A Comparison of Multi-cloud Provisioning Platforms

Domenico Calcaterra, Vincenzo Cartelli, Giuseppe Di Modica and Orazio Tomarchio
Department of Electrical, Electronic and Computer Engineering, University of Catania, Catania, Italy

Keywords: Cloud Service Provisioning, Multicloud, Cloud orchestration, TOSCA, BPMN

Abstract: Although cloud computing has been around for over a decade now, many issues of the first hour still persist. Vendor lock-in, poor interoperability and portability hinder users from taking full advantage of main cloud features. On the industry side, the big players often adopt proprietary solutions to guarantee a seamless management and orchestration of cloud applications; on the research side, the TOSCA specification has emerged as the most authoritative effort for the interoperable description of cloud services. In this paper, we discuss the design and implementation of a TOSCA-based orchestration framework for the automated service provisioning. We also compare this approach to two open-source orchestration tools (Heat, Cloudify) with respect to the deployment of a two-tier application on an OpenStack environment. Test results show our method performs comparably to the aforementioned tools, while maintaining full compliance with TOSCA.

1 INTRODUCTION

The adoption of the cloud computing paradigm is nowadays very common in many companies and organizations. A very recent technological trend within the cloud computing scenario is the use of services and resources from multiple clouds, the so-called multi-cloud paradigm (Grozev and Buyya, 2014; Ferry et al., 2013). However, in order to enable an effective multi-cloud paradigm, it is essential to guarantee an easy portability of a cloud application from a cloud provider to another one (Petcu and Vasilakos, 2014; Ferry and Rossini, 2018).

Cloud orchestration frameworks have emerged as systems to address this need as well: their goal is to manage cloud resources through the lifecycle phases, respectively to select, describe, configure, deploy, monitor and control them (Weerasiri et al., 2017; Ranjan et al., 2015; Baur et al., 2015).

Most of the cloud providers offer their own orchestration platform (Weerasiri et al., 2017; Bousselmi et al., 2014): however, none of these commercial products are open, neither their solutions are portable on different providers. Moreover, since cloud applications may have varying resource requirements during different phases of their lifecycle, the need to partition or migrate parts of a cloud application to different platforms arises. Therefore, designing effective cloud orchestration techniques to cope with large-scale multi-cloud environments remains a deeply challenging problem.

Several efforts have been made in the last few years to overcome these issues: as a notable example, TOSCA (Topology and Orchestration Specification for Cloud Applications) has been proposed as an open standard for representing and orchestrating cloud resources (OASIS, 2013). It describes a composite cloud service using a service template, which captures the topology of component resources and sets a plan for orchestrating them (Binz et al., 2014).

In a previous paper we proposed the design and implementation of TORCH, a TOSCA-based orchestration framework for automated cloud service provisioning (Calcaterra et al., 2018). In this paper we compare this approach with two open-source orchestration tools such as Heat (OpenStack, 2016) and Cloudify (GigaSpaces, 2016), showing that our solution achieves the automated provisioning over multi-cloud environments, while maintaining a full TOSCA compliance. Other benefits include substantial model reusability and abstraction, allowing to avoid provider-specific customisation.

The remainder of the paper is organised as follows. In Section 2 we present the related work. Section 3 briefly presents the fundamentals of our system TORCH. Section 4 describes the reference scenario used to compare TORCH with Heat and Cloudify. Details on comparative tests and systems performance are provided in Sections 5 and 6, respectively. Finally, the work is concluded in Section 7.

Calcaterra, D., Cartelli, V., Di Modica, G. and Tomarchio, O.
A Comparison of Multi-cloud Provisioning Platforms.
DOI: 10.5220/0007765005070514
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 507-514
ISBN: 978-989-758-365-0

2 RELATED WORK

Many IT organizations and DevOps adopters look at the cloud orchestration as a way to speed up the delivery of services to end users and reduce costs. All the big cloud industry players use their own orchestration platform to manage the provisioning of cloud services (e.g. Amazon CloudFormation[1], Rightscale Cloud Management Platform[2], RedHat CloudForms[3], IBM Cloud Orchestrator[4]). Such platforms, to varying degrees, promise to provide automation in three fundamental steps: *cloud configuration*, *cloud provisioning* and *cloud deployment*. The most advanced also offer services and tools for the management of the cloud applications' lifecycle. None of these commercial products are open to the community, and the solutions they offer are not portable across third-party providers either.

TOSCA is a standard proposed by OASIS to enable the portability of cloud applications and the related IT services (OASIS, 2013). This specification permits to describe the structure of a cloud application as a *service template*, that is in turn composed of a *topology_template* and the types needed to build such a template. The TOSCA Simple Profile is an isomorphic rendering of a subset of the TOSCA v1.0 XML specification in the YAML language. It provides a more accessible syntax as well as a more concise and incremental expressiveness of the TOSCA language in order to speed up the adoption of TOSCA to describe portable cloud applications.

Cloudify (GigaSpaces, 2016) is an open-source orchestration framework based on TOSCA. It provides services to model applications and automate their entire lifecycle through a set of built-in workflows. Alien4Cloud (Application LIfecycle ENabler for Cloud) (Alien4Cloud, 2018) is a TOSCA-based designer and cloud application lifecycle management platform. It is natively integrated with Cloudify for runtime orchestration. Ubicity (Corporation, 2018) is yet another TOSCA-based solution, which proposes a model-driven approach to simplify service management by providing a path towards a unified view of applications and services, independent of the underlying cloud platform. The OpenStack platform[5] includes an orchestration service which provides a template-based way to describe cloud applications.

In the literature some minor research initiatives propose open-source cloud provisioning platforms which are not based on TOSCA. Roboconf (Pham et al., 2015) is a hybrid cloud orchestrator for application deployment. It proposes a Domain Specific Language (DSL) for fine-grain definition of applications and execution environments. In (Giannakopoulos et al., 2017) authors formulates the application deployment problem as a directed acyclic graph traversal and re-execution. The proposed tool is intended for deploying applications over providers that tend to present transient failures.

3 TORCH: A TOSCA-BASED PROVISIONING SYSTEM

This section briefly describes TORCH, a Tosca-inspired ORCHestration framework introduced in (Calcaterra et al., 2018). The framework orchestrates the provision of cloud services separating the orchestration of the provisioning tasks from the invocation of the provisioning services themselves. Provisioning services are supplied by third-party service providers, while the provisioning tasks orchestrated by the workflow engine draw on those services in a SOA (Service Oriented Architecture) fashion. Figure 1 depicts the framework's architecture.

The core component is the **Cloud Orchestrator** which takes a TOSCA service template (in the YAML format) as input, translates it into a collection of proper *BPMN data objects*, and feeds such a collection as *BPMN data inputs* of a new instance of the top-level process (see Figure 2) running in the *BPMN Engine*, which eventually orchestrates all the provisioning tasks. Provisioning tasks, in turn, have their data inputs populated with a collection of *data objects* describing both the demanded resources and the state of each component, and do nothing but send service requests to the *Service Broker* and receive back the new state of the invoked provisioning services.

Two categories of provisioning services are contemplated: *Cloud Services* and *Packet-based Services*. The former comprise the resources offered through any of the Cloud delivery models (IaaS, PaaS, SaaS), whereas the latter include all the downloadable software "packets" requiring a pre-configured runtime environment to run. An intermediate *Service Connectors Layer* is populated with REST services called *Service Connectors* (advertised in a *Service Registry*), which provide a unified interface model for the invocation of the provisioning services. When it comes to *Cloud Services*, additional configuration is usually required for Service Connectors to ac-

[1]https://aws.amazon.com/cloudformation/

[2]https://www.rightscale.com/

[3]https://www.redhat.com/en/technologies/management/cloudforms/

[4]https://www.ibm.com/us-en/marketplace/deployment-automation/

[5]https://www.openstack.org/

cess the cloud environment. Since TOSCA abstracts application-specific data away from templates for portability reasons, extra configuration properties can be fed into TORCH as a separate input (*Application Properties* in Figure 1). The framework is responsible to properly inject this information into service requests. Service requests issued by the BPMN plans are directed to the Service Broker, which browses the Service Registry for the most appropriate Connectors meeting the specified requirements.

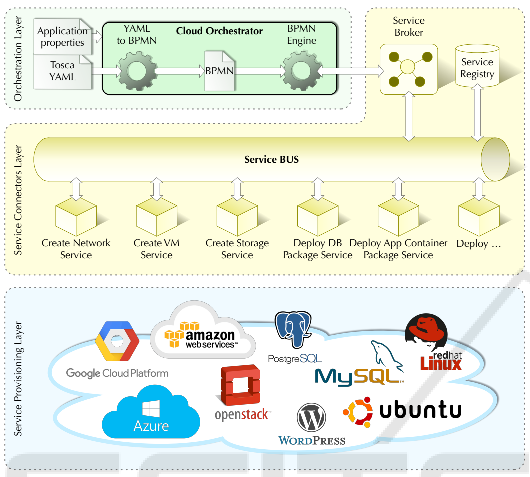


Figure 1: TORCH architecture.

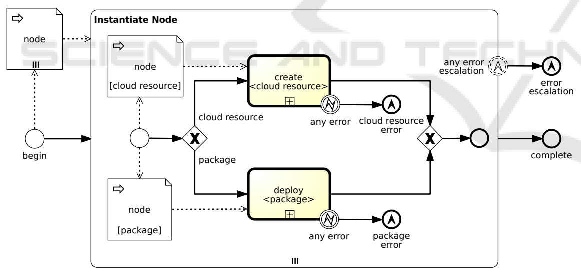


Figure 2: Cloud Orchestrator BPMN process model.

The *YAML-TO-BPMN* component (see Figure 1) is responsible for the conversion of the TOSCA YAML template into the BPMN data object collection used as data inputs in the provisioning process. The orchestration of the provisioning tasks is realised by three BPMN process models. More precisely, a new top-level process (see Figure 2) is instantiated with the output of YAML-TO-BPMN as its *"node" data input collection*. Since nodes comprise both cloud resources and software packages, the top-level process feeds each node of the input collection to a new instance of either the *create cloud resource* or the *deploy package* sub-processes. The orchestration of the provisioning tasks is carried out by many instances of these last two processes, which delegate the execution of the provisioning services to the *Service Broker*, and globally collect the status data that it returns.

4 REFERENCE SCENARIO

The application model taken into consideration deploys a WordPress web application on an Apache web server, with a MySQL DBMS hosting the application content on a separate server. Figure 3 shows the overall TOSCA-compliant architecture (wordpress, php and apache node types are non-normative, though).

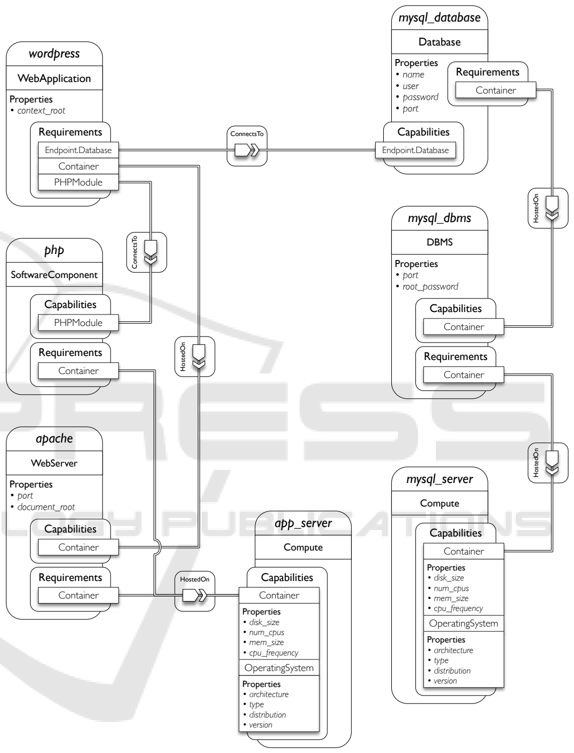


Figure 3: WordPress Deploy - TOSCA Template.

There are two separate servers: *app_server* for the web server hosting and *mysql_ server* for the DBMS hosting. Both servers are configurable on *hardware capacity* (e.g., disk size, number of cpus, memory size and CPU frequency) and *operating system environment* (e.g., OS architecture, OS type, OS distribution and OS version). The *apache* node features *port* and *document_root* properties, and is dependent upon the app_server via a *HostedOn* relationship as well. Likewise, the *php* node is dependent upon the app_server via a *HostedOn* relationship. The *mysql_dbms* node features *port* and *root_password* properties, and a *HostedOn* dependency relationship upon the mysql_server. The *mysql_database* node features *name*, *username*, *password* and *port* properties, and a *HostedOn* dependency relationship upon the mysql_dbms. The *wordpress* node features the *context_root* property, and depends on mysql_database and php via two *ConnectsTo* relationships and on apache via a *HostedOn* relationship, respectively.

5 COMPARISON OF PROVISIONING STRATEGIES

In this section the focus is put on the provisioning strategies of the aforementioned reference scenario enforced by Heat, Cloudify and TORCH, respectively, highlighting the main similarities and differences among the three methods.

5.1 Heat

Heat orchestrates composite cloud applications using either a CloudFormation compatible template format (CFN) or the native OpenStack Heat Orchestration Template format (HOT). A Heat template describes the infrastructure of a cloud application in a declarative fashion. The resources, once created, are referred to as *stacks*. HOT templates are defined in YAML and require to include at least the *heat_template_version* key and the *resources* section. Bearing in mind the use case in Section 4, Listing 1 shows how the "mysql_server" node was described.

Listing 1: MySQL Server for Heat.

```
mysql_server:
 type: OS::Nova::Server
 properties:
  name: mysqlserver
  image: { get_param: image }
  flavor: { get_param: flavor }
  key_name: { get_param: key_pair_name }
  security_groups:
   - { get_param: security_group_name }
  networks:
   - network: { get_param: network_name}
     subnet: { get_param: subnet_name}
  user_data_format: RAW
  user_data:
   str_replace:
    template: { get_file: scripts/db_install.sh }
    params:
     $db_root_password: { get_param: db_root_password }
     $db_port: { get_param: db_port }
     $db_name: { get_param: db_name }
     $db_user: { get_param: db_user }
     $db_password: { get_param: db_password }
```

The *OS::Nova::Server* allows to create a Compute instance. The *flavor* property is the only mandatory one, but a boot source needs to be defined using one of the *image* or *block_device_mapping* properties. The *key_name* property defines the key-pair to enable remote access via SSH. The *security_groups* property associates a security group to an instance. The *networks* property indicates which networks an instance should connect to. Each network contains one of the following keys: port, network. The *network* key refers to the name or ID of an existing network. The *subnet* key specifies the name or the ID of an existing subnet for the provided network. The *user_data* property enables to configure the software running on the server instance. A shell script (e.g. *db_install.sh*) can be executed by the server during boot.

Heat templates are consumed by the *OpenStack-Client*, which provides a command-line interface to OpenStack APIs. Before any Heat commands can be run, credentials need to be supplied as options on the command-line or as environment variables. As for the latter, Listing 2 shows typical credential information.

Listing 2: Credential info for Heat.

```
export OS_USERNAME=<username>
export OS_PASSWORD=<password>
export OS_PROJECT_NAME=<project-name>
export OS_USER_DOMAIN_NAME=<user-domain-name>
export OS_PROJECT_DOMAIN_NAME=<project-domain-name>
export OS_AUTH_URL=<url-to-openstack-identity>
export OS_IDENTITY_API_VERSION=<identity-api-version>
export OS_IMAGE_API_VERSION=<image-api-version>
```

5.2 Cloudify

Cloudify, based on TOSCA, automates the entire lifecycle of applications, including deployment on any cloud environment. Templates are referred to as *blueprints*, which are YAML documents written in *Cloudify's DSL (Domain Specific Language)*. Cloud services are supported through built-in plugins. In relation to the use case in Section 4, Listing 3 shows how the "mysql_server" node was described.

The *cloudify.openstack.nodes.Server* type, defined in the OpenStack plugin, allows to create a Compute instance featuring *image* and *flavor* specified in the *server* property. The *openstack_config* property declares OpenStack credentials. In contrast to Heat, they are stored as secrets and accessed via an alias referencing a named anchor in the *dsl_definitions* section. The *agent_config* property defines how to configure the "host agent", which executes orchestration operations locally and collects metrics. Configuration properties for host agents installed via SSH include: *user*, *key* and *port*. The *relationships* property defines how nodes relate to one another. Three "connected_to" relationships describe the key-pair, security group and network which the server instance should be connected to, and one "depends_on" relationship delineates the subnet which it should depend on. Conversely to Heat, Cloudify allows to model software components running on a server instance as separate nodes. Listing 4 displays the description for the *mysql_dbms* node.

Listing 3: MySQL Server for Cloudify.

```
dsl_definitions:
 openstack_config: &openstack_config
  username: { get_secret: keystone_username }
  password: { get_secret: keystone_password }
  tenant_name: { get_secret: keystone_tenant_name }
  auth_url: { get_secret: keystone_url }
  region: { get_secret: keystone_region }

node_templates:
 mysql_server:
  type: cloudify.openstack.nodes.Server
  properties:
   openstack_config: *openstack_config
   agent_config:
    user: { get_input: agent_user_name }
    key: { get_property: [ keypair, private_key_path ] }
    install_method: remote
    port: 22
   server:
    name: mysqlserver
    image: { get_input: image }
    flavor: { get_input: flavor }
  relationships:
   - type: cloudify.openstack.server_connected_to_keypair
     target: keypair
   - type: cloudify.openstack.server_connected_to_security_group
     target: security_group
   - type: cloudify.relationships.connected_to
     target: network
   - type: cloudify.relationships.depends_on
     target: subnet
```

Listing 4: MySQL DBMS for Cloudify.

```
mysql_dbms:
 type: MySqlDBMS
 properties:
  port: { get_input: db_port }
  root_password: { get_input: db_root_pwd }
 interfaces:
  cloudify.interfaces.lifecycle:
   create:
    implementation: scripts/mysqldbms/create.sh
    executor: host_agent
    inputs:
     db_root_password: { get_property: [SELF, root_password] }
   configure:
    implementation: scripts/mysqldbms/configure.sh
    executor: host_agent
    inputs:
     db_ip_address: { get_attribute: [mysql_server, ip] }
     db_port: { get_property: [SELF, port] }
 relationships:
  - type: cloudify.relationships.contained_in
    target: mysql_server
```

Two properties are defined: *port* and *root_password*. A "contained_in" relationship relates the node to *mysql_server*. Shell scripts are also specified for *create* and *configure* lifecycle operations, whose execution (via the *Script plugin*) is the host agent's responsibility. Blueprints are normally consumed by the Cloudify *Command Line Interface (CLI)*, which includes all of the commands necessary to run any actions on *Cloudify Manager*.

5.3 TORCH

TORCH offers a standardised way to deploy and orchestrate the lifecycle of applications across multiple cloud environments. Applications are modelled in YAML - according to TOSCA language - as *service templates*. Unlike Cloudify, the primary focus is only on design time aspects (i.e., the description of services). Runtime aspects (such as orchestration, service mapping and invocation) are addressed by the TOSCA-based provisioning framework (see Section 3). With reference to the use case in Section 4, Listing 5 shows how the "mysql_server" node was described.

Listing 5: MySQL Server for TORCH.

```
mysql_server:
 type: tosca.nodes.Compute
 properties:
  keypair:
   protocol: ssh
   token_type: identifier
   token: { get_input: keypair_id }
 capabilities:
  host:
   properties:
    disk_size: { get_input: ms_host_disk_size }
    num_cpus: { get_input: ms_host_cpus }
    mem_size: { get_input: ms_host_mem_size }
    cpu_frequency: { get_input: ms_host_cpu_frequency }
  os:
   properties:
    architecture: { get_input: ms_os_architecture }
    type: { get_input: ms_os_type }
    distribution: { get_input: ms_os_distribution }
    version: { get_input: ms_os_version }

network:
 type: tosca.nodes.network.Network
 properties:
  network_name: { get_input: network_name }

mysql_port:
 type: tosca.nodes.network.Port
 properties:
  ip_range_start: { get_input: subnet_starting_ip }
  ip_range_end: { get_input: subnet_ending_ip }
 requirements:
  - binding:
     node: mysql_server
  - link:
     node: network
```

The *tosca.nodes.Compute* type allows to create a Compute instance, which is configurable in terms of both hosting and operating system capabilities. The *keypair* property specifies the keypair for remote access via SSH. The "network" node represents an existing logical network, which the server instance should be connected to by means of the "mysql_port" node. Unlike Heat and Cloudify, no platform-dependent information is provided. For instance, neither *flavor* nor *image* concepts are specified. Service Connectors infer them from *tosca.capabilities.Container* and *tosca.capabilities.OperatingSystem* capability properties. The same considerations apply to *keypair*, *network* and *subnet* concepts.

Similar to Cloudify, TORCH enables to describe software components running on a server instance as separate nodes. Listing 6 exhibits the description for the *mysql_dbms* node. Two properties are defined in this case as well: *port* and *root_password*. A "tosca.relationships.HostedOn" dependency relationship relates the node to *mysql_server*. Shell scripts are also associated to *create* and *configure* lifecycle operations, whose execution is delegated to *Service Connectors* via the *Service Broker*.

Listing 6: MySQL DBMS for TORCH.

```
mysql_dbms:
 type: tosca.nodes.DBMS
 properties:
  port: { get_input: db_port }
  root_password: { get_input: db_root_pwd }
 requirements:
  - host: mysql_server
 interfaces:
  Standard:
   create:
    implementation: scripts/mysqldbms/create.sh
    inputs:
     db_root_password: { get_property: [ SELF, root_password ] }
   configure:
    implementation: scripts/mysqldbms/configure.sh
    inputs:
     db_addr: { get_attribute: [mysql_server, private_address] }
     db_port: { get_property: [ SELF, port ] }
```

Conversely to Heat and Cloudify, TORCH supplements templates with application-specific information when *Service Connectors* require additional configuration (see Section 3). By way of example, Listing 7 shows a JSON excerpt with extra configuration properties for accessing an OpenStack environment.

Listing 7: Configuration info for TORCH.

```
{
  "compute.url": <url-to-openstack-compute>,
  "identity.url": <url-to-openstack-identity>,
  "image.url": <url-to-openstack-image>,
  "network.url": <url-to-openstack-network>,
  "domain.id": <domain-id>,
  "username": <username>,
  "password": <password>,
  "security.group": <security-group>
}
```

5.4 Discussion

Table 1 generalises the achievements of the three orchestration tools. *Cloud Features* relate to the cloud infrastructure, whereas *Application Features* refer to the deployment and automation of applications. Regarding the *Cloud Features*, the following applies:

- Both Cloudify and TORCH support a distributed deployment over *multi-Cloud* environments, while Heat only supports a single-Cloud deployment on an OpenStack environment.
- Both Cloudify and TORCH are *infrastructure agnostic*; but the former does not provide an *abstraction layer*, which hides differences and avoids provider-specific customisation.

As for the *Application Features*, the main points are as follows:

- Heat does not natively support TOSCA; Cloudify is aligned with TOSCA, but it does not reference the standard types. On the contrary, TORCH is fully in compliance with the specification.
- Heat and Cloudify exclusively supports *manual binding* for cloud resources (i.e. the user needs to reference provider-specific node types). TORCH supports *automatic binding* (i.e. the user defines abstract requirements, e.g. number of cores).
- All three tools provide support for *shell scripts* to deploy applications. While Cloudify and TORCH rely on TOSCA lifecycle interfaces, Heat relies on user-data boot scripts associated to resources.
- All three tools support a way to configure communication relationships between components, and resolve dependencies via *attribute passing*.
- Heat and Cloudify support both *manual* and *automatic workflows*, while TORCH only supports automatic workflows.
- Both Cloudify and TORCH use the same *reusability* mechanisms as TOSCA (e.g., type inheritance). Heat partially supports reusability via input parameters, and template composition.

Table 1: Comparison of Heat, Cloudify, and TORCH.

	Orchestration Tools		
Cloud Features	**Heat**	**Cloudify**	**TORCH**
Multi-Cloud support	✗	✓	✓
Infrastructure agnostic	✗	✓	✓
Abstraction Layer	✗	✗	✓
Application Features			
TOSCA compliance	✗	**0**	✓
Resource Selection			
- Manual Binding	✓	✓	✗
- Automatic Binding	✗	✗	✓
Lifecycle Description			
- Shell Script	✓	✓	✓
Wiring & Workflow			
- Attribute Passing	✓	✓	✓
- Manual Workflow	✓	✓	✗
- Automatic Workflow	✓	✓	✓
Reusability	**0**	✓	✓

✗= not fulfilled, **0** = partially fulfilled, ✓= fully fulfilled

6 PERFORMANCE COMPARISON

In order to evaluate the effectiveness of the proposed approach, extensive tests for the reference scenario in Section 4 were conducted, by using Heat, Cloudify and TORCH. Deployment tests were performed on a minimal *OpenStack* environment, which comprises at least two hosts: *Controller node* and *Compute node*.

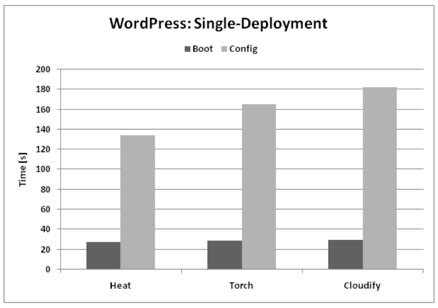
The Controller node runs the Identity service, Image service, management portions of Compute, management portion of Networking, various Networking agents, and the Dashboard. The Compute node runs the hypervisor portion of Compute that operates instances. It also runs a Networking service agent that connects instances to virtual networks. The ”provider networks” option is assumed as virtual networking infrastructure, which deploys the OpenStack Networking service with primarily layer-2. Both Controller and Compute nodes require a minimum of two network interfaces for the following: a) *Management network*, and b) *Provider network*.

A local cluster consisting of two identical off-the-shelf PCs was considered in order to create a minimal *OpenStack Queens* set-up, i.e., Controller node and Compute node. The former also runs portions of the Heat orchestration services, whilst the latter also hosts the Cloudify Manager and the TORCH as well. The features of the physical hosts can be summarised as follows: Intel Core i7-4770S Quad-Core CPU @ 3.10 GHz, 8 GB SO-DIMM DDR3 SDRAM @ 1.6 GHz, 1 TB Hard Disk, 2 NICs. Both nodes run Ubuntu Server 16.04.5 LTS x86_64 Linux distribution.

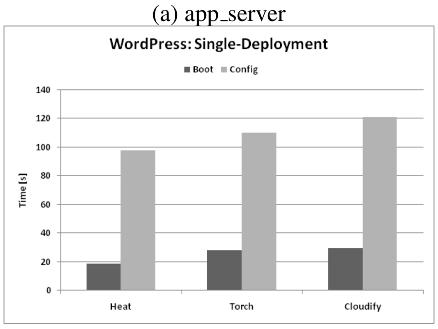
Two different kinds of deployment tests were carried out for Heat, Cloudify and TORCH; namely, *WordPress Single-Deployment* and *WordPress Multi-Deployment*. The former concerns the basic use case, the latter refers to the scalable instantiation of the former instead. Regardless of the orchestration deployment scenario, the following assumptions were made about each VM instance: a) Ubuntu Server 14.04 cloud image, b) “m1.small” flavor (vcpus = 1, ram = 1 GB, disk = 5 GB), and c) IPv4 address on a DHCP-enabled external network. The following parameters were calculated for performance evaluation: *Boot time*, *Configuration time*, and *Workflow time*. The first pertains to the time taken for a VM instance to be booted and *cloud-init* modules to be executed on it. The second refers to the time software installation and configuration on a VM instance takes. The third concerns the time the workflow takes for the entire deployment to be complete.

Figure 4 illustrates the results obtained for the single-deployment scenario. Both *app_server* (Figure 4a) and *mysql_server* (Figure 4b) instances were considered. Boot and Configuration times (on the Y-axis) for each VM instance were computed depending on Heat, TORCH and Cloudify approaches (on the X-axis). Both charts show that TORCH performance lies halfway between Heat and Cloudify ones.

Test results for the multi-deployment scenario are displayed in Figure 5. Three different experiments were conducted by triggering the multiple instantiation of the single-deployment scenario for 2 VMs (Figure 5a), 4 VMs (Figure 5b) and 6 VMs (Figure 5c), respectively. Boot, Configuration and Workflow times (on the Y-axis) for VM instances were calculated according to Heat, TORCH and Cloudify methods (on the X-axis). In contrast to the single-



(a) app_server



(b) mysql_server

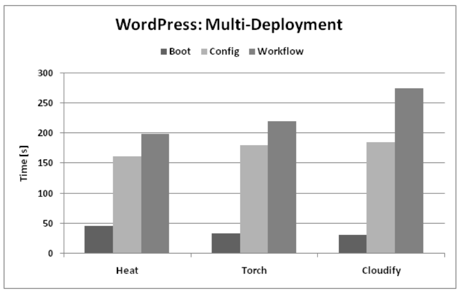
Figure 4: WordPress Single-Deployment - Boot and Configuration.

deployment case, both Boot and Configuration times are meant as the maximum Boot and Configuration times. For the sake of clarity, Boot time is calculated as the difference between the maximum Boot end-time and the minimum Boot start-time among all the VMs. The same applies to Configuration time. Even in this case, the charts show that TORCH performs midway between Heat and Cloudify. In particular, regardless of the number of VMs, the following remarks can be made: a) both TORCH and Cloudify prove to be more effective in lowering Boot times, and b) Configuration and Workflow times gradually increase from Heat to Cloudify, passing through TORCH.

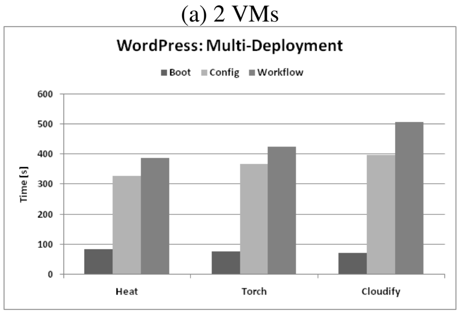
7 CONCLUSION

Automating service orchestration has become one of the driving factors behind the growth of cloud providers. Several initiatives and tools have emerged to facilitate the portable, automated management of cloud services throughout their lifecycle.

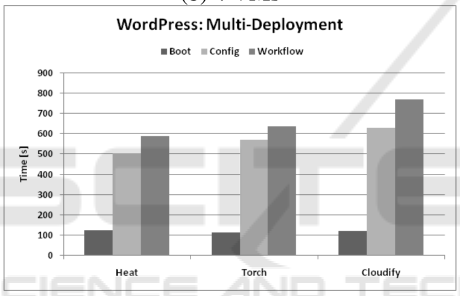
In this work, we presented TORCH, i.e. a TOSCA-based orchestration framework that automates the cloud service provisioning. In addition, we selected two open-source orchestration tools (Heat, Cloudify) and compared them on both quality and quantity aspects using the experience from deploying a sample use case. Test findings lead to the insight



(a) 2 VMs



(b) 4 VMs



(c) 6 VMs

Figure 5: WordPress Multi-Deployment: Boot, Configuration and Workflow.

that our method compares to Heat and Cloudify, while remaining TOSCA-compliant. Future work will include support for containerisation using Docker[6].

REFERENCES

Alien4Cloud (2018). FastConnect. https://alien4cloud.github.io/index.html. Last accessed on 24-01-2019.

Baur, D., Seybold, D., Griesinger, F., Tsitsipas, A., Hauser, C. B., and Domaschka, J. (2015). Cloud Orchestration Features: Are Tools Fit for Purpose? In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing, UCC 2015*, pages 95–101.

Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2014). TOSCA: Portable Automated Deployment and Management of Cloud Applications. *Advanced Web Services*.

Bousselmi, K., Brahmi, Z., and Gammoudi, M. M. (2014). Cloud services orchestration: A comparative study of existing approaches. In *IEEE 28th International Conference on Advanced Information Networking and Applications Workshops, (WAINA 2014)*, pages 410–416.

Calcaterra, D., Cartelli, V., Di Modica, G., and Tomarchio, O. (2018). Exploiting BPMN features to design a fault-aware TOSCA orchestrator. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 533–540, Funchal-Madeira (Portugal).

Corporation, U. (2018). Ubicity. https://ubicity.com/. Last accessed on 24-01-2019.

Ferry, N. and Rossini, A. (2018). CloudMF: Model-Driven Management of Multi-Cloud Applications. *ACM Trans. Internet Technol*, 18(2):16–24.

Ferry, N., Rossini, A., Chauvel, F., Morin, B., and Solberg, A. (2013). Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 887–894. IEEE.

Giannakopoulos, I., Konstantinou, I., Tsoumakos, D., and Koziris, N. (2017). Recovering from cloud application deployment failures through re-execution. In *Algorithmic Aspects of Cloud Computing: Second International Workshop, ALGOCLOUD 2016, Aarhus, Denmark*, pages 117–130.

GigaSpaces (2016). Cloudify. http://getcloudify.org/. Last accessed on 24-01-2019.

Grozev, N. and Buyya, R. (2014). Inter-Cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*, 44(3):369–390.

OASIS (2013). Topology and Orchestration Specification for Cloud Applications Version 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html. Last accessed on 10-04-2018.

OpenStack (2016). OpenStack Heat. https://wiki.openstack.org/wiki/Heat. Last accessed on 15-02-2017.

Petcu, D. and Vasilakos, A. V. (2014). Portability in clouds: approaches and research opportunities. *Scalable Computing: Practice and Experience*, 15(3):251–270.

Pham, L. M., Tchana, A., Donsez, D., de Palma, N., Zurczak, V., and Gibello, P. Y. (2015). Roboconf: A hybrid cloud orchestrator to deploy complex applications. In *Proceeding of IEEE 8th International Conference on Cloud Computing*, pages 365–372.

Ranjan, R., Benatallah, B., Dustdar, S., and Papazoglou, M. P. (2015). Cloud Resource Orchestration Programming: Overview, Issues, and Directions. *IEEE Internet Computing*, 19:46–56.

Weerasiri, D., Barukh, M. C., Benatallah, B., Sheng, Q. Z., and Ranjan, R. (2017). A taxonomy and survey of cloud resource orchestration techniques. *ACM Comput. Surv.*, 50(2):26:1–26:41.

[6]https://www.docker.com/