Exploiting BPMN Features to Design a Fault-aware TOSCA Orchestrator

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

Keywords: Cloud Provisioning, Cloud Orchestration, Fault Tolerance, TOSCA, BPMN.

Abstract: Cloud computing is nowadays a consolidated paradigm that enables scalable access to computing resources and complex services. One of the greatest challenges Cloud providers have to deal with is to efficiently automate the service “provisioning” activities through Cloud orchestration techniques. By relying on TOSCA, a well-known standard specification for the interoperable description of Cloud services, we developed a fault-aware orchestrator capable of automating the workflow for service provisioning. The BPMN notation was used to define both the workflow and the data associated with workflow elements. To corroborate the proposal, a software prototype was developed and tested with a sample use case which is discussed in the paper.

1 INTRODUCTION

Nowadays, Cloud computing is gaining an increasing popularity over traditional systems in terms of flexibility, which brings both great opportunities and challenges. One of the critical challenges is *resiliency*, defined as the capacity of a system to remain reliable, fault tolerant, survivable, dependable, and secure in case of any malicious or accidental malfunctions or failures that result in a temporary or permanent service disruption (Colman-Meixner et al., 2016). Then, resiliency strongly relates to *fault tolerance*, meaning that a system can provide its services even in the presence of faults. As for service provisioning, a number of special activities need to be put into place with proper timing in order to build up a Cloud service. If any of them fails, in the absence of fault-tolerance mechanisms, the system might not be able to provide the required service.

In this work we analyse faults that occurr at *provisioning time*. Considerations stemmed from the analysis drove the design of a **fault-aware orchestrator** capable of pipelining the service provisioning activities according to fault-tolerant schemes. The provisioning activities scheduled by the orchestrator can detect faults when they occur, and automatically put in force counteractions to recover from faults, thus minimizing the recourse to human intervention. In order to operate on top of different Cloud platforms, we leveraged the Cloud application model description of the TOSCA specification (OASIS, 2013).

The remainder of the paper is organized in the following way. In Section 2 we report some literature works addressing fault management in the Cloud context. Section 3 provides a bird’s eye view of the OASIS TOSCA standard. In Section 4 we delve into faults in Cloud service provisioning, while in Section 5 the design of the proposed fault-aware TOSCA orchestrator is presented. We provide a sample use case in Section 6 and conclude the work in Section 7.

2 RELATED WORK

When it comes to fault tolerance, there is a distinction made among fault, error and failure (Agarwal and Sharma, 2015). A fault is defined as the inability of a system to do its required task caused by an anomalous state or bug in one or more than one parts of a system. An error is a part of a system’s state that may lead to a failure, which refers to misconduct of a system that can be observed by a user. Faults can be of various types including: network faults, physical faults, media faults, processor faults, process faults and service expiry faults (Sivagami and EaswaraKumar, 2015). Faults may also be generally classified as transient, intermittent and permanent ones. Various fault tolerance techniques can be used at either task level or workflow level to resolve the faults, which are classified into two types: proactive and reactive (Patra et al., 2013) (Amin et al., 2015) (Poola et al., 2017).

Calcaterra, D., Cartelli, V., Di Modica, G. and Tomarchio, O.
Exploiting BPMN Features to Design a Fault-aware TOSCA Orchestrator.
DOI: 10.5220/0006775605330540
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 533-540
ISBN: 978-989-758-295-0
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

By leveraging on the policies of proactive and reactive methods, several architecture models have also been proposed to provide fault tolerance (Cheraghlou et al., 2016).

Different challenges in the context of Cloud computing can be encountered both during and after the deployment of applications. Specifically, manual deployment of large-scale distributed systems is time-consuming and error-prone (Hamilton, 2007) (Leite et al., 2014). That's the reason why the deployment process must be automated and resilient. Speaking of automation, BPEL and BPMN are the most applied standards for service composition and orchestration (Vargas-Santiago et al., 2017). Three basic fault handling concepts are provided by the BPEL engine: compensation handlers, fault handlers, and event handlers. Nevertheless, BPEL only manages predefined faults specified by application designers. Similarly, the BPMN engine provides error events, cancel events and compensation events.

The scientific community has also taken an interest in research proposals for fault-tolerant frameworks. In (Varela-Vaca et al., 2010), a framework for developing business processes with fault tolerance capabilities was provided. The framework presents different mechanisms in the fault tolerance scope, contemplating both replication solutions and software fault-tolerant techniques. In (Jayasinghe et al., 2013), the authors presented a fault-tolerant runtime (AESON) to recover applications from failures that could possibly happen during and after deployment. Three types of failures are supported: node crashes, node hangs and application component failures. Even though deployment failures were addressed, this work suffers from two major drawbacks: a) AESON was designed as a P2P system, and b) application models are not TOSCA-compliant. In (Giannakopoulos et al., 2017), a deployment methodology with error recovery features was proposed. It bases its functionality on identifying the script dependencies and re-executing the appropriate configuration scripts. Nevertheless, this approach can only resolve transient failures occurring during the deployment phase.

3 THE TOSCA SPECIFICATION

TOSCA is the acronym for *Topology and Orchestration Specification for Cloud Applications*. It is a standard designed 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 composed of a *topology_template* and the types needed to build such a template. The topology_template is a typed directed graph, whose nodes (called *node_templates*) model the application components, and edges (called *relationship_templates*) model the relations occurring among such components. Each node of a topology can also be associated with the corresponding component's requirements, the operations to manage it, the capabilities it features, and the policies applied to it.

TOSCA supports the deployment and management of applications in two different flavours: *imperative processing* and *declarative processing*. The imperative processing requires that all needed management logic is contained in the *Cloud Service Archive (CSAR)*. *Management plans* are typically implemented using workflow languages, such as BPMN or BPEL. The declarative processing shifts management logic from plans to runtime. TOSCA runtime engines automatically infer the corresponding logic by interpreting the application topology template. The set of provided management functionalities depends on the corresponding runtime and is not standardized by the TOSCA specification.

The TOSCA Simple Profile is an isomorphic rendering of a subset of the TOSCA specification (OASIS, 2013) in the YAML language (OASIS, 2017). The TOSCA Simple Profile defines a few normative workflows that are used to operate a topology, and specifies how they are declaratively generated: *deploy*, *undeploy*, *scaling-workflows* and *auto-healing workflows*. Imperative workflows can still be used for complex use-cases that cannot be solved in declarative workflows. However, they provide less reusability as they are defined for a specific topology rather than being dynamically generated based on the topology content. Moreover, by default, any activity failure of the workflow will result in the failure of the whole workflow. Although some constructs (e.g., *on_failure*) allow to execute rollback operations, neither policies nor mechanisms are defined to automatically recover from failures happening during the deployment of the topology.

The work described in this paper heavily grounds on the TOSCA standard and, specifically, on the TOSCA Simple Profile.

4 ANALYSIS OF FAULTS IN SERVICE PROVISIONING

IaaS providers usually allow to create and manage Cloud resources using web-based dashboards, CLI clients, REST APIs, and language-specific SDKs. Although all the aforementioned approaches provide ac-

cess to different Cloud services, REST APIs do it without any restrictions whatsoever. Besides, since REST architectural style uses well-known W3C standards, it comes with several advantages in terms of flexibility, portability, simplicity and scalability. As a result, REST-APIs can be considered as the preferential way to access Cloud infrastructure services.

Notwithstanding that most of IaaS providers offer a set of REST APIs, infrastructure differences often reflect badly upon the semantics of interaction with heterogeneous services. There is, therefore, a need for models capable of masking these differences in order to achieve a unified interaction paradigm. In particular, a process of homogenization should be carried out for *REST-API error codes* and *Resource status codes*.

Without loss of generality, OpenStack APIs and Amazon EC2 APIs have been taken into account for mapping heterogeneous, platform-dependent error and status codes onto homogeneous, platform-independent error and status codes. Specifically, mapping templates have been produced for the provision of the following Cloud resources: ***VM***, ***Storage***, ***Network*** and ***Subnet***.

Table 1: Error and status codes mapping for VMs.

	VM	
	Error Code	***Error Mapping***
EC2	IdempotentParameterMismatch	400 Bad Request
	InsufficientFreeAddressesInSubnet	
	InvalidAMIID.Malformed	
	InvalidAMIID.Unavailable	
	InvalidBlockDeviceMapping	
	InstanceLimitExceeded	
	InvalidInterface.IpAddressLimitExceeded	
	InvalidParameter	
	InvalidParameterCombination	
	InvalidParameterValue	
	MissingParameter	
	SecurityGroupLimitExceeded	
	Unsupported	
	UnsupportedOperation	
	VolumeTypeNotAvailableInZone	
	AuthFailure	401 Unauthorized
	InvalidAMIID.NotFound	404 Not Found
	InvalidGroup.NotFound	
	InvalidKeyPair.NotFound	
	InvalidNetworkInterfaceID.NotFound	
	InvalidSnapshot.NotFound	
	InvalidSubnetID.NotFound	
OpenStack	badRequest	400 Bad Request
	unauthorized	401 Unauthorized
	forbidden	403 Forbidden
	itemNotFound	404 Not Found
	conflict	409 Conflict
	Status Code	***Status Mapping***
EC2	pending	wip
	running	ok
	terminated	error
OpenStack	BUILD	wip
	ACTIVE	ok
	ERROR	error

With regard to *VM* provisioning, let us consider Table 1, which represents mappings for error codes and status codes as well. The second column contains error codes (and status codes, in that order), that are grouped by both the APIs they refer to (as specified in the first column) and the HTTP error codes (and the generic status codes, in that order) they can be mapped onto (as specified in the third column). For instance, according to Table 1, the "InvalidAMIID.Malformed" entry identifies an Amazon EC2 APIs error code, which can be mapped onto the HTTP "400 Bad Request" error code; the "itemNotFound" entry identifies an OpenStack APIs error code, which can be mapped onto the HTTP "404 Not Found" error code. Similarly, the "pending" entry identifies an Amazon EC2 APIs status code, which can be mapped onto the "wip" (i.e., work in progress) status code; the "ACTIVE" entry identifies an OpenStack APIs status code, which can be mapped onto the "ok" status code.

Table 2: Error and status codes mapping for Storage.

	Storage	
	Error Code	***Error Mapping***
EC2	MaxIOPSLimitExceeded	400 Bad Request
	UnknownVolumeType	
	VolumeLimitExceeded	
	IncorrectState	
	MissingParameter	
	AuthFailure	401 Unauthorized
	InvalidSnapshot.NotFound	404 Not Found
	InvalidZone.NotFound	
	Status Code	***Status Mapping***
EC2	creating	wip
	available	ok
	error	error

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 3: Error and status codes mapping for Networks.

	Network	
	Error Code	***Error Mapping***
EC2	InvalidVpcRange	400 Bad Request
	VpcLimitExceeded	
	AuthFailure	401 Unauthorized
	Status Code	***Status Mapping***
EC2	pending	wip
	available	ok

Table 4: Error and status codes mapping for Subnets.

	Subnet	
	Error Code	***Error Mapping***
EC2	SubnetLimitExceeded	400 Bad Request
	AuthFailure	401 Unauthorized
	InvalidVpcID.NotFound	404 Not Found
	InvalidSubnet.Conflict	409 Conflict
	Status Code	***Status Mapping***
EC2	pending	wip
	available	ok

In a similar way, Table 2 displays mappings for *Storage* provisioning, whereas Table 3 exhibits mappings for *Network* provisioning, and Table 4 illustrates mappings for *Subnet* provisioning instead. For the sake of brevity, it should be noted that only Table 1 shows error codes and status codes for both OpenStack APIs and Amazon EC2 APIs.

5 DESIGN OF A FAULT-AWARE TOSCA ORCHESTRATOR

The objective of this work is to design a TOSCA Orchestrator which enriches the deployment process with ad-hoc activities to deal with faults, whenever they occur. The fault model described in Section 4 was used to design a fault-aware deploying process. The Orchestrator is part of a bigger framework that was designed for the provisioning of applications in the Cloud. Section 5.1 provides a description of such a framework and its components. In Section 5.2 the fault-tolerant BPMN schemes enforced by the Orchestrator to deploy Cloud applications are discussed.

5.1 System Architecture

In (Calcaterra et al., 2017) a software framework that automates the provision of Cloud services was discussed. In this section we briefly report the description of the architecture of the provisioning framework. Starting from the Cloud application model description, the proposed framework is capable of devising and orchestrating the workflow of the provisioning operations to execute. We have designed and implemented a **TOSCA Orchestrator** which transforms the YAML model into an equivalent BPMN model, which is fed to a BPMN engine that will instantiate and coordinate the relative process. The process will put in force all the provisioning activities needed to build up the application stack. The overall provisioning scenario is best depicted in Figure 1.

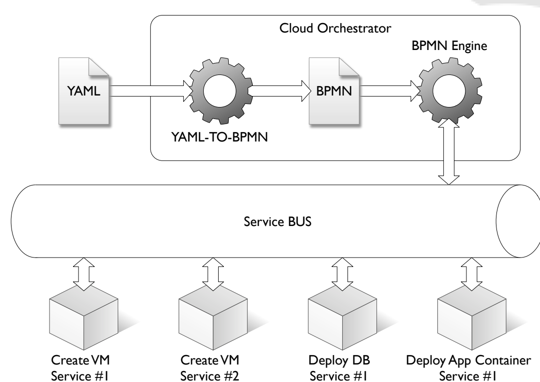

Figure 1: Cloud Orchestrator scenario.

Providers can design their services according to specific templates and offer them to Customers through an *Enterprise Service Bus* (ESB). There are mainly two categories of Provisioning Services that need to be integrated in the ESB: *Cloud Services* and *Packet-based Services*. In order to integrate all the mentioned services in the ESB, we deploy a layer of *Service Connectors* which are responsible for connecting requests coming from the Provisioning Tasks with the Provisioning Services.

Figure 2 shows the ESB-based architecture. The Connectors layer provides a unified interface model for the invocation of the services, which allows to achieve *service location transparency* and *loose coupling* between Provisioning BPMN plans (orchestrated by the *Process Engine*) and Provisioning Services. The *Service Registry* is responsible for the registration and discovery of Connectors. The *Service Broker* is in charge of taking care of the requests coming from the Provisioning Tasks.

Cloud Services Connectors implement interactions with Cloud Providers for the allocation of Cloud resources. For each service type a specific Connector needs to be implemented. For instance, **Instantiate VM** represents the generic Connector interface to the instantiation of Cloud resources of “Virtual Machine” type. All concrete Connectors to real VM services (Amazon, OpenStack, Azure, etc.) must implement the **Instantiate VM** interface. Likewise, **Add Storage** is the generic Connector interface to storage services that concrete Connectors to real storage services in the Cloud must implement.

Packet-based Services Connectors are meant to implement interactions with all service providers that provide packet-based applications. When the YAML-to-BPMN conversion takes place, three types of BPMN service tasks might be generated: “Create”, “Configure” and “Start”. To each of these tasks corresponds a generic connector interface (**Create**, **Configure** and **Start**). Those interfaces are then extended in order to manage many types of applications (DBMS, Web Servers, etc.). The latters are the ones that concrete Connectors must implement in order to interact with real packet-based application providers.

5.2 Fault-tolerant BPMN Schemes

The YAML-TO-BPMN component (see Figure 1) is responsible for the conversion of the TOSCA YAML template into the BPMN schemes that the BPM engine will have to execute. Those schemes define a workflow of fault-tolerant provision activities, that can detect faults and react accordingly. All recoverable faults are autonomously managed by the process tasks thus minimizing the recourse to human intervention (which we will refer to as *escalation*).

From the analysis of the most common deployment faults in Section 4, it appears that some faults may occur on the Cloud Provider side, others are due to bad service requests issued by the Customers. As for the faults at the Provider side, some might turn to

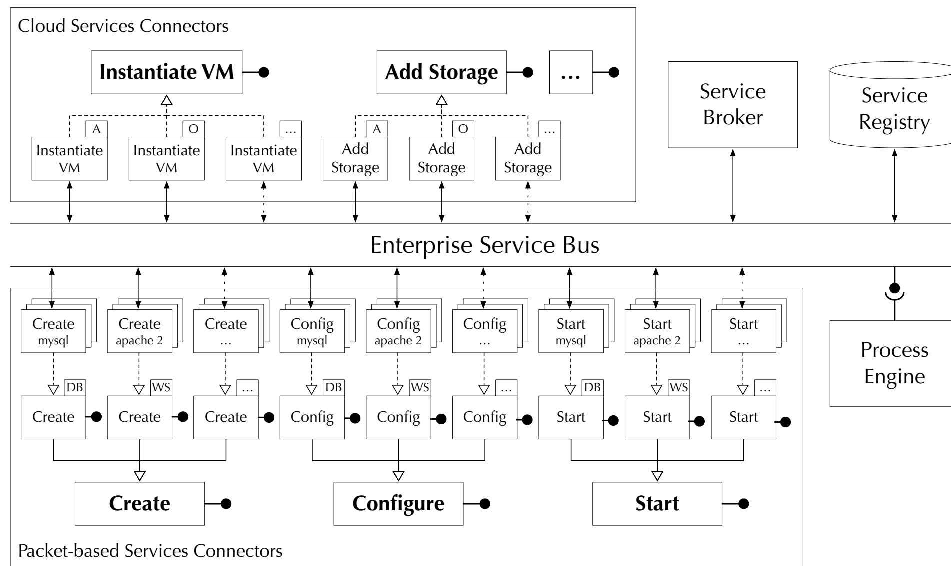

Figure 2: Enterprise Service Bus and Service Connectors.

be transient, some others are permanent and require the intervention of an operator. When Customers issue bad requests, instead, there is no way but to call for human intervention. Also, most of Cloud service APIs are by their nature *asynchronous*. The timing of Cloud service activation is an aspect that needs to be taken into account when pipelining the provision activities. A faulty access to a service instance might be mistaken for a faulty service, when instead the problem could just be that the service is being activated and, therefore, is not yet ready for use.

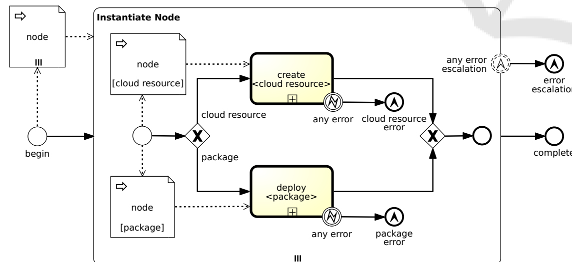

Figure 3: BPMN scheme of the service provision workflow.

The BPMN provisioning schemes we devised take care of all the discussed requirements for the Cloud services fault management. In the case of *Packet-based Services*, at the moment only schemes for the detection of faults and the subsequent escalation have been defined. The definition of more elaborated schemes to deal with faulty packet-based services will be object of future work. In Figure 3 the overall service provision workflow is depicted. The diagram is composed of a parallel multi-instance sub-process, i.e., a set of sub-processes (called "Instantiate Node") each processing a TOSCA *node* in a parallel fashion.

Since a TOSCA *node* can be either a Cloud resource or a software package, a sub-process will proceed to either a "create cloud resource" or a "deploy package" sub-process. Here, whenever an error is detected, an escalation is thrown by the relative "escalation end event" ("cloud resource error" or "package error") in the parent sub-process ("Instantiate Node"). The selective escalation ends the faulty "Instantiate Node" sub-process and keeps all other sub-processes alive and running, while the faulty one is being recovered.

In Figure 4 the workflow of the "create cloud resource" sub-process is depicted. The top pool called "Node Instance" represents the pool of all instances of either the "create cloud resource" sub-process or the "deploy package" sub-process, which are running in parallel to the "create cloud resource" sub-process being analysed. The bottom pool called "Cloud Service Connectors" represents the pool of the software connectors deployed in the ESB, which interface to the Cloud Service APIs. In the middle pool the sequence of tasks carried out to create and instantiate a Cloud resource are depicted. Interactions of the middle pool with the "Node Instance" pool represent points of synchronization between the multiple installation instances, that may be involved in a provision process. The creation of a Cloud resource starts with a task that awaits notifications coming from the preceding sub-processes. A service task will then trigger the actual instantiation by invoking the appropriate Connector on the ESB. Here, if a fault occurs, it is immediately caught and the whole sub-process is cancelled. Following the path up to the parent process (see Figure 3), the escalation is engaged. If the creation step is

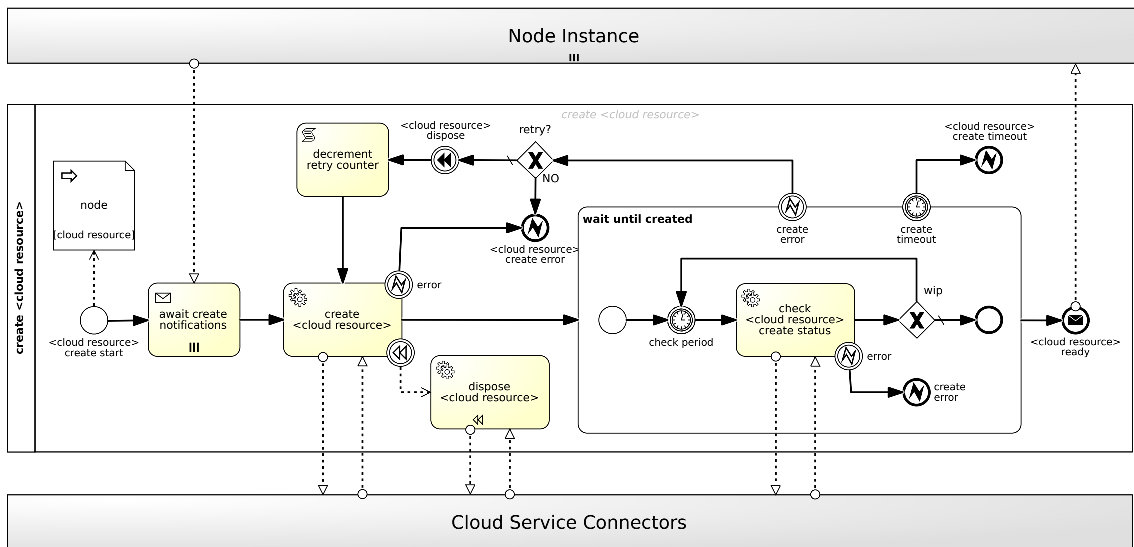

Figure 4: BPMN scheme of the Cloud Service provision workflow.

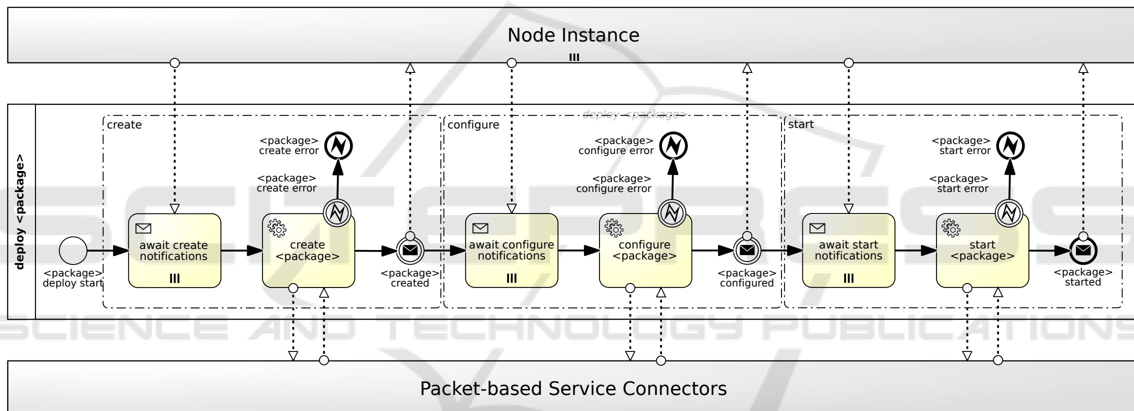

Figure 5: BPMN scheme of the Packet-based Service provision workflow.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

successful, a “wait-until-created” sub-process is activated. Checks on the resource status are iterated until the latter becomes available for use. The “check cloud resource create status” service task is committed to invoke the Connector on the ESB to check the resource status on the selected Provider. Checking periods are configurable, so is the timeout put on the boundary of the sub-process. An error event is thrown either when the timeout has expired or when an explicit error has been signalled in response to a resource check call. In the former case, the escalation is immediately triggered; in the latter case, an external loop will lead the system to autonomously re-run the whole resource creation sub-process a configurable number of times, before giving up and eventually calling up the escalation. Moreover, a compensation mechanism (“dispose cloud resource” task) allows to dispose of the Cloud resource, whenever a fault has occurred.

In Figure 5 the workflow of the “deploy package” sub-process is depicted. This sub-process presents three synchronization points with the “Node Instance” pool. Notifications from preceding TOSCA node instances must be awaited before executing the “create package”, “configure package” and “start package” tasks, respectively. Also, the sub-process is in charge of sending out notifications when each of the mentioned tasks successfully terminates its execution. Regarding the management of potential faults, they are caught and handled via escalation.

6 USE CASE

The Application model taken into consideration deploys a WordPress web application on an Apache web server, with a MySQL DBMS hosting the database content of the application on a separate server. Figure 6 shows the overall TOSCA-compliant architec-

ture (although wordpress, php and apache node types are non-normative).

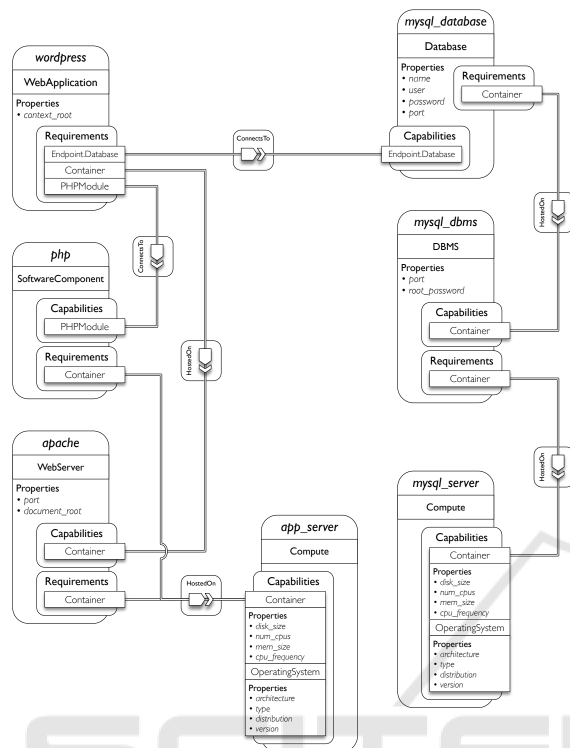

Figure 6: Wordpress Deploy - TOSCA Template.

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 side* (e.g., disk size, number of cpus, memory size and CPU frequency) and *software side* (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. In the same way, 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. Finally, the *wordpress* node features the *context_root* property, and depends on mysql_database and php by means of two *ConnectsTo* relationships and on apache by means of a *HostedOn* relationship, respectively.

The TOSCA template was fed to the YAML-TO-BPMN converter which produced the BPMN schemes implementing the provision process. The lack of space prevented us from reporting all the workflow diagrams describing the instantiation sub-processes. In Figure 7, instead, we have reported the *BPMN 2.0 Collaboration diagram* describing only the interactions among sub-processes. For the sake of simplicity, in all sub-processes only the synchronization messages have been reported. Messages give a clear idea of the precedence constraints that affect the provision. For instance, the "deploy apache" sub-process awaits the notification coming from the "create application server" sub-process, which in turn awaits notifications coming from both the "create private network" and the "create public network" sub-processes; the "deploy Wordpress" awaits the notification from the "deploy Apache" sub-process before executing the creation task, but after that, and before starting the configure task, it needs two further notifications from the "deploy database" and the "deploy PHP" sub-processes, respectively.

7 CONCLUSION

The automation of the provisioning of complex Cloud applications is becoming a crucial factor for the competitiveness of Cloud providers. Several frameworks and standards addressing this issue have appeared: however, as shown in this work, the resilience of the provisioning process is not adequately dealt with. In this work, leveraging on the TOSCA specification, we proposed the definition of a Cloud orchestration and provisioning framework that automates the Cloud service deployment, explicitly taking into account the occurrence of failures during the provisioning process. The resulting fault-aware orchestrator includes specific plans to recover from those failures, minimizing human intervention. In our future work, the framework will be enhanced with more complex plans to deal with different fault types.

REFERENCES

Agarwal, H. and Sharma, A. (2015). A comprehensive survey of fault tolerance techniques in cloud computing. In *2015 International Conference on Computing and Network Communications (CoCoNet)*, pages 408–413.

Amin, Z., Singh, H., and Sethi, N. (2015). Review on fault tolerance techniques in cloud computing. *International Journal of Computer Applications*, 116(18):11–17.

Calcaterra, D., Cartelli, V., Di Modica, G., and Tomarchio, O. (2017). Combining TOSCA and BPMN to Enable Automated Cloud Service Provisioning. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 187–196, Porto (Portugal).

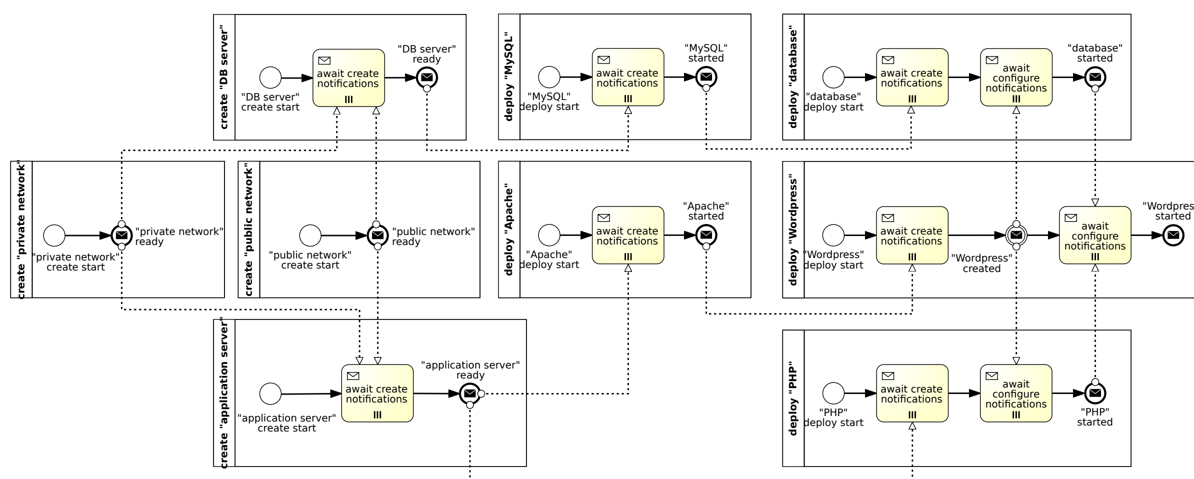

Figure 7: Wordpress Deploy - Collaboration diagram.

Cheraghlou, M. N., Khadem-Zadeh, A., and Haghparast, M. (2016). A survey of fault tolerance architecture in cloud computing. *Journal of Network and Computer Applications*, 61:81 – 92.

Colman-Meixner, C., Develder, C., Tornatore, M., and Mukherjee, B. (2016). A survey on resiliency techniques in cloud computing infrastructures and applications. *IEEE Communications Surveys Tutorials*, 18(3):2244–2281.

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.

Hamilton, J. (2007). On designing and deploying internet-scale services. In *Proceedings of the 21st Conference on Large Installation System Administration Conference*, LISA'07, pages 18:1–18:12.

Jayasinghe, D., Pu, C., Oliveira, F., Rosenberg, F., and Eilam, T. (2013). AESON: A Model-Driven and Fault Tolerant Composite Deployment Runtime for IaaS Clouds. In *2013 IEEE International Conference on Services Computing*, pages 575–582.

Leite, L., Moreira, C. E., Cordeiro, D., Gerosa, M. A., and Kon, F. (2014). Deploying Large-Scale Service Compositions on the Cloud with the CHOReOS Enactment Engine. In *IEEE 13th International Symposium on Network Computing and Applications*, pages 121–128.

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 22-01-2018.

OASIS (2017). TOSCA Simple Profile in YAML Version 1.2. http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2. Last accessed on 22-01-2018.

Patra, P., Singh, H., and Singh, G. (2013). Fault tolerance techniques and comparative implementation in cloud computing. *International Journal of Computer Applications*, 64:37–41.

Poola, D., Salehi, M. A., Ramamohanarao, K., and Buyya, R. (2017). Chapter 15 - A Taxonomy and Survey of Fault-Tolerant Workflow Management Systems in Cloud and Distributed Computing Environments. In *Software Architecture for Big Data and the Cloud*, pages 285 – 320.

Sivagami, V. and EaswaraKumar, K. (2015). Survey on fault tolerance techniques in cloud computing environment. *International Journal of Scientific Engineering and Applied Science*, 1(9):419–425.

Varela-Vaca, A. J., Gasca, R. M., Borrego, D., and Pozo, S. (2010). Towards Dependable Business Processes with Fault-Tolerance Approach. In *2010 Third International Conference on Dependability*, pages 104–111.

Vargas-Santiago, M., Hernández, S. E. P., Rosales, L. A. M., and Kacem, H. H. (2017). Survey on Web Services Fault Tolerance Approaches Based on Checkpointing Mechanisms. *JSW*, 12(7):507–525.