DynTail - Dynamically Tailored Deployment Engines for Cloud Applications

Johannes Wettinger, Uwe Breitenbücher, Frank Leymann

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

DynTail – Dynamically Tailored Deployment Engines for Cloud Applications

Johannes Wettinger, Uwe Breitenbücher, Frank Leymann

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

***Abstract*—Shortening software release cycles increasingly becomes a critical competitive advantage, not exclusively for software vendors in the field of Web applications, mobile apps, and the Internet of Things. Today's users, customers, and other stakeholders expect quick responses to occurring issues and feature requests. DevOps and Cloud computing are two key paradigms to enable rapid, continuous deployment and delivery of applications utilizing automated software delivery pipelines. However, it is a highly complex and sophisticated challenge to implement such pipelines by installing, configuring, and integrating corresponding general-purpose deployment automation tooling. Therefore, we present a method in conjunction with a framework and implementation to dynamically generate tailored deployment engines for specific application stacks to deploy corresponding applications. Generated deployment engines 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 different deployment automation tooling.**

***Keywords*—Deployment, Deployment Engine, Provisioning, Application Topology, APIfication, DevOps, Cloud Computing**

I. 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 own 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 [1]. 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* [2] is an emerging paradigm to bridge the gap between these two groups to enable efficient collaboration.

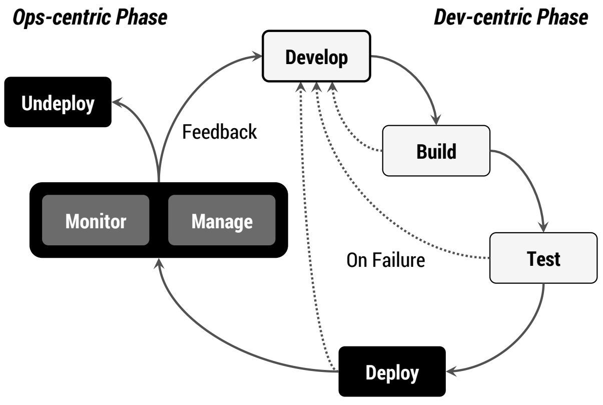

Figure 1. DevOps lifecycle (adapted from [2], [3])

Organizational and cultural changes are typically required to eliminate this split [1]. In addition, the deployment process needs to be highly automated to enable continuous delivery of software [3]. The constantly growing DevOps community supports this by providing a huge variety of approaches such as tools and artifacts to implement holistic deployment automation. Prominent examples are Chef [4], Puppet [5], and Ansible [6]. Reusable artifacts such as scripts, modules, and templates are publicly available to be used for deployment automation. Furthermore, Cloud computing [7] 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)[1] 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 [8]. Moreover, all phases of the DevOps lifecycle[2] 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, e.g., to run test cases or

[1]Amazon Web Services (AWS): http://aws.amazon.com
[2]New Relic's DevOps lifecycle: http://newrelic.com/devops/lifecycle

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 (i.e., comprising exactly the deployment logic required for the given application stack) to efficiently run in different environments. The key contributions of this paper can therefore be summarized by:

- A fully integrated end-to-end method to dynamically generate tailored deployment engines for Cloud applications, covering design time, build time, and runtime.
- An architectural framework and an implementation based an various open-source building blocks to support all phases and steps of the previously introduced method.
- An evaluation to analyze the overhead of generating deployment engines.
- A case study on applying the presented approach to the emerging paradigm of microservice architecture.

The remainder of this paper is structured as follows: Section II discusses the problem statement and outlines a motivating scenario. Important fundamentals are presented in Section III to understand the dynamic tailoring method for deployment engines discussed in Section IV. The framework, its architecture, and the implementation to support the individual phases and steps of the method are presented in Section V and Section VI. Based on them, Section VII and Section VIII discuss the evaluation and a case study. Finally, we outline related work in Section IX and conclude the paper with Section X.

II. 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[3]), 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.

[3]Amazon EC2: http://aws.amazon.com/ec2

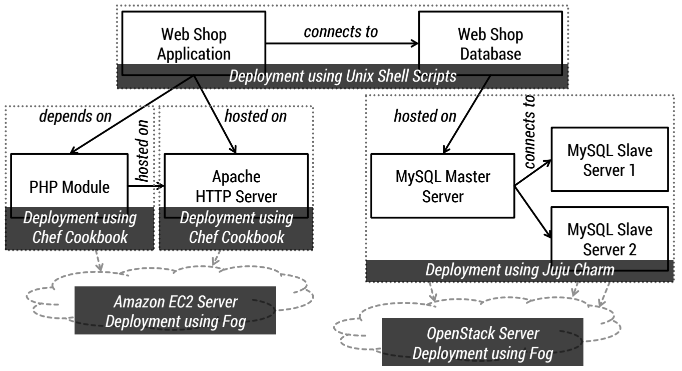
Web Shop Application
connects to
Web Shop Database
Deployment using Unix Shell Scripts
depends on
hosted on
hosted on
hosted on
connects to
PHP Module
Apache HTTP Server
MySQL Master Server
MySQL Slave Server 1
MySQL Slave Server 2
Deployment using Chef Cookbook
Deployment using Chef Cookbook
Deployment using Juju Charm
Amazon EC2 Server Deployment using Fog
OpenStack Server Deployment using Fog

Figure 2. Web shop application topology (adapted from [9])

In terms of deployment automation of the different parts involved in the application topology, the Apache HTTP server and the associated PHP modules are deployed using the Chef cookbooks[4] `apache2` and `php`. In addition, we could use the `mysql` cookbook to deploy a MySQL database server. However, the `mysql` cookbook does not 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 before, 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 [10] 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 [11], Terraform[7], and Apache Brooklyn[8] enable the orchestration of different deployment executables and approaches by introducing

[4]Chef Supermarket: https://supermarket.chef.io/cookbooks

[5]Juju charms: https://jujucharms.com

[6]fog: http://fog.io

[7]Terraform: https://terraform.io

[8]Apache Brooklyn: https://brooklyn.incubator.apache.org

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 2 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 [12] to have a lightweight environment, whereas a cluster of VMs would be preferable for a production environment to ensure high availability. In the following Section III, 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.

III. DynTail Concepts & Fundamentals

As motivated previously in Section II, 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., *dyn*amically *tail*ored 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.* In the following we focus on deployment logic. However, 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 II for the Web shop application can be technically specified using various languages such as the Topology and Orchestration Specification for Cloud Applications (TOSCA) [13], [14]. 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 I. In this context we want to 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,

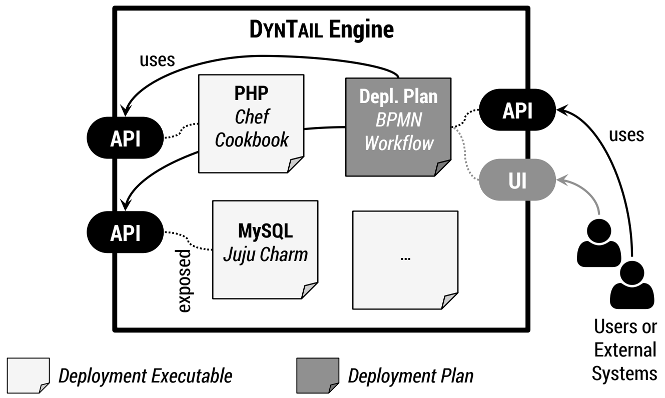

Figure 3. Example for a DynTail engine, deployment plan and executables

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 [15] or a CloudFormation template[9] 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 a deployment plan to create instances of an application topology. As shown in the example outlined in Figure 3, 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 [16].

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 II:

- A generated DynTail engine can be *minimal* by only including deployment executables implementing the deployment actions required by a certain 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 such as a service bus [10]. As a result, these engines

[9]Amazon CloudFormation: http://aws.amazon.com/cloudformation

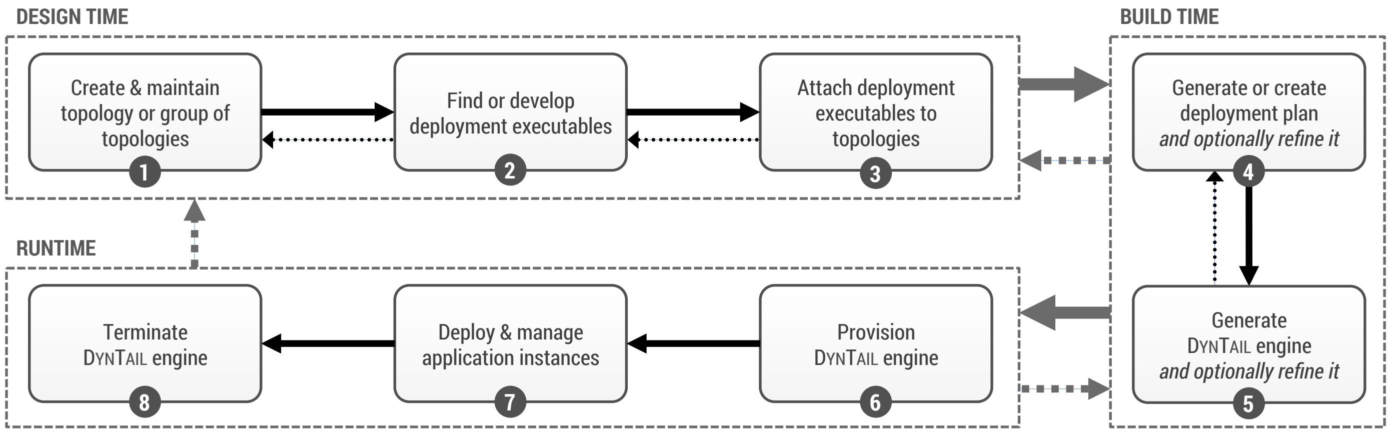
DESIGN TIME
BUILD TIME
Create & maintain topology or group of topologies
1
Find or develop deployment executables
2
Attach deployment executables to topologies
3
Generate or create deployment plan *and optionally refine it*
4
RUNTIME
Terminate DynTail engine
8
Deploy & manage application instances
7
Provision DynTail engine
6
Generate DynTail engine *and optionally refine it*
5

Figure 4. Overview of the DynTail method consisting of the phases *design time*, *build time*, and *runtime*, each comprising multiple steps

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, but 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) [17] 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 IV presents a method to dynamically generate DynTail engines, targeting the benefits of this approach as described previously.

IV. DynTail Method

In this section we introduce a method to dynamically generate tailored engines as outlined in Figure 4, namely the DynTail *method*. It follows an abstract and generic approach that can be implemented in various ways. On a conceptual level there are 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 or a group of related topologies. After modeling a topology, deployment executables are required to deploy all individual infrastructure, middleware, and application components that are involved. In *step 2*, these executables can be either developed individually or already available artifacts and frameworks can be used as they are publicly shared as open-source software (Chef cookbooks, Juju charms, fog[10], jclouds[11], etc.). After finding and/or developing the required deployment executables, in *step 3* these need to be attached to the application topologies that were originally created. The three steps can be arbitrarily repeated to eventually end up with properly designed application topologies including the required application deployment executables.

The next major phase is build time, which starts with *step 4* of generating or manually creating a deployment plan. As defined in Section III, a deployment plan is a deployment executable that orchestrates all required deployment executables to create an instance of a certain application topology. In previous work we presented an approach to automatically derive and generate a deployment plan based on a given application topology [16]. Such approaches can be utilized to generate deployment plans, e.g., based on service composition languages such as BPEL [18]. Optionally, a generated plan can be refined manually if necessary, e.g., for customization purposes as it is typically required for complex application topologies. 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, for some cases, a preliminary version of the DynTail engine needs to be generated (without a deployment plan) to have the APIs for the underlying deployment executables in place. This is to know how the API endpoints are structured in detail to enable the generation or development of deployment plans utilizing these API endpoints.

Finally, the runtime phase is entered through *step 6* by provisioning an instance of the generated DynTail engine. As defined in Section III, 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 deployment plan. In case no further application instances need to be deployed and managed, the DynTail engine can be terminated in *step 8*. All three phases (design time, build time, and runtime) are linked among 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 topologies, deployment executables, etc.

As stated before, the DynTail method is abstract and

[10]fog: http://fog.io
[11]jclouds: http://jclouds.apache.org

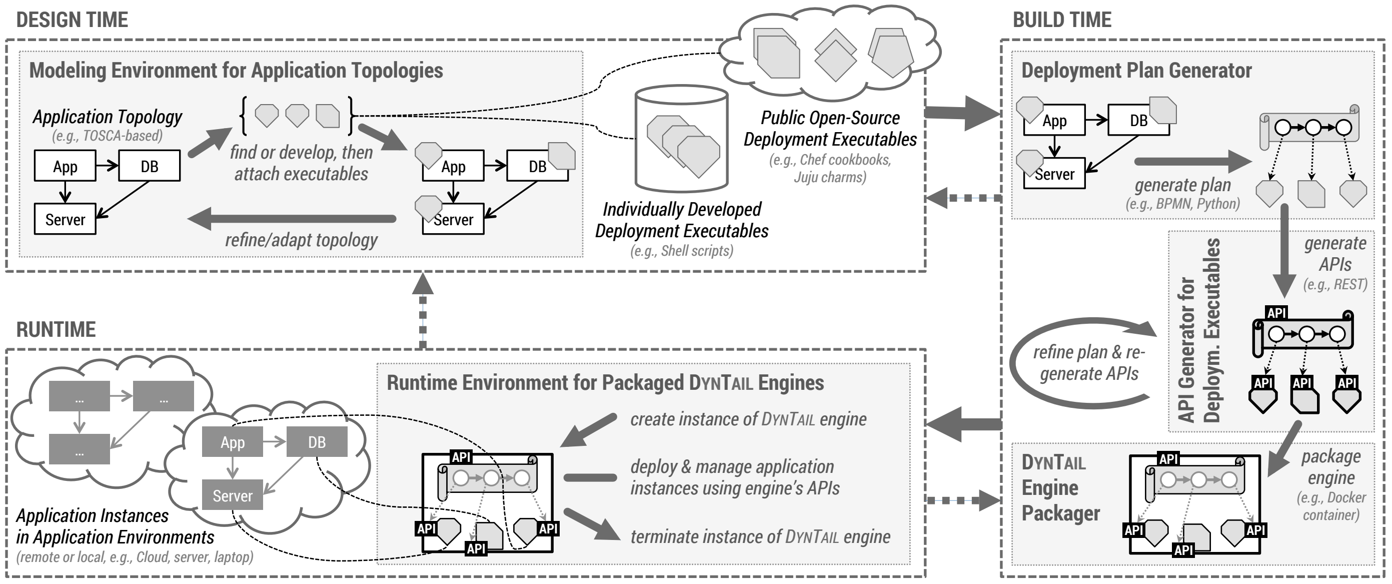
DESIGN TIME
Modeling Environment for Application Topologies
Application Topology
(e.g., TOSCA-based)
App
DB
Server
find or develop, then attach executables
refine/adapt topology
Public Open-Source Deployment Executables
(e.g., Chef cookbooks, Juju charms)
Individually Developed Deployment Executables
(e.g., Shell scripts)
BUILD TIME
Deployment Plan Generator
generate plan
(e.g., BPMN, Python)
generate APIs
(e.g., REST)
API Generator for Deploym. Executables
refine plan & re-generate APIs
DynTail Engine Packager
package engine
(e.g., Docker container)
API
RUNTIME
Runtime Environment for Packaged DynTail Engines
create instance of DynTail engine
deploy & manage application instances using engine's APIs
terminate instance of DynTail engine
Application Instances in Application Environments
(remote or local, e.g., Cloud, server, laptop)

Figure 5. DynTail framework supporting the DynTail method

generic enough to be implemented in various ways in order to provide the benefits outlined in Section III. Therefore, in the following section we provide a framework to implement the phases with their individual steps.

V. DynTail Framework

The DynTail *framework* outlined in Figure 5 provides a way to implement and support all phases and steps of the DynTail method as discussed previously in Section IV. For the design time phase a *modeling environment for application topologies* such as the TOSCA-based Winery tooling [19] is required. Other options include Flexiant's Bento Boxes[12] and GENTL [20]. Such a modeling environment is used to create and maintain application topologies as well as to attach corresponding deployment executables to them. 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. In order to dynamically and automatically derive a deployment plan from a given application topology at build time, a *deployment plan generator* as, for instance, presented in previous work [16] is required. The plan generator must be able to process the application topology created before. To decouple the modeling environment from the plan generator, it is highly recommended to utilize standards-based modeling approaches such as TOSCA [13], [14]. In case the deployment plan needs to be refined for customization purposes or is not generated at all but created manually from scratch, a corresponding development environment is required. The Eclipse BPEL Designer[13] is an example for such a development environment for BPEL workflows [18].

To make the deployment executables, including deployment plans accessible through APIs as outlined in Section III, an *API generator* is required. The main purpose of the API generator is to wrap a deployment executable without modifying it and make its functionality available through an API, hiding and abstracting from the technical details of 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. 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 [21]. 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 [12], [22] or Vagrant [23] can be used for portable packaging and execution of DynTail engines in very different environments on various platforms. In the following Section VI, we present the architecture we used to implement the DynTail framework to support the DynTail method discussed in Section IV.

VI. Architecture & Implementation

In order to validate and evaluate the DynTail framework and the underlying DynTail method we implemented a prototype based on the architectural building blocks outlined in Figure 5. It is based on a toolchain, which covers design time, build time, and runtime. Especially portability and extensibility were considered when implementing and establishing the toolchain. The modeling of application topologies during design time is covered by the Winery tooling [19]. It provides a comprehensive and extensible modeling environment based on

[12]Flexiant Bento Boxes: http://goo.gl/8JDk52
[13]Eclipse BPEL Designer: http://eclipse.org/bpel

the TOSCA standard [13]. By utilizing TOSCA as an emerging standard in the field of Cloud application modeling, the portability gets significantly improved. The OpenTOSCA plan generator [16] is utilized at build time to dynamically produce a deployment plan skeleton for a given application topology. Such a skeleton is customized to meet the specific deployment needs of the associated application topology. Because the deployment plans are based on service composition languages such as BPMN [15] or BPEL [18], corresponding APIs need to be generated at build time to expose the functionality of deployment executables that are attached to an application topology. These APIs are generated using the APIfication framework ANY2API [21]. As an example, SOAP/WSDL-based Web service APIs [24] may be generated for deployment exectuables to enable a deployment plan implemented in BPEL to properly invoke and handle the underlying exectuables. Alternatively, RESTful Web APIs [25] may be generated when implementing a deployment plan using a scripting language such as Python or Ruby in conjunction with libraries such as rest-client[14]. The deployment plan, which is itself a deployment executable, is exposed through an API, too. Finally, Docker [12] is used as a portable container virtualization approach to (i) package DYNTAIL engines at build time using Dockerfiles[15] and (ii) to execute them potentially anywhere at runtime. The packaging is 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 [23] may be utilized to run Docker containers inside a dedicated virtual machine. All parts of the toolchain including Winery[16], OpenTOSCA's plan generator[17], ANY2API[18], Docker[19], and Vagrant[20] are available as open-source software.

VII. 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 IV and Section V: the 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 [21] as APIfication framework. We evaluate the impact of the APIfication-related overhead at build time and runtime. Therefore, we use the ANY2API framework to generate individual API implementations for a set of deployment executables required to deploy the Web shop application topology outlined in Section II. These are in particular, three publicly 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)[21]. 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.medium* size) 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[25]). 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 I 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.

VIII. CASE STUDY: MICROSERVICE ARCHITECTURES

Microservices [26] are an emerging architectural style to develop complex applications in a strictly modular manner, avoiding monolithic architectures that are hard to maintain. Each component is developed, packaged, and deployed as an independent entity, providing a service-based interface

[14]rest-client library: http://github.com/rest-client/rest-client
[15]Dockerfile: https://docs.docker.com/reference/builder
[16]Winery Dockerfile: http://github.com/jojow/winery-dockerfile
[17]OpenTOSCA on GitHub: https://github.com/OpenTOSCA
[18]ANY2API: http://any2api.org
[19]Docker: http://www.docker.com
[20]Vagrant: http://www.vagrantup.com
[21]Amazon RDS: http://aws.amazon.com/rds
[25]boot2docker: http://boot2docker.io

Table I. MEASUREMENTS: OVERHEAD OF API IMPLEMENTATIONS (SUBSEQUENT EXECUTIONS IN BRACKETS)

	`mysql` cookbook	`php` cookbook	`apache2` cookbook	`aws-ec2` script	`aws-rds` script
Avg. duration to generate API implementation	12s (0s)	33s (0s)	13s (0s)	9s (0s)	8s (0s)
Avg. execution duration **with** API implementation	65s (9s)	75s (15s)	40s (10s)	403s (183s)	1136s (827s)
Avg. execution duration ***without*** API implementation	64s (7s)	72s (11s)	35s (10s)	398s (174s)	1139s (819s)
Max. memory usage **with** API implementation	403M (407M)	295M (297M)	242M (243M)	507M (509M)	358M (358M)
Max. memory usage ***without*** API implementation	343M (344M)	230M (231M)	179M (179M)	459M (459M)	282M (281M)
Used ANY2API invoker module for execution	Chef invoker[22]	Chef invoker	Chef invoker	Ruby invoker[23]	Ruby invoker
Used ANY2API generator module to generate API impl.	REST generator[24]	REST generator	REST generator	REST generator	REST generator

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 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.

IX. RELATED WORK

The DYNTAIL method and framework (Section IV and Section V) 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 [10], [11], [27] 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 very 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 [28] 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 [9] 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 [13]. 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 create instances of an application topology. Several Cloud providers offer modeling and orchestration tooling such as Amazon OpsWorks [29], covering design time, build time, and runtime. In addition, platform-as-a-service (PaaS) offerings such as Heroku [30] and IBM Bluemix[26] are provided to directly deploy and manage application instances without explicitly modeling application topologies. These approaches can only

[26]IBM Bluemix: http://bluemix.net

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.

X. Conclusion

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. 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. 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 corresponding 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.

Acknowledgment

This work was partially funded by the BMWi project CloudCycle (01MD11023) and the DFG project SitOPT (610872).

References

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

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

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

[4] M. Taylor and S. Vargo, *Learning Chef: A Guide to Configuration Management and Automation*. O'Reilly Media, 2014.

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

[6] M. Mohaan and R. Raithatha, *Learning Ansible*. Packt Publishing Ltd, 2014.

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

[8] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and J. Wettinger, "Integrated Cloud Application Provisioning: Interconnecting Service-centric and Script-centric Management Technologies," in *Proceedings of the 21st International Conference on Cooperative Information Systems (CoopIS)*, 2013.

[9] J. Wettinger, U. Breitenbücher, and F. Leymann, "Standards-based DevOps Automation and Integration Using TOSCA," in *Proceedings of the 7th International Conference on Utility and Cloud Computing (UCC)*, 2014.

[10] 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.

[11] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA – A Runtime for TOSCA-based Cloud Applications," in *Proceedings of the 11th International Conference on Service-Oriented Computing*, ser. LNCS. Springer, 2013.

[12] J. Turnbull, *The Docker Book*. James Turnbull, 2014.

[13] 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

[14] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, ser. Advanced Web Services. Springer, 2014, pp. 527–549.

[15] OMG, "Business Process Model and Notation (BPMN) Version 2.0," 2011.

[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.

[17] D. Petcu, "Consuming resources and services from multiple clouds," *Journal of Grid Computing*, vol. 12, no. 2, pp. 321–345, 2014.

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

[19] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery - A Modeling Tool for TOSCA-based Cloud Applications," in *Proceedings of the 11th International Conference on Service-Oriented Computing*, ser. LNCS, vol. 8274. Springer Berlin Heidelberg, 2013.

[20] V. Andrikopoulos, A. Reuter, S. Gomez Saez, and F. Leymann, "A GENTL Approach for Cloud Application Topologies," in *Service-Oriented and Cloud Computing*, ser. Lecture Notes in Computer Science. Springer, 2014, vol. 8745.

[21] J. Wettinger, U. Breitenbücher, and F. Leymann, "Any2API - Automated APIfication," in *Proceedings of the 5th International Conference on Cloud Computing and Services Science*. SciTePress, 2015.

[22] J. Fink, "Docker: a Software as a Service, Operating System-Level Virtualization Framework," *Code4Lib Journal*, vol. 25, 2014.

[23] M. Hashimoto, *Vagrant: Up and Running*. O'Reilly Media, 2013.

[24] W3C, "SOAP Specification, Version 1.2," 2007.

[25] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs*. O'Reilly Media, Inc., 2013.

[26] J. Lewis and M. Fowler, "Microservices," 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html

[27] H. Lu, M. Shtern, B. Simmons, M. Smit, and M. Litoiu, "Pattern-based Deployment Service for Next Generation Clouds," 2013.

[28] K. Vukojevic-Haupt, D. Karastoyanova, and F. Leymann, "On-demand Provisioning of Infrastructure, Middleware and Services for Simulation Workflows," in *Proceedings of the 6th International Conference on Service-Oriented Computing & Applications (SOCA)*. IEEE, 2013, pp. 91–98.

[29] T. Rosner, *Learning AWS OpsWorks*. Packt Publishing Ltd, 2013.

[30] M. Coutermarsh, *Heroku Cookbook*. Packt Publishing Ltd, 2014.