Institute of Architecture of Application Systems

Enabling Dynamic Deployment of Cloud Applications Using a Modular and Extensible PaaS Environment

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{Wettinger2013a,
  author    = {Johannes Wettinger and Vasilios Andrikopoulos and
               Steve Strauch and Frank Leymann},
  title     = {Enabling Dynamic Deployment of Cloud Applications
               Using a Modular and Extensible PaaS Environment},
  booktitle = {Proceedings of the 6th IEEE International Conference on Cloud
               Computing, CLOUD 2013, 27 June – 2 July 2013,
               Santa Clara, CA, USA},
  year      = {2013},
  pages     = {478--485},
  publisher = {IEEE Computer Society}
}
```


Universität Stuttgart
Germany

Enabling Dynamic Deployment of Cloud Applications Using a Modular and Extensible PaaS Environment

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*—The Platform as a Service (PaaS) model promotes the development and deployment of applications on top of middleware solutions offered by the provider. Deploying an application in this model entails both the deployment of the application on top of the platform, but potentially also the deployment of the middleware components required by the application. Existing works enable the abstraction from the underlying infrastructure and allow for the creation of generic deployment plans for middleware components that automate the deployment of applications. In this paper we propose a middleware-oriented deployment approach that defines how the deployment of middleware components can be defined in a manner that allows to offer them as PaaS building blocks, and enable the automatic deployment of application components on them. We also present an evaluation of our proposal, together with the lessons learned during this activity.**

***Keywords*-middleware abstraction; infrastructure abstraction; middleware-oriented deployment; PaaS model; Cloud computing**

I. Introduction

As the Cloud service market is maturing, the Platform as a Service (PaaS) model is becoming more popular with application developers. The PaaS model, as defined by NIST [1], provides the capability to the consumer to deploy onto the Cloud infrastructure applications based on languages and tools supported by the provider. In this manner, PaaS allows application developers to pick and choose *middleware components* from one or more Cloud service providers, on top of which they can develop their application-specific logic as *application components*. Deploying an application in this context involves the deployment and wiring of the application components across one or more middleware components. Deployment of the middleware components themselves may also be required in case the PaaS solution is built on top of an Infrastructure as a Service (IaaS) service or another PaaS solution in a nested manner (e.g., deploying a BPEL engine on Google App Engine). This latter case is the focus of this work.

Enabling the PaaS model in this context requires the ability to offer middleware components that can be deployed across different infrastructures on demand. Both IaaS and PaaS offerings can be used for hosting these components. This in turn, would empower application developers to define their applications in an implementation-agnostic manner that allows for different middleware components to be used for their purposes. There are therefore two distinct goals to be satisfied: *1)* to allow the portability of application deployment across different Cloud infrastructures, and *2)* to enable flexibility in middleware component selection on the application developer side through the automation of the deployment of both application and middleware components.

Existing approaches in the literature such as [2], [3], and [4] are geared towards deploying whole application stacks in Virtual Machine images (VMs), following the IaaS model that has been dominant in the Cloud service market. While useful in their own right, the lack of separation between application and middleware components makes such approaches not directly applicable for the purposes of deploying application in PaaS solutions. On the other hand, tools originating in the DevOps community [5] such as Chef[1] or Puppet[2], allow for a finer degree of granularity in deploying both application and middleware components in a flexible manner. However, they require application developers to create their own, application-specific deployment plans with little or no reusability. Nevertheless, such tools can be leveraged to deploy middleware components in a generic, reusable across applications manner; this is the approach followed here.

The main contributions of this work can therefore be summarized as:

- A middleware-oriented deployment methodology, geared towards the PaaS delivery model, that prescribes a separation of concerns in the deployment of applications, and focuses on decoupling the application logic from the necessary middleware and infrastructure resources.
- An evaluation of this methodology through a scenario of non-trivial complexity demonstrating the efficacy and efficiency of our proposal.
- A set of best practices in creating deployment plans for middleware components enabling PaaS solutions, expressed as requirements in implementing the proposed methodology.

The remaining of this paper is structured as follows:

[1]Chef: http://www.opscode.com/chef
[2]Puppet: https://puppetlabs.com

Section II motivates our work by presenting a scenario developed as part of a European Union-funded project that is used throughout the paper. Section III surveys the existing work to establish the State of the Art in component deployment in the Cloud and position our work with respect to it. Section IV presents our proposal for a middleware-oriented deployment methodology that can be used both to enable PaaS solutions, and to facilitate the automation of application deployment on top of such solutions. The methodology is evaluated in Section V based on the scenario discussed in Section II. A set of requirements on how to define the deployment plans for middleware components is enumerated in Section VI as the lessons learned during the evaluation of our methodology. Finally, Section VII concludes the paper and presents some future work.

II. Motivating Scenario

The European Union's research project 4CaaSt [6] aims to create a Cloud platform to design services and compositions based on Cloud-aware building blocks provided by the platform, offer them in a marketplace, and operate them at Internet-scale. The goal of the 4CaaSt platform is to lower the entry barrier for small and medium enterprises by offering an advanced environment reducing the effort to create innovative applications leveraging the benefits of Cloud computing. In particular the 4CaaSt platform addresses the needs of *application developers* and *service providers*. The latter are using 4CaaSt to offer applications as a service to customers via the 4CaaSt marketplace. An application developer utilizes 4CaaSt to build new applications by combining external and 4CaaSt platform internal building blocks and services, e.g., Cloud-enabled application server, workflow engine, database, or Google Maps Web Services [7].

In the scope of 4CaaSt the *Taxi Scenario* has been defined, where a service provider offers a taxi management software as a service to different taxi companies, i.e., *tenants*. Taxi company customers, who are the *users* of the tenant, submit their taxi transportation requests to the company that they are registered with. The taxi company uses the taxi management software to contact nearby taxi drivers. Once one of the contacted taxi drivers has confirmed the transportation request, the taxi management software sends a transport notification containing the estimated arrival time to the customer. A video demonstrating the Taxi Scenario is available at http://tiny.cc/4caast-taxi-demo.

Figure 1 provides an overview of the realization of the Taxi Scenario. The topology consists of *middleware components, application components,* and external services. The taxi management software (back-end) is implemented as a business process using BPEL [8] and hosted on the 4CaaSt platform. The taxi management software leverages 4CaaSt platform internal context integration processes to retrieve context information from 4CaaSt platform internal Context as a Service, which provides context information about taxi cab locations and taxi driver contact details. Moreover, Google Maps Web Services [7] provide distance calculations between the location of a taxi cab and the pick up location.

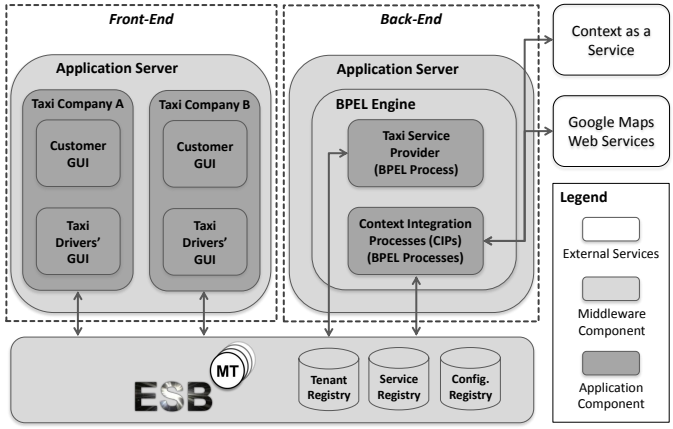

Figure 1. Overview of the Taxi Scenario

The taxi companies specific front-ends consist of a Customer GUI and a Taxi Drivers' GUI. The registered customers are sending transportation requests and receiving transport notifications. The taxi drivers are supported by a map-based interface, which navigates them to their customer's location.

Introducing an ESB as the messaging middleware (Fig. 1) enables loose coupling and provides a flexible integration solution by avoiding hard-coded point-to-point connections. This makes the monitoring, management, and maintenance of the taxi application easier and more effective. Furthermore, enabling multi-tenancy at the ESB level allows *multiple* taxi companies to use the same taxi application offered as a service by a *single* provider. Thus, apart from allowing taxi companies to outsource the development, deployment, operation, and management of such an application to a service provider, this solution also maximizes the benefits on the provider side.

The realization shown in Fig. 1 is currently deployed in one virtual machine (VM) using Flexiant's Flexiscale offering[3], but it is also possible to split the deployment into several VMs, e.g., by distributing the front-end, back-end, and integration middleware as we have done for the purpose of performance evaluation of ESB^{MT} [9].

III. Background

To deploy an application owning a complex architecture including multiple tiers such as the Taxi Scenario described in Section II, different approaches can be used that are State of the Art. According to the NIST definition of Cloud computing [1], there are three different service models: Infrastructure as a Service (IaaS), Platform as a Service

[3]Flexiscale: http://www.flexiscale.com

(PaaS), and Software as a Service (SaaS). In the following, we discuss different deployment approaches based on the IaaS and PaaS model. Deployment is not relevant for the SaaS model because this service model transparently exposes software services that can be configured and used directly.

The first approach is based on the IaaS model, meaning to encapsulate the different middleware components and application components in virtual machine images. Today, there are many IaaS providers that can be used to deploy virtual machine images such as Amazon Web Services (AWS)[4]. In addition, there are open source products such as OpenStack [10] available to create an IaaS environment for deploying virtual machine images. The Open Virtualization Format (OVF) [11] aims to be a standardized format for such images. For the Taxi Scenario described in Section II, a virtual machine image can be created to host the ESB. Another image may encapsulate the application server and the BPEL engine of the taxi service provider (back-end). The application server for each taxi company (front-end) can be hosted on separate virtual machines. Alternatively, multiple application servers can share a single virtual machine. Several approaches are available that are focused on optimized provisioning of virtual machines and deploying virtual machine images such as [2], [3], and [4].

Focusing on the IaaS model, there is another deployment approach available: instead of completely pre-installing and pre-configuring virtual machine images, simple standard images can be used that basically provide a plain operating system only. Then, configuration management tooling originating in DevOps communities [5] such as Chef[5] or Puppet[6] can be used to install and configure the actual middleware and application components. Scripts are used to perform the installation and configuration [12]. To manage topologies consisting of several machines and different components hosted on them, model-driven tooling such as AWS CloudFormation[7], AWS OpsWorks[8], or Juju[9] can be used. An efficient way of combining configuration management with model-driven management is described in [13]. In addition, there are holistic management services available such as EnStratus [14] or RightScale [15]. These use Chef scripts in the background to perform the actual deployment. For the Taxi Scenario 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.

Moving toward the PaaS model, the deployment can be performed using existing platform offerings such as Google App Engine [16] or Amazon Elastic Beanstalk [17]. 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 Scenario using the PaaS model, several "middleware services" are required to be exposed by the platform such as an ESB and a BPEL engine. Such middleware services may not be offered out of the box by PaaS providers. Consequently, a custom PaaS environment can be built based on existing infrastructure resources. As an example, the PaaS framework Cloud Foundry[10] enables this approach. Further, there is research going on focused on realizing a federated multi-Cloud PaaS environment [18].

For both the IaaS model and the PaaS model, the structure and behavior of a complex application consisting of different types of components and multiple tiers can be specified by a higher-level model-driven approach. The goal is to have a holistic model for a particular Cloud service to be used to deploy the service. Two examples to realize this approach are the Topology and Orchestration Specification for Cloud Applications (TOSCA) [19] and Blueprints [20]. Whereas TOSCA is an emerging standard [21], Blueprints are originating in the 4CaaSt project [6]. In addition, there are commercial products available that implement the model-driven approach. An example for these products is the IBM SmartCloud Orchestrator[11]. 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.

The approach proposed in the following section is not isolated from the deployment approaches described before; it is meant to be combined with these to specify both the lower-level aspects (scripts and platform services) and the higher-level aspects (service topology) of a particular Cloud service.

IV. Proposed Methodology

Based on the discussion in the previous section, automating the deployment of applications in Cloud environments revolves around two types of approaches:

1) *Top-down* approaches that start with an abstract application topology and resolve it into concrete deployment plans by adding application-specific logic in the deployment (e.g., TOSCA and Blueprint-based approaches).
2) *Bottom-up* approaches that propose the composition of existing deployment plans (usually, but not necessarily defined for one infrastructure solution) as the means to deploy an application stack (DevOps-oriented tools).

[4]Amazon Web Services: http://aws.amazon.com
[5]Chef: http://www.opscode.com/chef
[6]Puppet: https://puppetlabs.com
[7]AWS CloudFormation: http://aws.amazon.com/cloudformation
[8]AWS OpsWorks: http://aws.amazon.com/opsworks
[9]Juju: http://juju.ubuntu.com
[10]Cloud Foundry: http://www.cloudfoundry.org
[11]IBM SmartCloud Orchestrator: http://ibm.co/CPandO

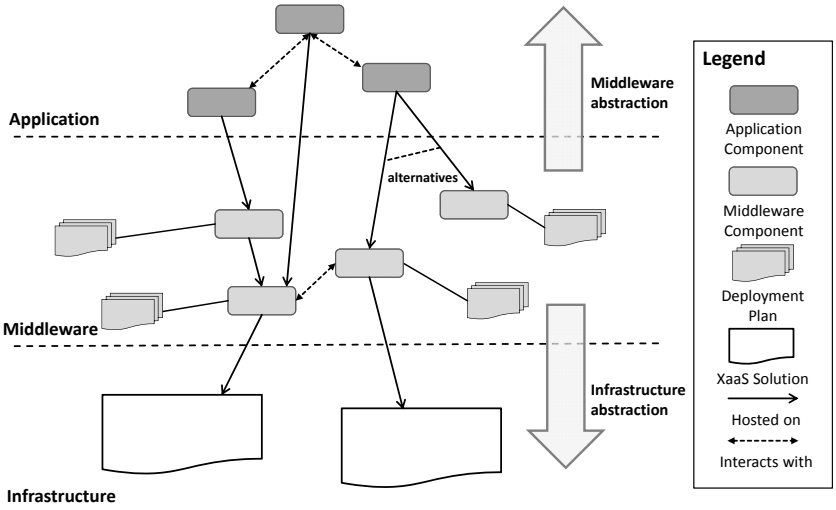

Figure 2. Middleware-oriented Deployment Methodology

Top-down approaches enforce a *middleware abstraction* allowing for flexibility in choosing which middleware solution to be incorporated. In the case of the Taxi Scenario for example, starting from a Blueprint description of the application the application developer may choose for the application server component (Fig. 1) either an application server such as JOnAS[12], or a simple servlet container such as Apache Tomcat[13]. Application-specific logic pertaining to how the application components (e.g., Taxi Company A/B WAR files) should be deployed, configured, and wired for communication in the chosen server for a chosen infrastructure solution can then be semi-automatically generated. In this respect, top-down approaches promote flexibility, but require additional effort that will need to be replicated across different applications.

Bottom-up approaches on the other hand promote a view of *infrastructure abstraction,* that minimizes the effort required for deployment, configuration, and wiring. This is achieved through the definition of generic deployment plans on the level of both application and middleware components that can then be reused for different applications across different infrastructures. Some configuration and wiring effort may be required for application-specific components, but this could ideally be folded in the deployment plans for the components. While these approaches focus on reusability, the available component options for the application developer are limited to the ones that deployment plans have already been defined for, reducing flexibility in application design.

Addressing the gap between the two types of approaches, we propose a methodology to deploy the Taxi Scenario and other Cloud services in a way that we gain the benefits of the PaaS model and minimize its drawbacks. Our goal is to have generic middleware components that can be hosted on different infrastructures, on top of which application components defined in abstract application topologies can be deployed. The middleware components shall be reusable, meaning that the very same components can be used to deploy similar application components such as WAR files for a completely different Cloud service. Consequently, these generic middleware components can be used to build a modular and extensible PaaS environment without sticking to a particular PaaS provider or framework. A model-driven approach can be used to wire the involved components. In order to leverage the benefits of both top-down and bottom-up approaches, we combine them in a best-of-breed manner.

For existing PaaS offerings such as Google App Engine, developers have to follow a particular programming model because the platform may not expose all features of the underlying technologies [16]. Reasons for this may be security risks or scalability issues. The methodology described in this section does not restrict the programming model in general. These restrictions completely depend on the underlying middleware components that realize a particular PaaS environment. As an example, a scalable middleware component may not expose features that prevent it from scaling. Consequently, it depends on the actual use case which middleware components to choose, considering the restrictions regarding the programming model and the properties of the middleware components involved such as scalability or security.

Figure 2 summarizes the main concepts of our *middleware-oriented deployment methodology.* The figure is centered around a three level view of Cloud applications: an *ap-*

[12]JOnAS: http://jonas.ow2.org
[13]Apache Tomcat: http://tomcat.apache.org

plication model level focusing on application components such as the GUIs in the Taxi Scenario, a *middleware* level that consists of components facilitating the operation of applications (e.g., the application server and ESB^{MT}) together with their generic deployment plans, and an *infrastructure* level that represents the XaaS solution actually used to host the application.

Following this view, the automation of the deployment of an application on a Cloud infrastructure consists of two major tasks:

- The definition of an abstract topology of the application that distinguishes between application components implementing the application logic (not reusable), and middleware components to support the application components (with an emphasis on their reusability).
- The creation of a set of generic deployment plans for various middleware components that can be stored and reused for multiple applications across different infrastructures. These plans are defined in a generic manner that allow for the deployment, configuration, and wiring of application components on top of them on an application-specific basis.

Application deployment can then be seen either as the *selection of suitable middleware components* for the given set of application components (in a top-down manner), or as the *composition of available middleware components in a given infrastructure* to support the application components (bottom-up manner). In any case, the resulting software stack can then be deployed in the infrastructure solution of the developer's choice.

In the following section we explain how our proposal works in practice by using the Taxi Scenario to evaluate our proposal.

V. Evaluation

The purpose of the evaluation described in this section is to point out the benefits of the methodology described in Section IV compared to the State of the Art approaches discussed in Section III. This evaluation is done based on the Taxi Scenario outlined in Section II.

A. Evaluation Setting

As described in Fig. 1, the application components involved in the Taxi Scenario require several middleware components on which they are hosted. We deploy the Taxi Scenario in two variants. The first variant is shown in Fig. 3 and uses JOnAS version 5.3.0-M4 as an application server to host both the front-end and the back-end. The second variant, shown in Fig. 4, uses Apache Tomcat version 6.0.24 as a servlet container instead of JOnAS. The other middleware components and application components are the same. Consequently, the two variants differ in a single middleware component only: JOnAS application server versus Apache Tomcat servlet container.

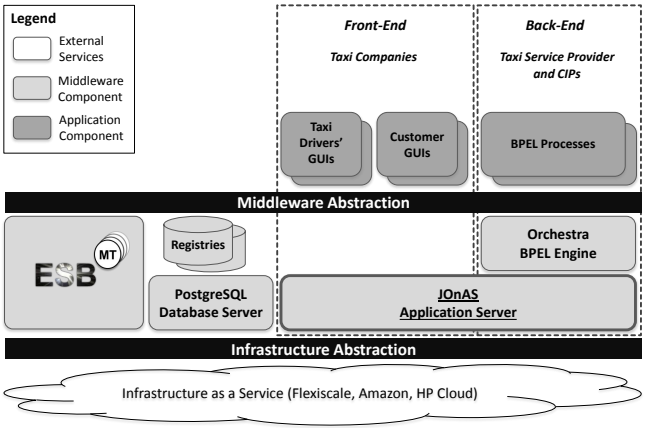

Figure 3. Deployment of the Taxi Scenario using JOnAS

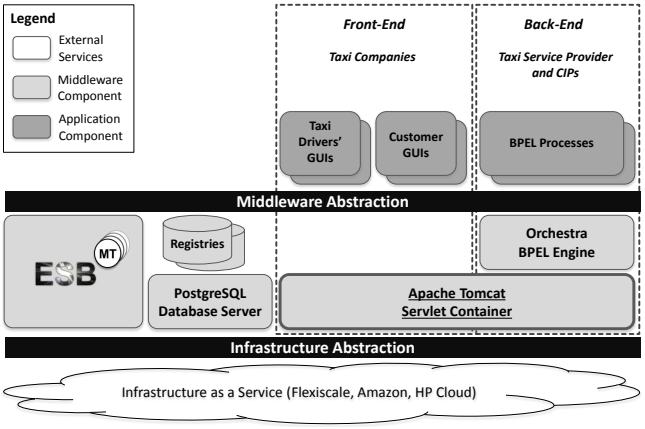

Figure 4. Deployment of the Taxi Scenario using Apache Tomcat

Whereas the integration of the application components from an architecture point of view is realized using Web service technology [22], the actual deployment of all the components and their wiring is realized using Chef. For this purpose, we created configurable Chef cookbooks consisting of recipes (scripts) to provide generic deployment plans for the middleware components. Following the 4CaaSt deployment approach [23], each cookbook provides four deployment recipes that are executed in the given order:

1) *Deploy-PIC* deploys the actual middleware component, also called "Platform Instance Component" (PIC) in 4CaaSt terminology.
2) *Start-PIC* starts the middleware component.
3) *Deploy-AC* deploys the application components (AC) hosted on top of the middleware.
4) *Start-AC* starts the application components that are running on top of the middleware.

Both variants are deployed on three different IaaS offerings, namely Amazon Web Services, Flexiscale, and HP Cloud. All middleware components and application components are deployed on a single VM with two CPU cores and four

gigabytes of RAM. Ubuntu Server 10.04 LTS (64-bit version) is the operating system installed on the VM. The ESB^{MT} implementation is based on Apache ServiceMix 3.4.0 and PostgreSQL 9.1 [9]. In addition, Orchestra 4.9.0-M3 provides the BPEL engine to execute the business processes.

B. Results and Lessons Learned

Table I shows the number of deployment plans that are required for the Taxi Scenario with and without middleware abstraction. In the case of not applying middleware abstraction, twice as many deployment plans (Chef cookbooks) have to be created because a separate deployment plan is required for each application component. Of course, the cookbooks enabling middleware abstraction are more complex because of their generality. However, the generality makes them reusable for many other deployment scenarios. For this evaluation both deployment variants are performed with middleware abstraction, i.e., creating generic Chef cookbooks, because this is the goal of our middleware-oriented deployment methodology.

Middleware components can be exchanged transparently without modifying the application components or other middleware components. In this evaluation we exchanged JOnAS with Apache Tomcat. The *Deploy-AC* recipe that is part of the deployment plan attached to the application server middleware component takes care of deploying the application components. Consequently, the application components are completely decoupled and do not refer to a specific middleware component. The middleware abstraction further enables the usage of the same generic *Deploy-AC* recipe of a particular middleware component to deploy other application components of the same type (e.g., WAR files or BPEL process models).

Furthermore, following this approach deployment plans are not bound to a specific IaaS provider. This is due to the fact that Chef runs on any kind of virtual machine. Furthermore, Chef cookbooks can be created in a way so that the same cookbook runs on multiple operating systems by using a domain-specific language that is platform-independent [12]. This enables the *infrastructure abstraction,* so the deployment plans are reusable across different infrastructures.

Table II summarizes our measurements for the deployment time of the Taxi Scenario in both variants. The measurements show a trend that the deployment using Apache Tomcat is faster. This behavior does not depend on a particular IaaS provider. We conclude that this is because Apache Tomcat is a simple servlet container, whereas JOnAS as an application server is a more complex middleware component. Consequently, the deployment time depends on the choice of middleware components. The differences in the deployment time regarding the IaaS providers is due to their internal resource management, which is out of scope of our evaluation.

Table I
DEPLOYMENT PLANS REQUIRED

With middleware abstraction	**Without middleware abstraction**
ESB^{MT}	ESB^{MT}
PostgreSQL	PostgreSQL
JOnAS	JOnAS
	Taxi Driver GUI
	Customer GUI
Apache Tomcat	Apache Tomcat
	Taxi Driver GUI
	Customer GUI
Orchestra	Orchestra
	BPEL Process
Sum: 5 deployment plans	**Sum:** 10 deployment plans

VI. DERIVED REQUIREMENTS

In the following, we present and discuss an initial list of requirements for enabling the middleware-oriented deployment methodology. We derive these requirements from the evaluation and lessons learned described in the previous section:

- R_1 *Middleware-orientation:* To enable middleware abstraction, deployment plans are always bound to middleware components. Application components do not have deployment plans attached. These are also deployed using the deployment plans of the middleware components. The plans are generic and not application-specific, so different application components of a particular type (e.g., WAR files) can be deployed.
- R_2 *Composable deployment plans:* Instead of implementing a monolithic artifact such as a huge script to perform the deployment of an application, composable deployment plans shall be implemented. These plans are focused on deploying specific middleware and application components and thus can be recombined and reused for different deployment scenarios.
- R_3 *Configurable deployment plans:* To make deployment plans reusable for different deployment scenarios they have to be configurable, e.g., by parameterization. Thus, configuration parameters shall not be hard-coded inside these plans.

Table II
DEPLOYMENT MEASUREMENTS

Taxi Scenario Deployment	with JOnAS	with Apache Tomcat
on Amazon Web Services	589 sec.	523 sec.
on Flexiscale	512 sec.	434 sec.
on HP Cloud	476 sec.	378 sec.

- R_4 *Distributed deployment:* Different components of a single application can be distributed across several machines. Thus, the corresponding deployment components need to be created in a way that they can deploy an application in a distributed manner. Consequently, different application components can be hosted on different machines.
- R_5 *Modularity:* Middleware components including their deployment plans shall be designed and created in a modular fashion: each "module" represents a particular middleware component. This enables separation of concerns. The goal is to combine a set of middleware components to establish a PaaS environment that can host applications which require the corresponding middleware.
- R_6 *Extensibility:* To enable the evolution, refinement, and adaptation of middleware components, their design and deployment plans shall foster extensibility. This is key to cover new features of updated versions of the middleware.
- R_7 *Portability:* Deployment plans shall be realized in a platform-independent manner to enable infrastructure abstraction. This increases the degree of portability, so they can be used to deploy middleware on different platforms, e.g., on different variants of Linux.

VII. Conclusion and Future Work

The previous sections discussed our proposal for enabling middleware components as building blocks in a PaaS solution. The existing work has been identified either as focused on deploying whole application stacks (top-down), or as DevOps tools that enable reusability but require additional effort by application developers to define them (bottom-up). Our proposal bridges the gap between the two approaches by proposing the clean separation between middleware and application components, with different requirements for their deployment. Plans for the deployment of middleware and application components are proposed to be as generic as possible, allowing for flexibility in selecting and deploying them as part of a bigger application. As shown by our evaluation, this approach significantly decreases the number of plans required for the deployment of an application. We also show that, overall, the time in deploying an application stack using our approach depends not only on the underlying infrastructure solution, but also on the selected components. Flexibility in selecting middleware components can therefore be seen as crucial for performance purposes.

In terms of future work, a more thorough evaluation of our proposal across more platforms and deployment tools (e.g., Puppet) is currently underway. Part of this evaluation is a comparison of the elapsed deployment time for application-specific deployment plans, i.e., those without middleware abstraction in Table I, against the performace of enacting the middleware-oriented deployment plans reported in Table II. This would allow us to evaluate the possible overhead of our proposal. The creation of a repository of reusable middleware components providing access to application developers would further contribute towards the direction of evaluating our proposal using different applications in practice.

Furthermore, two aspects not covered by this paper are also under investigation. Firstly, the possibility of enabling dynamic wiring between application and middleware components, meaning that the dependencies between components are resolved during deployment. Currently, we use a static wiring approach, i.e., we express explicitly these dependencies in the application model. Secondly, scalability of the application has been so far out of the scope of this discussion. Both the mechanisms required to be implemented by the middleware components, and the mechanisms offered by the underlying infrastructure have to be taken into consideration. Beyond enriching our methodology, incorporating these aspects is expected to result into further requirements on the deployment plan definition of middleware components.

Acknowledgments

The research leading to these results has received funding from the 4CaaSt project (http://www.4caast.eu) part of the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 258862.

References

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

[2] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski, "Introducing STRATOS: A Cloud Broker Service," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, 2012, pp. 891–898.

[3] S. Zaman and D. Grosu, "An Online Mechanism for Dynamic VM Provisioning and Allocation in Clouds," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, 2012, pp. 253–260.

[4] M. Bjorkqvist, L. Chen, and W. Binder, "Opportunistic Service Provisioning in the Cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, 2012, pp. 237–244.

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

[6] 4CaaSt Consortium, "4CaaSt Project Website." [Online]. Available: http://www.4caast.eu

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

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

[9] S. Strauch, V. Andrikopoulos, S. G. Sáez, F. Leymann, and D. Muhler, "Enabling Tenant-Aware Administration and Management for JBI Environments," in *Proceedings of SOCA'12*. IEEE Computer Society Conference Publishing Services, 2012, pp. 206–213.

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

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

[12] 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.

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

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

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

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

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

[18] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier, "A Federated Multi-Cloud PaaS Infrastructure," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 392–399.

[19] 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.

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

[21] 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

[22] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana *et al.*, "Web Services Description Language (WSDL) 1.1," 2001.

[23] S. García-Gómez, M. Jiménez-Gañán, Y. Taher, C. Momm, F. Junker, J. Bíró, A. Menychtas, V. Andrikopoulos, and S. Strauch, "Challenges for the Comprehensive Management of Cloud Services in a PaaS Framework," *Scalable Computing: Practice and Experience (SCPE)*, vol. 13, no. 3, pp. 201–213, November 2012.

All links were last followed on May 3, 2013.