDepHost(DepDA)
15:
16:         DepDAI ← INSTANTIATE(DepDA,
17:         DepDAC, DepHost)
18:
19:         DAI.InstanceDeps.add(DepDAI)
20:     end for
21:
22:     Access ← Host.connect(DAC)
23:     Exec ← DA.Impl.execute(DAC, Access)
24:
25:     Result ← Exec.getResult(Access)
26:
27:     Host.disconnect()
28:
29:     InitialRun ← new DAI Run
30:     InitialRun.Conf ← DAC
31:     InitialRun.Res ← Result
32:
33:     DAI.Runs.add(InitialRun)
34:
35:     return DAI
36:
37: end function
```

Figure 7. Algorithm to create DAIs

6. Related Work

In the previous Sections 3, 4, and 5 we defined and explained our generic meta-model for the domain of deployment

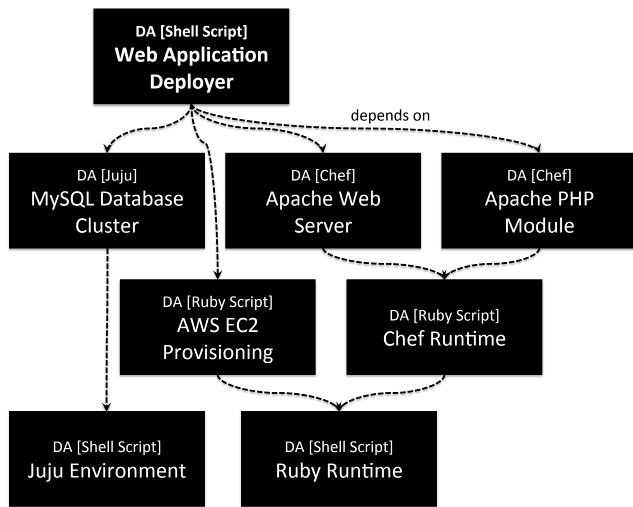


Figure 8. Sample DA: Web App. Deployer

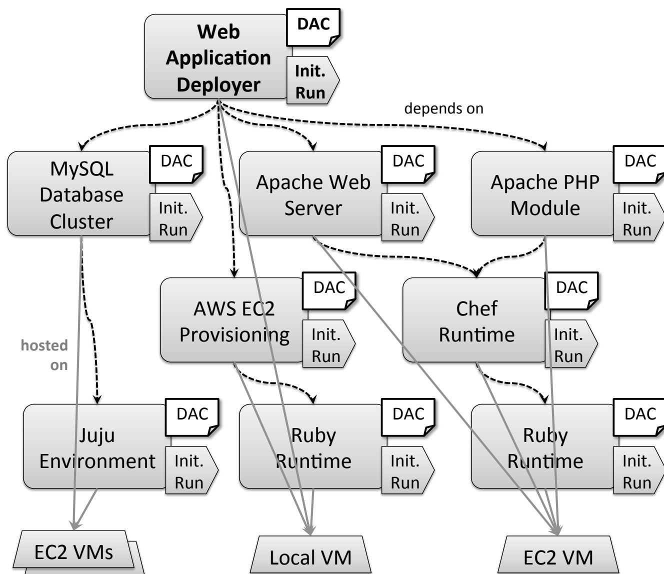


Figure 9. Sample DAI based on Web App. Deployer DA

automation based on deployment aggregates. Regarding the generic meta-modeling approach there is related work in the field of service science: [17] proposes a unified meta-model for executing service compositions. Moreover, choreographies of Web services are formalized in [18], providing a generic meta-model for such choreographies.

TOSCA [19] is an emerging higher-level standard in the field of Cloud management and operations. A major goal of TOSCA is to provide a higher-level abstraction based on application topologies and thus can be used to integrate different lower-level deployment automation approaches [20], [21] such as Chef or Puppet. Consequently, TOSCA's XML schema-based meta-model may be used as a generic meta-model for the deployment domain. However, especially pure TOSCA topology models in the sense of an application topology are not immediately deployable because they may require some interpretation or conventions. Thus, we need to transform and/or enrich them before deployment as shown in Figure 4 and 6.

Furthermore, TOSCA is completely built on the idea of creating topology models and management plans implemented as workflows to operate an application in an automated manner. However, deployment automation could also be implemented as a hierarchical collection of scripts, as a monolithic compiled program, or even as a holistic declarative configuration. The meta-model for deployment aggregates provides a generic and recursive way to use and orchestrate very different kinds artifacts in a unified manner.

PaaS frameworks such as Cloud Foundry[18] and Stratos[19] implicitly define a generic meta-model to be used to run and deploy different kinds of middleware and application components in a developer-centric manner. However, certain plugins may have to be developed to extend the framework, more or less following a strict programming model and using given APIs to integrate with a particular framework. The concept of deployment aggregates tries to minimize such restrictions and framework dependencies as much as possible.

In the field of multi-cloud applications there are related approaches dealing with unified APIs and management interfaces. Some of them stay on the level of IaaS [22], others are more holistic [23], [24] including the PaaS level, too. However, they are more focused on providing some kind of a central manager exposing such APIs to deploy and manage middleware and application components in a unified manner. Such a manager could be used as an intermediary DA to provide an environment as DAI to enable the execution of other DAs depending on such APIs.

7. Conclusions and Future Work

In this paper we introduced a generic meta-model centered around deployment aggregates to implement automated deployment of applications. We showed how different kinds of existing deployment automation approaches and artifacts can be used and orchestrated using deployment aggregates. Moreover, we presented concepts how existing, not yet deployable artifacts (pre-deployment aggregates) can be transformed and/or enriched to make deployment aggregates out of them. Finally, an algorithm was shown to instantiate deployment aggregates.

In terms of future work we plan to implement a holistic framework to manage deployment aggregates and instances of these. Based on this framework we will conduct a thorough evaluation to demonstrate the seamless orchestration and unified handling of deployment aggregates based on existing artifacts of different kinds. Furthermore, we plan to create a meta-model for deployment topologies as a generic technical means to build deployment aggregates. Technically, such deployment topologies may be rendered using XML, YAML, JSON, etc.

18. Cloud Foundry: http://cloudfoundry.org
19. Apache Stratos: http://stratos.incubator.apache.org

References

[1] F. Leymann, "Cloud Computing: The Next Revolution in IT," in *Photogrammetric Week '09*. Wichmann Verlag, 2009.

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

[3] B. Wilder, *Cloud Architecture Patterns*. O'Reilly Media, Inc., 2012.

[4] F. Leymann, C. Fehling, R. Mietzner, A. Nowak, and S. Dustdar, "Moving Applications to the Cloud: An Approach Based on Application Model Enrichment," *International Journal of Cooperative Information Systems*, vol. 20, no. 3, p. 307, 2011.

[5] T. Binz, F. Leymann, and D. Schumm, "CMotion: A Framework for Migration of Applications into and between Clouds," in *2011 IEEE International Conference on Service-Oriented Computing and Applications*. IEEE, 2011.

[6] K. Pepple, *Deploying OpenStack*. O'Reilly Media, 2011.

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

[8] K. Görlach and F. Leymann, "Orthogonal Meta-Modeling," *Journal of Systems Integration*, vol. 5, no. 2, pp. 3–17, 2014.

[9] OASIS, "Web Services Business Process Execution Language (BPEL) Version 2.0," 2007.

[10] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, "Portable Cloud Services Using TOSCA," *Internet Computing, IEEE*, vol. 16, no. 3, pp. 80–85, 2012.

[11] T. Binz, C. Fehling, F. Leymann, A. Nowak, and D. Schumm, "Formalizing the Cloud through Enterprise Topology Graphs," in *Proceedings of 2012 IEEE International Conference on Cloud Computing*. IEEE Computer Society Conference Publishing Services, 2012.

[12] M. Papazoglou and W. van den Heuvel, "Blueprinting the Cloud," *Internet Computing, IEEE*, vol. 15, no. 6, pp. 74–79, 2011.

[13] OMG, "Unified Modeling Language (UML), Version 2.4.1," 2011.

[14] 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 2014)*, ser. Lecture Notes in Computer Science (LNCS). Springer, 2014.

[15] R. Mietzner, "A Method and Implementation to Define and Provision Variable Composite Applications, and its Usage in Cloud Computing," Dissertation, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2010.

[16] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA," in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*. IEEE Computer Society, 2014, pp. 87–96.

[17] K. Görlach, F. Leymann, and V. Claus, "Unified Execution of Service Compositions," in *Proceedings of the 6th IEEE International Conference on Service Oriented Computing & Applications (SOCA 2013)*. IEEE Computer Society Conference Publishing Services, 2013, pp. 162–167.

[18] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo, "Formalizing Web Service Choreographies," *Electronic Notes in Theoretical Computer Science*, vol. 105, pp. 73–94, 2004.

[19] OASIS, "Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, Committee Specification 01," 2013. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html

[20] J. Wettinger, T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, and M. Zimmermann, "Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA," in *Proceedings of the 4th International Conference on Cloud Computing and Services Science*. SciTePress, 2014.

[21] J. Wettinger, M. Behrendt, T. Binz, U. Breitenbücher, G. Breiter, F. Leymann, S. Moser, I. Schwertle, and T. Spatzier, "Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA," in *Proceedings of the 3rd International Conference on Cloud Computing and Services Science*. SciTePress, 2013.

[22] T. Liu, Y. Katsuno, K. Sun, Y. Li, T. Kushida, Y. Chen, and M. Itakura, "Multi Cloud Management for Unified Cloud Services across Cloud Sites," in *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, 2011, pp. 164–169.

[23] D. Petcu, C. Craciun, M. Neagul, I. Lazcanotegui, and M. Rak, "Building an Interoperability API for Sky Computing," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, 2011, pp. 405–411.

[24] F. Moscato, R. Aversa, B. Di Martino, T. Fortis, and V. Munteanu, "An Analysis of mOSAIC Ontology for Cloud Resources Annotation," in *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, 2011, pp. 973–980.