Towards an Architecture for the Automated Provisioning of Cloud Services

Johannes Kirschnick*, Jose M. Alcaraz Calero†*, Lawrence Wilcock*, Nigel Edwards*
*Automated Infrastructure Laboratories
Hewlett Packard Laboratories
BS34 8QZ Bristol
United Kingdom
Email: {johannes.kirschnick, lawrence_wilcock, nigel.edwards}@hp.com
†Communications and Information Engineering Department
University of Murcia
30100 Murcia Spain
Email: jmalcaraz@um.es

***Abstract*—Cloud computing entails many challenges related to the management of on-demand virtual infrastructures. One of these challenges is the automated provisioning of services in cloud infrastructures. Users can request virtual machines from cloud infrastructure providers, but these machines have to be configured and managed properly in order to be useful. This paper describes an architecture that enables the automated provisioning of services in the cloud. The architecture orchestrates the different steps involved in the provisioning of services, such as the management of virtual infrastructures (creation and deletion of VMs, networks, *etc*) as well as the installation, configuration, monitoring and execution of software into the VMs. This architecture is extensible and able to deal with different software components and cloud providers to carry out the provisioning of the service. To fully support the automatic cloud service provisioning, a high level tool is introduced which enables a user to select and customize a predefined service from a services catalog. As a proof of concept, a prototype has been implemented. Implementation aspects and statistics results are provided to furthervalidate the proposal.**

I. Introduction

Cloud computing architectures provide computational resources like virtual machines (VM), storage and networking to third parties. Cloud services in this proposal are defined as software services which use the resources provided by cloud infrastructure providers. Cloud computing enables new business models in which businesses and researchers can create cloud services on-demand according to their continuously changing needs while only paying for the actual usage of the resources involved.

However, cloud computing entails many challenges related to the management of on-demand virtual infrastructures. One of these challenges is the automated provisioning of cloud services. Users can request virtual machines from cloud providers but these machines have to be configured and managed properly in order to be useful. This is especially relevant when requesting large numbers of VMs since the time needed to configure all of them can become a limiting factor. 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 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.

The main aim of this paper is to describe an architecture which enables the automated provisioning of cloud services. It orchestrates the different steps involved such as creating and removing VMs in the virtual infrastructure as well as installing, configuring, monitoring, running and stopping software in the VMs, *etc*. This architecture is extensible and able to manage different software components and use different cloud providers for deploying cloud services therein. Moreover, the architecture provides an integrated end-to-end system management solution, taking a system from user requirements down to an actual deployed system. A service catalog enables users to select a predefined service, customize it according to his requirements and deploy it automatically.

To describe the architecture, this paper has been structured as follows: section II provides some works related to the automated provisioning of services. Sections III and IV describe the architecture and the languages used to carry out the provisioning of services, respectively. After that, section V describes some implementation details and statistics about the implementation. Finally, section VI discusses some conclusions and gives an outlook to future work.

II. Related Work

Several research works related to the automated provisioning of services in distributed architectures have been done in recent years. For example, *PUPPET* [1] and *CHEF* [2] are software solutions to automate the installation and configuration of software in distributed environments. They are client-server architectures in which an orchestrator or *server* is in charge of controlling the provisioning of the services deployed into the computers or *clients*. Recently, *Control-tier* [3] and *Capistrano* [4] have been released as additions to

both *PUPPET* and *CHEF* respectively, providing orchestration capabilities over the installation and configuration processes. *CFEngine 3* [5] is another client-server architecture for installing software components in a distributed environment. *SmartFrog* [6] is a totally distributed P2P architecture for automated provisioning of services. It provides a fine-grained control of the life-cycle of the services managed in the architecture, enabling orchestrated deployment, installation, configuration, run-time service management, monitoring, *etc*. All these architectures are extensible since they provide a way to insert new software components to be managed by the architecture. Moreover, all of them have a common basis. They provide a language to enable users to define the desired services in a distributed environment.

While all these related works share the same aims, none of them is suitable for provisioning cloud services. They only cover the provisioning of services into physical machines or *resources* but they do not tackle the usage of cloud infrastructures as part of the provisioning of these services. It is one of the main requirements associated with the provisioning of cloud services, is fully addressed in the proposed architecture in this paper.

III. Architecture

From an architecture point of view, a cloud service can be defined as a number of software components with their accompanying configuration parameters, running on top of a cloud infrastructure platform, delivering a service over the *Internet*. The provisioning of a new cloud service involves the creation of the virtual IT infrastructure, followed by the installation of the necessary software components into this infrastructure and finally to configure and start them.

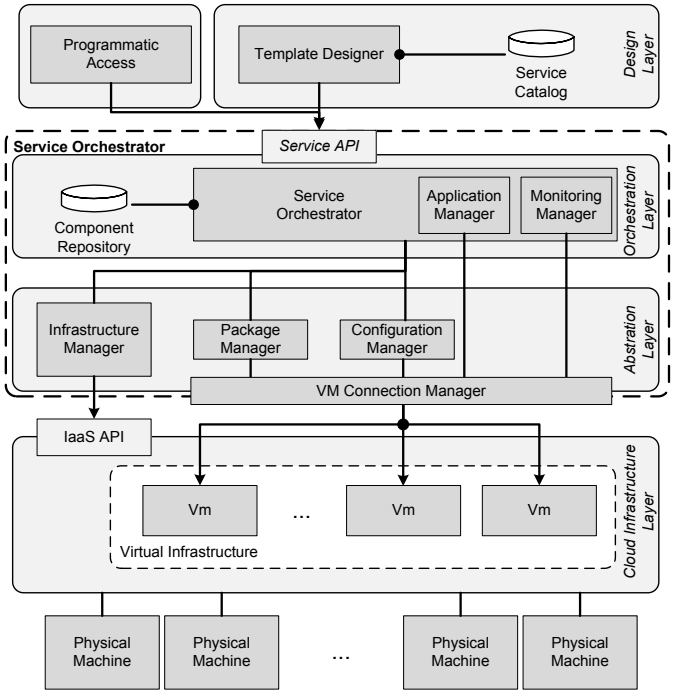

Figure 1. Overall system architecture and interaction

Figure 1 depicts an overview of our architecture. Firstly, the *Cloud Infrastructure Layer* represents the virtual IT resources provided by the cloud infrastructure platform. Secondly, the *Service Orchestrator* is the software element in charge of creating new cloud services. The orchestrator is composed of two layers: abstraction and orchestration layer. The *Orchestration Layer* provides the orchestration of all the steps involved in the automatic provisioning of cloud services. These steps have been identified by observing the manual provisioning of cloud services. The *Abstraction Layer* provides abstraction capabilities for managing heterogeneous cloud providers, different manners of installing and configuring software. Finally, the *Design Layer* offers a high level tool and graphical interface which provide final users a very intuitive and easy way to automatically provision cloud services based on a predefined service catalog. The following subsections describe in detail each of the layers.

A. Cloud Infrastructure Layer

Cloud infrastructure providers utilize a plurality of physical resources to deliver virtual infrastructures. This is known as *Infrastructure as a Service* (IaaS) since the functionality of managing these virtual infrastructures is provided by means of a *service API*. Currently, there exist a number of different commercial vendors with comparable infrastructure offerings, prominently *Amazon EC2* [7]. Even though the offerings differ, all of them have at least in common the ability to dynamically create and destroy virtual machines. Usually, the cloud infrastructure providers offer a limited set of management capabilities to deploy services into these virtual machines, focusing merely on providing connectivity to the virtual machines like remote desktop or SSH connections. Nurmi et Al [8] provides a comprehensible description of *Eucalyptus*, an open source cloud provider architecture.

B. Abstraction Layer for Deployment

The abstraction layer provides abstraction and extension capabilities to the *Service Orchestrator*, which is the component in charge of performing the automatic deployment of cloud services. This layer is composed of four different components: *Infrastructure Manager*, *Package Manager*, *Configuration Manager* and *VM Connection Manager*.

Although each cloud infrastructure provider offers its functionality via an *IaaS API*, the lack of a common standard creates the need for individual vendor-specific adapters for each cloud provider. For this reason, the *Infrastructure Manager* enables the integration with new cloud infrastructure providers to create and destroy virtual resources on-demand.

Notice that each VM can provide different ways to connect to it, depending on the operating system used, remote connection software involved, firewall policies, infrastructure provider policies, *etc*. For this reason, a *VM Connection Manager* components is available to deal with different ways to connect to a VM such as *SSH tunnel*, *Remote Desktop Connection*, *VNC*, *Telnet*, etc.

There are many different ways to install a software in an operating system. For example, using a package repository tool

like *apt*, *yum*, *yast*, *rpmtools* or *rubygems*, copying a set of files into a given folder, or copying source files and compile them. The *Packager Manager* component enables the management of different ways of installing software as well as the extension of new mechanisms.

Additionally, the *Configuration Manager* component is in charge of enabling the management of different ways of configuring software. It is motivated by the fact that there is no common interface to configure software components. For example, it is possible to configure an application by modifying configuration files such as *INI* or *XML* files; by using a template based configuration approach, or by utilizing the software component command-line API directly. The *Configuration Manager* provides an unified way to configure the software components, respecting these differences.

C. Orchestration Layer

This layer orchestrates the steps involved in the automatic provisioning of cloud services. To this end, a *service API* is exposed to enable the users to describe the desired state of the virtual infrastructure and the cloud services to be deployed. The description is expressed in the *Desired State Description Language (DSDL)*. This language is a declarative language which enables the definition of the different virtual IT resources, software elements, configurations and orchestration information involved in the offering of the cloud services. *DSDL* is discussed in section IV-B.

Figure 2 shows a sequence diagram representing how this layer carries out the automatic provisioning. The use case starts when the user select and instantiate a template generating an instance of the *DSDL* model to the *Service API*. Then, the *Service Orchestrator* inspects the *DSDL* model and splits it up into four distinct parts: infrastructure description, installation description, configuration description and run-time state description.

Firstly, the infrastructure description is passed to the *Infrastructure Manager*, which in turn, uses this information to select the appropriate cloud vendor provider and requests the on-demand virtual infrastructure from this provider. Additionally, the appropriate ways to connect to these VMs are registered with the *VM Connection Manager*. This manager is used later to connect to these VMs using the appropriate protocols and credentials.

Secondly, the installation description is passed to the *Package Manager* to correctly install all the software components required. For each software component, the *Package Manager* retrieves the packages specification from the *Component Repository*. This specification describes the required methods to carry out the installation process, i.e. apt, copy files, *etc*. The *Package Manager* utilizes the *VM Connection Manager* component as a gateway to connect to the associated VMs to perform the installation. For example, it can execute an *apt-get* command in the remote VM to install software packages from a package repository.

The *Component Repository* is the set of all the different software components that the *Service Orchestrator* is able to manage. This repository is extensible and new components can be described using the *Component Description Language* introduced in section IV-A.

Thirdly, the configuration information is passed to the *Configuration Manager* to perform the configuration of all the software components. The *Configuration Manager* retrieves for each component the configuration information from the *Component Repository* and performs the appropriate configuration actions. Similar to the Package Manager, it can utilize the *VM Connection Manager* to connect to the VMs as well. For example, an XML configuration file can be created from a template and copied to a specific location in the remote VM.

Finally, the run-time state description is passed to the *Application Manager* component to start the software on the VMs. However, the *Application Manager* component not only manages the starting of the software but also the stopping, restarting, *etc*.

When different software components have to be deployed, the default behaviour is to deploy them all in parallel. This is an added value of this architecture since it provides an efficient ways to deploy large numbers of components almost simultaneously. Orchestration information is in the *DSDL* provided dependencies to control the parallel deployment. A standard life-cycle of the service deployment process has been defined by the architecture. Figure 3 shows a state diagram of the life-cycle used during the deployment. It includes basic software states such as *Installed*, *Pre-Configured* or *Running*. A component is always associated with a concrete life-cycle state managed by the *Service Orchestrator*. Note that the configuration state is split into three parts, pre-configuration which occurs before a component is started and the post-configuration which occurs after starting and finally the re-configure step which occurs when the deployed service is changed during run-time requiring individual components to be reconfigured.

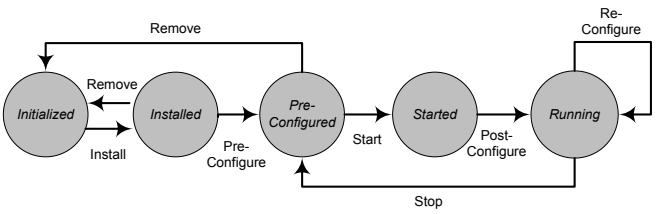

Figure 3. Simplification of the State Diagram of the life-cycle for the cloud services

The life-cycle is used to stablish orchestration constraints between different software components. These constraints define the synchronization points for the parallel deployment. For example, a web server may require that a database server is already 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.

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

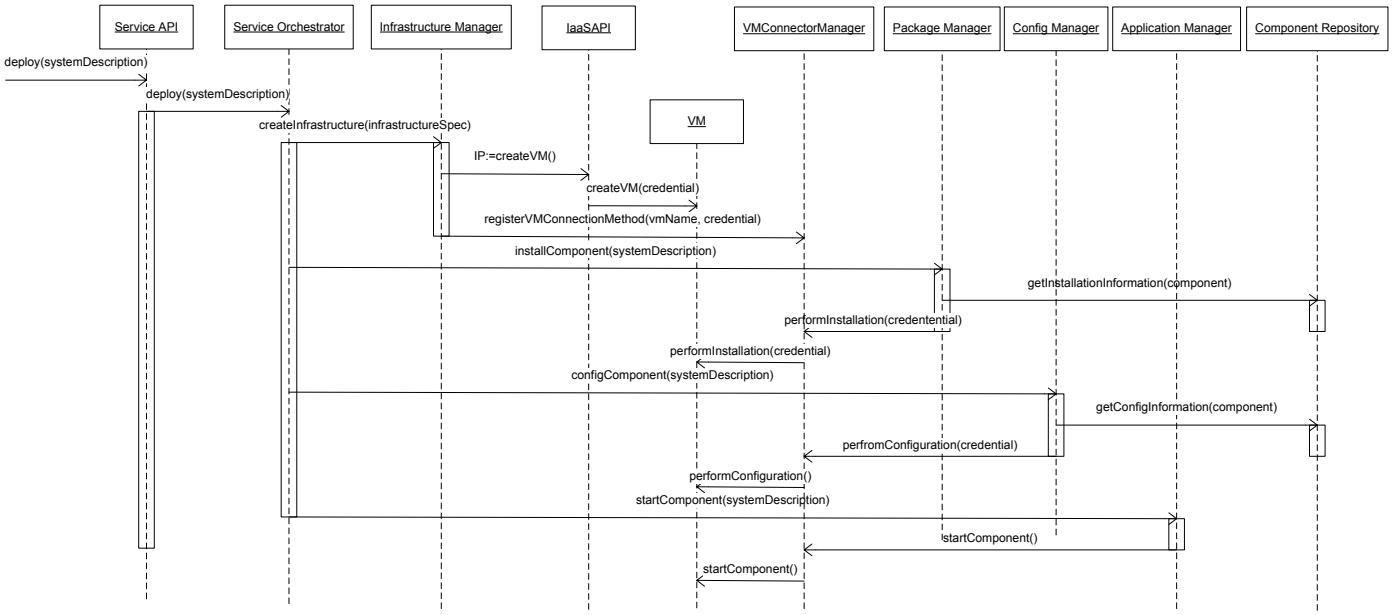

Figure 2. Sequence Diagram for all the steps involved in provisioning a cloud services

the VMs and their software components in real-time.

D. Design Layer

This layer enables an integrated end-to-end system management solution, taking a system from user requirements down to an actual deployed system. Different user requirements for a service translate into different realizations of it. A template is used to capture all these possible variations. The template describes the topology of a cloud service in term of scalable collections of software components and the constraints of how this components are map to VMs. This description have requirement focused parameters which when chosen creates different service realizations.

A collection of templates is stored is a service catalog. This catalog is used by a intuitive and easy to use graphical tool to provide users the ability to select a predefined cloud service, customize it according to his requirements and to deploy it automatically. This tool enables a requirement driven service design while hiding the complexity involved in defining the service. The instantiation of a template with a set of given parameters produces a *DSDL* description of the desired cloud service. This description is then deployed using the *Service Orchestrator*.

The language used to describe the service templates is the *Template Description Language*. This language is an extension of the *DSDL* language introduced in section IV-B. It enables to express parametrized software components and topologies. Topology information expresses allocations between components and VMs.

IV. Domain-Specific Languages for Provisioning Cloud Services

This proposal provides a set of domain specific languages for describing the desired state of the virtual infrastructure for cloud services, how the services are managed in the architecture and how these services are composed into software component. The following subsections explains these languages in detail.

A. Component Description Language

The *Component Description Language* (CDL) is used to define how a software component is managed by the architecture. The use of this language enables the incorporation of new software components to the component repository to extend the variety of manageable components by the architecture. The language itself shares the same syntax and semantics as *Groovy* [9], but it should be noted that a discussion of the complete syntax and semantics is beyond the scope of this paper. In fact, the inclusion of a new software component does not require a developer to know all *Groovy* features since the *CDL* language has been designed as a domain specific language to easily create such descriptions.

The definition of a new software component requires to create three different files: *Component Description*, *Configuration Specification* and *Package Specification*. The *Component Description* is a simple class that represents the software component. This class manages all the configuration information required to manage the component at run-time. The *DSDL* language reference parameters of these *Component Descriptions* in order to define the desired software component to be deployed.

The *Configuration* and *Package* specifications share a similar syntax. The former defines how to manage the configura-

tion process associated with the software component whereas the latter defines how to manage the installation of this component. The content of these files refers to individual methods for installing and configuring the software component. These methods are provided by the *Package Manager* and the *Configuration Manager* architectural components, respectively.

The following example shows an excerpt for the *Package Specification* of a *MySQL* database software component. This specification is interpreted by the *Package Manager* to carry out the installation of this component.

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

The example defines a *specification* section in which the installation methods *rpm* and *tar* are declared 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 correctly manage the configuration of the software component while taking into account the component's life-cycle states, previously described in section 3. 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 example ensures that a specific configuration file (innoDB.cnf) is placed into a well known location on the VM, before the database is started. It is indicated by the copying process declared in the *preconfigure* section. Furthermore, after the database has been started, the default password is changed to one which is retrieved from the *Component Description*, using a command line invocation on the remote machine. There is a reference in the example denoted as *comp.pass*. This reference is resolved against the run-time information available during the deployment.

B. Desired State Description Language

The *Desired State Description Language* (DSDL) is used to describe the architecture of both the virtual infrastructure and the cloud services 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 virtual machines which need to be present, the software components to be deployed into them, the cardinalities associated to these software components and the deployment dependencies that exist between them. The language itself it implemented on top of Groovy as well as the *CDL* language. The software components deployed into virtual machines are referenced from the component repository introduced in section III-C. In particular, these software components are referenced using the *Component Description* class previously introduced in section IV-A.

The following shows an example instance for the provisioning of a new cloud service. It is a typical 3-Tier web application, composed of a load balancer, a database and a varying number of web servers. It defines a *TikiWiki* service which is a web based groupware solution enabling team collaboration using a Wiki. This service is used as a running example in the rest of this proposal. The database used in this example corresponds to the *MySQL* software component used as example in section IV-A.

```
architecture(
  defaults : {
   vm(provider = "HP-internal",
      baseimage= "golden-ubuntu")
  }
  model : {
   // Static software components
   vm {
      lb = LoadBalancer(type: "apache")
   }
   vm {
      db = MySQL(type: "innoDB",
                 user:"alice",
                 pass: "share")
   }
   // Dynamic range components
   vmrange(count:1,name:"wsColl") {
    ws = Apache(memory: 256, loadbalancer: lb,
     webapp: TikiWiki(ver: "2.0", dbServer: db))
   }
  }
  dependencies: {
   // Start Apache web server after DB runs
   depends(op: "ws.start", on: ["db.started"])
  }
)
```

A *desired state description* is be split into three separate subsections. The *defaults* subsection defines default parameters used in the actual model, for example the cloud infrastructure provider to use. The *model* section defines the software components to be deployed and their allocation to VMs and finally the *dependencies* section defines deployment dependencies of individual software components. Each software component is initialized with a set of parameters as well as references to other software components to enable the exchange of late binding configuration information. Furthermore, as syntactic sugar, the language enable the definition for a range of VMs to make it easier to define large numbers of similar software components.

V. Implementation and Statistics

As a proof of concept, a prototype of the the architecture explained in section III using the languages exposed in section IV has been implemented. This prototype is called *SLIM* and used internally at Hewlett-Packard Laboratories. It has been implemented in *Java* while both the *Desired State Description Language* and the *Component Description Language* have been implemented in Groovy [9]. Groovy was chosen because

of its dynamic nature which simplifies rapid prototyping and domain specific language creation.

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 defined in section IV-B. The upper part of the figure shows the template designer interface loaded with the *TikiWiKi* service from the catalog. The lower part of the figure shows the graphical representation of the virtual infrastructure and the software components deployed. The picture corresponds to three separated 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 4 shows the high level parameters defined by the template in order to customize the cloud service before provisioning it.

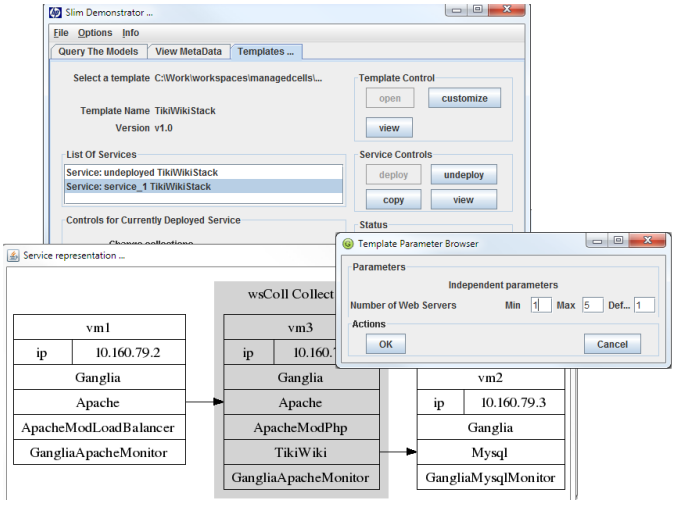

Figure 4. Snapshot of the graphical interface of the service catalog and template designer tools

To analyse the scalability of our *SLIM* prototype, statistics have been gathered to measure the time needed to deploy a new cloud service. Both the time for creating a new virtual infrastructure and the time it took to automatically provision *TikiWiki* have been measured independently. A series of tests have been executed in which the same *TikiWiki* service was deployed while varying the number of web application servers from 1 to 14. Each web application server was deployed in a different VMs. The tests has been executed using the *HP internal* cloud testbed.

Figure 5 shows the execution results. These results show that when creating a cloud service, the time needed to provision the software components is small compared to the infrastructure creation time. Furthermore, an almost constant trend can be observed for the time it takes to provision the service. This corresponds with our expectation that the parallel deployment capability should enable to deploy all the software component almost at the same time. Thus, increasing the number of VMs should not increase the overall amount of time it takes to deploy the cloud service for a software perspective.

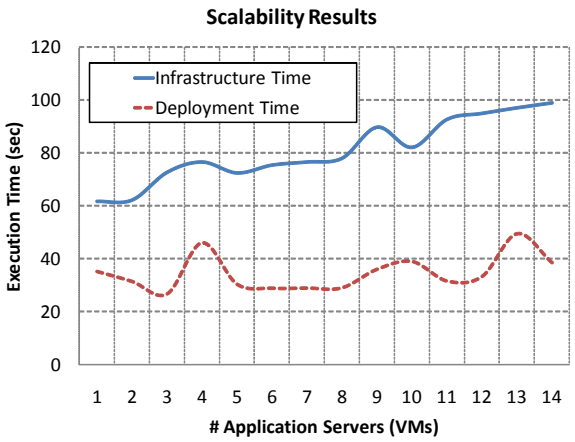

Figure 5. Scalability Results deploying TiKiWiKi service

VI. Conclusion

An architecture for automated provisioning of cloud services has been described and successfully validated in this proposal. The architecture can utilize multiple different cloud providers. It offers extensible ways to include new cloud services to be deployed. Furthermore, a parallel deployment method has been integrated into the architecture to significantly reduces the time needed to deploy large cloud services. To this purpose, an orchestration model has been integrated into the architecture. Moreover, the architecture supports the declarative definition of cloud services and its software components. *CDL* and *DSDL* language have been created for this purpose and successfully validated in this proposal. Finally, a high level template mechanisms for intuitively defining cloud services has been successfully implemented. This enables requirement driven rapid service provisioning while hiding the configuration complexities from the final user. As a proof of concept, a prototype has been implemented and validated with statistic results.

As a future work, it is excepted to include autonomic computing features into the *Service Orchestrator* in order to provide self-management capabilities such as fault tolerant cloud services, service level agreements management, quality of service assurance and intrusion detection.

Acknowledgement

Thanks to the *Spanish Ministerio de Educacion y Ciencia* and the *Fundacion Seneca* for sponsoring this research under the grants AP2006-4150, TIN2008-06441-C02-02 and 04552/GERM/06. Thanks to the *European Commission* for sponsoring this research under the project FP7-ICT-2007-1 SWIFT. Finally, the authors would like to thank *Matthias Schwegler* for his contribution to this research.

References

[1] J. Turnbull, *Pulling Strings with Puppet*. FristPress, 2007.
[2] A. Jacob, "Infrastructure in the cloud era," in *Proceedings at International O'Reilly Conference Velocity*, 2009.

[3] D. Solutions, "Control tier," Tech. Rep., 2010. [Online]. Available: $http://controltier.org/wiki/Main_Page$
[4] D. Frost, "Using capistrano," *Linux Journal*, vol. 177, p. 8, 2009.
[5] M. Burgess, "Knowledge management and promises," *LNCS Scalability of Networks and Services*, vol. 5637, pp. 95–107, 2009.
[6] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft, "The smartfrog configuration management framework," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 16–25, 2009.
[7] J. Murly, *Programming Amazon Web Services*. O'Really, 2008.
[8] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2009.
[9] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet, *Groovy in Action*. Manning Publications Co., 2007.