Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies

Uwe Breitenbücher, Tobias Binz, Oliver Kopp,
Frank Leymann, and Johannes Wettinger

Institute of Architecture of Application Systems, University of Stuttgart, Germany
Universitätsstraße 38, 70569 Stuttgart, Germany
{breitenbuecher,lastname}@iaas.uni-stuttgart.de

Abstract Modern Cloud applications employ a plethora of components and XaaS offerings that need to be configured during provisioning. Due to increased heterogeneity, complexity is growing and existing approaches reach their limits if multiple different provisioning and configuration technologies are involved. They are not able to integrate them in an automated, flexible, and customizable way. Especially combining proprietary management services with script-centric configuration management technologies is currently a major challenge. To enable automated provisioning of such applications, we introduce Generic Lifecycle Management Planlets that provide a means to combine custom provisioning logic with common provisioning tasks. We implemented planlets for provisioning and customization of components and XaaS offerings based on both SOAP and RESTful Web services as well as configuration management technologies such as Chef to show the feasibility of the approach. By using our approach, multiple technologies can be combined seamlessly.

Keywords: Cloud Application Provisioning, Integration, Management Scripts, Management Services.

1 Introduction

With growing adoption of Cloud computing, the automated provisioning of composite Cloud applications becomes a major issue as this is key to enable Cloud properties such as on-demand self-service, pay-as-you-go pricing, and elasticity. However, due to various kinds of different components and XaaS offerings employed in modern composite Cloud applications and the dependencies among them, the complexity and heterogeneity is constantly increasing. This becomes a challenge if the components and XaaS offerings employ different management technologies and need to be combined and customized during provisioning. Especially application-specific provisioning and customization tasks such as wiring custom components and standard XaaS offerings together cannot be implemented in a generic and reusable way. In addition, these tasks are typically

R. Meersman et al. (Eds.): OTM 2013, LNCS 8185, pp. 130–148, 2013.

implemented by various kinds of different heterogeneous provisioning technologies. Although wiring and configuration of (custom) components are typically implemented using script-based technologies such as Puppet[1], Chef[2], CFEngine[3], or Juju[4], provisioning and configuration of XaaS Cloud offerings such as *Infrastructure as a Service* or *Database as a Service* are typically provided through Web service APIs—mostly HTTP-based Query services, RESTful Web services, or SOAP Web services. As a result, available provisioning approaches reach their limits: in case multiple standard components, custom components, and XaaS offerings provided by different vendors and Cloud providers are combined and different provisioning and configuration technologies are involved, available solutions are unable to integrate them. In this paper, we tackle this issue. We present an approach to enable the seamless integration of script-centric and service-centric provisioning and configuration technologies in order to customize and automate provisioning of composite Cloud applications. Therefore, we extend our concept of Management Planlets [2] by *Generic Lifecycle Management Planlets (GLMPs)* that provide a means to bind abstract lifecycle tasks to script- or service-based operation implementations. This enables the seamless integration of different technologies to customize the generation of an overall provisioning flow that provisions the application fully automated. The extension enables application developers to benefit from reusable common provisioning logic implemented by third parties and individual customization possibilities. We validate the approach by creating several GLMPs that support the integration of service-based technologies such as RESTful and HTTP Query Web services as well as script-based technologies such as Chef. To prove the benefits, we evaluate the concept against existing approaches in terms of functionality and features. In addition, we implemented a prototype to show its practical applicability.

The remainder of this paper is structured as follows. In Section 2, we motivate our approach, introduce a motivating scenario, and describe why the related work is not able to tackle the analyzed issues. Afterwards, we describe Management Planlets and Provisioning Topologies in Section 3. In Section 4, we present our approach to integrate script- and service-centric provisioning technologies. We present a case study in Section 5 and evaluate the approach in Section 6. Finally, Section 7 concludes and provides an outlook on future work.

2 Motivation, Scenario, and Related Work

In this section, we motivate our approach and describe the type of applications whose provisioning is the focus of this paper. Afterwards, we describe the provisioning of a motivating scenario and identify the occurring challenges and problems. The related work, that does not provide a means to tackle these issues completely, is discussed in Section 2.3.

[1] `http://puppetlabs.com/puppet/what-is-puppet`
[2] `http://www.opscode.com/chef`
[3] `http://cfengine.com`
[4] `https://juju.ubuntu.com`

2.1 The Cloud Applications to be Provisioned

This paper considers Cloud applications that are of small and medium size and complexity such as CRM systems. They are based on multiple XaaS offerings possibly of different providers and employ common as well as individual software components. We use a PHP-based Web shop application that stores product data

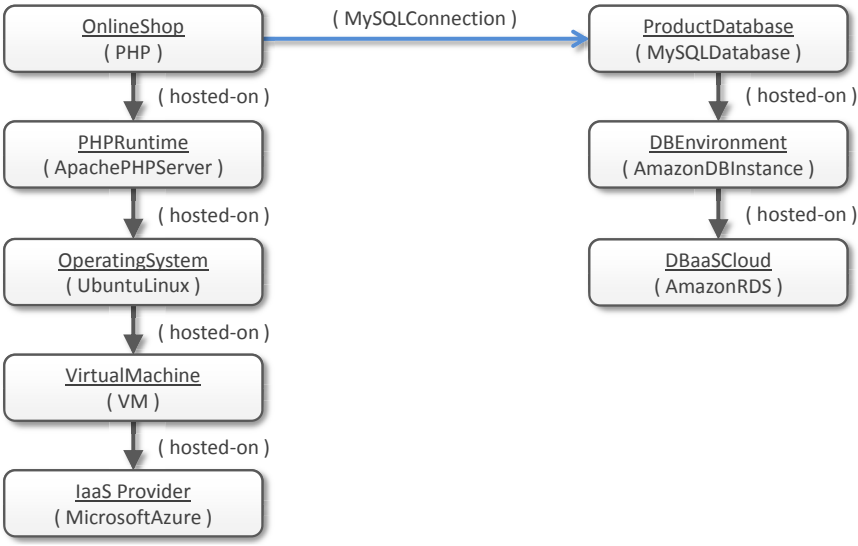

Fig. 1. Motivating Scenario

in a relational database as running example throughout this paper. The application is based on two Cloud offerings of type infrastructure and database as a service: the infrastructure is provided by Microsoft's Windows Azure Cloud offering[5] and the database by Amazon's Relational Database Service (AmazonRDS[6]). Figure 1 shows the application modeled as application topology. A topology is a graph consisting of nodes, which represent the components, and edges, which represent the relations between the components. We refer to nodes and relations as *elements* in the following. Each element has a certain type that defines its semantics and properties, which are key-value pairs. Types may inherit from a super type, e. g., Ubuntu inherits from Linux. We use Vino4TOSCA [1] to render topologies. Thus, types are denoted as text enclosed by parentheses and element ids as underlined text. The application itself consists of two connected stacks. The left stack hosts the business logic implemented in PHP. This is denoted by the node of type PHP on the top left. The PHP node is hosted on a PHP Runtime of type ApachePHPServer which runs on an Ubuntu Linux operating system. This Linux runs in a virtual machine (VM) hosted on Azure.

[5] `http://www.windowsazure.com/`
[6] `http://aws.amazon.com/rds/`

The product data are stored in a database node of type MySQLDatabase hosted on AmazonRDS. The connection between these stacks is established by a relation of type MySQLConnection which connects business logic with database backend. For simplicity, all other relations are modeled as "hosted-on" relations, which is the super type for "installed-on", etc. The architecture is a result of Cloud-related design rationales [5]. Reasons for using multiple Cloud providers are differences in pricing or quality of service and that a provider may not offer required services or features, e. g., AmazonRDS offers an automated backup functionality which is not supported by Azure currently.

2.2 Provisioning of the Web Shop Application

In this section, we describe the provisioning of our Web shop in detail. To provision the Ubuntu operating system and the virtual machine on Azure, the Windows Azure Service Management REST API[7] is invoked. The thereby instantiated VM is accessible via SSH. Hence, Chef can be used to install the Apache PHP Web server on it. Therefore, a Chef agent is installed on the operating system via SSH before. After the Web server is installed by executing the corresponding Chef recipes, we install God[8], which is a monitoring framework written in Ruby. To ensure high availability, we use God to make sure that the Web server is up—otherwise, a restart will be triggered automatically. After that, the PHP application files are transferred from an external storage onto the operating system and copied into the `htdocs` folder, which contains all applications that are hosted on the server. This is done via SSH and Secure Copy (scp), which is a means to securely transfer data between different hosts. To create the MySQL database instance on AmazonRDS, a single HTTPS call to Amazon's Query API[9] is sufficient. However, by default, network access to AmazonRDS instances is disabled. Thus, we authorize access before by creating a so called *security group* that defines the rules to make the database accessible for the PHP frontend hosted on another provider. This requires two HTTPS calls to Amazon's Query API. Afterwards, we setup frequently automated backups for the database to prevent data loss. This is also done by an HTTPS service call to the same API. After both application stacks are provisioned, initial product data is imported to the database. To do this, we employ an SQL batch update. In the last step, the PHP application needs to be connected to the database. Establishing this connection is application-specific as there is no standard or common way defining how to set such database endpoint information. Thus, only the Web shop developer knows how to configure the application to connect to the database. In our scenario, we employ a shell script that writes the database's endpoint information into a configuration file which is read by the PHP application. Such shell scripts typically need parameterization: the endpoint is passed to the script through environment variables which are read by the script.

[7] `http://msdn.microsoft.com/en-us/library/windowsazure/ee460799.aspx`
[8] `http://godrb.com/`
[9] `http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/`

2.3 The Challenge: How to Provision This Application Fully Automated? Tools and Related Work

The presented application architecture constitutes a set of problems. The fully automated provisioning is a challenge as various kinds of technologies are involved. Proprietary vendor- and Cloud-specific solutions such as Amazon Cloud Formation[10] are not able to tackle this provisioning issue completely as they consider a certain Cloud provider environment only and lack the integration of XaaS offerings from other providers. Proprietary multi-Cloud management services such as RightScale[11] aim to enable the provisioning and management of business-critical Cloud infrastructures across multiple IaaS providers based on automation scripts. However, the wiring of custom components and integration of various XaaS offerings is not possible. Cloud abstraction APIs such as OpenStack[12] provide a means to decouple dependencies to the underlying Cloud infrastructure and platform services in order to ease multi-Cloud applications. However, they do not solve the problem of integrating different management APIs, technologies, and XaaS offerings. The DevOps communities provide tools such as Puppet, Chef, or CFEngine and frameworks such as Marionette Collective[13] or Spiceweasel[14] to enable sophisticated configuration management [3, 6]. In addition, there are tools such as Juju to enable the orchestration of configuration management scripts. However, these script-centric approaches are mostly limited to installing and configuring software components on existing virtual machines. The deployment of complex composite Cloud applications that include the fully automated provisioning of various XaaS offerings with custom dependencies among each other is not trivial using these approaches as low-level scripts need to be written for integration. Another deficit of most approaches is that the wiring of different components across different machines, such as connecting the Web shop PHP frontend to the product database, cannot be modeled separately. Therefore, custom low level scripts must be manually embedded into the overall process what requires a lot of technical knowledge. Juju supports this kind of wiring, but also comes with the deficit that the used configuration management scripts are made to be executed on the target infrastructure such as a virtual machine that was provisioned before. To summarize, the main problem of all these tools is that a complete support for the provisioning, configuration, and wiring of virtual machines, storage, and other XaaS offerings, provided by different providers, is currently out of scope. Thus, they do not provide a means to integrate needed technologies to enable interoperable, multi-Cloud, and multi-vendor applications such as our Web shop. Besides implementing custom software or low level scripts from scratch that orchestrate all these technologies on their own, a common solution for this problem is implementing provisioning workflows that integrate required technologies, APIs, and abstraction frameworks as shown

[10] http://aws.amazon.com/cloudformation/
[11] https://www.rightscale.com
[12] https://www.openstack.org/
[13] http://docs.puppetlabs.com/mcollective
[14] http://wiki.opscode.com/display/chef/Spiceweasel

by Keller and Badonnel [7]. Provisioning workflows provide significant advantages in contrast to solutions from scratch: They inherit features from workflow technology such as recoverability, traceability, compensation-based recovery, and transactional support and they provide an accepted means for orchestration of heterogeneous software [9]. In addition, they support long-running processes and enable people involvement through human tasks, which may be needed to manage utilized physical hardware. Thus, provisioning workflows enable a flexible, reliable, and robust way to provision applications—even if multiple providers, vendors, and technologies are involved. However, implementing such workflows manually has two crucial drawbacks. (i) The required knowledge and effort is high. Developers do not only need the knowledge about the workflow language and its semantics itself but also have to know how to integrate and wrap technologies to make them accessible for workflows. This is a difficult, time-consuming, and error-prone task and often requires deep technical knowledge about certain technologies. Even using common workflow languages such as BPMN or BPEL needs a lot of detailed knowledge and, in addition, script-centric technologies have to be integrated which is not supported by BPMN and BPEL natively, for example. This causes a lot of glue code to wrap APIs and technologies. Especially script-based technologies provide some difficult challenges as they are typically tightly coupled to operating systems, need to be copied to target machines in advance, and employ different parameterization mechanisms, which are not interoperable. Thus, several steps are required before scripts can be executed. The seamless and transparent integration of different heterogeneous technologies is the major challenge if workflows are created manually. We developed a BPMN extension [8] that eases implementing provisioning workflows. However, this extension also does not solve the aforementioned problems completely as it also relies on the invocation of management services. Especially the second problem is not tackled by this extension: (ii) provisioning workflows are typically tightly coupled to a single application and hard to reuse and maintain [2]. If components or relations change, this needs to be adapted in the workflow. Thus, provisioning workflows must be created from scratch or by copying workflow fragments from other applications, which is an error-prone task. In summary, the manual implementation of provisioning workflows is hard, costly, and inefficient. Thus, we need a means to generate provisioning workflows for individual applications fully automated. The literature presents approaches that deal with this issue: Cafe is a framework that enables automating the provisioning of composite service-oriented Cloud applications [11]. It generates provisioning workflows by orchestrating so called "component flows" that implement a uniform interface to manage the provisioning of individual components. The work of Maghraoui et al. presents an operation oriented approach that enables transferring the current state of a data center into a desired state by orchestrating provisioning operations [10]. This orchestration is based on planning algorithms that investigate the preconditions and effects of each operation in order to determine the correct set and order of operations. The work of Eilam et al. also uses desired state models to provision applications [4]. In contrast to the previous work, it is based

on graph covering techniques to orchestrate so called "automation signatures" that implement provisioning logic. However, none of these approaches supports the direct and generic integration of script-centric and service-centric operations that provide custom provisioning logic implemented by the application developer itself. This is especially needed for wiring custom components as discussed in Section 2.2. Using the available approaches, application developers need to write glue code to embed custom logic, i. e., custom component flows, provisioning operations, and automation signatures need to be implemented. Thus, the application developer requires technical knowledge about the technologies that makes the automated provisioning of custom applications complicated and costly.

3 Management Planlets and Provisioning Topologies

In this section, we explain Management Planlets and Provisioning Topologies, which are a means to automate the provisioning of applications. We introduced Management Planlets in a former work [2] and extend them in this paper to support the direct and explicit integration of script-centric and service-centric provisioning technologies, which is not supported by the original approach. Management Planlets provide small reusable workflows that perform low level management tasks on a certain combination of nodes and relations, e. g., installing a Web server or instantiating a virtual machine. The purpose of planlets is to be orchestrated into an overall workflow that provides a higher-level functionality. Thus, they serve as generic building blocks for the generation of provisioning plans that provision an application fully automated. Planlets consist of two parts: (i) An *Annotated Topology Fragment* that depicts the management tasks performed by the planlet on the nodes and relations and (ii) a workflow implementing this functionality. The topology fragment contains (i) a graph of typed nodes that may be interconnected by typed relations and (ii) so called *Management Annotations* that are attached to the nodes or relations. These annotations describe abstract tasks to be performed on the associated element: each annotation has well-defined semantics but exposes no details about its actual implementation. Thus, they hide complexity and describe tasks decoupled from concrete implementations. All details about the technical implementation are hidden behind the topology fragment and implemented by the workflow. For example, the *Create-Annotation* specifies that the associated element gets instantiated or installed by the respective planlet. The concrete implementation is up to the planlet. In addition, planlets implement a uniform interface for invoking them and define input parameters that have to be provided by the caller, e. g., required account credentials. Due to these properties, planlets are capable of integrating different technologies into a common model without exposing technical details. As planlets implement their functionality as workflows, they inherit the features from workflow technology as described in Section 2.3.

Provisioning Topologies are used to define the provisioning of applications. A Provisioning Topology is an application topology that consists of nodes and relations annotated with Management Annotations. These annotations define which

tasks have to be performed to provision the application. Elements in Provisioning Topologies may specify properties and operations they provide. Operations provide information such as file references to scripts or URLs to service endpoints. This topology serves as input for a plan generator that generates a provisioning plan by orchestrating multiple planlets.

3.1 Management Annotations

There are two classes of Management Annotations: (i) *Structural Annotations* and (ii) *Domain-specific Annotations*. The first class is fixed and contains two generic annotations defining that the associated element gets *created* or *destroyed* by the planlet. Thus, they modify the topology in terms of its structure. The second class contains custom annotations that are needed to describe tasks of a certain domain. For example, an *ExportTables-Annotation* for database nodes defines that tables are exported by the planlet. Thus, this class makes the approach extensible towards all conceivable kinds of tasks. Furthermore, domain-specific annotations may define additional information in the form of properties which are used by the planlet for customization. For instance, the ExportTables-Annotation defines the tables that have to be exported and the target storage.

3.2 Preconditions and Effects

Planlets may define preconditions to be fulfilled prior to the execution of the planlet. Preconditions include value range restrictions on properties. Each property specified on an element contained in the planlet's topology fragment must be initially available in the topology model or set by another planlet before. To manage these properties, planlets write and read them from an *instance model*, which is based on the Provisioning Topology. The effects of planlets are expressed through Management Annotations and properties: Properties that are annotated with a Create-Annotation are created by the planlet and may be used by other planlets. For example, a planlet that deploys an application on a Web server may have preconditions in the form of server IP-Address and credentials which are set by another planlet that installed the Web server before. Thus, planlets use properties to communicate with each other indirectly. The preconditions and effects of planlets are matched against Provisioning Topologies to find a set of planlets that are able to provision the whole application.

Figure 2 presents an example. The Provisioning Topology depicted on the left gets provisioned by the two planlets shown on the right. The planlet on the bottom instantiates an Ubuntu Linux operating system running in a virtual machine on Amazon EC2. This is depicted by the topology fragment on the left of the planlet: the Ubuntu and VM nodes as well as the underlying hosted-on relations have a Create-Annotation attached and, thus, get instantiated by the planlet. This topology fragment is matched by the corresponding nodes and relations contained in the Provisioning Topology based on the types and annotations of the elements (depicted by the blue arrows). In addition, the state properties of both nodes get set to "instantiated" and the credentials and IP-address properties of the Ubuntu node

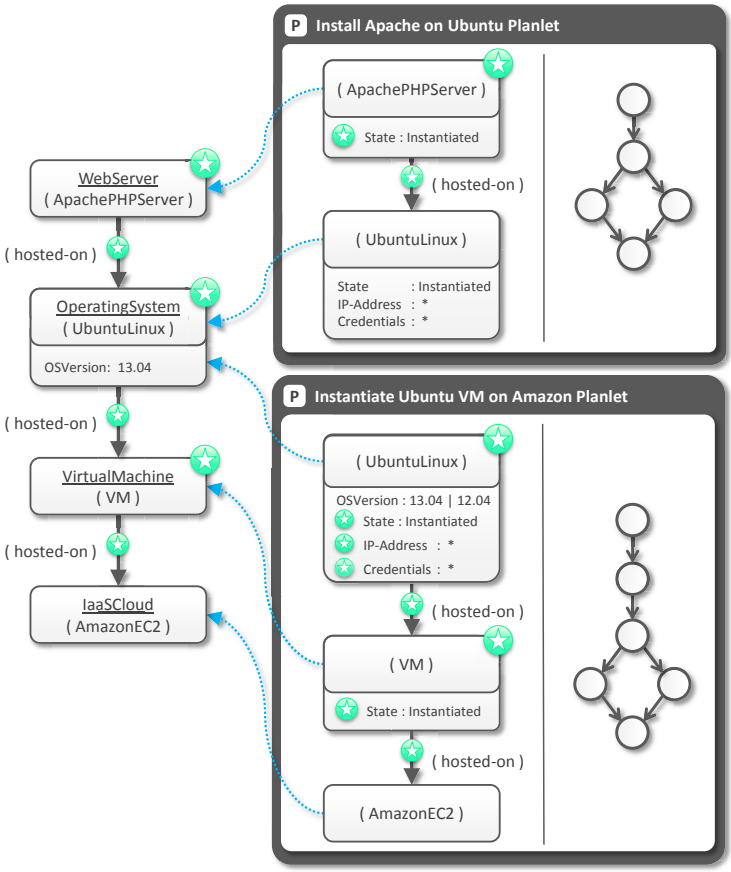

Fig. 2. Provisioning Topology (left) and matching Management Planlets (right)

to a meaningful value. This is expressed by the Create-Annotations attached to the corresponding properties. The planlet also defines a precondition in the form of the OSVersion property of the Ubuntu node: It is able to provision this stack for versions 13.04 and 12.04. This precondition is matched by the Provisioning Topology. As Amazon EC2 is always running, there is no need to instantiate that node explicitly. Thus, the planlet is applicable. The planlet on the top is able to install an ApachePHPServer on Ubuntu. It specifies preconditions on the Ubuntu node in terms of state, credentials, and IP-address, which are all fulfilled after the former planlet was executed. In general, preconditions determine the execution order of planlets.

3.3 Transparent Integration

In this section, we describe how Management Planlets are used to integrate script-centric and service-centric provisioning technologies transparently. Technical

details are hidden by planlets as the topology fragment exposes the functionality only in an abstract fashion. This abstraction allows combining service-centric and script-centric technologies without exposing any details about the implementation to the plan generator: The implementation details are up to the planlet creator. As a consequence, the plan generator and the application developer, who specifies the Provisioning Topology, do not have to care about the technical details. We call this *Transparent Integration* because all technical details are invisible. This kind of integration is suited for regularly used tasks in which standard or common components are involved that are both not specific to a certain application. For example, the instantiation of an Ubuntu virtual machine on Amazon EC2 is such a task in which service invocations are involved. Installing an Apache Web server on this virtual machine afterwards is a typical script task. Therefore the Chef community, for instance, provides recipes for this installation. All in all, both tasks are suited to be integrated transparently through *Generic Planlets* because each component as well as both tasks are common and likely to be reused. This transparent integration supports the reusability of expert knowledge across different domains and technologies. We implemented the plain provisioning of both stacks of our motivating scenario as Generic Planlets. That includes provisioning of VM and operating system, installation of the Web server, deployment of the PHP application, and the instantiation of DBEnvironment and product database.

3.4 Standards Compliance

Proprietary approaches as discussed in the related work (Section 2.3) are based on proprietary domain-specific languages (DSLs). DSLs prevent these approaches to be portable and accepted by a broad audience as the languages require special knowledge and technical skills. A main strength of Management Planlets is the applicability to the OASIS TOSCA standard [12], which provides a standardized format to describe application topologies and their management in a portable fashion. We proved this by a prototypical implementation [2].

3.5 Customization Drawback

As shown in the previous sections, planlets provide a means to abstract tasks from concrete operation implementations to automate their combined execution. Application developers only need to specify the desired tasks in a Provisioning Topology without having any doubt about the final execution. However, these Generic Planlets cannot serve all customization requirements the application developers may have. Custom components such as the Web shop frontend of our motivating scenario often need special consideration. For example, to establish the connection to the database, the application needs a configuration in the form of database credentials and endpoint information. As there is no standardized or common way for this task, it is not possible to implement a planlet that deals with this in a generic and reusable way. Thus, application developers need to implement specific planlets to inject custom behavior for an actually simple task. These so called *Custom Planlets* can be combined with the Generic Planlets in

a seamless way. This is, however, not always sufficient as writing planlets causes unnecessary overhead if only a simple service call or script execution is needed to serve the needs. In addition, the Custom Planlets are hardly reusable as they are targeted to a very special custom task. To tackle this issue, we extend the concept of Management Planlets in the following section. The presented extension enables configuration of provisioning by integrating service calls or script executions directly into the generated provisioning plan without corrupting the overall concept. The new approach enables application developers to customize provisioning without the need to write planlets by themselves.

4 Integrating Script- and Service-Centric Technologies

Management Planlets currently support the Transparent Integration of script-centric and service-centric provisioning technologies implicitly. However, if special tasks are needed, implementing Custom Planlets is not appropriate as discussed in Section 3.5. Thus, we need a means to *explicitly* integrate customization scripts and service calls generically. Therefore, we extend the concept of Management Planlets to enable an *Explicit Integration* of different technologies into the generated provisioning workflow. The combination of Transparent and Explicit Integration enables integrating script-centric and service-centric provisioning technologies in a seamless fashion. This allows application developers to benefit from Generic Planlets and to customize the provisioning at any point through using the Explicit Integration without the need to write own Custom Planlets.

4.1 Explicit Integration

In this section, we present the main contribution of this paper that provides a means to integrate script executions and service invocations of various types explicitly into the overall automatically generated provisioning flow. Therefore, we introduce *Generic Lifecycle Management Planlets* (*GLMP* from now on) that serve as generic technology integration mechanism to implement lifecycle actions. GLMPs enable custom implementation of provisioning and configuration logic specifically for a certain application, component, or relation. They are able to directly execute a specific script- or service-based operation implementation that implements one or multiple Management Annotations. Thus, GLMPs enable binding custom operation implementations to abstract tasks which are represented by Management Annotations. This enables application developers to inject own provisioning and configuration logic without the need to implement Custom Planlets. Thus, the operational logic gets distributed over the Provisioning Topology of an application and Management Planlets.

Figure 3 shows the general concept by an example describing how an HTTPS service call can be used to configure a node. On the left, it depicts the DBEnvironment of the motivating scenario. This node needs to be created and configured, as denoted by a Create-Annotation and a Configure-Annotation. It provides a SetupFrequentBackup operation of type HTTPS (1 in Figure 3) that implements (2)

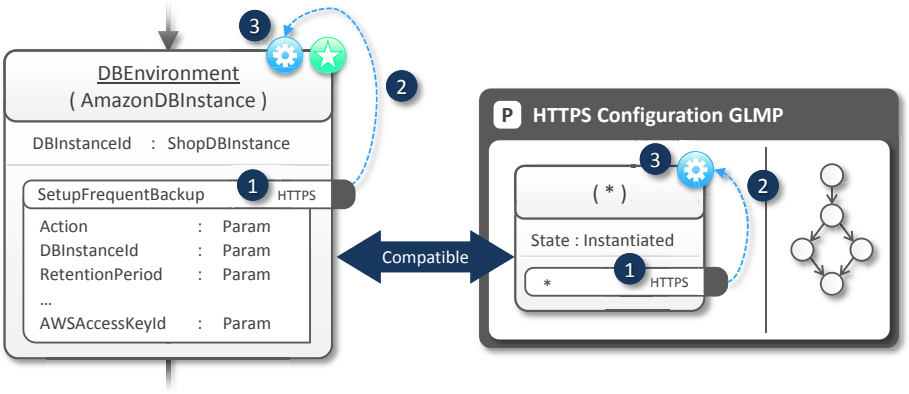

Fig. 3. DBEnvironment node with custom HTTPS operation bound to Configure-Annotation (left) and compatible HTTPS Configuration GLMP (right)

the Configure-Annotation (3). On the right hand side, the figure shows an HTTPS Configuration GLMP that matches the DBEnvironment node: The GLMP is applicable to single nodes of any type (denoted by the star symbol in the type parentheses). It is able to invoke services via HTTPS (1 in Figure 3) in order to configure a node (3). This is shown by the operation-annotation-binding (2). The planlet, however, has a precondition in the form of a state property that must be set to instantiated. Thus, this GLMP is applicable after the DBEnvironment node was created by another planlet that sets the state-property accordingly.

A GLMP is responsible to perform one or multiple lifecycle actions on an element. For this paper, we use a simple lifecycle that is sufficient for most provisioning scenarios [4]: each node and relation goes through the lifecycle actions *instantiation*, *configuration*, and *termination*. The instantiation action is represented by the Create-Annotation. Each planlet that performs this annotation sets the state-property on the corresponding element to "instantiated" after completion. This state-property serves as precondition for planlets that perform the Configure-Annotation on that element. Thus, the Configure-Annotation is always processed after the Create-Annotation. This enables a fine grained injection of provisioning logic in the desired phase of an element's lifecycle. Termination is out of scope for this paper. This lifecycle may be extended to support more complex needs, e. g., by executing Prepare-Annotations before instantiation.

Operation-Annotation-Binding. An operation may be implemented through various kinds of technologies. Thus, their integration mechanisms differ significantly from each other and need specific additional information. GLMPs offer a way to define information for each operation type individually: Each operation type defines its custom binding information as shown in Figure 4. The type of the operation defines its semantics and the meaning of this information. The example shows binding information for an HTTPS call. All elements contained

in the binding having the prefix "http" are technology-specific and ignored by the plan generator. They are used to provide needed information used by the GLMP, such as request URI and HTTP method in this example. The *Implements* elements specify the annotations that are implemented by the operation. This information is used by the plan generator to select GLMPs, which is explained in Section 4.1. Binding-information may not be available at design time, e. g., endpoints of configuration services provided by components themselves are typically not known until the component is provisioned. The *Data Handling Specification* introduced in the next section enables defining lazy bindings.

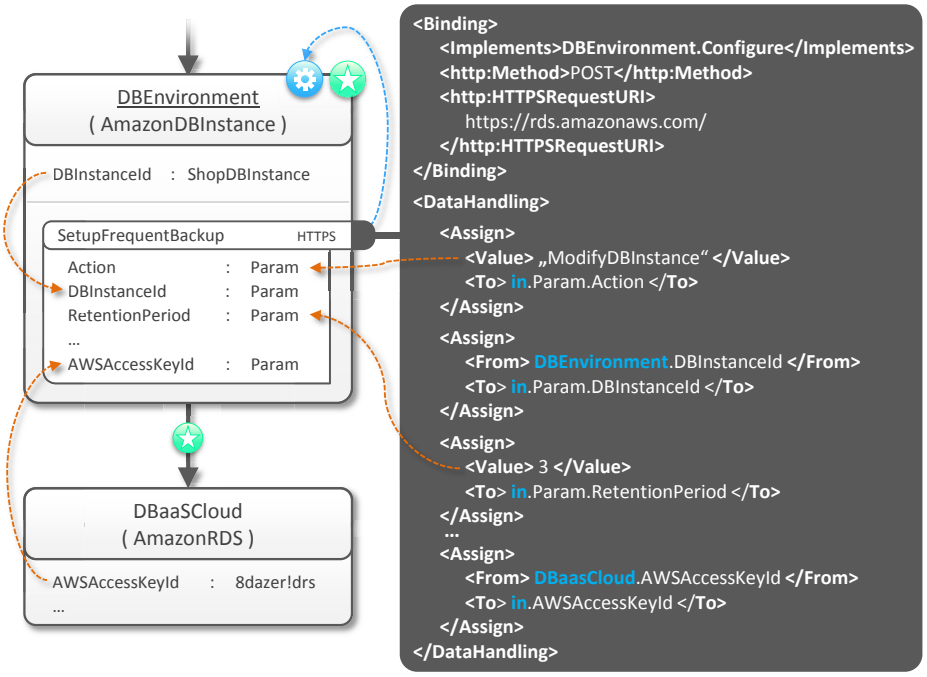

Fig. 4. Data Handling Specification for GLMP that invokes an HTTPS Web service

Data Handling Specification. Operations may have several input and output parameters. This depends on the implementation technology: Services have typically one output parameter, scripts may define several environment variables as output parameters. Thus, data handling is operation type specific and up to the corresponding GLMP. Many kinds of operations need input parameters that depend on properties in the topology. For example, to setup the frequent backup on the DBEnvironment node of our motivating scenario, an HTTPS call to the Amazon Query API is required. This call needs query parameters such as DBInstanceId and AWSAccessKeyId, which are properties of nodes: The DBInstanceId is

a property of the DBEnvironment node itself whereas the AWSAccessKeyId is a property of the underlying DBaaSCloud node. Therefore, we introduce a Data Handling Specification as shown in Figure 4. This allows assigning properties and default values to input parameters (path element "in") and output parameters (path element "out") to element properties as shown in Figure 4. This specification is read by the plan generator and GLMP. The plan generator uses the information to check the applicability of a GLMP by analyzing assigns: If an assigned topology property is not available, the implemented annotation is not executable. GLMPs use this specification to retrieve the needed input parameters by accessing the specified element properties and default values. Converting data is up to the GLMPs, e. g., a string property to an environment variable for scripts. All input parameters which are not assigned with a default value or property are exposed to the input message of the generated provisioning plan. These parameters have to be set by the caller and are routed by the provisioning plan to the corresponding GLMP that receives an additional list of parameters that are not defined in the specification as input. This kind of data handling is similar to data assign activities in BPEL and Variability Points in Cafe [11]. To enable lazy binding, the Data Handling Specification may be used also to assign properties to operation-specific elements in the binding specification. This information can be read by the GLMP to complete the binding at runtime.

Technology-Specific Preconditions. The preconditions of a GLMP depend on the technology to be integrated. Services offered by nodes have other preconditions than scripts that typically run on the underlying operating system. Thus, the planlet's topology fragment depends on characteristics of the respective technology and provides a means to define preconditions in the form of nodes, their dependencies, and properties. To provide an additional means that enables defining very specific requirements of operations, we extend management operations by properties: each property of an operation defined in the topology must be compatible with a property of the corresponding operation defined in the planlet's topology fragment. The set of properties is prescribed by the operation's type. In contrast to the Data Handling Specification, which is processed by the planlets, these properties are evaluated by the framework to select appropriate planlets during plan generation. Properties are required, for example, for defining script-based management operations on relationships. If a script implements the Create-Annotation of a directed relationship in the Provisioning Topology, a property is used to define if it has to be executed on the infrastructure of the source node or on the infrastructure of the target node. Depending on the used operating systems, different planlets may be needed for different property values.

Extended Workflow Generation. In Breitenbücher et al. [2], we presented a concept for provisioning plan generation based on planlets. We extend this algorithm to support GLMPs. In general, GLMPs are used similarly to normal planlets: they are checked for compatibility in terms of types, preconditions, effects, and Management Annotations. However, the generation algorithm needs

to be extended: (i) first, the matchmaking of topology and planlet fragments is extended to support operation-annotation binding specification of GLMPs: If the plan generator checks compatibility of annotations that are implemented by an operation, both annotation and implementing operation in GLMP and Provisioning Topology must have the same types. In addition, the properties of both operations must be compatible. (ii) The generator has to consider the Data Handling Specification, which influences applicability of GLMPs: an annotation implemented by an operation must not be processed by a GLMP until all assigned properties are available. In addition, output parameters that are written into properties must be considered, too. (iii) The generator prefers GLMPs whenever possible to Generic Planlets in order to treat custom implementations with a higher priority. This enables application developers to customize the provisioning.

5 Case Study

In this section, we present a case study to prove the approach's feasibility based on the Web shop. As already mentioned in Section 3.3, the plain provisioning of both stacks is completely done by Generic Planlets. To perform the custom tasks, we developed GLMPs that support the following technologies and tasks:

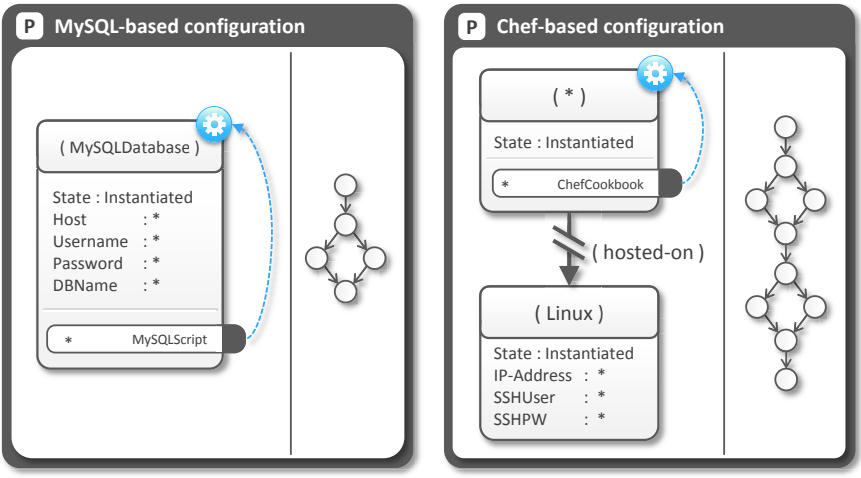

Fig. 5. GLMPs executing MySQL scripts (left) and Chef cookbooks (right)

To establish the MySQLConnection from PHP application to database, we implemented a small shell script that gets the endpoint of the database and credentials as input via environment variables. To setup the frequent database backup, we use the Amazon Query API that provides HTTPS services for this task. The corresponding GLMP is shown in Figure 3, the corresponding binding

and Data Handling Specification in Figure 4. The initial data import into the product database is done via an SQL script. The corresponding GLMP is shown in Figure 5 on the top left. The binding specifies the location of the SQL script, database endpoint and credentials are extracted by the GLMP from the MySQL-Database node automatically. To ensure that this information is available, the GLMP defines a target node of type MySQLDatabase that must be running already. Thus, this example shows how technology specific preconditions can be used to model the requirements and characteristics of technologies. To install God to monitor the Apache Web server, we use a Chef cookbook to implement the corresponding operation which is bound to a Configure-Annotation attached to the Web server. The GLMP shown in Figure 5 on the right executes this. It uses a transitive relation, which ignores all elements between source and target, of type hosted-on and requires an underlying Linux operating system with corresponding SSH credentials and IP-Address. Internally, it installs the Chef agent on the operating system by using SSH and executes the specified cookbook.

6 Evaluation

In this Section, we evaluate our approach against the most similar publicly accessible implemented approaches. The features compared in Table 1 are derived from the challenges discussed in Section 2.3 and explained in the following. Thus, they represent requirements that must be fulfilled to be able to provision the kind of Cloud applications used in our motivating scenario (cf. Section 2.1) fully automated. An x denotes that the approach supports the corresponding functionality without limitations. An x in parentheses denotes partial support.

Table 1. Feature Evaluation

Feature	GLMP	CloudForm.	Heat	Puppet	Chef	Juju	Workflows	Cafe
Component Wiring	x	(x)	(x)	(x)	(x)	x	x	x
XaaS Integration	x						x	x
Multi-Cloud	x		(x)	x	x	(x)	x	x
Full Customization	x						x	x
Multi-Script	x	(x)	(x)	(x)	(x)	x	(x)	x
Multi-Service	x	(x)	(x)	(x)	(x)	(x)	x	x
Explicit Integration	x							
Transparent Integration	x			x	x	x		x
Fully-Automated	x	x	x	(x)	(x)			x
Standards Compliant	x						(x)	(x)
Complete Top. Model	x							

Component Wiring means establishing relations between nodes, e. g., connecting a PHP application to a database. CloudFormation enables this by embedding hard-wired scripts into the template that access properties of resources. This is similar to our data flow definitions but limited to resources that are contained in

the template and, thus, provisioned on Amazon. Heat implements the CloudFormation specification and comes with similar problems. The script-centric technologies are able to wire components on a very low level based on custom scripts. However, this approach is limited as discussed in Section 2.3. *XaaS Integration* means the capability to provision and configure multiple service offerings of different types. CloudFormation and Heat are coupled to the service types provided by Amazon and cannot deal with others in a practicable way. The *Multi-Cloud* feature enables integrating different Cloud providers. All approaches support this except CloudFormation that is bound to Amazon. Heat supports this partially as OpenStack is used as Cloud operating system. All script-centric technologies are not affected by the underlying Cloud infrastructure and, thus, multi-Cloud ready. Juju can be used in conjunction with different Cloud providers, but a particular composite service instance is bound to one provider. *Full Customization* means that provisioning may be customized by the application developer in each detail. The concept of GLMPs enables this in three ways: (i) custom Planlets may be implemented to customize certain node combinations, (ii) GLMPs are able to integrate low level logic in a seamless fashion, and (iii) as Management Planlets are used to generate a Provisioning Workflow that is executed after the generation, this workflow may be adapted arbitrarily. All other features except workflows and Cafe are not able to provide this feature. *Multi-Script* and *Multi-Service* means that various kinds of script and management service calls respectively can be integrated seamlessly into the provisioning process. GLMPs support this integration directly as shown in Section 5. With Juju it is possible to combine different scripting languages. All other technologies, except workflows, that are made to orchestrate services, and Cafe, that employs workflows, only indirectly as they need wrapping code (scripts) for integration. *Explicit Integration* means that no visible glue code is needed for integrating any technology. GLMPs support this feature as technical execution details are hidden. Script-centric approaches do not support this feature as they need to create glue code, e. g., for invoking SOAP services. Even workflows, and thus Cafe, do not support this feature as they do not abstract from fine-grained tasks and wrapping code is needed to invoke scripts. *Fully-Automated* provisioning is the key feature for Cloud computing. The script-centric technologies Puppet and Chef do not support this feature directly. To enable the fully-automated provisioning of multi-stack applications, there are tools such as Marionette Collective or Spiceweasel. Workflows need to be created by hand. *Standards Compliant* ("de jure") are only Management Planlets as they support TOSCA as topology model and BPEL as workflow language. Workflows and Cafe are partially compliant as they may use the BPEL or BPMN standards but do not support any standard for the modeling of applications. This feature is important to create portable applications. *Complete Topology Model* means that nodes as well as relations are explicitly modeled. Only Management Planlets support this feature that is important to maintain the application: if relations are modeled implicitly only, it is hard to recognize them. Of course, most of the missing features of CloudFormation, Heat,

and the script-based approaches can be emulated by using low-level shell scripts. However, this is not efficient and the result is hard to maintain.

7 Conclusion and Future Work

In this paper, we presented an approach that enables the integration of script-centric and service-centric provisioning and configuration technologies. The approach is based on Management Planlets and enables to customize and automate the provisioning of composite Cloud applications. The validation showed the feasibility of our approach and the detailed evaluation, comparing the supported features to other approaches in this area, proved its relevance. We plan to extend our approach in the future to support influencing the execution order of Management Annotations in order to provide a fully customizable approach independent from restricted lifecycle operations. In addition, we focus on management of applications and apply the presented approach also in this area.

Acknowledgments. This work was partially funded by the BMWi project CloudCycle (01MD11023).

References

1. Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., Schumm, D.: Vino4TOSCA: A Visual Notation for Application Topologies Based on TOSCA. In: Meersman, R., et al. (eds.) OTM 2012, Part I. LNCS, vol. 7565, pp. 416–424. Springer, Heidelberg (2012)
2. Breitenbücher, U., et al.: Pattern-based runtime management of composite cloud applications. In: CLOSER (2013)
3. Delaet, T., Joosen, W., Vanbrabant, B.: A Survey of System Configuration Tools. In: 24th Large Installations Systems Administration Conference (2010)
4. Eilam, T., et al.: Pattern-based composite application deployment. In: Integrated Network Management. IEEE (2011)
5. Fehling, C., et al.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer (2013)
6. Günther, S., Haupt, M., Splieth, M.: Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures. Tech. rep., Very Large Business Applications Lab Magdeburg (2010)
7. Keller, A., Badonnel, R.: Automating the provisioning of application services with the BPEL4WS workflow language. In: Sahai, A., Wu, F. (eds.) DSOM 2004. LNCS, vol. 3278, pp. 15–27. Springer, Heidelberg (2004)
8. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: BPMN4TOSCA: A domain-specific language to model management plans for composite applications. In: Mendling, J., Weidlich, M. (eds.) BPMN 2012. LNBIP, vol. 125, pp. 38–52. Springer, Heidelberg (2012)
9. Leymann, F., Roller, D.: Production workflow: concepts and techniques. Prentice Hall PTR (2000)

10. El Maghraoui, K., Meghranjani, A., Eilam, T., Kalantar, M., Konstantinou, A.V.: Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 404–423. Springer, Heidelberg (2006)
11. Mietzner, R.: A method and implementation to define and provision variable composite applications, and its usage in cloud computing. Dissertation, Universität Stuttgart (August 2010)
12. OASIS: Topology and Orchestration Specification for Cloud Applications Version 1.0 (May 2013)