

Enhancing Cloud Application DevOps Using Dynamically Tailored Deployment Engines

Johannes Wettinger, Uwe Breitenbücher, Frank Leymann
Institute of Architecture of Application Systems (IAAS), University of Stuttgart, Germany
{wettinger, breitenbuecher, leymann}@iaas.uni-stuttgart.de

Abstract

Shortening software release cycles increasingly becomes a critical competitive advantage as today’s users, customers, and other stakeholders expect quick responses to occurring issues and feature requests. DevOps practices and Cloud computing are two key paradigms to tackle these issues by enabling rapid and continuous delivery of applications, utilizing automated software delivery pipelines. However, it is a complex and sophisticated challenge to implement such pipelines by installing, configuring, orchestrating, and integrating the required deployment automation solutions. Therefore, we present a method in conjunction with a framework and implementation to dynamically generate tailored deployment automation engines for specific application stacks, which are packaged in a portable manner to run them on various platforms and infrastructures. The core of our work is based on generating APIs for arbitrary deployment executables such as scripts and plans that perform different tasks in the automated deployment process. As a result, deployment tasks can be triggered through generated API endpoints, abstracting from lower-level, technical details of diverse deployment automation tooling. Beside a quantitative evaluation, we discuss two case studies in this context, one focusing on microservice architectures, the other one considering application functionality and its relation to deployment functionality.

Keywords: Deployment, Deployment Engine, Provisioning, Application Topology, APIfication, DevOps, Cloud Computing, Microservices

1. Introduction

In many of today’s organizations, development and operations are strictly split, e.g., across different groups or departments in a company. They usually follow different goals, have contrary mindsets, and suffer from incompatible processes. This conventional split was mainly implemented to foster clear separation of concerns and responsibility. However, it is a major obstacle for fast and frequent releases of software as required in many environments today. Typically, developers aim to push changes into production quickly, whereas the operations personnel’s goal is to keep production environments stable (Hüttermann, 2012). For this reason, collaboration and communication between developers and operations personnel is mainly based on slow, manual, and error-prone processes. As a result, it takes a significant amount of time to put changes, new features, and bug fixes into production. However, especially users and customers of Web applications and mobile apps expect fast responses to their changing and growing requirements. Thus, it becomes a competitive advantage to enable fast and frequent releases by implementing highly automated processes. However, this cannot be achieved without closing the gap between development and operations. DevOps (Humble, 2011) is an emerging paradigm to bridge the gap between these two groups to enable efficient collaboration.

Organizational and cultural changes are typically required to eliminate this split (Hüttermann, 2012). In addition, the deployment process needs to be highly automated to enable continuous delivery of software (Humble, 2010). The constantly growing DevOps community supports this by providing a huge variety of approaches such as tools and artifacts to implement deployment automation. Prominent examples are Chef (Taylor, 2014), Puppet (Uphill, 2014), and Ansible (Mohaan, 2014). Reusable artifacts such as scripts, modules, and templates are publicly available to be used for deployment automation. Furthermore, Cloud computing (Mell, 2011) is heavily used to provision the underlying resources such as virtual servers, storage, network, and databases on
demand in a self-service and pay-per-use manner. Cloud providers such as Amazon Web Services (AWS) expose APIs to be used in automated deployment processes. However, the efficient and automated deployment of complex composite applications typically requires a combination of various approaches, APIs, and artifacts because each of them solves different kinds of challenges (Breitenbücher, 2013). Moreover, all phases of the DevOps lifecycle as outlined in Figure 1 must be targeted to enable comprehensive automation. Application instances are not only deployed to production environments (ops-centric phase). When developing and testing the application, instances of it have to be deployed, too (dev-centric phase). This is required, for example, to run test cases or to quickly double-check the correctness of recent code changes on a developer’s machine. Consequently, the application is constantly deployed and re-deployed to different environments, so efficient and comprehensive deployment automation is of utmost importance. Furthermore, the deployment logic has to be portable and minimal to seamlessly run in various environments, from a simple developer machine to a distributed, multi-Cloud infrastructure in production. As a major goal, our work aims to dynamically generate tailored deployment engines for individual application stacks by generating APIs for deployment scripts and other executables (APification). These engines are portable and minimal to efficiently run in different environments, i.e., they comprise the deployment logic required for the given application stack. The key contributions of this paper can be summarized as follows:

1) A fully integrated end-to-end method to dynamically generate tailored deployment engines for Cloud applications, covering design time, build time, and runtime.

2) An architectural framework and an implementation based on various open-source building blocks to support all phases and steps of the introduced method.

3) A meta model to enable the systematic selection of reusable artifacts, which are the foundation for generating deployment automation engines.

4) An evaluation that quantitatively analyzes the overhead of generating deployment engines.

5) A case study on applying the presented approach to the emerging paradigm of microservice architectures.

6) A case study on merging deployment and application functionality in the context of generated deployment engines to foster DevOps practices.

Our work presented in this paper is based on previous research that has been published as part of the paper entitled DynTail — Dynamically Tailored Deployment Engines for Cloud Applications (Wettinger, 2015) at the 8th International Conference on Cloud Computing (IEEE CLOUD 2015). For reasons of better comprehension, we repeat the refined contributions of the original paper and present the additional work and research that has been done on top.

The remainder of this paper is structured as follows: Section 2 discusses the problem statement and outlines a motivating scenario. Key fundamentals are presented in Section 3 to understand the dynamic tailoring method for deployment engines discussed in Section 4. The framework, the newly introduced meta model, the framework’s architecture, and the implementation to support the individual phases and steps of the method are presented in Section 5, Section 6, and Section 7. Based on them, Section 8 and Section 9 discuss the evaluation and presents a case study. Section 10 discusses an additional case study, which considers merging deployment and application functionality by applying our approach. Finally, we outline related work in Section 11 and conclude the paper with Section 12.

2. Motivation & Problem Statement

In this section we introduce a Web shop application as motivating scenario and running example for our work. The application structure is outlined in Figure 2 as topology model consisting of the actual Web shop application stack on the left and the associated database stack on the right. In terms of infrastructure, we assume a hybrid Cloud deployment: the application itself is hosted on Amazon (EC2 virtual server), whereas the database resides on an OpenStack environment to avoid outsourcing sensitive data to public Cloud providers. The middleware consists of an Apache HTTP server combined with a PHP module to serve the user interface and run the business logic of the Web shop application. For the database, a MySQL master/slave setup is used to improve the application’s scalability and to enable high availability of the database: data that are written to the master instance are replicated to the slave instances, so reading requests can be load-balanced between slave instances. In case the master instance breaks, a slave instance can be selected to be the new master instance.

In terms of deployment automation of the different parts involved in the application topology, we discuss possible approaches in the following. An overview is provided by Figure 3, which adds diverse deployment automation approaches to the topology model described previously. The Apache HTTP server and the associated PHP modules are deployed using the Chef cookbooks apache2 and php. In addition, we could use the mysql cookbook to deploy a MySQL database server. However, the mysql cookbook does not immediately

---

1 Amazon Web Services (AWS): http://aws.amazon.com
2 DevOps lifecycle: http://newrelic.com/devops/lifecycle
3 Amazon EC2: http://aws.amazon.com/ec2
4 Chef Supermarket: https://supermarket.chef.io/cookbooks
support a master/slave setup as specified in the Web shop application topology. For this reason, we use the mysql charm provided by the Juju community\(^5\) to have this setup covered. On top of the middleware layer, we utilize individually developed Shell scripts to deploy the actual application components. Deployment on the infrastructure layer including the provisioning of virtual machines and networking properties happens through fog\(^6\), a Ruby-based library to interact with APIs of Cloud providers and Cloud management frameworks such as Amazon and OpenStack.

As outlined previously, there are multiple kinds of artifacts and approaches involved in the implementation of the automated deployment of the Web shop application topology. Consequently, a lot of technical details and differences have to be considered when implementing an automated deployment process for that topology. A plain Chef-based or Juju-based deployment engine does not suffice because both Chef and Juju need to be supported by the engine for the middleware deployment in addition to the application-specific Shell scripts. Furthermore, the application topology is deployed in a multi-Cloud fashion with Amazon as a public Cloud provider and OpenStack as a private Cloud management platform involved. General-purpose deployment engines as, for instance, discussed in (Wettinger, 2014a) that support a multitude of deployment approaches and technologies tackle these issues, but tend to be heavyweight and complex to maintain. This is because of their generic nature to support a huge and potentially ever-growing variety of application topologies, covering different deployment approaches. Moreover, such an engine may become a single point of failure if it is used as a centralized piece of middleware for the deployment of various application topologies in a particular environment. Generic deployment automation frameworks such as OpenTOSCA (Binz, 2013), Terraform\(^7\), and Apache Brooklyn\(^8\) enable the orchestration of different deployment executables and approaches by introducing unified meta models for the purpose of abstraction. However, this abstraction does not happen automatically. Specialized glue code needs to be implemented to make different approaches available through the corresponding meta model.

To tackle the previously described issues, we present an approach to dynamically generate tailored deployment engines for applications using automated APIfication, i.e., generating APIs for different kinds of deployment executables. Such individually generated engines cover exactly the deployment actions required for a certain application topology or a group of related application topologies. An example for such a group could be the specialization and refinement of an application topology as shown in Figure 3 for different target environments: developers typically prefer to run an application on their developer machine by keeping the overhead as low as possible. Consequently, a developer-oriented topology may run all required components in one VM or one Docker container (Turnbull, 2014) to have a lightweight environment, whereas a cluster of VMs would be preferable for a

\(^5\) Juju charms: https://jujucharms.com

\(^6\) fog: http://fog.io

\(^7\) Terraform: https://terraform.io

\(^8\) Apache Brooklyn: https://brooklyn.incubator.apache.org
production environment to ensure high availability. In the following Section 3, we describe the basic concepts and fundamentals of our approach, explaining how the dynamic tailoring targets the previously identified challenges. Furthermore, we define key terms that are relevant for the dynamic tailoring of deployment engines.

3. DynTail Concepts & Fundamentals

As motivated previously in Section 2, individually generated deployment engines are required to efficiently enable the deployment of Cloud applications, especially considering multi-Cloud setups and the combination of differently specialized deployment automation approaches. In the remainder of this paper, we refer to a DynTail engine (i.e., dynamically tailored deployment engine, in short engine) as a portable and executable package of deployment logic that exposes at least one API endpoint to deploy instances of at least one application topology. Although we focus on deployment logic in the following, a DynTail engine may also cover management logic, e.g., to scale certain components that have been deployed.

An application topology, as outlined in Section 2 for the Web shop application, can be technically specified using various languages, such as the Topology and Orchestration Specification for Cloud Applications (TOSCA) (OASIS, 2013; Binz, 2014). The portable packaging of DynTail engines enables their usage in very different environments (development on a laptop, test on a local server, production in the Cloud, etc.), targeting the DevOps lifecycle as discussed in Section 1. In this context, we emphasize the fact that the DynTail engine and any of the created application instances may run in different environments. As an example, the DynTail engine may run directly on a developer machine, whereas the application instance that it creates may run remotely in the Cloud. However, both environments (DynTail engine and application instance) can also be the same such as a developer laptop.

The deployment logic packaged as a DynTail engine is technically implemented by at least one deployment executable (in short executable), which can be any kind of runnable artifact such as a script, a configuration definition, or a compiled program. A deployment executable may implement ‘atomic’ deployment actions, e.g., a deployment script to install and configure certain software packages on a Linux machine, or a piece of code to invoke a Cloud provider API to provision virtual servers. Alternatively or additionally, it composes other deployment executables, e.g., a deployment
workflow implemented in BPMN (OMG, 2011) or a CloudFormation template to invoke and orchestrate several deployment scripts, potentially in parallel. In this context, we refer to a deployment plan (in short plan) as a deployment executable that orchestrates all required deployment executables to create an instance of a certain application topology. Consequently, each DynTail engine exposes an API endpoint to trigger the invocation of such a deployment plan to create instances of the respective application topology. As shown in the example outlined in Figure 4, this API endpoint is either utilized by a user or an external system, e.g., a higher-level scheduler that provisions additional instances of an application depending on the current load. In addition to exposing API endpoints, user interface (UI) endpoints could be provided to ease the interaction between users and the DynTail engine. Alternatively to developing a deployment plan manually, it may be derived from a given application topology model dynamically at runtime (Breitenbücher, 2014).

By following the approach of dynamically generating tailored engines for certain application topologies or groups of related topologies, several benefits appear that help to tackle the issues outlined in Section 2:

- A generated DynTail engine can be minimal by only including deployment executables implementing the deployment actions required to deploy a specific application topology. Consequently, it provides an optimized performance due to minimal resource consumption and minimal setup efforts.

- A generated DynTail engine can be optimized for the deployment of a given application topology in terms of which kind of API is exposed (REST, SOAP/WSDL, JSON-RPC, etc.), how it is packaged (Docker container, VM image, etc.), and further aspects.

- Generated DynTail engines are independent of each other because they package all required deployment executables for deploying a particular application in a self-contained manner. Consequently, they do not rely on centralized, self-hosted middleware components, e.g., a central management service bus (Wettinger, 2014a; Wettinger, 2014b). As a result, these engines are more robust by avoiding a single point of failure as it would be implied by a centralized middleware component.

- Glue code that is required for exposing the functionality of deployment executables through APIs does not have to be developed and maintained manually as it is generated automatically.

- Consequently, contents of existing, diverse open-source ecosystems providing reusable artifacts such as Chef cookbooks, Docker containers, and Juju charms can be utilized and combined without developing custom glue code to make their functionality available through APIs fitting the context of their usage.

- Multi-Cloud (multiple Cloud providers) (Petcu, 2014) and hybrid Cloud deployments (e.g., private and public Cloud) are supported by providing corresponding deployment executables (e.g., to provision and connecting virtual servers) to be used when generating tailored engines.

The following Section 4 presents a method to dynamically generate DynTail engines, targeting the benefits of this approach as described previously.

---

9 CloudFormation: http://aws.amazon.com/cloudformation
4. **DynTail Method**

In this section, we introduce a method to dynamically generate tailored DynTail engines as outlined in Figure 5, namely the DynTail method. The method defines eight steps and follows an abstract and generic approach that can be implemented in various ways using different kinds of technologies. In Section 7, we show one possible realization of the method in the context of a prototypical implementation.

On an organizational level, the method groups the eight steps by three major phases namely design time, build time and runtime. The entry point is the design time phase with its initial step to create and maintain an application topology model or a group of related topology models. After modeling the topology, deployment executables have to be selected that are capable of deploying the modeled infrastructure, middleware, and application components involved. In step 2, these executables can be either developed individually or already available artifacts and frameworks can be used, for example, publicly shared as open-source software such as Chef cookbooks, Juju charms, fog, jclouds, etc. After finding and/or developing the required deployment executables, in step 3 these are attached to the application topologies that were originally created in step 1. The three steps can be arbitrarily repeated to eventually end up with properly designed topology models including the required deployment executables.

Afterwards, the build time phase starts with step 4, in which a deployment plan has to be created. As defined in Section 3, a deployment plan is a deployment executable that orchestrates all required deployment executables to create an instance of a certain application topology model. The creation of this deployment plan can be done either manually, i.e., modeling the plan by hand, or by using a technology that is capable of automatically generating such plans. For example, in previous work, we presented an approach to automatically derive and generate a deployment plan for a given application topology model that is modeled using TOSCA (Breitenbücher, 2014). The generated deployment plan is realized using the standardized BPEL workflow modeling language (OASIS, 2007), which enables using standards-compliant BPEL engines to automatically execute the generated process model. Thus, approaches like this can be utilized to generate deployment plans that are based on established service composition languages such as BPEL, BPMN (OMG, 2011), or domain-specific application management extensions of these languages, e.g., BPMN4TOSCA (Kopp, 2012). Optionally, a generated plan may be refined manually if necessary, e.g., for customization purposes as it is typically required for complex topology models. With creating the deployment plan all required deployment executables are in place, so a DynTail engine can be generated and optionally be refined for customization purposes in step 5. This step typically concludes the build time phase. However, if deployment executables shall be wrapped by APIs, a preliminary version of the DynTail engine needs to be generated (without a deployment plan) to have the APIs for the respective executables in place. This is to know how the API endpoints are structured in detail to enable the automated generation or the manual development of the deployment plan.

---

10 Chef Supermarket: https://supermarket.chef.io

11 Juju charms: https://jujucharms.com

12 fog: http://fog.io

13 jclouds: http://jclouds.apache.org
Finally, the runtime phase is entered through step 6 by provisioning an instance of the generated DynTail engine. As defined in Section 3, the DynTail engine exposes at least one API endpoint to trigger and manage the deployment of application instances based on the originally modeled application topology. This happens in step 7 of the runtime phase. Typically, an invocation of this API endpoint runs the created deployment plan. In case no further application instances need to be deployed and managed, the DynTail engine can be terminated in the final step 8. All three phases, namely design time, build time, and runtime, are linked with each other, i.e., feedback loops are supported to go from the runtime phase back to the build time and design time phase in order to refine application topology models, deployment executables, deployment plans, etc.

As stated before, the DynTail method is abstract and generic on purpose to be implemented in various ways in order to provide the benefits outlined in Section 3. Therefore, in the following section, we provide a framework to implement the phases with their individual steps.

5. DynTail Framework

In this section, we present the DynTail framework that supports applying the DynTail method. The concept of the framework is outlined in Figure 6 and provides a way to implement and support all phases and steps of the DynTail method. For supporting the design time phase, a modeling environment for application topology models such as the TOSCA-based Winery tooling (Kopp, 2013) is required. Other options include Flexiant’s Bento Boxes14 and GENTL (Andrikopoulos, 2014). Such a modeling environment is used to create and maintain application topology models as well as to attach the required deployment executables to the components and dependencies described by these topology models. Optionally, a development environment for deployment executables, such as deployment scripts, is required in case such deployment executables, e.g., custom Shell scripts, are developed individually for a certain application. In order to dynamically and automatically derive a deployment plan for a given application topology model at build time, a deployment plan generator (Breitenbücher, 2014) is required. The plan generator must be able to process the application topology model created before. To decouple the modeling environment from the plan generator, it is highly recommended to utilize standards-based modeling approaches such as TOSCA (OASIS, 2013; Binz, 2014). In case the deployment plan needs to be refined for customization purposes or is created manually from scratch, a corresponding deployment environment is required. The Eclipse BPEL Designer15 is an example for such an environment for BPEL workflows (OASIS, 2007).

The manual creation and adaptation of deployment plans or their manual refinement in case they have been generated is a major issue since typically multiple heterogeneous deployment automation technologies are involved in the deployment of non-trivial applications (Breitenbücher, 2013). Especially when workflow modeling languages such as BPEL are used, integrating various kinds of technologies is a complex and error-prone issue (Breitenbücher, 2015). Therefore, we suggest wrapping the individual characteristics of these technologies and their invocation mechanisms by APIs that can be invoked using standardized protocols such as HTTP. This also supports the automated generation of deployment plans since plan generators do not need to understand the variety of different invocation mechanisms of heterogeneous technologies but only how to invoke APIs. To make the deployment executables and created deployment plans accessible through APIs, an API generator is required. The main purpose of the API generator is to wrap a deployment executable without modifying it and to make its functionality available through an API, hiding and abstracting from the technical details of their implementation, i.e., how to invoke the executable, how to pass input parameters, and how to collect output data. Such a generated API implementation does not only provide the API endpoint but also includes additional logic, e.g., to run the wrapped executable remotely using SSH. Consequently, the generated API implementation significantly enhances the scope of how the underlying deployment executable can be used. Automating the creation of such APIs is especially required if multiple application-specific deployment executables are involved because manually implementing APIs to wrap these executables for a single application only is not efficient.

In order to produce a packaged, self-contained DynTail engine including the generated APIs, another building block is required, namely the DynTail engine packager. The output is a DynTail engine as a portable, executable package to be used later at runtime to deploy application instances. Both building blocks, the API generator and the DynTail engine packager can, for instance, be implemented based on an APIfication framework such as Any2API (Wettinger, 2015a). Finally, a runtime environment for packaged DynTail engines is required to provision the DynTail engine and then utilize its APIs to deploy and manage application instances. The kind of runtime environment depends on the packaging format of the DynTail engine. Portable virtualization approaches, such as Docker (Turnbull, 2014; Fink, 2014), can be used for portable packaging and execution of DynTail engines in very different environments and platforms.

6. Selection of Deployment Executables

Step 2 of the DynTail method (Figure 5) deals with selecting or developing deployment executables, respectively. This step is covered by the DynTail framework since deployment

14 Flexiant Bento Boxes: http://goo.gl/8JDk52

15 Eclipse BPEL Designer: http://eclipse.org/bpel
executables for the topology model can either be implemented individually or existing deployment executables may be reused, e.g., in the form of artifacts publicly shared by open-source communities. While the latter option is suited to utilize the maturity and diversity of existing ecosystems such as Chef Supermarket\(^\text{16}\) or Docker Hub\(^\text{17}\), the process of finding and selecting the most appropriate artifacts is a major challenge. This issue results from the lack of a comprehensive knowledge base, which holds diverse and interlinked artifacts including metadata. Currently, each community maintains separate and isolated repositories with limited search capabilities.

We aim to tackle this issue by proposing a meta model that serves as the foundation for such a comprehensive, integrated knowledge base. Figure 8 depicts a UML class diagram representation of the meta model, which is centered around deployment executables. More specific kinds of deployment executables are Chef cookbooks, Juju charms, etc. Deployment requirements are expressed by requiring referenceables. A referenceable is an abstract modeling construct and can either be a concrete deployment executable or a specific capability, which is provided by an executable. In the context of our DynTail approach, deployment requirements result from application topologies. Beside the provides relation, deployment executables point to referenceables, i.e., other deployment executables or capabilities. This is to express that a particular deployment executable requires, recommends, or conflicts with other deployment executables. Finally, a deployment executable does not have to be an atomic entity, but can be composed of existing executables.

Capabilities are hierarchically organized using taxonomies (Wetinger, 2016) as outlined in Figure 7. Such taxonomies are used to systematically categorize deployment executables. The presented meta model in conjunction with corresponding taxonomies can be used to technically implement a comprehensive knowledge base, which provides the foundation for systematically querying and selecting deployment executables. Furthermore, automated crawling techniques (Wetinger, 2016) may be utilized to gather deployment executables and their metadata from diverse existing repositories to populate the knowledge base.

7. Architecture & Implementation

In order to validate and evaluate the practical feasibility of the DynTail framework shown in Figure 6 and the underlying

---

\(^{16}\) Chef Supermarket: https://supermarket.chef.io

\(^{17}\) Docker Hub: https://hub.docker.com
DynTail method, we implemented a prototype based on the building blocks outlined in Figure 9. The prototype encompasses a toolchain, which covers design time, build time, and runtime. Especially portability and extensibility have been considered as major criteria when designing and implementing the toolchain. Therefore, we decided to base the prototype on TOSCA, which provides a standardized meta model to create application topology models. As mentioned previously, the modeling of application topology models during design time is covered by Winery (Kopp, 2013), which also provides a back-end system to manage different types of components and dependencies. Thus, the modeling tool provides a comprehensive and extensible standards-based modeling environment. As a result, by utilizing TOSCA as an emerging standard in the field of Cloud application modeling, the portability gets significantly improved.

To dynamically produce a deployment plan skeleton for a given application topology model, the OpenTOSCA plan generator (Breitenbücher, 2014) can be utilized at build time. Such a plan skeleton may be customized manually to meet the specific deployment needs of the associated application topology. However, for well-known types of components, such as MySQL databases and Apache Web servers, the plan generator is able to generate complete deployment plans that do not need further refinement, i.e., plans that can be executed without manual adaptation. Because the generated deployment plans are based on workflow modeling languages such as BPMN (OMG, 2011) or BPEL (OASIS, 2007), corresponding APIs need to be generated at build time to expose the required functionality of deployment executables that are attached to the components and dependencies in the application topology model. These APIs are generated using the APIfication framework Any2API (Wettinger, 2015a). As an example, SOAP/WSDL-based Web service APIs (W3C, 2007) may be generated for deployment executables to enable a deployment plan implemented in BPEL to properly invoke and handle the underlying executables. Alternatively, RESTful Web APIs (Richardson, 2013) may be generated when implementing a deployment plan using a scripting language such as Python or Ruby in conjunction with libraries such as rest-client18. The generated deployment plan, which itself is a deployment executable, is exposed through an API, too. Thus, combining multiple different topology models is supported by this recursive aggregation concept based on APIs. Finally, Docker (Turnbull, 2014) is used as a portable container virtualization approach to (i) package DynTail engines at build time using Dockerfiles19 and (ii) to execute them potentially anywhere at runtime. The packaging is also performed by Any2API. To provide even more isolation at runtime, e.g., in case a Docker container running a DynTail engine should not be placed directly on a machine, Vagrant (Hashimoto, 2013) may be utilized to run Docker containers inside a dedicated virtual machine.

All parts of the presented toolchain including the modeling tool Winery20, OpenTOSCA’s plan generator21, the Any2API framework22, Docker23, and Vagrant24 are available as open-source software. Thus, the presented prototype provides a full open-source end-to-end toolchain that can be customized arbitrarily for individual needs.

8. Evaluation & Discussion
The APIfication of deployment executables is the key enabler for dynamically generating DynTail engines in the context of the DynTail framework and method as discussed in Section 4 and Section 5: functionality of arbitrary deployment executables is exposed through APIs. On the one hand this approach significantly eases the usage and integration of deployment functionality, on the other hand an additional layer is added, namely the API implementation invoking the executable. This may result in performance degradation at runtime. Moreover, additional overhead occurs at build time because an individual API is generated for each deployment executable. Our current architecture and implementation utilizes Any2API (Wettinger, 2015a) as APIfication framework. We evaluate the impact of the APIfication-related overhead

---

18 rest-client library: http://github.com/rest-client/rest-client
19 Dockerfile: https://docs.docker.com/reference/builder
20 Winery Dockerfile: http://github.com/jojow/winery-dockerfile
21 OpenTOSCA on GitHub: https://github.com/OpenTOSCA
22 Any2API: http://any2api.org
23 Docker: http://www.docker.com
24 Vagrant: http://www.vagrantup.com
Particularly, three publically shared Chef cookbooks (`apache2`, `php`, and `mysql`) to deploy an Apache HTTP server, a PHP runtime environment, and a MySQL database server. In addition, two Ruby scripts are implemented using `fog` (aws-`ec2` and aws-`rds`) to provision a virtual server on the Amazon EC2 Cloud infrastructure and to provision a managed MySQL database server instance using Amazon Relational Database Service (RDS)\(^{25}\). In particular, the aws-`ec2` script provisions a virtual machine running Ubuntu 14.04 (of `m3.medium` size) in the us-east-1 region; the aws-`rds` script provisions a MySQL 5.6 RDS instance (of `db.m3.mediu-`msize`) in the us-east-1 region. We generate a RESTful Web API for each of them and measure (i) how long it takes to generate the API implementation at build time. Furthermore, we perform runtime measurements, i.e., (ii) how long it takes for the deployment executable to run and (iii) how much resources it takes in terms of memory for the deployment executable to run. To analyze the performance overhead of the APIfication approach, we further run the plain deployment executables directly to measure and compare the execution time and memory consumption. The evaluation was run on a clean virtual machine (4 virtual CPUs clocked at 2.8 GHz, 64-bit, 4 GB of memory) on top of the VirtualBox hypervisor, running a minimalist Linux system (boot2docker\(^{26}\)). The processing and invocation of a particular deployment executable was done in a clean Docker-based Debian Linux container with exactly one container running on the virtual machine at a time. We did all measurements at container level to completely focus on the workload that is linked to the executable and the API implementation. To produce representative results, we run each deployment executable 20 times (10 times with, 10 times without API implementation). 5 of each set of 10 runs were initial executions (run in a clean environment without any execution run before), the other 5 times were subsequent executions (run in an environment, in which an initial execution was run before). Table 1 shows the results of our evaluation. Results of subsequent executions are depicted in brackets. The measured average duration to generate an API implementation (in the range from 8 to 33 seconds) is the overhead at build time, including the retrieval of all dependencies of the given executable.

Generally, there is a minor overhead in terms of execution duration and memory consumption at runtime. In most of today’s environments this overhead should be acceptable, considering the significant simplification of using the generated APIs compared to the plain executables. In addition, when using the plain executables directly, much of the complexity hidden by the generated API implementation has to be covered at the orchestration level. So, the overall consumption of resources may be the same or even worse, depending on the selected means for orchestration. Furthermore, instances of API implementations can be reused to run an executable multiple times and potentially in different remote environments. Through this reuse, the overhead can be quickly compensated in large-scale environments.

### 9. Case Study: Microservice Architectures

Microservices (Newman, 2015; Familiar, 2015) are an emerging architectural style to develop complex applications in a strictly modular manner, avoiding monolithic and hard to maintain architectures. Each component is developed, packaged, and deployed as an independent entity, providing a service-based interface to interact with other microservices making up a certain application. Microservices interact among each other through language-agnostic APIs (e.g., HTTP-based REST APIs), so each application component can potentially be implemented based on a different technology stack. Each individual microservice may be deployed on different servers or even different infrastructures or Cloud providers. Consequently, not only the internal application structure is modularized (which is state of the art, e.g., using Java packages or Ruby modules), but also the deployment of application components can be highly distributed. Therefore, this architectural style enables the independent deployment

---

\(^{25}\) Amazon RDS: http://aws.amazon.com/rds

\(^{26}\) boot2docker: http://boot2docker.io
and re-deployment of individual application components (e.g., only the ones that have been changed) without re-deploying the application as a whole. As a result, deployment processes can be much faster and more flexible, enabling rapid and continuous delivery of an application by quickly responding to required changes, occurring problems, and additional functionality. DynTail engines as proposed in this paper are a great fit for microservice architectures. Because each microservice (i.e., application component) is managed and deployed independently, an individually tailored engine can be dynamically generated for each of them. This approach significantly improves the decoupling of microservices by not only treating them as independently deployable entities, but also providing and individually assigning tailored deployment logic to each of them. Consequently, microservices do not share general-purpose deployment facilities and thus are not depending on centralized middleware for deployment purposes. A DynTail engine is minimal and thus comprises exactly the deployment logic required by a certain microservice. Furthermore, proper APIs can be provided by a DynTail engine for each microservice individually. Choosing the most appropriate kind of API can therefore be completely determined by the context and environment of a particular microservice, including developers’ preferences and established practices.

Furthermore, microservice architectures follow the principle of smart endpoints and dumb pipes. Consequently, the middleware is typically lightweight and minimal in the sense of pushing intelligence and complexity (message transformation, routing, filtering, etc.) to the endpoints and using simple middleware solutions such as dumb messaging systems to interconnect microservices. The aim of this approach is to make microservice architectures scale by keeping the maintenance of centralized middleware components as simple as possible. This architecture principle is immediately supported by the DynTail approach: deployment executables to install and configure individually required middleware components are part of a DynTail engine. As a result, such middleware components do not have to be provided as centralized building blocks. Moreover, middleware components can be dynamically tailored (Sáez, 2015) and added to an application stack through corresponding deployment executables.

Another characteristic of microservice architectures is decentralized data management, i.e., each microservice owns its separate data storage. In terms of deployment this implies the installation and configuration of an individual database instead of sharing a monolithic database instance. Since this is yet another middleware component in the application topology, a DynTail engine is able to deploy a corresponding data storage component.

Microservices are typically organized around business capabilities, so the functionality they provide usually has a business meaning. We propose to broaden the definition of a microservice, so technical management capabilities such as deployment and monitoring are covered, too. Such a service does not immediately provide a business capability, but is a key enabler to run microservices, which actually provide business capabilities. In the context of this broadened understanding of microservices, we consider DynTail engines as they are presented in this paper as a particular kind of microservice. Such an engine provides a service-oriented interface to perform deployment tasks and it is minimal in the sense of individually tailored for a specific application topology.

10. Case Study: DynTail+ Engines — Merging Deployment & Application Functionality

We presented DynTail engines in this paper as a means to efficiently bundle deployment logic in the form of deployment executables to eventually create instances of a particular application topology model. Therefore, the focus clearly was on deployment logic and exposing the respective artifacts through automatically generated APIs. However, DynTail engines are conceptually not limited to deployment and management logic but also enable packaging and providing application functionality in the form of automatically generated APIs.

As outlined in Figure 11, a deployment engine may also include application functionality in the form of automatically generated APIs. Referring to our Web shop application scenario presented in Section 2, application functionality of the Web Shop, such as AddUser and GetInventory, can be covered by a generated DynTail engine in addition to the actual deployment functionality. In the illustrated scenario, we generate an API for providing the functionality of adding a user to the database. In this example, the application functionality logic is implemented in the form of an executable Shell script that is wrapped by the generated API and executed on the underlying virtual machine hosted on OpenStack. Thus, by automatically generating APIs for application functionality, the development of loosely coupled components that abstract from individual technical details of the underlying technologies as well as their packaging and deployment gets supported and reduced to a single technology. We refer to such
engines, which contain both application functionality and deployment logic, as DynTail+ engines as outlined in Figure 10 to respect the included implementations concerning application functionality, e.g., in the form of scripts or other artifacts. As a result, the separation between deployment, management, and application functionality gets increasingly blurred, fostering the use of DevOps practices: a single team is not only responsible for implementing application functionality, but also bundling it with corresponding deployment logic to eventually operate instances of the application.

11. RELATED WORK

The DynTail method and framework (Section 4 and Section 5) presented previously focus on generating DynTail engines for individual application topologies. The resulting engines are self-contained and portable and thus do not rely on centralized middleware components at runtime such as general-purpose deployment engines. Related work (Wettinger, 2014a; Binz, 2013; Lu, 2013) proposes approaches to deploy and manage application topologies with deployment executables attached directly, utilizing general-purpose deployment engines. Such engines are typically extensible to deal with a broad variety of deployment executables. Consequently, the additional step of generating an individually tailored engine can be skipped by using such engines, which may reduce the overhead at build time. However, several drawbacks appear compared to the dynamic tailoring approach presented in this paper. A general-purpose engine is often used as centralized middleware component for multiple deployment scenarios. This makes it hard to maintain and potentially a single point of failure. Furthermore, the APIs provided by general-purpose engines are not ideal for all deployment scenarios because there is no ‘one-size-fits-all’ approach. Typically, the ideal solution depends on multiple factors such as existing expertise, established practices, and the utilized orchestration technique. As a result, custom glue code is developed (e.g., in the form of scripts or plugins, potentially hard to reuse) to wrap existing APIs correspondingly. Additionally, general-purpose engines are not minimal because they are not specialized for a given application topology. Consequently, the overhead at runtime is typically higher compared to tailored engines. This makes a significant difference in case the engine is provisioned and used several times in different environments.

The concept of initially provisioning a deployment engine and then using it to deploy application instances is similar to the “bootware” approach (Vukojevic-Haupt, 2013) used in the context of modeling and running scientific workflows. It follows a two-step bootstrapping process, which initially provisions a deployment engine. In the second step, the deployment engine is used to deploy the target environment including all required middleware and application components. However, in contrast to our DynTail approach, these deployment engines are not dynamically generated. They are general-purpose deployment engines, i.e., non-specialized complex systems, which are not specifically tailored for certain application topologies. Further related work (Wettinger, 2014) aims to ease the reuse of contents provided by existing, diverse open-source ecosystems. This is achieved by transforming different kinds of deployment executables toward standards-based artifacts utilizing TOSCA (OASIS, 2013). While this is an efficient approach to cover the design time and build time phases of deployment automation, a general-purpose deployment engine is still required at runtime to cre
ate instances of an application topology. Several Cloud providers offer modeling and orchestration tooling such as Amazon OpsWorks (Rosner, 2013), covering design time, build time, and runtime. In addition, platform-as-a-service (PaaS) offerings such as Heroku (Coutermarsh, 2014) and IBM Bluemix are provided to immediately deploy and manage application instances without explicitly modeling application topologies. These approaches can only be used efficiently in case the whole application is hosted at a single provider. When implementing a multi-Cloud or hybrid Cloud deployment scenario these approaches are generally not feasible. In addition, vendor lock-in appears when sticking to such provider-specific offerings. This makes it hard to move the application or parts of the application to a different provider, considering deployment automation. Moreover, the APIs are predefined by the provider and must be manually wrapped by developing custom glue code in case they are not appropriate for a given deployment scenario.

12. CONCLUSION AND FUTURE WORK

Rapid and highly automated deployment processes are key to shorten software release cycles, which is a critical competitive advantage. Especially for Cloud-based applications such as Web applications, mobile apps, and the Internet of Things, today’s users and customers expect quick responses to occurring issues and feature requests. The leading paradigms of DevOps and Cloud computing help to implement comprehensive and fully automated deployment processes that aim to shorten release cycles by continuously delivering new iterations of an application. We outlined limitations and issues of current deployment automation techniques and proposed an alternative approach to efficiently and dynamically generate DynTail engines for individual application topologies. The core of the approach is based on the automated APIfication of arbitrary deployment executables such as scripts and plans. More specifically, we presented the DynTail method and framework to support the proposed approach, covering design time, build time, and runtime. In addition, we proposed a meta model to enable the systematic selection of reusable artifacts, which are the foundation for generating deployment automation engines. This meta model can form the foundation for a comprehensive knowledge base to streamline the selection and combination of diverse deployment executables. We validated our approach by implementing the DynTail framework based on an end-to-end, open-source toolchain. Furthermore, our evaluation showed that the overhead (at build time and runtime) introduced by the DynTail approach and the APIfication of deployment executables is reasonable for many environments today, considering the significant benefits gained by the proposed approach. Finally, we conducted two case studies showing the applicability of the DynTail approach. The first study considered microservice architectures, which is an emerging architectural style to develop complex applications in a strictly modular manner, while the second study showed how the development of application functionality and executable deployment logic can be combined using our approach.

In terms of future work we plan to widen the scope of this work, considering management tasks beside deployment. As an example, additional executables may be packaged within a DynTail engine and exposed through APIs to cover management actions such as scaling in and out certain application components. Moreover, we plan to extend the implementation of the presented APIfication framework to support a broader variety of APIs for DynTail engines.

13. ACKNOWLEDGMENT

This work was partially funded by the BMWi projects CloudCycle (01MD11023) and SmartOrchestra (01MD16001F) as well as the DFG project SitOPT (610872).

14. REFERENCES


27 IBM Bluemix: http://bluemix.net


the 6th International Conference on Service-Oriented Computing & Applications (SOCA 2013), IEEE.


Authors

**Johannes Wettinger** is a research associate and PhD student at the University of Stuttgart. His research focuses on developing and operating continuously delivered software applications, addressing DevOps collaboration and efficient end-to-end automation. Beside his participation in the EU-funded 4CaaS project and his teaching activities, Johannes is involved in further projects related to Cloud computing.

**Uwe Breitenbücher** is a research associate and postdoc at the University of Stuttgart. His research vision is to improve cloud application provisioning and management through automating the application of management patterns. Uwe was part of the CloudCycle project, which researches portable cloud services with guaranteed security and compliance.

**Frank Leymann** is a full professor of computer science and director of the Institute of Architecture of Application Systems at the University of Stuttgart. His research interests include cloud computing, architecture patterns, workflow and business process management, service-oriented computing, and integration technology. His background includes having been IBM Distinguished Engineer and elected member of the IBM Academy of Technology.