Middleware-oriented Deployment Automation for Cloud Applications

Johannes Wettinger, Vasilios Andrikopoulos, Frank Leymann, Steve Strauch
Institute of Architecture of Application Systems (IAAS)
University of Stuttgart, Stuttgart, Germany
{wettinger, andrikopoulos, leymann, strauch}@iaas.uni-stuttgart.de

Abstract—Fully automated provisioning and deployment of applications is one of the most essential prerequisites to make use of the benefits of Cloud computing in order to reduce the costs for managing applications. A huge variety of approaches, tools, and providers are available to automate the involved processes. The DevOps community, for instance, provides tooling and reusable artifacts to implement deployment automation in an application-oriented manner. Platform-as-a-Service frameworks are available for the same purpose. In this work we systematically classify and characterize available deployment approaches independently from the underlying technology used. For motivation and evaluation purposes, we choose Web applications with different technology stacks and analyze their specific deployment requirements. Afterwards, we provision these applications using each of the identified types of deployment approaches in the Cloud to perform qualitative and quantitative measurements. Finally, we discuss the evaluation results and derive recommendations to decide which deployment approach to use based on the deployment requirements of an application. Our results show that deployment approaches can also be efficiently combined if there is no ‘best fit’ for a particular application.

Index Terms—middleware-oriented deployment; application-oriented deployment; Cloud computing; DevOps; decision support

1 INTRODUCTION

Public and relatively cheap infrastructure offerings such as Amazon Web Services\(^1\) are the foundation for the popularity and success of Cloud computing. Following the Infrastructure as a Service (IaaS) delivery model [1], resources such as virtual machines and virtual disks can be provisioned and decommissioned on demand, so the customers only pay for what they are actually using. According to the NIST definition of Cloud computing [1] services provided through the Cloud are not limited to the infrastructure level to provision infrastructure resources such as computing power and storage. There are higher-level service delivery models available, namely Platform as a Service (PaaS) and Software as a Service (SaaS). As the Cloud service market is becoming more mature, these service models gain more traction in the market with further offerings such as the Google App Engine\(^2\) becoming available. In particular in the case of the PaaS model, providers essentially offer Cloud-enabled middleware solutions to their customers. Such solutions can either be provided as middleware services (e.g., database as a service) or reusable middleware components that can be used as a foundation to deploy the actual application components. Different deployment approaches are available in this environment, depending on how the provider-driven automation of the offered middleware solutions is leveraged by application developers. The implementation of holistically automated deployment processes is especially driven by the rapidly emerging DevOps paradigm [2], [3] and continuous delivery practices [4], [5].

The research presented in this article builds on our previous work [6], [7] and focuses on the characterization of these deployment approaches, with the intention of identifying the most efficient deployment of different types of application stacks on PaaS and IaaS solutions. The overall goal of our work is to build the foundation for a decision support system for Cloud application deployment. In addition, this work addresses some of the open issues identified in [6] and [7], in terms of providing an extended evaluation of the discussed approaches. Beside refined measurements and presentation of results, we evaluated the reusability of deployment artifacts when following different deployment automation strategies. Thus, the original findings of [7] are refined and the resulting lessons learned are further refined. Moreover, we provide a comprehensive, systematic analysis and categorization of the state of the art as a foundation for our work. The main contributions of this work can therefore be summarized as follows:

- We present a systematic classification of the state of the art and the limitations of current operations automation approaches.
- Based on this classification, we define and charac-

\(^{1}\) Amazon Web Services: http://aws.amazon.com

\(^{2}\) Google App Engine: https://cloud.google.com/appengine
terize two distinct types of deployment approaches.

- We analyze the deployment requirements of three different applications covering some of the most popular technologies for building Web applications today.
- Based on these requirements, we implement the automated deployment of all three applications using both types of deployment approaches and different Cloud infrastructures for evaluation purposes. This results in eight deployment scenarios for which we measure both qualitative and quantitative properties and we derive a number of findings.
- We present a list of lessons learned based on these findings that can be used to support decision making concerning which deployment approach is more appropriate for a particular application.

The remaining of this article is structured as follows: Section 2 introduces the applications to be used for evaluation purposes and analyzes their specific deployment requirements in order to motivate our work. Based on our investigation of the state of the art in Section 3, two major types of deployment automation approaches are identified in Section 4. Our evaluation of these deployment approaches is presented in Section 5 based on the three applications and their deployment requirements described in Section 2. The evaluation results are discussed in Section 6, including the reporting of our findings. Furthermore, an initial list of lessons learned is derived from these findings. Finally, Section 7 concludes this article and provides an outlook in terms of future work.

2 Motivation

In this section we introduce the three applications that are realized based on different technologies (Section 2.1). These applications are used for the evaluation of the different deployment approaches we propose in this paper. Moreover, we identify application-specific deployment requirements for all three applications (Section 2.2) to be addressed in the evaluation.

2.1 Applications

We chose the applications Taxi Application (Taxi App), SugarCRM, and Chat Application (Chat App) because they cover a set of the most important and established technologies used for Web application development today. These are in particular, Java EE and Web services, PHP and the LAMP stack, and Node.js, respectively. As confirmed by various programming language and technology stack popularity statistics such as GitHub³ and LangPop⁴, these applications target the current TOP 5 technology stacks, and can therefore be considered as representative for a significantly large set of today’s Web applications. In the context of our current research we focus on Web applications as a major class of applications, which is highly relevant for diverse fields such as software as a service, mobile apps, and the Internet of things. The topologies of all three applications consist of middleware components, application components, and external services.

Taxi App – An overview of the Taxi App’s architecture developed in the scope of the European Project 4CaaSt as a demonstrator of the PaaS offerings of the project is shown in Figure 1 [8]. A service provider offers taxi management software as a service to different taxi companies, i.e., tenants. Taxi company customers, who are users of the tenant, submit their taxi transportation requests to the company that they are registered with. The taxi management software (back-end) is realized as a set of business processes using BPEL [9]. The taxi management software leverages context integration processes also implemented in BPEL to retrieve context information about taxi cabs such as location and taxi driver contact details from the 4CaaSt platform-provided Context as a Service. Moreover, Google Maps Web Services [10] provide distance calculations between the pick-up location and the location of the taxi cab. All BPEL processes are deployed in the open source BPEL engine Orchestra⁵ version 4.9.0-M3, which itself is deployed in the Java Open Application Server (JOnAS)⁶ version 5.3.0-M4. The taxi company-specific front-ends consist of a Customer GUI and a Taxi Drivers’ GUI, which are both deployed in JOnAS version 5.3.0-M4. The multi-tenant, open source Enterprise Service Bus ESBRMT [11] as messaging middleware (Figure 1) enables loose coupling and provides a flexible integration solution by avoiding hard-coded point-to-point connections. ESBRMT is based on Apache ServiceMix² version 4.3.0 and comes with three registries realized as PostgreSQL⁸ version 9.1

Figure 1. Architecture of the Taxi Application

3. GitHub: http://github.info
4. LangPop: http://langpop.com
5. OW2 Orchestra: http://orchestra.ow2.org
6. OW2 JOnAS: http://jonas.ow2.org
8. PostgreSQL: http://www.postgresql.org
databases [12].

**SugarCRM** – Figure 2 provides an overview of the architecture of SugarCRM, an open source Customer Relationship Management Software (CRM), which is used for interoperability demonstration purposes by the technical committee of the Topology and Orchestration Specification for Cloud Applications (TOSCA) [13]. All relevant data such as contact details of the customers are stored in the SugarCRM Database Server within a MySQL Database Server version 5.5.32. The SugarCRM Web Application is implemented in PHP and is running on an Apache HTTP Server version 2.2.22 using a PHP runtime version 5.3.10.

**Chat App** – The Chat App’s architecture is presented in Figure 3. The user information and the chat logs are stored in the Chat Log database using a Redis Database Server version 2.6.14. The Chat App is based on Node.js version 0.10.13. All clients (Web browsers) communicate with the Node.js-based chat server using the WebSocket protocol.

In the next section we define the general and application-specific deployment requirements for each of the three applications introduced. These requirements have to be considered when creating corresponding deployment plans. A deployment plan (e.g., a script) implements the logic necessary to deploy application or middleware components.

### 2.2 Deployment Requirements

It can be easily observed that there are requirements that apply to the deployment of all three presented applications. Since the scope of our research is on deployment automation and approaches for its technical implementation we are focusing only on technically relevant deployment requirements. We therefore do not consider additional non-functional requirements regarding performance, costs, compliance, etc. As such, the following general deployment requirements need to be considered in this context:

**GR<sub>1</sub>** *Middleware deployment:* Configurable deployment plans are required to deploy all middleware components involved, such as the JOnAS application server for the Taxi App, or the MySQL database server for SugarCRM.

**GR<sub>2</sub>** *Application deployment:* Configurable deployment plans are required to deploy all application components on top of the middleware, such as the customer user interface for the Taxi App or the chat log database for the Chat App.

**GR<sub>3</sub>** *Wiring of components:* The middleware as well as the application components are not deployed in an isolated manner. Application components need to be wired with each other, or with some of the middleware components to enable communication between components as outlined in Figure 1, Figure 2, and Figure 3.

Further requirements regarding the design of deployment plans such as modularity, configurability, extensibility, and portability are discussed in [6]. In addition to these general requirements, each presented application imposes additional deployment requirements. For the Taxi App [11] we identified the following requirements when we implemented its automated deployment:

**TR<sub>1</sub>** *Deployment of additional tenants:* After the initial deployment has been performed, mechanisms are needed to deploy additional tenants. This involves sending SOAP messages to the Web service-based management interface of ESB<sup>MT</sup> as well as deploying additional WAR files (customer GUI and taxi drivers’ GUI) to the JOnAS application server.

**TR<sub>2</sub>** *Registration and configuration of tenant endpoints:* In order to register and configure tenant endpoints inside ESB<sup>MT</sup> several SOAP messages need to be sent to its management interface both upon initial deployment of the application and when additional tenants are deployed. For the Taxi App these messages are sent by running a number of test cases by SoapUI’s<sup>15</sup> command line-based test runner.

---

10. MySQL Server: http://www.mysql.com
12. Redis: http://redis.io
14. WebSocket: https://www.websocket.org
15. SoapUI: http://www.soapui.org
When automating the deployment of SugarCRM, the deployment plans is easier. There were two requirements identified during the implementation of the deployment plans:

- **Requirement 1**: **Database**: The database is hosted on two different virtual machines. In mind, meaning that the application and the database are hosted on two different virtual machines. When automating the deployment of SugarCRM, the following requirements were identified:

  - **SR1**: **Permission settings**: The permissions of particular files and directories of the PHP application need to be set. As an example, the cache directory needs to be writable. This is a typical requirement for the deployment of PHP applications.

  - **SR2**: **Silent installation**: After extracting and placing all the PHP script files of the application, a so-called silent installation needs to be triggered by calling the `install.php` script using an HTTP GET request. As a result, the database structure is created and the application gets configured by creating a default user (administrator).

  - **SR3**: **Dynamic wiring at deployment time**: The SugarCRM application needs to be wired with the database dynamically at deployment time; the endpoint of the database needs to be put to the application’s configuration, so the connection can be established (this requirement is a refinement of GR3).

Compared to the Taxi App and SugarCRM, the Chat App’s architecture is relatively simple. Thus, automating the deployment based on deployment plans is easier. There were two requirements identified during the implementation of the deployment plans:

- **CR1**: **Node package manager integration**: Node.js applications such as the Chat App typically provide a `package.json` file that holds some metadata of the application as well as its dependencies on other Node.js modules. These dependencies are resolved at deployment time using the node package manager (npm)\(^\text{16}\).

- **CR2**: **Pointer to application’s entry point**: Node.js applications typically consist of several scripts implemented in JavaScript. To start such an application, a particular script needs to be defined as entry point. This script is then called by the Node.js runtime environment to initialize and start the actual application.

Different applications have therefore different requirements regarding their deployment. These individual requirements have to be taken into account when implementing plans to automate the deployment process of a particular application. In principle, there exist different approaches to realize deployment automation such as the one described in [6]. Because there is not a single approach that fits all the different requirements, it is necessary to be able to identify the most suitable approach for each type of application. Thus, in the following we categorize existing approaches to automate operations especially focusing on automated deployment. However, all relevant deployment requirements for a particular application must be satisfied when implementing its automated deployment using any of the categorized approaches, i.e., the identified requirements are orthogonal to the approaches outlined in the following classification.

### 3 State of the Art

In the following we survey various deployment approaches from the literature. We consider each deployment approach as a key part of a certain operations automation approach because operating applications as described in Section 2 needs to cover additional aspects beside deployment. Typical examples include doing a database backup or scaling certain parts of the application. Thus, we consider the automated deployment of middleware and application components as a sub-discipline of operations automation.

Figure 4 categorizes state of the art approaches to implement operations automation. The categories are arranged in a hierarchic structure with leaves grouping the actual automation approaches. Approaches in each leaf group (category) in the hierarchy are further classified to one of three types: (1) a provider-dependent approach such as using a PaaS solution offered by a certain provider (e.g., Heroku\(^\text{17}\)), (2) a tooling-dependent approach such as an abstraction library to interact with multiple providers (e.g., fog\(^\text{18}\)), or a piece of software that does not have any dependencies on certain providers (e.g., Docker\([15]\)). Some of the provider-dependent approaches are implicitly tooling-dependent, too. For instance, Amazon provides a set of tools\(^\text{19}\) to interact and integrate with their services programmatically. (3) Moreover, standards-based approaches exist such as the Topology and Orchestration Specification for Cloud Applications (TOSCA)\([16]\), which enables model-based management of Cloud applications and infrastructure resources.

**Platform-centric management** approaches enable deployment in the PaaS model. These are usually based on provider-dependent platform offerings such as Google App Engine\([17]\) or Amazon Elastic Beanstalk\([18]\). The goal of the PaaS model is to provide a platform that abstracts from the underlying infrastructure resources and provides “middleware

---

16. npm: http://npmjs.org
17. Heroku: http://www.heroku.com
18. fog: http://github.com/fog/fog
as a service”. Thus, the application components are directly hosted on the platform. To host the Taxi App, SugarCRM, or Chat App using the PaaS model, several “middleware services” are required to be exposed by the platform. These middleware services can for example provide an ESB, a BPEL engine, or a database server depending on the topology of the corresponding application. As these middleware services may not be offered out of the box by PaaS providers, a custom PaaS environment can be built based on existing infrastructure resources, e.g., inside the private boundaries of a particular organization. As an example, the PaaS framework Cloud Foundry[20] enables a tooling-dependent platform-centric management approach. All platform-centric approaches have the drawback of restricting control over the underlying infrastructure. For instance, the filesystem cannot be accessed in most cases. Moreover, the programming model is often limited to the use of certain frameworks and APIs. This is necessary in most cases to implement an efficient layer of abstraction for the services offered by the platform.

Higher-level model-based management approaches enable the definition of a holistic application-centric model of a particular Cloud service for the deployment, its structure and behavior. Such a holistic model can be provider-dependent (e.g., AWS OpsWorks [19]), tooling-dependent (e.g., Juju[21] or Blueprints [20]), or standards-based (e.g., TOSCA [16]). In addition, there are commercial products available that implement the model-based approach. An example of these products is the IBM SmartCloud Orchestrator[22]. The goal of the model-based approach is to enable top-down modeling by starting with a higher-level model for a Cloud application. To enable the deployment of such a model, additional artifacts such as scripts may have to be attached to the model to perform the actual deployment of the components that are involved.

Another model-based technique is to create infrastructure-centric models. For instance, AWS CloudFormation[23] enables the creation of provider-dependent models by defining a set of resources such as virtual machines (VMs). Furthermore, scripts can be embedded to install and configure middleware and application components on these machines. However, in contrast to the application-centric models discussed before, these models focus on the orchestration of resources on the infrastructure level and do not explicitly define relations between middleware and application components.

The configuration management approaches can be seen as an alternative or a complement to the model-based approaches. Either plan-based approaches or image-based virtualization techniques can be used to implement operations automation, most probably under some kind of version control. Plans such as scripts are used to install and configure middleware and application components. The image-based approach with respect to the IaaS service model encapsulates the different middleware and application components in VM images. Today, there are many IaaS providers offering the deployment of VM images such as Amazon Web Services (AWS)[24]. In addition, there are open source products such as OpenStack [21] available to create an IaaS environment for deploying VM images. The Open Virtualization Format (OVF) [22] aims to be

Figure 4. Categorized State of the Art Approaches to Implement Operations Automation

---

22. IBM SmartCloud Orchestrator: http://ibm.co/CPandO
configuration management tools such as Chef knife plugins\(^{26}\) to cover infrastructure management aspects, so these two management disciplines can be integrated seamlessly. To reduce dependencies on providers and tools, standards are emerging in all categories. Some examples are Cloud Infrastructure Management Interface (CIMI)\(^{36}\) on the level of infrastructure management, OVF for VM images (hypervisor-based virtualization), and TOSCA for model-based management.

### 4 Plan-based Deployment Automation

In the previous section we discussed plan-based configuration management as an operations automation approach, and as an enabler for several other approaches to implement the automated deployment of middleware and application components. For instance, plans such as scripts are attached to topology models of applications in order to make them deployable. Furthermore, plans are used to install and configure software components hosted on virtual machines and containers. Platform-centric management environments based on PaaS frameworks can be set up and operated based on such plans. Thus, in the following we focus on characterizing, refining, and evaluating plan-based approaches to implement deployment automation as a key part of operations automation.

Taking a look at the state of the art, the DevOps community focuses on providing pragmatic solutions for the automation of application deployment. Technically, this is often (immediately or implicitly) based on plan-based configuration management approaches, which drive the following discussions and evaluation presented in this article. The focus is on deploying predefined application stacks across several (virtual or physical) machines. Reusability only occurs when similar application stacks are being deployed by the same teams of people. Such a deployment can be characterized as application-oriented. The communities affiliated with some of the popular DevOps tools such as Chef\(^{26}\) or Puppet\(^{28}\) provide artifacts such as Chef cookbooks\(^{27}\) to build deployment plans for certain application stacks. Deployment plans based on these artifacts can be used to automate the deployment of the applications described in Section 2.1. Such plans can be implemented as Chef cookbooks by (i) orchestrating existing cookbooks that are already available and (ii) implementing some application-specific deployment logic. Existing cookbooks are typically available to deploy popular middleware components such as an Apache HTTP server or a PHP runtime environment. Consequently, this deployment approach can be summarized as follows:

**Application-oriented Deployment:** Application-specific but portable deployment plans enable the deployment of a particular application including all

\(^{25}\) RightScale and Chef integration: http://goo.gl/sn1OtT

\(^{26}\) Chef knife plugins: https://docs.chef.io/plugin_knife.html

\(^{27}\) Chef supermarket: https://supermarket.chef.io
application components and middleware components involved.

The portability of deployment plans enables infrastructure abstraction [6], meaning that the plans are not bound to a specific infrastructure such as a particular XaaS provider. As an example, most of the cookbooks provided by the Chef community realize infrastructure abstraction because Chef cookbooks are implemented using a domain-specific language that is not bound to a specific platform. Deployment plans that follow the application-oriented deployment approach are typically limited in their reuse because they were created to automate the deployment of a specific application. In addition, the deployment plans involved are typically hard-wired, i.e., they have explicit dependencies that cannot be exchanged dynamically without changing the plans themselves. To provide an approach enabling the creation of deployment plans with improved reusability, in [6] we proposed the middleware-oriented deployment approach:

Middleware-oriented Deployment: Generic and reusable middleware components that are not bound to a specific application enable the deployment of Cloud applications including (i) the middleware functionality required by the application and (ii) the application components involved.

In this case, application deployment is performed by parameterizing and executing portable deployment plans that are attached only to the middleware components. We assume that middleware components are not bound to specific applications, so these middleware components including their deployment plans can be reused to deploy different applications of the same type. There are no deployment plans attached to any application component when following the pure middleware-oriented deployment approach. Consequently, an overarching orchestration model or system to wire multiple deployment plans as it is typically required for non-trivial application stacks can purely focus on the configuration of reusable plans. This is in contrast to developing individual application-specific plans and corresponding orchestration models, which are hard to reuse.

In case the application components are not bound to specific middleware components, this approach enables middleware abstraction. Consequently, particular middleware components can be arbitrarily exchanged, e.g., based on functional or non-functional requirements. As an example, [6] shows how the JOnAS application server in the Taxi App stack can be exchanged by a Tomcat servlet container. Because the Tomcat servlet container consumes less memory and gets deployed faster, it could be typically used in a development environment instead of deploying a complete application server such as JOnAS.

The Chef community has published cookbooks that can be classified under the middleware-oriented deployment approach such as the application_php28 cookbook to deploy arbitrary applications or application components implemented in PHP. However, these cookbooks do not enable middleware abstraction because they contain hard-wired dependencies to other middleware components. For example, the application_php cookbook mentioned before has a hard-wired dependency to the Apache HTTP server as its underlying middleware. Consequently, this middleware component that provides a PHP runtime environment cannot be exchanged dynamically without changing the cookbook. In addition, these cookbooks cannot be used as deployment plans that can be parameterized. Usually, they provide resources that can be used in other cookbooks. A developer needs to create an additional deployment plan that implements the configuration and wiring of these resources. Consequently, these cookbooks cannot be immediately used as middleware-oriented deployment plans as they are.

In the following we evaluate these types of approaches using the applications presented in Section 2.1 as the means to identify which approach fits better which application.

5 Evaluation

For evaluation purposes we implemented deployment plans to automate the deployment of all three applications described in Section 2.1 using both the application-oriented and the middleware-oriented approach. This results in six different deployment scenarios (three applications multiplied by two approaches). In addition, we deploy the SugarCRM application in two manners (centralized in one VM, and distributed across different VMs), again using both the application-oriented and the middleware-oriented deployment approach to further broaden the scope of our evaluation. This adds two additional deployment scenarios, so our evaluation covers eight deployment scenarios in total.

Table 1 provides an overview of all deployment plans involved in the scenarios. Technically, all plans are implemented as Chef cookbooks. However, we do not rely on any unique feature of Chef, so the deployment plans can be implemented using different deployment automation tools such as Puppet, Juju29, OpenTOSCA30, and other plan-based management approaches [37]. Each plan is characterized by its type: middleware means that the plan deploys one or more middleware components. Plans of type middleware & app generic are typically used for the middleware-oriented deployment approach to deploy middleware components as well as application components on top of them as discussed in [6]. App specific plans are mostly used for the application-oriented deployment

30. OpenTOSCA: http://www.iaas.uni-stuttgart.de/OpenTOSCA
approach because they implement specific logic to automate the deployment of particular application components. Furthermore, the input parameters (Chef attributes) for each plan are identified. Figure 5 and Figure 6 show the dependencies between plans when following the application-oriented or the middleware-oriented deployment.

All deployments have been performed using two different Cloud infrastructures: FlexiScale\textsuperscript{31} and Amazon Web Services EC2\textsuperscript{32}, in order to ensure independence from a specific provider. Table 2 shows the evaluation settings in detail: the Taxi App has been deployed on Ubuntu Linux 10.04 Server (64-bit) on a virtual machine providing 2 CPU cores and 4 GB (FlexiScale) or 7.5 GB (AWS EC2) of RAM. Both SugarCRM and the Chat App have been deployed on Ubuntu Linux 12.04 Server (64-bit) based on 1 CPU core and 1 GB (FlexiScale) or 1.7 GB (AWS EC2) of RAM. For the distributed SugarCRM deployment, two virtual machines were used: one for the database and another one for the application itself. Each machine has 1 CPU core and 1 GB (FlexiScale) or 1.7 GB (AWS EC2) of RAM. The deployment of the database was running in parallel to the deployment of the actual application. Then, these two machines were dynamically wired at deployment time by exchanging the database endpoint information using an AWS S3 bucket\textsuperscript{33}. In all cases we are using hypervisor-based VMs with a plain operating system running the plans (Chef cookbooks) on top of it, i.e. no pre-configured VM images are used. Consequently, we are clearly following the plan-based configuration management approach as discussed in Section 3.

For each deployment scenario, five measurements are performed in total. These include three qualitative and three quantitative measurements.

5.1 Qualitative Measurements
First, we measure the following three qualitative properties for each deployment plan. For all three properties we use an ordinal \textit{Low/Medium/High} scale, from worst to best:

- **Flexibility**: This property expresses the degree of customizability by configuring a particular deployment plan using input parameters at runtime. Measurable degrees:
  - \textit{Low}: No input parameters, i.e., no dynamics at runtime.
  - \textit{Med}: Configuration options for predefined components, e.g., database credentials.
  - \textit{High}: Dynamic processing of arbitrary application components of a particular type.

- **Assumption Freedom**: This refers to the number of assumptions made regarding the input of a particular deployment plan. Obviously this can only be measured in case the plan has any input parameters at all. Measurable degrees:
  - \textit{Low}: Application-specific assumptions, e.g., a tenant ID needs to be written into a WAR file (application component) either by the deployment plan itself or by some kind of preprocessor.
  - \textit{Med}: Common assumptions for the corresponding type of application component, e.g., a Node.js application typically owns a package.json file that specifies its dependencies.
  - \textit{High}: No specific assumptions.

- **Reusability**: The following degrees specify the reusability of a particular deployment plan:
  - \textit{Low}: No reusability, i.e., the plan was specifically created for a particular application.
  - \textit{Med}: The plan can be used to deploy reusable but fixed components such as middleware components that are required in different application stacks.
  - \textit{High}: The plan can be used to deploy arbitrary application components of a particular type.
Table 1: Types and Input Parameters of Deployment Plans

<table>
<thead>
<tr>
<th>Plan Type</th>
<th>Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application-oriented deployment of Taxi App:</strong></td>
<td></td>
</tr>
<tr>
<td>ESB</td>
<td>Middleware DB credentials</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Middleware DB credentials</td>
</tr>
<tr>
<td>JOnAS</td>
<td>JOnAS configuration</td>
</tr>
<tr>
<td>Orchestra</td>
<td>none</td>
</tr>
<tr>
<td>App_Helper_Services</td>
<td>none</td>
</tr>
<tr>
<td>App_ESB_Components</td>
<td>none</td>
</tr>
<tr>
<td>PostgreSQL_DBs</td>
<td>DB credentials</td>
</tr>
<tr>
<td>App_BPEL_Processes</td>
<td>none</td>
</tr>
<tr>
<td>App_Tenant</td>
<td>URLs of WAR files (taxi driver GUI, customer GUI), URL of SoapUI test suite</td>
</tr>
<tr>
<td><strong>Middleware-oriented deployment of Taxi App:</strong></td>
<td></td>
</tr>
<tr>
<td>ESB</td>
<td>Middleware &amp; App Generic DB credentials, URLs of SoapUI test suites</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Middleware &amp; App Generic DB credentials, DB specifications</td>
</tr>
<tr>
<td>JOnAS</td>
<td>JOnAS configuration, URLs of preprocessed WAR files (taxi driver GUI, customer GUI)</td>
</tr>
<tr>
<td>Orchestra</td>
<td>URLs of BPEL processes</td>
</tr>
<tr>
<td><strong>Application-oriented deployment of SugarCRM:</strong></td>
<td></td>
</tr>
<tr>
<td>Apache_HTTP_Server</td>
<td>Middleware Apache configuration</td>
</tr>
<tr>
<td>PHP_Runtime_Env</td>
<td>none</td>
</tr>
<tr>
<td>MySQL</td>
<td>none</td>
</tr>
<tr>
<td>SugarCRM_DB</td>
<td>DB credentials</td>
</tr>
<tr>
<td>SugarCRM_App</td>
<td>none</td>
</tr>
<tr>
<td>Connect_App_to_DB</td>
<td>DB credentials</td>
</tr>
<tr>
<td><strong>Middleware-oriented deployment of SugarCRM:</strong></td>
<td></td>
</tr>
<tr>
<td>Apache_HTTP_Server</td>
<td>Middleware &amp; App Generic Apache configuration, URL of ZIP file (SugarCRM PHP scripts), permission information</td>
</tr>
<tr>
<td>PHP_Runtime_Env</td>
<td>Middleware &amp; App Generic none</td>
</tr>
<tr>
<td>MySQL</td>
<td>DB credentials</td>
</tr>
<tr>
<td>Connect_App_to_DB</td>
<td>DB credentials, SugarCRM admin password</td>
</tr>
<tr>
<td><strong>Application-oriented deployment of Chat App:</strong></td>
<td></td>
</tr>
<tr>
<td>Node.js</td>
<td>Middleware none</td>
</tr>
<tr>
<td>Redis</td>
<td>none</td>
</tr>
<tr>
<td>Chat_App</td>
<td>none</td>
</tr>
<tr>
<td><strong>Middleware-oriented deployment of Chat App:</strong></td>
<td></td>
</tr>
<tr>
<td>Node.js</td>
<td>Middleware URL of ZIP file (Chat App scripts)</td>
</tr>
<tr>
<td>Redis</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 3 shows the measured degrees for the properties flexibility, assumption freedom, and reusability for each deployment plan.

5.2 Quantitative Measurements

In addition to the qualitative measurements, we also measure the following three quantitative properties for each deployment scenario:

- **Total Complexity**: This property expresses the number of “atomic actions”, i.e., Chef resources executed during deployment.
- **Total Number of Plan Dependencies**: This is the total number of dependencies between plans for a particular deployment scenario.
- **Total Execution Time**: This is the total time required for the execution of all deployment plans involved in a single scenario, measured in seconds.

Table 4 shows the measured total complexity of each deployment scenario, i.e., the number of Chef resources executed at deployment time for a particular deployment scenario. Moreover, Figure 5 and Figure 6 compare the total number of plan dependencies for each deployed application. Table 5 outlines the average execution time in total for each deployment scenario. The average time is based on five deployment runs.
Table 2
Evaluation Settings

<table>
<thead>
<tr>
<th>Application</th>
<th>Machine Type</th>
<th>#VMs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FlexiScale:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taxi App</td>
<td>2 CPU cores, 4 GB memory</td>
<td>1</td>
</tr>
<tr>
<td>SugarCRM</td>
<td>1 CPU core, 1 GB memory</td>
<td>1</td>
</tr>
<tr>
<td>SugarCRM (distrib.)</td>
<td>1 CPU core, 1 GB memory</td>
<td>2</td>
</tr>
<tr>
<td>Chat App</td>
<td>1 CPU core, 1 GB memory</td>
<td>1</td>
</tr>
<tr>
<td><strong>Amazon Web Services EC2:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taxi App</td>
<td>2 CPU cores, 7.5 GB memory (m1.large)</td>
<td>1</td>
</tr>
<tr>
<td>SugarCRM</td>
<td>1 CPU core, 1.7 GB memory (m1.small)</td>
<td>1</td>
</tr>
<tr>
<td>SugarCRM (distrib.)</td>
<td>1 CPU core, 1.7 GB memory (m1.small)</td>
<td>2</td>
</tr>
<tr>
<td>Chat App</td>
<td>1 CPU core, 1.7 GB memory (m1.small)</td>
<td>1</td>
</tr>
</tbody>
</table>

5.3 Reusability of Plans in Practice

In order to evaluate the reusability of deployment plans in practice, we reused the plans created for SugarCRM to deploy another PHP-based application, namely WordPress\(^{35}\). The deployment requirements of WordPress are similar to those of SugarCRM. In case of middleware-oriented deployment we only had to adapt the Connect_App_to_DB plan because the configuration file format for storing the database endpoint information is different. However, in case of application-oriented deployment two new plans had to be created based on the SugarCRM_DB and SugarCRM_App plans in addition. These are WordPress_DB and WordPress_App to deploy the WordPress application itself as well as the database. In summary, three plans were reused, one plan was modified, and no additional plan was implemented for the middleware-oriented deployment; for the application-oriented deployment, three plans were reused, one plan was modified, and two plans were additionally implemented.

6 Discussion

Section 5 described the process and results of our evaluation based on eight deployment scenarios. This section discusses the evaluation results and presents findings as well as lessons learned. In our previous work [6] we evaluated and discussed the impact of using the middleware-oriented deployment approach in order to reduce the number of deployment plans for the Taxi App. The evaluation presented in this paper broadens the horizon by looking beyond the number

of plans involved for the deployment of a single application. Therefore, in the following we analyze the measurements reported in the previous section.

### 6.1 Findings

One of the first findings that can be derived from the evaluation results shown in Table 3 is that the middleware-oriented deployment approach generally improves the reusability of deployment plans. Furthermore, our measurements verify the observations reported in [6] that the number of deployment plans decreases when using the middleware-oriented deployment approach in general. This is due to the fact that application-specific actions are covered by parameterizing generic deployment plans attached to middleware components.

However, the pure middleware-oriented deployment approach cannot be implemented for all deployment scenarios. Middleware-oriented deployment is typically implemented based on plans of type middleware and middleware & app generic. Because app specific plans are specifically created for particular application components they should be avoided when following the middleware-oriented approach. In case of implementing a middleware-oriented deployment of SugarCRM, for example, Table 3 shows that there is a plan to wire the application with the database (Connect_App_to_DB). This plan is of type app specific because it cannot be implemented in a generic manner. As discussed in Section 2.2, the requirements SR2 and SR3 require very application-specific actions to be performed. These are implemented using the wiring plan.

In case generic deployment plans for deploying application components are attached to middleware components, typically assumptions are made beforehand regarding the application components deployed using these plans. For instance, the Node.js plan used in the middleware-oriented deployment expects (i) a package.json file as described in the requirement CR1 and (ii) a pointer to the script that is the entry point for a particular application (CR2). Another example is the JOnAS plan to deploy additional tenants (WAR files) for the Taxi App (TR1, TR3).

The flexibility of deployment plans following the application-oriented deployment approach is, as expected, worse compared to middleware-oriented deployment because their implementation is tightly coupled to specific application components. They typically expect a few rudimentary input parameters only such as configuration options for the middleware.

When looking at Table 4 it becomes clear that the complexity of deploying a particular application is equal in both cases, and therefore it does not depend on the deployment approach chosen. This is because there is no difference on the level of “atomic actions”, i.e., on the level of Chef resources that are executed at deployment time such as creating a particular directory, storing a configuration file, or installing a software package. However, Figure 5 and Figure 6 denote a decreased number of plans and plan dependencies for all deployed applications in case the middleware-oriented deployment approach is used.

Table 5 shows minimal differences between application-oriented and middleware-oriented deployment for the average deployment time of each scenario. These measurements match the complexity measurements shown in Table 4: because the Chef resources executed at deployment time are the same, it is only logical that there are no significant differences in terms of the plans’ total execution time. The minor differences are due to the fact that files (middleware and application components) are downloaded from the Web during deployment. Because the quality of the underlying HTTP connections for these downloads could differ, there are small deviations.

Our further evaluation of reusability of plans in practice based on SugarCRM and WordPress (Section 5.3) confirms our previous conclusion: middleware-oriented deployment clearly improves reusability of plans. Moreover, the number of plans that need to be adapted for specific applications can be significantly reduced.

### 6.2 Lessons Learned

Based on the findings presented in the previous section, we now summarize several of the lessons learned to support the decision which deployment approach fits which type of application:

---

**Table 5**

<table>
<thead>
<tr>
<th>Application</th>
<th>App.-oriented</th>
<th>Middleware-oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FlexiScale:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taxi App</td>
<td>$\lambda = 899$ sec.</td>
<td>$\lambda = 937$ sec.</td>
</tr>
<tr>
<td></td>
<td>$\sigma = 25$ sec.</td>
<td>$\sigma = 45$ sec.</td>
</tr>
<tr>
<td>SugarCRM</td>
<td>$\lambda = 264$ sec.</td>
<td>$\lambda = 262$ sec.</td>
</tr>
<tr>
<td></td>
<td>$\sigma = 9$ sec.</td>
<td>$\sigma = 13$ sec.</td>
</tr>
<tr>
<td>SugarCRM (distrib.)</td>
<td>$\lambda = 184$ sec.</td>
<td>$\lambda = 180$ sec.</td>
</tr>
<tr>
<td></td>
<td>$\sigma = 4$ sec.</td>
<td>$\sigma = 10$ sec.</td>
</tr>
<tr>
<td>Chat App</td>
<td>$\lambda = 219$ sec.</td>
<td>$\lambda = 228$ sec.</td>
</tr>
<tr>
<td></td>
<td>$\sigma = 6$ sec.</td>
<td>$\sigma = 11$ sec.</td>
</tr>
<tr>
<td><strong>Amazon Web Services EC2:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taxi App</td>
<td>$\lambda = 1316$ sec.</td>
<td>$\lambda = 1294$ sec.</td>
</tr>
<tr>
<td></td>
<td>$\sigma = 53$ sec.</td>
<td>$\sigma = 37$ sec.</td>
</tr>
<tr>
<td>SugarCRM</td>
<td>$\lambda = 372$ sec.</td>
<td>$\lambda = 358$ sec.</td>
</tr>
<tr>
<td></td>
<td>$\sigma = 47$ sec.</td>
<td>$\sigma = 36$ sec.</td>
</tr>
<tr>
<td>SugarCRM (distrib.)</td>
<td>$\lambda = 233$ sec.</td>
<td>$\lambda = 235$ sec.</td>
</tr>
<tr>
<td></td>
<td>$\sigma = 9$ sec.</td>
<td>$\sigma = 16$ sec.</td>
</tr>
<tr>
<td>Chat App</td>
<td>$\lambda = 314$ sec.</td>
<td>$\lambda = 325$ sec.</td>
</tr>
<tr>
<td></td>
<td>$\sigma = 7$ sec.</td>
<td>$\sigma = 18$ sec.</td>
</tr>
</tbody>
</table>
L₁ – Middleware-oriented deployment plans are typically preferred for scenarios where conventions for application components are already established, so there are no or minimum assumptions regarding the application components. Examples for such conventions are the package.json file (requirement CR₁ in Section 2.2), setting permissions for PHP applications (SR₁), or sending SOAP messages using SoapUI test suites (TR₂). From this perspective, middleware-oriented deployment is preferred for the Chat App because the application-specific deployment requirements CR₁ and CR₂ can be transferred to other Node.js applications.

L₂ – There are application-specific requirements that cannot be generally transferred to other applications of the same type, so the implementation of middleware-oriented deployment for the Taxi App and SugarCRM is not as straightforward as it is for the Chat App. For SugarCRM the Connect_App_to_DB plan prevents the implementation of pure middleware-oriented deployment because application-specific deployment requirements (SR₂, SR₃) need to be fulfilled. These cannot be transferred to other PHP applications (e.g., WordPress) in general because the configuration of each PHP application is different. For the Taxi App, middleware-oriented deployment can be realized. However, to fulfill requirement TR₂, WAR files that are deployed using the JOnAS plan need to be preprocessed before deployment: the tenant ID and the tenant endpoint are stored inside the WAR files. Furthermore, TR₁ cannot be fulfilled by a single deployment plan because this is application-specific knowledge. To deploy additional tenants both the JOnAS and the ESB plan need to be used in combination with certain parameters.

L₃ – In case of distributed deployments, such as we did for SugarCRM, the wiring logic can be implemented in a middleware-oriented manner only if it does not need any application-specific knowledge of how and where to store the endpoint information. Typically, wiring plans are application-specific such as the Connect_App_to_DB plan in case of deploying SugarCRM. This is because most of the times endpoint information needs to be stored in application-specific configuration files.

L₄ – Even if the middleware-oriented deployment approach cannot be implemented completely, a hybrid approach can be realized. In this case as many deployment plans as possible are implemented in a middleware-oriented manner to improve flexibility and reusability. However, application-specific deployment actions are performed using plans that follow the application-oriented deployment approach. Examples for such actions are deploying additional tenants for the Taxi App (TR₁) or wiring the SugarCRM application with the database (SR₃). We followed this approach implicitly when we implemented middleware-oriented deployment for SugarCRM because it is impossible to implement the logic of the application-specific Connect_App_to_DB plan in a generic manner.

L₅ – The usage of middleware-oriented deployment plans instead of application-oriented ones does not affect the total complexity of a deployment scenario or the total execution time. Consequently, performance aspects do not have to be considered. Nonetheless, the total number of plans and plan dependencies can be reduced by following the middleware-oriented approach.

L₆ – However, the development of plans to realize middleware-oriented deployment might be more complex for the plan developers. As shown in Table 3 these plans can be parameterized using sets of input parameters. These parameters imply more dynamics in the plans' implementation increasing the complexity of the development. The gain of such an investment is a higher degree of flexibility and reusability as shown in Table 3.

6.3 Conventions for Application Components

As mentioned in lesson L₁, middleware-oriented deployment is typically preferred if common conventions for application components are established. Beside the conventions for Node.js applications (CR₁) as well as PHP applications (SR₁) discussed before, there are further conventions that are also implemented by PaaS providers such as Heroku or Google App Engine:

- For Java applications it is common to express all dependencies of a particular application component using a pom.xml file. These dependencies are resolved using Maven at deployment time. Furthermore, a system.properties file can be used to define the Java version required by a particular application component.
- In case the Java application is built using the Play framework, dependencies are typically defined using a dependencies.yml file instead of a pom.xml file.
- For Scala applications a build.properties file is typically used to specify dependencies. This file gets processed using sbt at deployment time.
- Ruby-based applications commonly make use of Bundler to process Gemfiles. A Gemfile is used to define both the dependencies and the Ruby version to be used for a certain application component.
- Applications based on Python usually utilize pip for dependency management. Therefore, a requirements.txt file must be created to point to other Python packages.

Moreover, PaaS providers implement mechanisms to expose endpoint information such as the IP address of the database instance using context or environment

37. Java system properties: http://goo.gl/Nov781
38. Play framework: http://www.playframework.com
40. sbt: http://www.scala-sbt.org
41. Bundler: http://bundler.io
42. pip: http://www.pip-installer.org
variables. These variables can be read, for instance, in a script-based configuration file (PHP, Python, Ruby, etc.) to connect application components to the database.

7 Conclusions and Future Work

The automated provisioning and deployment of applications on IaaS and PaaS solutions is one of the major enablers in the reduction of the operational costs by migrating to the Cloud. Tooling and approaches mostly from the DevOps community have provided the means for such an automation through deployment plans. However, these approaches focus on the deployment of individual, specific application stacks at a time, sacrificing reusability for efficiency and ease in the development of such deployment plans. For this reason, in previous work [6], and approaching the problem from a PaaS offering perspective, we proposed a middleware-oriented deployment approach that promotes reusability of deployment plans across different applications.

In this work, we expand on previous work to identify and characterize two different types of deployment approaches (application- and middleware-oriented) based on a systematic classification of existing operation automation approaches presented in the literature and available in the market. We developed deployment plans for three applications (Taxi App, SugarCRM, and Chat App) with significantly different deployment requirements using the identified approaches, and we evaluated the results across both qualitative and quantitative dimensions. Our findings show better reusability, portability, and flexibility of middleware-oriented plans when compared to application-oriented ones, without a loss in performance (i.e., deployment time). The results of our evaluation are independent of the Cloud provider. The trade-off however for this improvement is in the difficulty of creating such plans. In order to demonstrate the reusability potential of such plans in practice we reused the plans created for SugarCRM to deploy WordPress in an application- and middleware-oriented manner. WordPress has a similar application stack and comparable deployment requirements. Finally, based on what we derived from this evaluation we provide recommendations as lessons learned with respect to deciding which approach to use when deploying an application.

In terms of future work, we plan to extend our evaluation to cover even more application stacks based on different technologies such as Ruby on Rails or Django based on Python. Moreover, we aim to broaden the scope of the evaluation by considering additional aspects such as the number of code changes applied to deployment plans over time and training costs for developers and operations personnel. Based on this additional evaluation, further findings and lessons learned can be derived, in addition to verifying or falsifying the existing ones. Furthermore, as discussed in the introduction, the overall goal of this work is to provide a decision support system for deployment of applications in the Cloud. Toward this goal, a decision support matrix based on the lessons learned from this work is currently under development. The immediate goal of this matrix is to provide the systematic means for decisions related to the creation of deployment plans, as well as how to use and combine them to automate the deployment of a particular application stack. Based on such a matrix, a decision support system prototype can then be implemented. In this context, existing deployment plans such as cookbooks provided by the Chef community can be linked and proposed to the person using the decision support system. The decision support system will not only cover plan-based configuration management approaches, but also further operations automation approaches as shown in our classification. Moreover, we plan to focus on providing decision support on how multiple operations automation approaches can be combined. For instance, the combination of container virtualization and plan-based configuration management has also to be investigated.

Acknowledgments

The research leading to these results has partially received funding from the EU’s Seventh Framework Programme (FP7/2007-2013) projects 4CaaSt (grant agreement no. 258862), ALLOW Ensembles (grant agreement no. 600792), and from the BMBF project ECHO (01ZX13023G).

References


All links were last followed on February 15, 2016.