Toward an Architecture for the Automated Provisioning of Cloud Services

Johannes Kirschnick, Hewlett Packard Laboratories
Jose M. Alcaraz Calero, University of Murcia and Hewlett Packard Laboratories
Lawrence Wilcock and Nigel Edwards, Hewlett Packard Laboratories

ABSTRACT

The automated provisioning of services in cloud computing presents many challenges. Users can request virtual machines from cloud infrastructure providers, but these machines have to be configured and managed properly. This article describes an architecture that enables the automated deployment and management of the virtual infrastructure and software of services deployed in the cloud. The architecture takes a template description of a service, which encapsulates requirements, options, as well as behavior for a collection of resources and orchestrates the provisioning of this service into a newly created set of virtual resources. The template is used for integrating the deployment and reconfiguration behavior of a service in which logical components are described along with options to scale them and appropriately change their configuration. Services are described through a set of components, which can easily be mapped and remapped to dynamically created resources, letting services take full advantage of flexible cloud resources.

INTRODUCTION

Cloud computing infrastructure providers offer computational resources like virtual machines (VMs), storage, and networking to third parties. Cloud services in this proposal are defined as software services that use these resources. A complete service consists of appropriately configured software components deployed into a set of dynamically allocated infrastructure resources. Cloud computing enables new pay-per-use models in which services can be created and scaled on-demand, while only incurring charges for the actual usage of the resources involved.

However, this comes at the cost of increased management overhead, to dynamically deploy, redeploy, and reconfigure software components to take advantage of the flexible compute resources available. Users can request VMs from cloud infrastructure providers, but these machines have to be configured and managed properly. This is especially relevant when requesting large numbers of VMs, since the time needed to configure all of them can become a limiting factor, potentially offsetting the advantages of flexible compute infrastructures. For this reason, new tools and methods for managing and orchestrating VMs are required in order to automate the different steps involved in the provisioning and continuous operation of cloud services. How thousands of VMs can be dynamically created and configured automatically for a particular purpose is still an open issue for the users of cloud providers.

Current automated service deployment solutions do not fit well with cloud scenarios, as they treat infrastructure deployment as a separate problem. The main aim of this article is to describe an integrated architecture that enables the automated provisioning and management of cloud services. It orchestrates the different steps involved, such as virtual infrastructure management, in addition to installing, configuring, monitoring, running, and stopping software components in these virtual machines. This architecture is extensible, able to manage arbitrary software components and use different cloud providers to deploy infrastructure in the cloud. The architecture provides an integrated end-to-end service management solution, taking a service from user requirements down to an actual deployed system. It enables users to select a service from a catalog of predefined service templates allows them to customize them according to his/her requirements, and deploy them automatically. A template integrates the service deployment and reconfiguration behavior with a description of the service topology, in which logical collections of resources are described. This approach incorporates the flexible allocation of computing resources into the service design stage, naturally supporting operations to scale the deployed components and change their configuration appropriately, to fully take advantage of the new computing paradigm.

To describe the architecture, this article has been structured as follows. The next section pro-

0163-6804/10/$25.00 © 2010 IEEE

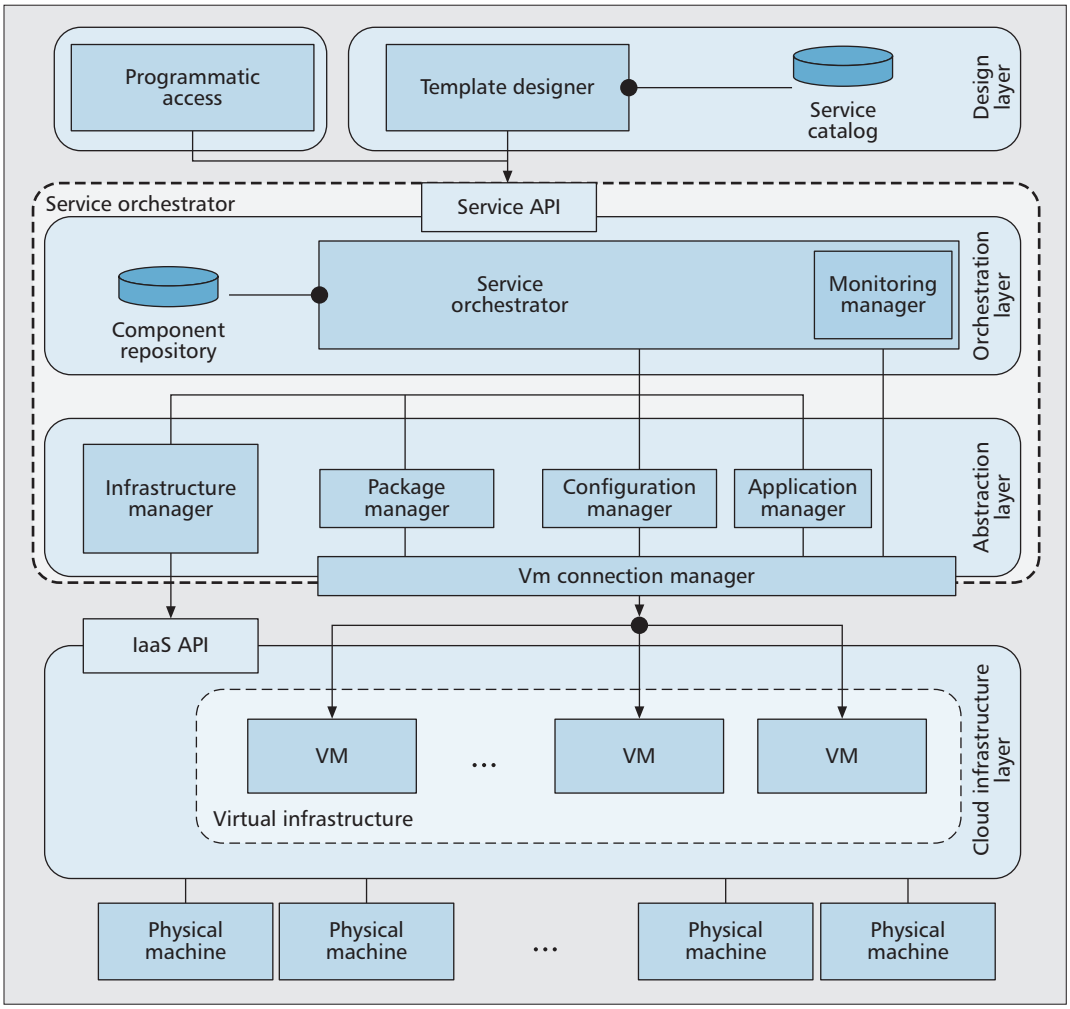

Figure 1. *Overall system architecture and interaction.*

The provisioning of a new cloud service involves the creation of the virtual IT infrastructure, followed by an initial installation of the necessary software components into this infrastructure, along with the correct configuration and subsequent ignition of the system.

vides some previous efforts related to the automated provisioning of services. We then describe the architecture and the languages used to carry out the provisioning of services. After that, we describe implementation details and provide statistics about the implementation. Finally, we discuss some conclusions and give an outlook to future work.

Related Work

Several research works related to the automated provisioning of services in distributed architectures have been published in recent years. *PUPPET* [1] and *CHEF* [2] are software solutions to automatically deploy software in distributed environments. Other more advanced tools are *SmartFrog* [3], *CHAMPS* [4], and *Quartermaster* [5], which provide integrated solutions to manage not only the automatic deployment of software but also the entire life cycle of the running software components, similar to a number of general approaches for managing services, such as COPS, WS-Management, and WSDM protocols. Furthermore extension to existing deployment engines have been developed to orchestrate service deployments, such as *Capistrano* [6] and *Control Tier* [7].

While these solutions share the same aim, they do not deal well with some of the essential problems related to cloud scenarios. In particular, they do not integrate virtual infrastructure management as part of the software life cycle, and thus do not target the elastic and the dynamic nature of scalable infrastructure inherent to the cloud. These are key features of our proposed architecture, which make it suitable for deploying services in cloud scenarios.

Most of the current cloud providers, such as *Amazon EC2*, *ElasticHosts*, *GoGrid*, *TerraMark*, and *Flexiant*, merely offer infrastructure on demand, and do not offer support to automatically deploy and configure software therein. For this reason, some vendors and companies have recently appeared, such as *Right Scale* [8] and *Scalr* [9], which provide scalable managed services on top of these infrastructures. This article describes an architecture that directly addresses the need for intuitive abstractions that integrate both the deployment and configuration of software with the ability to scale the associated infrastructure to provide such services in the cloud.

Architecture

From an architectural point of view, a cloud service can be defined as a number of software components with their accompanying configuration parameters, running on a cloud infra-

The Design Layer offers a high-level tool and graphical interface to provide end users with an intuitive way to customize and automatically provision cloud services based on a predefined service catalogue.

structure. The provisioning of a new cloud service involves the creation of the virtual IT infrastructure, followed by an initial installation of the necessary software components into this infrastructure, along with the correct configuration and subsequent ignition of the system. A particular characteristic of cloud services is that as load on the system changes, the service needs to be rescaled to either allocate additional or release unneeded infrastructure resources and appropriately reconfigure the deployed software components.

Figure 1 depicts an overview of our architecture. First, the *cloud infrastructure layer* represents the virtual IT resources provided by the cloud infrastructure platform. Second, the *service orchestrator* is responsible for creating new cloud services. The orchestrator is composed of two layers: the abstraction and orchestration layers. The *orchestration layer* coordinates the steps involved in the automatic provisioning of cloud services, and the *abstraction layer* provides abstractions of the various interfaces to manage a heterogeneous set of cloud providers, as well as abstractions of various mechanisms to install, configure, and start software components. Finally, the *design layer* offers a high-level tool and graphical interface to provide end users with an intuitive way to customize and automatically provision cloud services based on a predefined service catalogue. The following subsections describe each of the layers in detail.

Cloud Infrastructure Layer

Currently, there are a number of different commercial vendors with comparable infrastructure offerings. Prominent in the commercial space are Amazon EC2 as well as *RackSpace*. *Eucalyptus* [10] provides an open source cloud provider stack, which can be used to implement a private cloud. Even though the actual offering, pricing, and interactions differ, all of them have in common at least the ability to dynamically create and destroy infrastructure through means of a service application programming interface (API) (IaaS). They mostly offer only limited management capabilities, thus our architecture only assumes that we can request a virtual machine from a cloud provider and programmatically connect to it.

Abstraction Layer for Deployment

The abstraction layer hides the complexity involved in managing different aspects of the life cycle stages. It is composed of five different components: the *infrastructure*, *package*, *application*, *configuration*, and *VM connection manager*. Each of these is responsible for its respective aspect of the life cycle and presents a high-level interface to the *service orchestrator*. They act as configurable actuators that enable the service orchestrator to enact a service deployment, for each of the individual virtual resources and software components, as a set of coordinated interactions with these high-level components, from infrastructure deployment to software installation, configuration, startup, and reconfiguration.

Each cloud infrastructure provider offers its functionality via an IaaS API, but the lack of a common standard across them creates the need for individual vendor-specific adaptors. The *infrastructure manager* encapsulates the specifics of each vendor, and provides a common interface to create and destroy virtual resources on demand. Furthermore, each vendor can also require particular protocols for connecting to newly deployed virtual resources, thus making automation systems highly vendor-specific. To hide this complexity and abstract from operating-system-specific interaction protocols, the *VM connection manager* provides a consistent way to access VMs, depending on the target operating system, remote connection software involved, firewall policies, infrastructure provider policies, and so on; it offers support for a number of possible connection protocols, such as *SSH tunnel*, *Remote Desktop Connection*, *VNC*, and *Telnet*.

There are many different mechanisms to install, configure, and start software components, which depend on the operating system and the peculiarities of a particular software component. There is a need to abstract from these specifics, and direct the responsibility for the installation, configuration, and runtime management of a software component to the corresponding managers. Each manager interacts with the environment to actuate the deployment of the software components, using dynamically loaded component-specific descriptions of the set of required actions to carry out the high-level operations.

The *package manager* is responsible for the installation of software packages. It can delegate to existing software package installation tools such as apt (Advanced Package Tool), or interact directly with the file system to install software components. The *configuration manager* is utilized to dynamically configure and reconfigure software components. It provides a set of convenient tools to hide component-specific configuration details, such as access to API-based software manipulation, file template mechanisms, or web session replay capabilities for web components. Finally, the *application manager* manages the runtime state of software components, and interacts with the installed and configured software components to initiate startup or shutdown sequences.

Orchestration Layer

This layer orchestrates the steps involved in the automated provisioning of cloud services, utilizing the aforementioned managers. A *service API* is exposed to enable users to submit an initial model of the service to be deployed, and subsequently trigger reconfiguration and topology changes of existing services. The model is a topological description of the desired state of the service, consisting of named collections and sub-collections of logical components, their required configurations, and orchestration dependencies for the various actions required to install, configure, and ignite them. It furthermore specifies the desired mappings of components to virtual resources. The model can be constructed programmatically, or by using a template design approach. The model holds all the necessary configuration parameters, such as the cloud provider to use, the size of the virtual resources, as well as individual software component parameters, to deploy a running service. The template language used to generate an

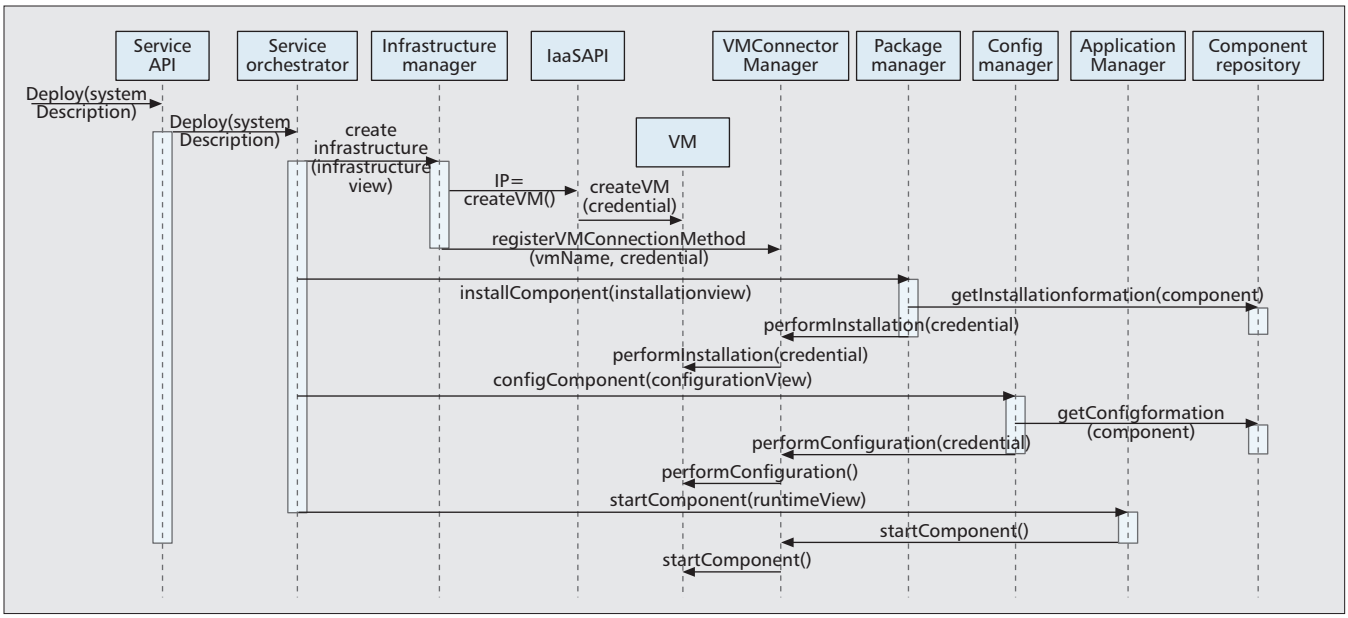

Figure 2. *Sequence diagram for all the steps involved in provisioning cloud services.*

instance of a model is discussed later, and serves as the starting point of the discussion of the sequence diagram depicted in Fig. 2.

Figure 2 shows a sequence diagram of the individual steps involved in the automated provisioning of a service instance. The use case starts when the user selects a template from a preexisting set of service definitions. This parameterized template, together with a set of user customizations, is used to generate a full model of the service and submitted to the *service API* for deployment. The *service orchestrator* coordinates the actions to reify four logical views for deploying the system, while taking into account the specific dependencies expressed in the service model. It inspects the model and subsequently passes parts of it down to the individual managers, delegating work on individual aspects of the service deployment. As the deployment progresses, the actions of the individual managers manipulate the model to reflect the current state of the model, such as the running state of a VM or life cycle state of software components.

First, the infrastructure view, which describes the set of required virtual resources, is passed to the *infrastructure manager*, which in turn uses this information to select the appropriate cloud provider and request the virtual infrastructure creation. Additionally, the appropriate mechanisms to connect to these VMs are registered with the *VM connection manager*.

Second, the installation view is passed to the *package manager* to correctly install all required software components. For each software component, the *manager* finds the correct component-specific package specification from the *component repository*. This specification describes the required sequence of abstract commands to execute to carry out the installation, such as installing a specified set of packages from a central location. The *package manager* utilizes the *VM connection manager* component as a gateway to connect to the associated VMs to perform the installation.

The *component repository* holds both a package description and a configuration description for all the different software components that the *service orchestrator* is able to manage. These descriptions are written in a *component description language* introduced later. The managers execute these descriptions for each individual component in response to requests from the *service orchestrator*. When the descriptions are executed at runtime, they have access to the instantiated model to retrieve individual configuration parameters and runtime state of the system. Abstract commands referenced in the descriptions are implemented by the individual managers through a set of tools that can carry out the work on the system.

Third, the configuration view is passed to the *configuration manager* to perform the configuration. Software components can be configured both before they are started (*preconfiguration*) and immediately after they are started (*post-configuration*), or as a result of topological changes to the system (*reconfigure*). To perform these configuration steps of a component, the *configuration manager* retrieves the corresponding configuration description from the *component repository*, and performs the encoded configuration. A typical configuration involves the generation of a configuration file from a template, using the component's configuration attributes stored in the model, and then copying the generated file to a specific location in the remote VM's file system.

Finally, the runtime state view is passed to the *application manager* component, to initiate the startup of the individual software components in the VMs. As with other managers, the component-specific actions to carry out operations such as *start* or *stop*, are encoded in a description retrieved from the *component repository*.

Figure 3 shows a state diagram of the life cycle associated with each software component. It includes basic states such as *Installed*, *Pre-Con-*

After the successful deployment, the Service Orchestrator is able to monitor the different software components. The architecture relies on third party monitoring solutions, such as Nagios or Ganglia, which are optionally installed in the VMs as part of the software deployment.

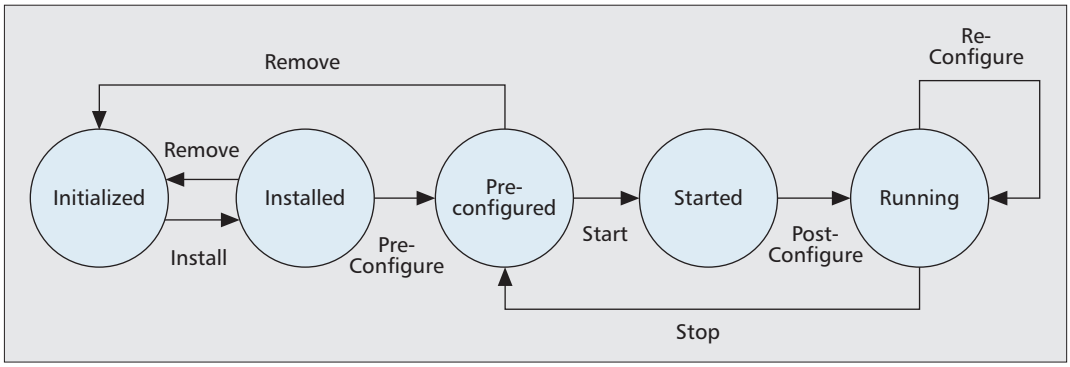

Figure 3. *State diagram of the life cycle for cloud services.*

figured, and *Running*. The configuration state is split into pre-, post-, and reconfigure phases to account for late binding information, which only becomes available at certain deployment stages.

The transitions between states are managed by the *service orchestrator*, and take into account any dependencies between components expressed in the model. These constraints define the synchronization points for the parallel deployment of a service instance. For example, a web server may require that a database server is running before it can be started. It is defined as a dependency between the state *running* of the web server and the state *running* of the database component.

The orchestrator will try to parallelize individual deployment steps. This is an important characteristic of this architecture, since it provides an efficient way to deploy large numbers of components simultaneously. Orchestration information in the model can constrain the parallel deployment if needed by component-specific requirements.

After successful deployment, the *service orchestrator* is able to monitor the different software components. The architecture relies on third party monitoring solutions, such as *Nagios* or *Ganglia*, which are optionally installed in the VMs as part of the software deployment. The reported monitoring information enables the tracking of the VMs and their software components in real time.

Design Layer

The design layer can automatically generate service model instances. An instance is generated from a parameterized template that describes the generic topology of a cloud service in terms of named collections of software components, deployment orchestration information, and constraints about how these components are mapped to VMs. The template describes best practice for how to instantiate and subsequently flex a service. The parameters of the template are expressed to capture important requirements, and control the instantiation process from the template to a specific model. For example, a high-level sizing parameter might be specified, with possible values high, medium, or low, which when selected determines the initial number of web servers or application servers in the service.

Using the Collection abstraction, the template describes possible flexing points of the service to allow dynamic scaling of the service deployment in response to observed behavior. These flexing points facilitate an integrated end-to-end management solution for deploying cloud services and subsequently right-sizing them to measured demands. Flexing points offer simple addition and subtraction operations to higher-level measurements-driven policies.

Templates are stored in a central service catalog. This catalog can be used by a graphical tool to provide users the ability to customize the initial instantiation of a cloud service according to his/her requirements and trigger the automated deployment. It enables a requirement-driven service design, while hiding the complexity involved in defining valid service topologies.

Domain-Specific Languages for Provisioning Cloud Services

This proposal introduces a set of domain-specific languages (DSLs) to describe the desired state of the virtual infrastructure and software components for cloud services, and how the software components are installed, configured, and managed to meet the desired state. The languages are implemented using Groovy [11], taking advantage of its dynamic nature to create declarative-style DSLs, combined with powerful scripting features to simplify rapid prototyping without any additional requirement for compilers or interpreters. It should be noted that a discussion of the complete syntax and semantics is beyond the scope of this article.

The following subsections explain the DSLs in detail. A typical three-tier web application, composed of a load balancer, database, and varying number of web servers, is used as a running example. The specific example illustrates a *TikiWiki* service, a web-based groupware solution for team collaboration.

Component Description Language

The Component Description Language (CDL) is used to define the deployment and configuration behavior of specific types of software components, such as a web server or application server. The language itself is structured in a way that resembles a declarative style, but contains executable logic that is executed in the context of the model for a specific component. The logic

can therefore reach into the model to resolve or modify additional variables that represent configuration parameters, desired state, or actual state.

Each software component has three different descriptions: the *component*, *configuration*, and *package* descriptions. The *component description* defines the attributes of the model entity for a software component, and serves as a configuration store holding runtime state. Templates reference these *descriptions* to define topologies of desired software components.

The *configuration* and *package* descriptions share a similar syntax. The former defines how to manage the configuration process associated with a software component, whereas the latter defines how to manage the installation of this component. The content of these files define the sequence of steps to be executed. They reference abstract commands provided by the execution environment of the *package* and *configuration manager*s, respectively.

The following example shows an excerpt for the *package specification* of a *MySQL* database software component, which is part of the *TikiWiki* service, interpreted by the *package manager*.

```
package_specification {
  applies { version == "5"}
  specification{
    rpm("mysql-shared," "mysql-5.0.26")
    tar("mysql-db," "/mysql/data")
  }
}
```

The *package manager* first selects the appropriate *specification* section to be applied, by evaluating the *applies* clause. The example defines a *specification* section in which the worker tools *rpm* and *tar* are referenced to first install a set of packages and then to un-tar a specific file into a given directory in the VM.

The configuration specification follows a similar approach. It defines how to manage the configuration of the software component, taking into account the component's life cycle states previously described. The following is an excerpt of the corresponding configuration specification for a *MySQL* database.

```
config_specification{
   applies(version == "5")
   preconfigure {
      file("innoDB.cnf," "/mysql/innoDB.cnf")
   }
   postconfigure{
            command("mysqladmin-newpw
${comp.pass}")
   }
}
```

The description defines sections that correspond to actions to be taken in transitions of the component's life cycle states. The example specifies that in the *preconfigure* phase a configuration file (innoDB.cnf) is copied to the VM before the start phase is initiated. The actions of a declared phase are evaluated before the corresponding component state is reached. Similarly, in the *postconfigure* phase, the default password is changed to a value retrieved from the *component description*. The reference to the new value of the password, denoted *comp.pass* in the example, is resolved against the runtime model available during the deployment. The load balancer, Apache webserver, and *TikiWiki* components of the *TikiWiki* service can be described in a similar manner.

Template Design Language

The Template Design Language (TDL) is used to describe the architecture of a cloud service, in terms of both the virtual infrastructure and the software components to be deployed. It represents the touch point between the *service orchestrator* and a user trying to deploy a new service. It captures all the VMs that need to be present, the software components to be deployed into them, the cardinalities associated with these software components, and the deployment dependencies that exist between them. It can also define default values for model entities. The software components deployed into virtual machines are referenced from the component repository introduced in the previous section. In particular, these software components are referenced using the *component description* classes previously introduced.

The following shows an example template to provision a new *TikiWiki* service. The database used in this example corresponds to the *MySQL* software component used earlier.

```
architecture(
  defaults: {
 vm(provider = "HP-internal," baseimage =
"golden-ubuntu")
  }
  model : {
   // Software components — Static
   vm {
     lb = LoadBalancer(type : "apache")
   }

  vm {
    db = MySQL(type: "innoDB," user: "alice,"
pass: "share")
  }

   // Dynamic Range Components
   vmrange(count: 1, name: "webserver
Collection"){
     ws = Apache(memory: 256, loadbalancer:
lb,
                  webapp: TikiWiki(ver: "2.0,"
dbService: db))
   }

   dependencies {
     // Start Apache web server after DB runs
     depends(op: "ws.start," on: ["db.started"])
   }
   }
```

The *template* is split into three subsections. The *defaults* subsection defines default parameters used in the model, for example, the cloud provider to use. The *model* section defines the software components to be deployed and their allocation to VMs. Finally, the *dependencies* sec-

The Template Design Language (TDL) is used to describe the architecture of a cloud service, in terms of both the virtual infrastructure and the software components to be deployed. It represents the touch point between the service orchestrator and a user trying to deploy a new service.

Templates enable requirements-driven, rapid service provisioning, while hiding the configuration complexities from the end user. As a proof of concept, a prototype has been implemented to confirm the advantage of the parallel deployment approach.

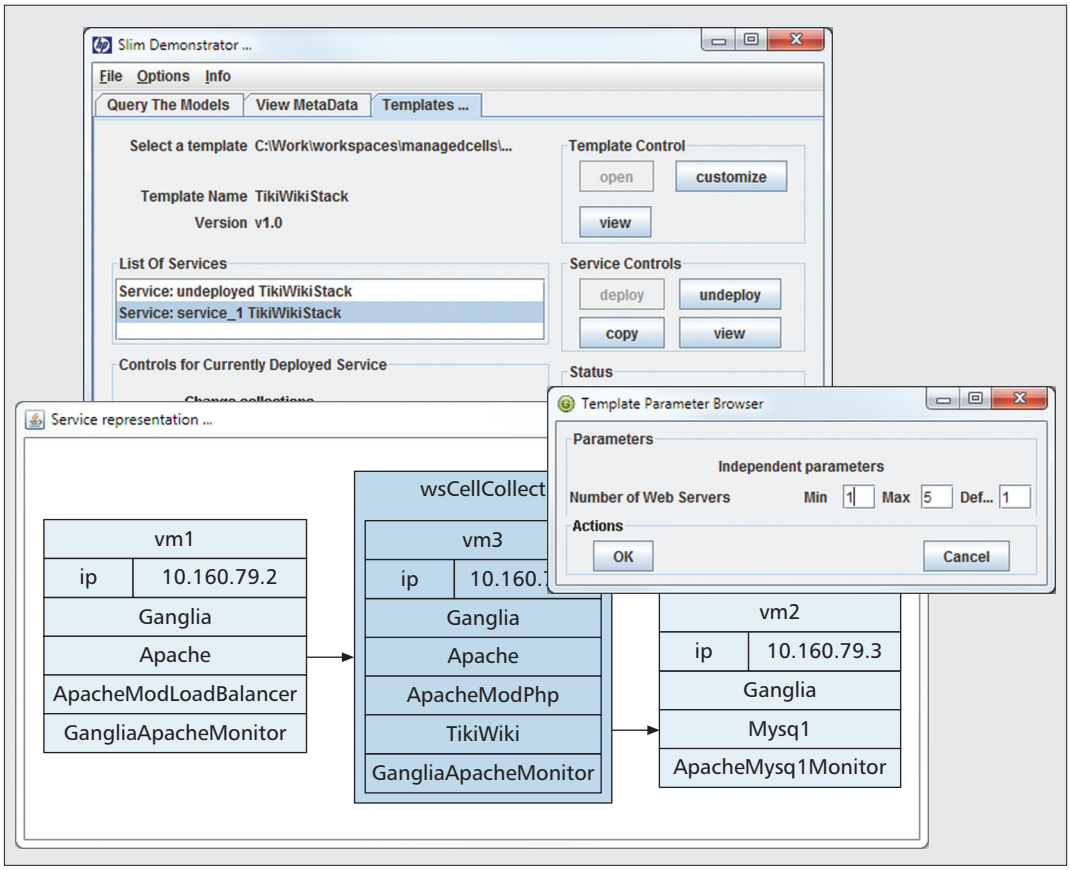

Figure 4. *Snapshot of the graphical interface of the service catalog and template designer tools.*

tion defines deployment dependencies between life-cycle states of software components. Each software component is initialized with a set of attributes, as well as references to other software components to enable the exchange of late binding configuration information. The values of these attributes can be configured from the parameters chosen when the template is instantiated from the service catalog. Furthermore, the language enables the definition of collections of named components that define service flexing points, specified using *vmrange*, which in the example defines a scalable collection of Apache web servers.

Implementation

As a proof of concept, a prototype of the architecture explained earlier, using the languages exposed in the previous section, has been implemented. This prototype is called *SLIM* and is used internally at Hewlett-Packard Laboratories to develop cloud services. It has been implemented in Java, while both the *Template Description* and the *Component Description Language* have been implemented in Groovy [11].

Figure 4 shows a snapshot of the graphical interface of the template designer tool. This snapshot corresponds to a successful deployment of the *TikiWiKi* example outlined in the previous section. The upper part of the figure shows the template designer interface loaded with the service. The lower part shows a graphical representation of the virtual infrastructure and the software components deployed within them. It corresponds to three separate VMs, one representing the load balancer (*vm1*), a second representing a *MySQL* database instance (*vm2*), and a third representing an Apache web server running the *TikiWiKi* web application (*vm3*). Finally, the right part of the figure shows the high-level parameters defined by the template in order to customize the cloud service before provisioning it.

To analyze the scalability of our *SLIM* prototype, a series of tests have been executed to measure the time needed to deploy new cloud services, while varying the number of web application servers from 1 to 14. Both the time for creating the infrastructure and the time it took to automatically provision *the software* have been measured independently. These tests were executed on the *HP internal* cloud testbed.

Figure 5 shows the execution results, which highlight that the time needed to provision the software components is small, compared to the infrastructure creation time. Infrastructure provisioning time is related to a particular cloud provider, whereas deployment time is related to our implementation. Note that an almost constant trend can be observed for the time it takes to deploy the service, demonstrating good scalability results. This corresponds with our expectation that the dependency-constrained parallel deployment orchestrator would find the maximal parallelism to deploy the software components. Thus, increasing the number of VMs should not significantly increase the overall amount of time

it takes to deploy the cloud service, in contrast to a sequential deployment approach, which exposes a linear time increase.

Conclusion

An architecture for the automated provisioning of cloud services has been described and successfully validated in this proposal. It offers extensible ways to include new cloud services definitions. Furthermore, a parallel deployment method supported by an orchestration model has been integrated into the architecture to significantly reduce the time needed to deploy large cloud services. Moreover, the architecture enables the declarative definition of services using a *Template Description Language* for the topology design and a *Component Description Language* to specify the individual configuration and deployment behavior of software components. Templates enable requirements-driven rapid service provisioning, while hiding the configuration complexities from the end user. As a proof of concept, a prototype has been implemented to confirm the advantage of the parallel deployment approach.

It is worth mentioning that bandwidth requirements for provisioning new services are considerably reduced using our approach, as current solutions offered by cloud providers require users to upload complete volume images, (usually several gigabytes) containing preinstalled software. By contrast, our approach makes it feasible to create service descriptions that can ignite a complete service simply by applying deltas to generic operating system images to install the software components

As future work, it is expected to include autonomic computing features in the *service orchestrator* in order to provide self-management capabilities such as fault-tolerant cloud services, service level agreement management, and quality of service assurance, utilizing the definition of named collections to adapt systems programmatically.

Acknowledgment

Thanks to the Fundacion Seneca for sponsoring this research under its post-doctoral grants and project 04552/GERM/06. Thanks to the European Commission for partially supporting this research under project FP7- CIP-ICT-PSP.2009.7.1-250453 SEMIRAMIS. The authors would like to thank Matthias Schwegler for his valuable contribution to this research.

References

[1] J. Turnbull, *Pulling Strings with Puppet*, APress, 2007.
[2] A. Jacob, "Infrastructure in the Cloud Era," *Proc. Velocity Conf.*, 2009.
[3] P. Goldsack *et al.*, "The Smartfrog Configuration Management Framework" *ACM SIGOPS Operating Sys. Rev.*, vol. 43, no. 1, 2009, pp. 16–25.
[4] A. Keller *et al.*, "The CHAMPS System: Change Management with Planning and Scheduling," *Proc. IEEE/IFIP NOMS*, vol. 1, 2004, pp. 395–408.

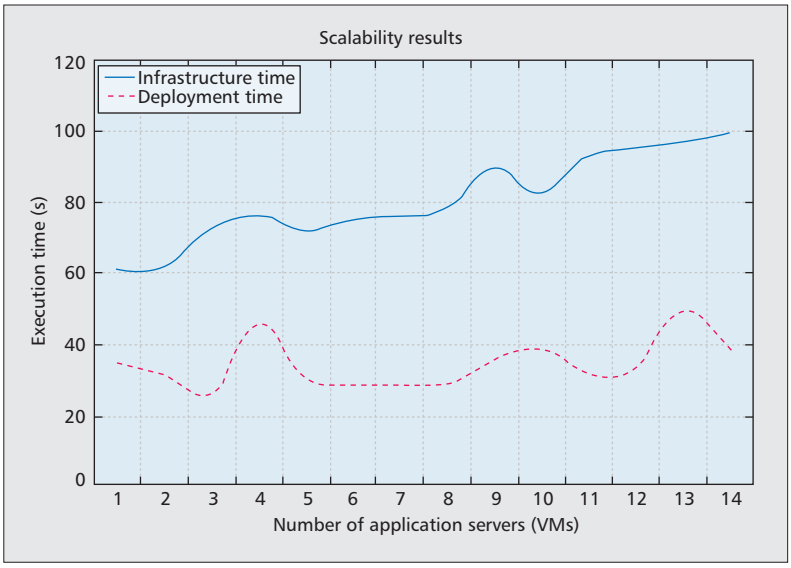

Figure 5. *Scalability results deploying TikiWiki service.*

[5] S. Singhal *et al.*, "Quartermaster — A Resource Utility System." *Proc. 9th IFIP/IEEE Int'l. Symp. Integrated Net. Mgmt.*, 2005, pp. 265–78.
[6] D. Frost, "Using Capistrano," *Linux J.*, vol. 177, 2009, p. 8.
[7] D. Solutions, "Control Tier," Tech. Rep., 2010; http://controltier.org/wiki/Main_Page.
[8] Right Scale, "Cloud Management Platform"; http://www.rightscale.com/.
[9] Scalr; https://www.scalr.net/.
[10] D. Nurmi *et al.*, "The Eucalyptus Open-Source Cloud-Computing System," *Proc. 9th IEEE/ACM Int'l. Symp. Cluster Computing and the Grid*, 2009.
[11] D. Koenig *et al.*, *Groovy in Action*, Manning Publications, 2007.

Biographies

Johannes Kirschnick (Johannes.kirschnick@hp.com) holds a degree in computer science from the Technical University of Munich and is currently a researcher at HP Labs, Bristol, United Kingdom. He is part of the Automated Infrastructure Laboratory, focusing on developing technologies for highly automated, secure, and dynamic instantiation and management of cloud computing infrastructure and services.

Jose M. Alcaraz Calero [M] (jmalcaraz@um.es) holds a Ph.D in computer science from the University of Murcia. He has been working at the University of Murcia on several European and international projects. Currently, he is a researcher at HP Labs. His research areas include cloud computing, semantic web, policy-based systems, and security. He is a member of the ACM.

Lawrence Wilcock (Lawrence.wilcock@hp.com) has worked on a variety of telecommunications, cloud, and distributed system projects within HP Labs and has been granted more than 28 patents. He is currently a senior researcher in the Automated Infrastructure Laboratory at HP Labs. He holds an M.Eng. from the University of Bath.

Nigel Edwards (nigel.edwards@hp.com) has worked on a variety of security and distributed system projects and products within HP businesses and the Labs. He is currently a senior researcher in the Automated Infrastructure Laboratory. He holds a Ph.D. from the University of Bristol. He is a member of the ACM and IET.