Institute of Architecture of Application Systems

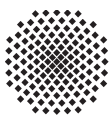
Characterizing and Evaluating Different Deployment Approaches for Cloud Applications

Johannes Wettinger, Vasilios Andrikopoulos, Steve Strauch, Frank Leymann

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

BibTeX:

```
@inproceedings{Wettinger2014,
  author    = {Johannes Wettinger and Vasilios Andrikopoulos and
               Steve Strauch and Frank Leymann},
  title     = {Characterizing and Evaluating Different Deployment Approaches
               for Cloud Applications},
  booktitle = {Proceedings of the IEEE International Conference on Cloud
               Engineering, IC2E 2014, 10-14 March 2014,
               Boston, Massachusetts, USA},
  year      = {2014},
  pages     = {205--214},
  publisher = {IEEE Computer Society}
}
```

Universität Stuttgart
Germany

Characterizing and Evaluating Different Deployment Approaches for Cloud Applications

Johannes Wettinger, Vasilios Andrikopoulos, Steve Strauch, Frank Leymann
Institute of Architecture of Application Systems (IAAS)
University of Stuttgart, Stuttgart, Germany
lastname@iaas.uni-stuttgart.de

***Abstract*—Fully automated provisioning and deployment in order to reduce the costs for managing applications is one of the most essential requirements to make use of the benefits of Cloud computing. Several approaches and tools are available to automate the involved processes. The DevOps community, for example, provides tooling and artifacts to realize deployment automation on Infrastructure as a Service level in a mostly application-oriented manner. Platform as a Service frameworks are also available for the same purpose. In this paper we categorize and characterize available deployment approaches independently from the underlying technology used. For this purpose, 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. Finally, we discuss the evaluation results and derive recommendations which deployment approach to use based on the deployment requirements of an application.**

***Keywords*-middleware-oriented deployment; application-oriented deployment; Cloud computing; DevOps; decision support**

I. Introduction

The popularity and success of Cloud computing was built on public infrastructure offerings such as Amazon Web Services[1] providing relatively cheap resources such as virtual machines and virtual disks (in the *Infrastructure as a Service (IaaS)* delivery model [1]). These resources can be provisioned and decommissioned on demand, so the customer only pays for what he is 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. The higher-level service delivery models *Platform as a Service (PaaS)* and *Software as a Service (SaaS)* are also available. As the Cloud service market is getting more mature, these service models gain more traction in the market with more 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 middleware 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 therefore available in this environment, depending on how the provider-driven automation of the offered middleware solutions is leveraged by the application developers. The research presented in this work focuses in particular on the characterization of these deployment approaches, with the intention of identifying the more 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 [2], in terms of providing a more thorough evaluation of the discussed approaches.

The main contributions of our work can therefore be summarized as follows:

- We define and characterize two types of deployment approaches based on the state of the art and the limitations of current tools.
- We analyze the deployment requirements of three different applications covering a set of the most popular technologies for developing Web applications.
- Based on these requirements, we implement the automated deployment of all three applications using both types of deployment approaches for evaluation purposes. This results in eight deployment scenarios measuring both qualitative and quantitative properties from which we derive a number of findings.
- Finally, we present an initial list of lessons learned based on these findings to support the decision when to use which deployment approach.

The remaining of this paper is structured as follows: Section II motivates our work by introducing the applications to be used for evaluation purposes and analyzing their specific deployment requirements. Based on the state of the art and current limitations two types of deployment approaches are identified in Section III. Our evaluation of these deployment approaches is presented in Section IV based on the three applications and their deployment requirements described in

[1]Amazon Web Services: http://aws.amazon.com
[2]Google App Engine: https://cloud.google.com/products/app-engine

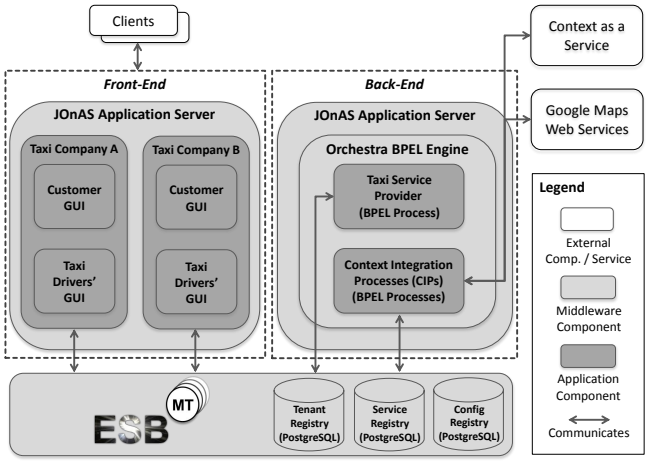


Figure 1. Architecture of the Taxi Application

Section II. Consequently, the evaluation results are discussed in Section V including the presentation of several findings. Furthermore, an initial list of lessons learned is derived from these findings in the same section. Section VI presents an overview of related work. Finally, Section VII concludes this paper and describes some ideas for future work.

II. Motivation

In this section we introduce the three applications that are realized based on different technologies (Section II-A). 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 II-B) to be addressed later on during the evaluation.

A. 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. The topologies of all three applications consist of *middleware components, application components,* and external services.

An overview of the Taxi App's architecture developed in the scope of the European Project 4CaaSt[3] as a demonstrator of the PaaS offerings of the project is shown in Figure 1 [3]. 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 [4]. The taxi management software leverages context integration

[3]EU Project 4CaaSt: http://www.4caast.eu

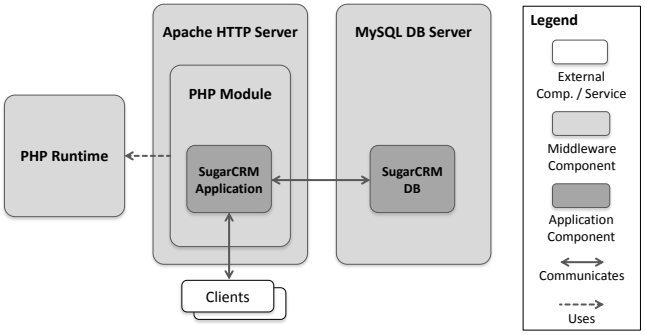


Figure 2. Architecture of SugarCRM

processes also implemented in BPEL to retrieve context information about taxi cabs such as location and taxi driver contact details from the 4CaaSt platform-internal Context as a Service. Moreover, Google Maps Web Services [5] 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 Orchestra[4] version 4.9.0-M3, which itself is deployed in the Java Open Application Server (JOnAS)[5] 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 ESB^{MT}[6] as messaging middleware (Figure 1) enables loose coupling and provides a flexible integration solution by avoiding hard-coded point-to-point connections. ESB^{MT} is based on Apache ServiceMix[7] version 4.3.0 and comes with three registries realized as PostgreSQL[8] version 9.1 databases [6], [7].

Figure 2 provides an overview of the architecture of SugarCRM[9], an open source Customer Relationship Management Software (CRM). All relevant data such as contact details of the customers are stored in the SugarCRM Database within a MySQL Database Server[10] version 5.5.32. The SugarCRM Web Application is implemented in PHP and is running on an Apache HTTP Server[11] version 2.2.22 using a PHP runtime version 5.3.10.

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[12] version 2.6.14. The Chat App is based on Node.js version 0.10.13[13].

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

[4]OW2 Orchestra: http://orchestra.ow2.org
[5]OW2 JOnAS: http://jonas.ow2.org
[6]ESB^{MT}: http://www.iaas.uni-stuttgart.de/esbmt
[7]Apache ServiceMix: http://servicemix.apache.org
[8]PostgreSQL: http://www.postgresql.org
[9]SugarCRM: http://www.sugarcrm.com
[10]MySQL Server: http://www.mysql.com
[11]Apache HTTP Server: http://httpd.apache.org
[12]Redis: http://redis.io
[13]Node.js: http://nodejs.org

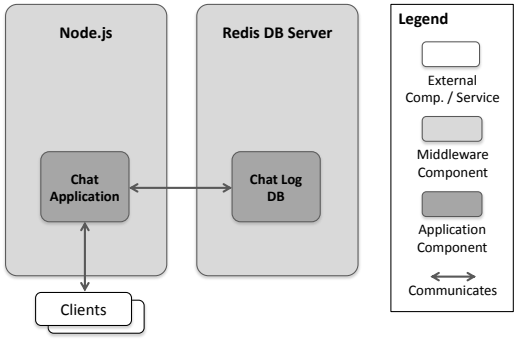


Figure 3. Architecture of the Chat Application

considered when creating corresponding deployment plans. A deployment plan (e.g., a script) implements the logic necessary to deploy application or middleware components.

B. Deployment Requirements

There are requirements that apply to the deployment of all the presented applications. These general deployment requirements are in particular:

- GR_1 *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_2 *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_3 *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 technical requirements regarding the design of deployment plans are discussed in [2]. In addition to these general requirements, each presented application imposes additional deployment requirements. For the Taxi App [3] we identified the following requirements when we implemented its automated deployment:

- TR_1 *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 the ESB as well as deploying additional WAR files (customer GUI and taxi drivers' GUI) to the JOnAS application server.
- TR_2 *Sending SOAP messages using SoapUI*[14]*:* To register tenant endpoints inside the ESB 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 the SoapUI's command line-based test runner.
- TR_3 *Modification of WAR files:* The WAR files that are deployed both upon the application's initial deployment and when adding additional tenants need to be modified at deployment time. This is because they need to connect to the tenant endpoints registered in the ESB. Therefore, the address of the corresponding tenant endpoint and the tenant ID have to be stored as key-value pairs inside a properties file located in the WAR file.

In contrast to the Taxi App (and the Chat App, later) the deployment automation of SugarCRM as described in [8] was realized with *distributed deployment* 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:

- SR_1 *Permission settings:* The permissions of particular files and directories of the PHP application need to be set specifically. As an example, the *cache* directory needs to be writable. This is a typical requirement for the deployment of PHP applications.
- SR_2 *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 preconfigured by creating a default user (administrator).
- SR_3 *Dynamic wiring at deployment time:* This requirement refines GR_3: the SugarCRM application needs to be wired with the database *dynamically at deployment time.* Thus, the endpoint of the database needs to be put to the application's configuration, so the connection can be established.

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:

- CR_1 *Node package manager:* 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)[15].
- CR_2 *Pointer to application's entry point:* Node.js applications typically consist of several scripts implemented using JavaScript. To start such an application, a

[14]SoapUI: http://www.soapui.org

[15]npm: http://npmjs.org

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.

This section outlined the fact that different applications have 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 general, there are different approaches to realize deployment automation such as the one described in [2]. Because there is not a single approach that fits all the different requirements, it is necessary to be able to identify the "best" approach for each type of application. Thus, in the following we categorize existing deployment approaches based on the purpose and granularity of their deployment plans.

III. Deployment Approaches

Taking a look at the state of the art, the DevOps community focuses on providing pragmatic solutions for the automation of application deployment. 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 [9] or Puppet [10] provide artifacts such as Chef cookbooks[16] 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 II-A. 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 application components and middleware components involved.*

The portability of deployment plans enables *infrastructure abstraction* [2], 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, we proposed the *middleware-oriented deployment approach* in [2]:

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 approach of middleware-oriented deployment.

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, [2] 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_php`[17] 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. Thus, a developer needs to create an additional deployment plan that implements the configuration and wiring of these resources.

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

[16]Chef community: http://community.opscode.com/cookbooks

[17]http://community.opscode.com/cookbooks/application_php

IV. Evaluation

For evaluation purposes deployment plans are implemented to automate the deployment of all three applications described in Section II 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. Technically, the deployment plans are implemented using Chef. 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, Juju[18], OpenTOSCA[19], and other plan-based management approaches [11].

For each deployment scenario, five measurements are performed in total. These include three qualitative and two quantitative measurements. All properties are measured for this purpose in an ordinal non-normalized scale.

A. Qualitative Measurements

First, we measure the following three qualitative properties for each deployment plan:

- *Flexibility:* This property expresses the degree of customizability by configuring a particular deployment plan using input parameters at runtime. Measurable degrees:
  1) No input parameters, i.e., no dynamics at runtime.
  2) Configuration options for predefined components, e.g., database credentials.
  3) Dynamic processing of arbitrary application components of a particular type.
- *Assumptions:* These are the assumptions made regarding the input of a particular deployment plan. Obviously this can only be measured in case the plan has any parameters at all. Measurable degrees:
  1) 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.
  2) 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.
  3) No specific assumptions.
- *Reusability:* The following degrees specify the reusability of a particular deployment plan:
  1) No reusability, i.e., the plan was specifically created for a particular application.
  2) Plan can be used to deploy reusable but fixed components such as middleware components that are required in different application stacks.
  3) Plan can be used to deploy arbitrary application components of a particular type.

For all three properties described above, the degrees are in a monotonically increasing scale (that is, larger is perceived as better). Table I provides an overview of the deployment plans that have been implemented to realize the deployment scenarios described before. Each plan is characterized by its type: (i) *middleware* means that the plan deploys one or more middleware components. (ii) 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 [2]. (iii) *App specific* plans are mostly used for the application-oriented deployment 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. Finally, Table I shows the measured degrees for the properties *flexibility, assumptions,* and *reusability* for each deployment plan.

B. Quantitative Measurements

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

- *Total Complexity:* This property expresses the number of "atomic actions", i.e., Chef resources[20] executed during deployment.
- *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 II shows the quantitatively measured total complexity of each deployment scenario, i.e., the number of Chef resources executed at deployment time. Table III outlines the average execution time in total for each deployment scenario. The average time is based on five deployment runs per scenario. All deployments have been performed using the Cloud infrastructure provided by FlexiScale[21]. The Taxi App has been deployed on Ubuntu Linux 10.04 Server (64-bit) on a virtual machine providing 2 CPU cores and 4 gigabytes of memory. Both SugarCRM and the Chat App have been deployed on Ubuntu Linux 12.04 Server (64-bit) based on 1 CPU core and 1 gigabyte of memory. For the distributed SugarCRM deployment, two virtual machines were involved: one for the database and another one for the application itself. Each machine has 1 CPU core and 1 gigabyte of memory. The deployment of the database was running in parallel to the deployment of the actual application. Then, these two

[18]Juju: http://juju.ubuntu.com

[19]OpenTOSCA: http://www.iaas.uni-stuttgart.de/OpenTOSCA

[20]Chef resources: http://docs.opscode.com/resource.html

[21]FlexiScale: http://www.flexiscale.com

Table I
QUALITATIVE MEASUREMENTS OF DEPLOYMENT PLANS

Plan	Type	Input Parameters	Flexibility	Assumptions	Reusability
Application-oriented deployment of Taxi App:					
ESB^{MT}	Middleware	DB credentials	2	3	2
PostgreSQL	Middleware	DB credentials	2	3	2
JOnAS	Middleware	JOnAS configuration	2	3	2
Orchestra	Middleware	none	1	–	2
App_Helper_Services	App Specific	none	1	–	1
App_ESB_Components	App Specific	none	1	–	1
PostgreSQL_DBs	App Specific	DB credentials	2	3	1
App_BPEL_Processes	App Specific	none	1	–	1
App_Tenant	App Specific	URLs of WAR files (taxi driver GUI, customer GUI), URL of SoapUI test suite	3	1	1
Middleware-oriented deployment of Taxi App:					
ESB^{MT}	Middleware & App Generic	DB credentials, URLs of SoapUI test suites	3	3	3
PostgreSQL	Middleware & App Generic	DB credentials, DB specifications	3	3	3
JOnAS	Middleware & App Generic	JOnAS configuration, URLs of preprocessed WAR files (taxi driver GUI, customer GUI)	3	1	3
Orchestra	Middleware & App Generic	URLs of BPEL processes	3	3	3
Application-oriented deployment of SugarCRM:					
Apache_HTTP_Server	Middleware	Apache configuration	2	3	2
PHP_Runtime_Env	Middleware	none	1	–	2
MySQL	Middleware	none	2	–	2
SugarCRM_DB	App Specific	DB credentials	2	3	1
SugarCRM_App	App Specific	none	1	–	1
Connect_App_to_DB	App Specific	DB credentials	2	3	1
Middleware-oriented deployment of SugarCRM:					
Apache_HTTP_Server	Middleware & App Generic	Apache configuration, URL of ZIP file (SugarCRM PHP scripts), permission information	3	1	3
PHP_Runtime_Env	Middleware & App Generic	none	1	–	2
MySQL	Middleware & App Generic	DB credentials	3	3	3
Connect_App_to_DB	*App Specific*	DB credentials, SugarCRM admin password	2	3	1
Application-oriented deployment of Chat App:					
Node.js	Middleware	none	2	–	2
Redis	Middleware	none	2	–	2
Chat_App	App Specific	none	1	–	1
Middleware-oriented deployment of Chat App:					
Node.js	Middleware & App Generic	URL of ZIP file (Chat App scripts)	3	2	3
Redis	Middleware	none	2	–	2

machines were dynamically wired at deployment time by exchanging the database endpoint information using an AWS S3 bucket[22].

This section described how we performed the evaluation based on eight deployment scenarios. We presented the results of the evaluation and explained how to understand them. The following Section V discusses the evaluation results and presents findings as well as lessons learned.

[22]Amazon Web Services S3 (AWS S3): http://aws.amazon.com/s3

V. DISCUSSION

In our previous work [2] we evaluated and discussed already 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. For this purpose, in the following we analyze the measurements reported in the previous section.

Table II
QUANTITATIVE MEASUREMENTS OF SCENARIOS' TOTAL COMPLEXITY

Application	App.-oriented	Middleware-oriented
Taxi App	139	139
SugarCRM	57	57
SugarCRM (distributed)	57	57
Chat App	16	16

Table III
TOTAL EXECUTION TIME OF DEPLOYMENT PLANS (AVERAGE)

Application	App.-oriented	Middleware-oriented
Taxi App	899 sec.	937 sec.
SugarCRM	264 sec.	262 sec.
SugarCRM (distributed)	184 sec.	180 sec.
Chat App	219 sec.	228 sec.

A. Findings

One of the first findings that can be derived from the evaluation results shown in Table I is that the middleware-oriented deployment approach generally improves the reusability of deployment plans (Table I, column *reusability*). Furthermore, our measurements verify the observations reported in [2] 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 I 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 II-B, the requirements SR_2 and SR_3 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 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 CR_1 and (ii) a pointer to the script that is the entry point for a particular application (CR_2). Another example is the `JOnAS` plan to deploy additional tenants (WAR files) for the Taxi App (TR_1, TR_3).

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 II 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.

Table III 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 II: 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.

B. Lessons Learned

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

L$_1$ *– Middleware-oriented deployment plans are typically preferred for scenarios where conventions for application components are established,* so there are no or minimal assumptions regarding the application components. Examples for such conventions are the `package.json` file (requirement CR_1 in Section II-B), setting permissions for PHP applications (SR_1), or sending SOAP messages using SoapUI test suites (TR_2). From this perspective, middleware-oriented deployment is preferred for the Chat App because the application-specific deployment requirements CR_1 and CR_2 can be transferred to other Node.js applications in general.

L$_2$ *– 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 pure implementation of middleware-oriented deployment because application-specific deployment requirements (SR_2, SR_3)

need to be fulfilled. These cannot be transferred to other PHP applications 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 TR_3, 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, TR_1 cannot be fulfilled by a single deployment plan because this is application-specific knowledge. To deploy additional tenants both the `JOnAS` and the $\texttt{ESB}^{\texttt{MT}}$ plans have to be used in combination with certain parameters.

L$_3$ *– 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.

L$_4$ *– 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 (TR_1) or wiring the SugarCRM application with the database (SR_3). 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.

L$_5$ *– The usage of middleware-oriented deployment plans instead of application-oriented ones does neither affect the total complexity of a deployment scenario nor the total execution time* (Table II and Table III). Consequently, performance aspects do not have to be considered.

L$_6$ *– However, the development of plans to realize middleware-oriented deployment might be more complex for the plan developers.* As shown in Table I 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 I.

VI. Related Work

To the extent of our knowledge there is no existing work beyond [2] attempting to evaluate and classify different deployment approaches. For this purpose in the following we survey various deployment approaches from the literature.

In order to have a holistic model of a particular Cloud service for the deployment, its structure and behavior can be specified by using a higher-level model-driven approach. This holistic model is independent from the deployment approach used to deploy the service. Two examples to realize this approach are the Topology and Orchestration Specification for Cloud Applications (TOSCA) [12] and Blueprints [13]. Whereas TOSCA is an emerging standard [14], Blueprints are originating in the 4CaaSt project. In addition, there are commercial products available that implement the model-driven approach. An example for these products is the IBM SmartCloud Orchestrator[23]. The goal of the model-driven approach is to enable top-down modeling by starting with a higher-level model for the Cloud service. To enable the deployment of such a model, scripts may have to be attached to the model to perform the actual deployment of the middleware and application components.

For the deployment of simple and complex application stacks including multiple components such as the three applications introduced in Section II many alternative deployment approaches exist that are state of the art. The first approach with respect to the IaaS service model is to encapsulate the different middleware components and application components in virtual machine images. Today, there are many IaaS providers offering the deployment of virtual machine images such as Amazon Web Services (AWS)[24]. In addition, there are open source products such as OpenStack [15] available to create an IaaS environment for deploying virtual machine images. The Open Virtualization Format (OVF) [16] aims to be a standardized format for such images.

For each of the three applications introduced in Section II, a virtual machine image can be created to host the whole application or each application component can be hosted in a separate virtual machine (VM). Another possibility is a compromise between both options by hosting some components on separate VMs and other components of the same application might be grouped together to be hosted in the same VM, e.g., depending on the resource requirements of each component. Several approaches are available that are focused on optimized provisioning of virtual machines and deploying virtual machine images such as [17], [18], and [19].

Focusing on the IaaS model, the alternative is to use standard images that basically provide a plain operating system only, instead of completely pre-installed and pre-configured virtual machine images. Configuration management tooling proposed by the DevOps community such as Chef [9], Puppet [10], or CFEngine [20] can be used to install and configure the actual middleware and application components. Scripts are used to perform the installation and configuration [21]. In order to manage topologies that consist

[23]IBM SmartCloud Orchestrator: http://ibm.co/CPandO

[24]Amazon Web Services: http://aws.amazon.com

of several machines and different components hosted on them, model-driven tooling such as AWS CloudFormation[25], AWS OpsWorks [22], or Juju[26] can be used. An efficient way of combining configuration management with model-driven management is described in [8]. In addition, there are holistic management services available such as EnStratus [23] or RightScale [24]. These use Chef scripts in the background to perform the actual deployment. For the three applications considered in this work this deployment approach implies to have at least several scripts to install and configure all the middleware and application components that are involved. In addition, there may be a specification that orchestrates all these scripts.

Contrary to the IaaS model, deployment in the PaaS model can only be performed based on existing platform offerings such as Google App Engine [25] or Amazon Elastic Beanstalk [26]. The goal of the PaaS model is to provide a platform that abstracts from the underlying infrastructure resources and provides "middleware as a service". Thus, the application components are directly hosted on the platform. To host the Taxi App, SugarCRM, or the Chat App using the PaaS model, several "middleware services" are required to be exposed by the platform. These middleware services should 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. As an example, the PaaS framework Cloud Foundry[27] enables this approach.

VII. 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 the time, sacrificing reusability for efficiency and ease in the development of such deployment plans. For this reason, in previous work [2], and approaching the problem for 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 the literature and existing tooling. We develop deployment plans for three applications with significantly different deployment requirements using the identified approaches, and we evaluate 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 trade-off however for this improvement is in the difficulty of compiling such plans. 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[28] or Django[29] based on Python. Based on such an expanded evaluation, further findings and lessons learned can be derived, in addition to verifying or falsifying the existing findings and lessons learned. 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.

[25] AWS CloudFormation: http://aws.amazon.com/cloudformation

[26] Juju: http://juju.ubuntu.com

[27] Cloud Foundry: http://www.cloudfoundry.org

[28] Ruby on Rails: http://rubyonrails.org

[29] Django: http://www.djangoproject.com

Acknowledgments

The research leading to these results has received funding from the EU's Seventh Framework Programme (FP7/2007-2013) projects 4CaaSt (grant agreement no. 258862) and ALLOW Ensembles (grant agreement no. 600792), and from the BMBF project ECHO (01XZ13023G).

References

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," 2009.

[2] J. Wettinger, V. Andrikopoulos, S. Strauch, and F. Leymann, "Enabling Dynamic Deployment of Cloud Applications Using a Modular and Extensible PaaS Environment," in *Proceedings of IEEE CLOUD 2013*. IEEE Computer Society, pp. 478–485.

[3] S. Strauch, V. Andrikopoulos, F. Leymann, and D. Muhler, "ESB^{MT}: Enabling Multi-Tenancy in Enterprise Service Buses," in *Proceedings of CloudCom 2012*. IEEE Computer Society Press, pp. 456–463.

[4] A. Alves *et al.*, "Web Services Business Process Execution Language Version 2.0," Comitee Specification, 2007.

[5] Google, Inc., "Google Maps API Web Services." [Online]. Available: http://developers.google.com/maps/documentation/webservices

[6] S. Strauch, V. Andrikopoulos, S. Gómez Sáez, and F. Leymann, "Implementation and Evaluation of a Multi-tenant Open-Source ESB," in *Proceedings of ESOCC 2013*, ser. Lecture Notes in Computer Science, vol. 8135. Springer, 2013, pp. 79–93.

[7] ——, "ESB^{MT}: A Multi-tenant Aware Enterprise Service Bus," *International Journal of Next-Generation Computing*, vol. 4, no. 3, pp. 230–249, 2013.

[8] J. Wettinger, M. Behrendt, T. Binz, U. Breitenbücher, G. Breiter, F. Leymann, S. Moser, I. Schwertle, and T. Spatzier, "Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA," in *Proceedings of CLOSER 2013*.

[9] S. Nelson-Smith, *Test-Driven Infrastructure with Chef*. O'Reilly Media, Inc., 2011.

[10] J. Loope, *Managing Infrastructure with Puppet*. O'Reilly Media, Inc., 2011.

[11] U. Breitenbücher, T. Binz, , O. Kopp, and F. Leymann, "Pattern-Based Runtime Management of Composite Cloud Applications," in *Proceedings of CLOSER 2013*.

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

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

[14] OASIS, "Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, Committee Specification Draft 04," 2012. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd04/TOSCA-v1.0-csd04.html

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

[16] DMTF, "Open Virtualization Format Specification (OVF) Version 2.0.0," 2013. [Online]. Available: http://www.dmtf.org/standards/ovf

[17] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski, "Introducing STRATOS: A Cloud Broker Service," in *Proceedings of IEEE CLOUD 2012*. IEEE Computer Society, pp. 891–898.

[18] S. Zaman and D. Grosu, "An Online Mechanism for Dynamic VM Provisioning and Allocation in Clouds," in *Proceedings of IEEE CLOUD 2012*. IEEE Computer Society, pp. 253–260.

[19] M. Bjorkqvist, L. Chen, and W. Binder, "Opportunistic Service Provisioning in the Cloud," in *Proceedings of IEEE CLOUD 2012*. IEEE Computer Society, pp. 237–244.

[20] D. Zamboni, *Learning CFEngine 3: Automated System Administration for Sites of Any Size*. O'Reilly Media, Inc., 2012.

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

[22] T. Rosner, *Learning AWS OpsWorks*, 2013.

[23] enStratus Networks, Inc., "Cloud DevOps: Achieving Agility Throughout the Application Lifecycle," 2012.

[24] RightScale, Inc., "Chef with RightScale," 2012. [Online]. Available: http://www.rightscale.com/solutions/managing-the-cloud/chef.php

[25] D. Sanderson, *Programming Google App Engine*. O'Reilly Media, 2009.

[26] J. Vliet, F. Paganelli, S. Wel, and D. Dowd, *Elastic Beanstalk*. O'Reilly Media, 2011.

All links were last followed on January 13, 2014.