SALSA: A Framework for Dynamic Configuration of Cloud Services

Duc-Hung Le, Hong-Linh Truong, Georgiana Copil, Stefan Nastic and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology
E-mail: {d.le,truong,e.copil,s.nastic,dustdar}@dsg.tuwien.ac.at

***Abstract*—Contemporary cloud services are constructed from different types of software and deployed on multiple cloud infrastructures, which offer various configuration options, and can change dynamically at runtime. Due to this complexity, such cloud services require substantial configuration efforts. Currently we lack techniques for automating the complex tasks and providing fine-grained configuration features for multi-cloud services. In this paper, we present a novel multi-level configuration approach for complex cloud services on multi-cloud environments. We develop techniques for automating configuration orchestration activities. Our solution enables the fine-grained configuration at different application abstraction levels and supports the dynamic change of cloud services at runtime. We provide the SALSA framework to implement our approach and demonstrate its usefulness with several real-world services.**

***Keywords*-Dynamic configuration, runtime management, multiple clouds, cloud computing, framework**

I. Introduction

Because cloud services are composed of components of various types, such as middleware services, containers and application-specific libraries, which are deployed on different types of cloud resources, the configuration and deployment of such cloud services require substantial efforts. In practices, it is possible to combine different tools (e.g. Chef [1] and SlipStream [2]) to accomplish complex configuration tasks, but it is difficult to automate the whole service configuration that deals with various types of artifacts and services.

Two types of users[1] would face above mentioned challenges. First, service controllers [3] [4], which control the dynamic changes of cloud services require mechanisms to bring the cloud services to the desired configurations. In order to provide more controlling abilities, the controllers need to perform complex configuration actions, which deal with different types of services. Second, cloud service developers or providers need coarse-grained configuration capabilities as they do not want to deal with underlying complex configuration tasks, such as dependencies resolving, cloud resources provisioning and deploying common components. This is difficult to achieve because of the dynamic and diverse nature of multi-cloud environments.

This work was supported by the European Commission in terms of the CELAR FP7 project (FP7-ICT-2011-8 #317790). The authors would like to thank Alessio Gambi, Daniel Moldovan, Oliver Moser and Tien Dung Nguyen for their fruitful insights.

[1]In this paper, we refer *users* as the service providers and cloud service controllers who use SALSA framework.

There are contemporary tools to support configuration tasks, such as SlipStream [2] and Google Cloud Deployment Manager [5] for declaring and deploying virtual machine-based environments, or Chef [1] and Puppet [6] for supporting artifact configuration. Research frameworks [7]–[10] have also been proposed for addressing the configuration problem. However, these solutions do not provide a full view of dynamic configuration of the complex cloud services, thus automating the configuration tasks is complex for both types of users.

This paper introduces a framework for configuration orchestration of cloud services with the following contributions:

- A model for providing the configuration capabilities on different application levels and deployment stacks. It supports the configuration on multiple granularities of cloud service structure and for different kinds of services.
- A service for configuring distributed services in which relationships between service units are complex, providing a full view of the cloud service and centralizing configuration functionalities.

We have implemented SALSA, the prototype of our framework for dynamic configuration functionalities. We show how our solution can simplify and enhance the configuration process of complex cloud services at runtime on multiple cloud infrastructures.

The remaining of the paper is organized as follows. Section 2 presents our motivation based on a real world example. Section 3 describes the multi-level configuration information. Section 4 shows techniques to achieve runtime configuration. Section 5 outlines some use cases and experiments. Section 6 concludes the paper and discusses the future work.

II. Motivation and Related work

A. Motivation

To analyze the challenges of the configuration for complex cloud services in multi-cloud environments, let us consider a Machine-to-Machine Data as a Service (M2MDaaS) whose structure is depicted in Figure 1. In this service, serveral *Gateways* receive sensoring data from numerous *Sensors*, and then aggregate and send the data to the *M2MDaaS* which includes several services. The *Event Processing* analyzes the data and stores the results into the *Data End*. The data can be transferred via a *Message Oriented Middleware* to guarantee the performance. These service units are hosted on virtual machines across different cloud systems with various APIs.

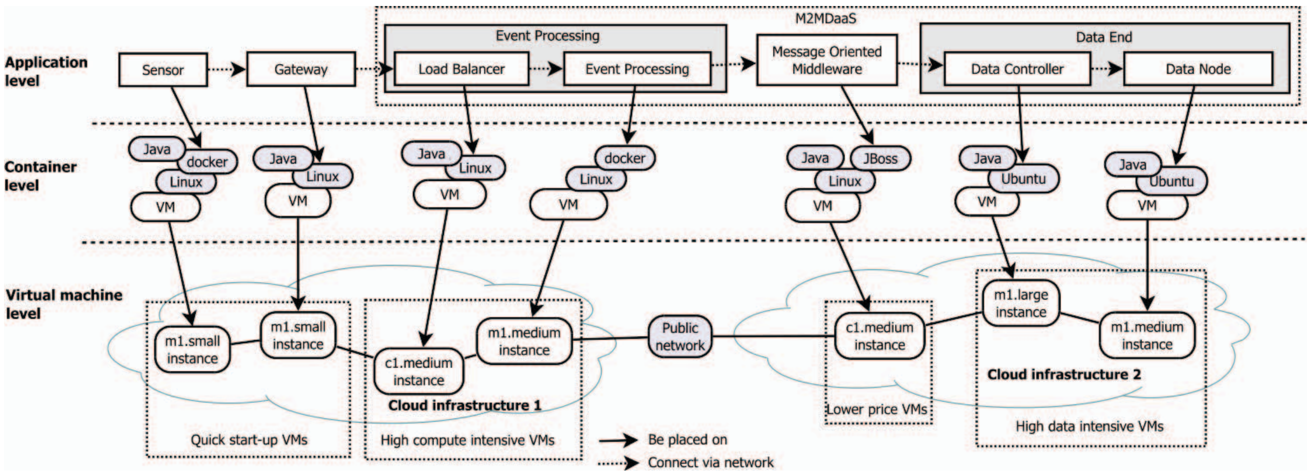


Fig. 1: Illustrative configuration complexity for the M2MDaaS service

Gateways, Sensors and M2MDaaS need to be deployed to several types of cloud resources provided by different IaaSs. This clearly requires various techniques to deploy and configure different types of software artifacts (e.g., executable sensors, lightweight gateway platforms, and heavyweight cloud service units). At runtime, data generated from sensors can change over time due to various conditions, such as the need to increase the number of sensor instances and the frequency of sensor measurements. This requires us to dynamically (re)configure and (re)deploy some software, including cloud service units, in different clouds.

For such scenarios, we need solutions for automatic configuring complex cloud services at multiple levels of abstraction that can work with different cloud infrastructures:

- Deploying diverse types of cloud services on heterogeneous environments requires extensible configuration functionalities that can interface to different underlying technologies and infrastructures.
- Configuring cloud services in different levels, such as executables, OS containers, service containers, and Web services, requires an extensive knowledge management about configuration capabilities, dependency analysis, and how to enact these capabilities at runtime.

Such solutions are needed not only for simplifying the deployment and configuration effort of the user (software developer and provider), but also for automatic software operation management tools (such as elasticity controllers [3], [4]). Although several tools and techniques have been developed for cloud deployment (see them in Section II-B), they are mostly developed for the end user and for single software stacks in single cloud environments. Therefore, we need a framework to support the configuration of the whole cloud service at both the deployment time and runtime, on multi-cloud systems and at different levels. Essential requirements for this framework are: (i) a novel model to represent configuration requirements of complex services, (ii) techniques to centrally manage configurations, (iii) integration with existing tools for executing complex configuration tasks in different environments.

B. Related work

The challenge of managing component diversity is addressed by a number of configuration management tools, e.g. Chef [1], CFEngine [11], Puppet [6]. TOSCA [12], CloudMF [13] and c-Eclipse [14] are the advanced tools that support the description and configuration of complex topologies. Oracle Virtual Assembly Builder [15] simplifies the configuration of multi-layer applications by packing components into VM appliances. Antonis et al. in [16] presented an architecture to capture the application deployment lifecycle. However, these works require low level information for all the components that take a lot of user's efforts and reduce the portability of cloud services.

The challenge of configuration dynamicity is dealing with the provisioning of configuration functionalities at runtime. Rui Han et al. in [7] introduce a platform for dynamic deployment and scaling cloud applications. Calheiros et al. in [17] introduce a framework that provisions resources from different sources and supports different application models. These solutions limit to the cloud resources level. In our work, by integrating other tools, we can configure cloud services dynamically at runtime, and not only at the cloud resource but also at application levels.

Krzysztof et al. in [18] introduce an approach of incorporating Domain Specific Languages(DSL) into the process of developing and deploying applications. Meriem et al. in [19] propose a deployment model to manage the dependencies for software components for the deployment. Binz et al. in [12] show approach to use TOSCA to resolve the dependencies of the cloud service. These models only support dependencies at the deployment time while our solution aims to resolve the dependencies also at runtime.

The challenges of orchestration of application topology have been observed in a number of tools, such as Juju [20], SlipStream [2], Brooklyn [21] which support the configuration of the application topology at the virtual machine level and support user to perform the configuration of all software stacks via scripts or their languages. Jacek Cala et al. in [9] introduce

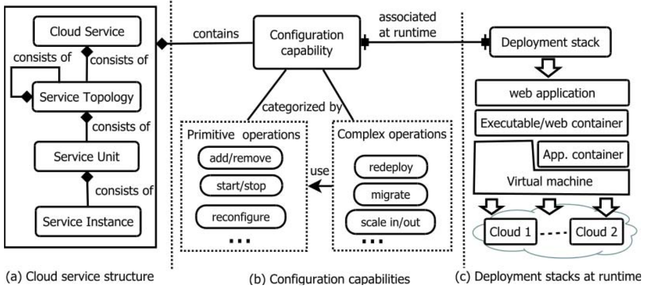


Fig. 2: The complexity of multi application levels and multi deployment stacks

an automatic deployment framework for Azure cloud platform. Gideon Juve et al. in [8] discussed about a deployment service which is able to handle complex dependencies and supported multiple clouds. Caballer et al. in [22] present a platform for dynamic management of virtual infrastructure using Ansible [23]. These solutions aimed to orchestrate virtual machine environments, while our solution aims to give fine-grained orchestration on different levels.

III. Multi-level Configuration Information of Cloud Services

A. Configuration capabilities at multiple application levels

To understand the need to support multi-level configuration of cloud services, we need to examine the complex structure of today's cloud services. As presented in [3], a cloud service can be decomposed into services topologies and service units (Figure 2(a)). Service units represent individual software or cloud offering services, and can be grouped in a service topology for semantically connecting and to enable more complex configuration tasks. A service instance represents a running service unit with its associated runtime information. This generic model can be used to represent different kinds of cloud applications/systems.

In order to support the configuration of such cloud services, each level of the cloud service must be able to expose *configuration capabilities* that indicate the type of available operations at the service deployment time or at runtime, which can affect the properties of that service. Figure 2(b) shows the categories of configuration operations as follows:

- The `primitive operations` abstract the core configuration actions of service units. Depending on the deployment stacks, the primitive operations are enforced by different methods, e.g. virtual machine can be configured via cloud providers' APIs, executables and libraries can be configured via scripts.
- The `complex operations` represent the composition of multiple primitive operations or other complex operations, that allows the service units to cooperate together. This enables complex configuration tasks, but still keeps the loosely-coupled relationships between these service units. For example, to migrate a web server from one cloud provider to another, we not only need to create new web server and move the data to the new place, but also allocate new VMs and configure the network for them.

Being able to capture these configuration capabilities at different software levels will enable a higher granularity of configuration, where users can manage their cloud services in a finer-grained manner.

B. Deployment relationships at multiple software stacks

For the above-mentioned cloud services, we need to support multiple deployment stacks, providing loosely-coupled configuration capabilities for different types of service units and artifacts. Thus, we can provision dependent services by an independent manner. We classified the deployment stacks that reflect the dependencies and mechanisms for configuring these services. Figure 2(c) shows the stacks we defined in our framework:

- `Virtual machine` (VM) provides an environment for running software components. This stack is provisioned by IaaS providers.
- `Application container` is an application that provides a generic environment for running other applications, such as Docker [24], Vagrant [25], Karaf [26].
- `Web container` provides an environment for running web applications, such as Tomcat [27], JBoss [28].
- `Application` represents the applications which run on top of a virtual machine or a container, for example a web service or a Java executable application.

Different parts of the cloud service structure need to be configured to work properly, which are represented by relationships between service units. As the dependencies can be varied, we distinguish among the following relationship types:

- The `vertical relationships` captures the relationships among a unit and other units hosted by that unit. The hosting instance must be ready before deploying dependent units, e.g. a web service and a web container.
- The `horizontal relationships` show that a service unit is needed for another to operate properly, for example a web service which needs a load balancer for directing requests. The service units with horizontal relationship usually connect to each others via network.
- The `local relationships` among units show that we need to deploy those service units on the same hosting service instance, such as two services run on a same VM.

These categories allow us to manage different types of software in more structured way during runtime.

C. Capturing configuration capabilities and software dependencies

In order to configure these diverse types of services with various types of operations, we need to capture the configuration capabilities and provide a unified way to access them. In Figure 3, we capture the configuration capabilities on multiple service levels. Moreover, the association among service units can be represented by the relationships between their configuration capabilities. For example, a web container

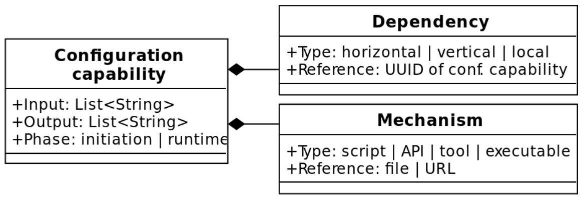

Configuration capability
+Input: List<String>
+Output: List<String>
+Phase: initiation | runtime
Dependency
+Type: horizontal | vertical | local
+Reference: UUID of conf. capability
Mechanism
+Type: script | API | tool | executable
+Reference: file | URL

Fig. 3: Model of configuration capabilities

has a capability to increase its memory resources, which means we need the hosting VM to extend its RAM. At the service topology level, configuration capabilities reflect the interactions between service units that provide complex operations. For example, to scale out a master-slave topology, we need to capture the capabilities of deploying new slave service instances and the *add-new-slave* operations from the master.

The configuration capabilities can be captured via a service registry or by users who specify the service. The model in Figure 3 can be interfaced with different service registries, then an implementation for performing the configuration can be referred depending on the mechanisms or types of the service. In the case that multiple mechanisms are available, we can select the appropriate one by the inputs, e.g., for a more secure configuration, a specific mechanism is applied. To associate different configuration capabilities, each of them has a list of dependencies, which link to other configuration capabilities of other services.

IV. Runtime service configuration in multi-cloud environments

Based on inputs containing cloud service structures, configuration capabilities and underlying cloud software infrastructures, at runtime, for dynamic configuration of cloud services, several activities are performed. Figure 4 shows the detailed flows of configuring different stacks. The configuration capabilities can be captured at the cloud service and service topology level, which includes multiple service unit configuration tasks. For each service unit, a service unit orchestrator is generated for handling the task. We separate the configuration process for VM level from that for other software levels. When creating a new VM, a bootstrap script and a client are run to configure the environment and respectively to support the configuration of higher software levels. In the following, we will discuss some steps in Figure 4.

A. Generating the deployment topology

Users, when describing their cloud service, do not know the complex details of the whole service at runtime. For each service unit, users define (i) artifacts for the configuration capabilities and (ii) configuration parameters. However, this description still misses the information regarding where to deploy the service units, and how to configure the dependencies, or what is the optimal configuration of cloud resources. For example, when we need a library which requires a specific VM, the information of suitable VM image and VM type need to be generated.

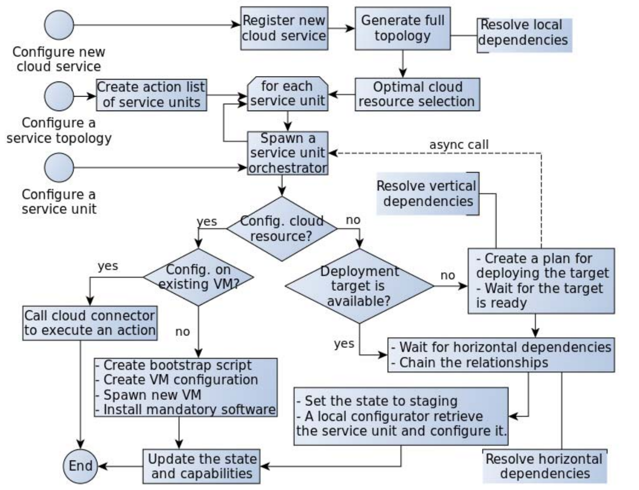

Configure new cloud service
Register new cloud service
Generate full topology
Resolve local dependencies
Configure a service topology
Create action list of service units
for each service unit
Optimal cloud resource selection
Spawn a service unit orchestrator
async call
Configure a service unit
Resolve vertical dependencies
Config. cloud resource?
yes
no
Config. on existing VM?
yes
no
Deployment target is available?
no
yes
- Create a plan for deploying the target
- Wait for the target is ready
Call cloud connector to execute an action
- Create bootstrap script
- Create VM configuration
- Spawn new VM
- Install mandatory software
- Wait for horizontal dependencies
- Chain the relationships
- Set the state to staging
- A local configurator retrieve the service unit and configure it.
End
Update the state and capabilities
Resolve horizontal dependencies

Fig. 4: General flow of configuring cloud services

The heterogeneous service unit types and relationship types in multi-cloud environments bring challenges in resolving dependencies and require a base representational model to deal with the problem. Studies in [12], [13], [19], [29] proposed service unit models for resolving dependencies for cloud services. These models focus on relationships between service units at the deployment time but not during runtime. In our solution, we associate the *configuration capability model* (Figure 3) to the *service unit model*, so that relationships between service units can be determined by the dependencies of their configuration capabilities.

Figure 5 illustrates the process of generating configuration information for a service unit. The meta information contains abstract nodes which define generic service unit types, which can have several implementation nodes for particular artifacts. In order to perform this process, we based on two analyses:

- Generate information of service units with appropriate configuration capabilities. For example, the *Increase Heap* capability requires the *VM scale up* capability, which can be enforced by Flexiant VM.
- Generate information of service units for performing the configuration capabilities. For example, the mechanism of deploying the *Service unit A* requires some dependency packages, and a VM to hosted all of them.

Due to the fact that the dependencies are generated for each configuration capability, the configuration detail of cloud services contains information for both deployment time and runtime. By this, not only the deployment but also the runtime configuration can de done without changing the environment.

B. Orchestrating service configurations

At the deployment time, the configuration orchestration initiates the service topology on different cloud resources. At runtime, it connects the newly-deployed service units with the already running cloud service units. To deal with the configuration of complex topologies, some studies considered the performance of cloud services, e.g. [8], [30], [31]. In

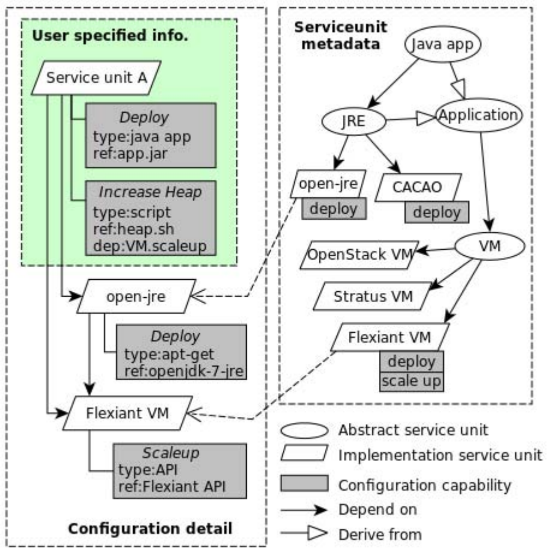


Fig. 5: Example of the generation of one service unit

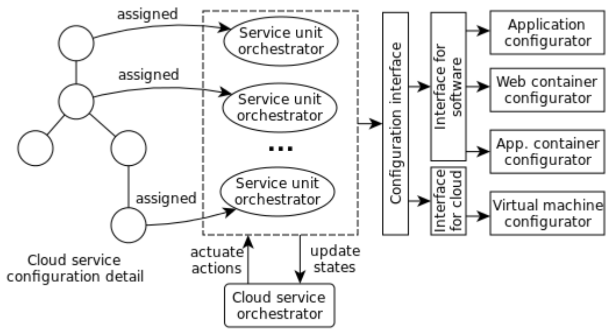

assigned
assigned
assigned
Service unit orchestrator
Service unit orchestrator
...
Service unit orchestrator
Cloud service configuration detail
actuate actions
update states
Cloud service orchestrator
Configuration interface
Interface for software
Interface for cloud
Application configurator
Web container configurator
App. container configurator
Virtual machine configurator

Fig. 6: Greedy orchestration process

our work, we build a generic mechanism based on a greedy approach to support finer-grained orchestration processes. In the future, better orchestration algorithms can be designed based on this and embedded within our framework.

Our greedy mechanism uses the cooperation of one cloud service orchestrator and multiple service unit orchestrators (Figure 6). From the generated cloud service configuration (in Section IV-A), we assigned each service unit to a service unit orchestrator which runs independently and interacts with the cloud service orchestrator. Each service unit orchestrator is aware of the conditions for the configuration operation and performing the assigned actions. The cloud service orchestrator manages service unit states and maintains a shared information space containing service units configuration information. By updating states and receiving commands from the cloud service orchestrator, service unit orchestrators can individually perform the tasks in the correct order.

C. Placing service units at runtime

To support the high level of configuration, we consider that users will not care about where to deploy and how to configure

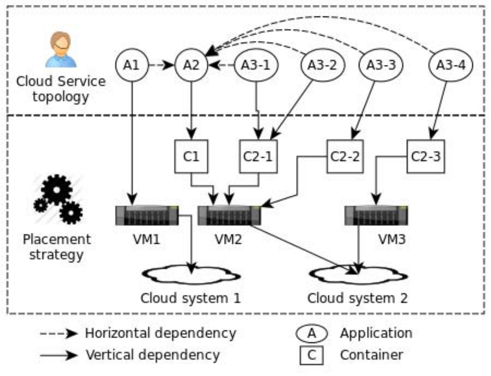


Fig. 7: Illustration of placement cases

service units. In many cases, a set of service units can be deployed on top of another service instance for sharing the resources and runtime environment. A placement strategy is used to decide where to deploy a new service units on the running cloud service or cloud system. Our framework allows to plug-in different placement strategies based on different deployment stacks. While the placement problem on top of cloud systems and on top of VMs are addressed by many studies [32]–[34], our framework integrates different strategies on different deployment stacks.

In our work, we use two mechanisms for the placement. First, we specify the maximum number of service instances can be hosted by another service instance, for example one VM can host multiple java applications but only one web server. Second, we specify thresholds for resources of the VM and use a monitoring system to evaluate the resource usage if it is over the threshold in order to decide the placement. On a new deployment, a service unit orchestrator (Section IV-B) will search for an available hosting service instance that still is able to host a new instance of the assigned service unit. If no suitable hosting service instance is found, the cloud service orchestrator will trigger a new deployment of the hosting service unit. Figure 7 illustrates the placement during several scaling out operations. The instances of the service unit `A3` are deployed on the application container `C2` and all of them are hosted on VMs across several cloud providers. At runtime, we can decide either deploy the `A3` on an existing container or on a new one depending the status of existing containers.

D. Managing configuration states

The cloud service configuration orchestrator needs to know the state of configuration tasks to trigger appropriate actions and manage service unit orchestrators. To manage the configuration state at multiple levels, we use a state aggregation algorithm to retrieve the states on the cloud service structure and deployment stacks. We identify the following configuration states: error(1), allocating(2), staging(3), configuring(4), deployed(5) and running(6). The state of a service instance is the progress of its configuration task. For the state aggregation, the states of a service will be the smallest state of services on lower level and stack. For example, a service unit has a service

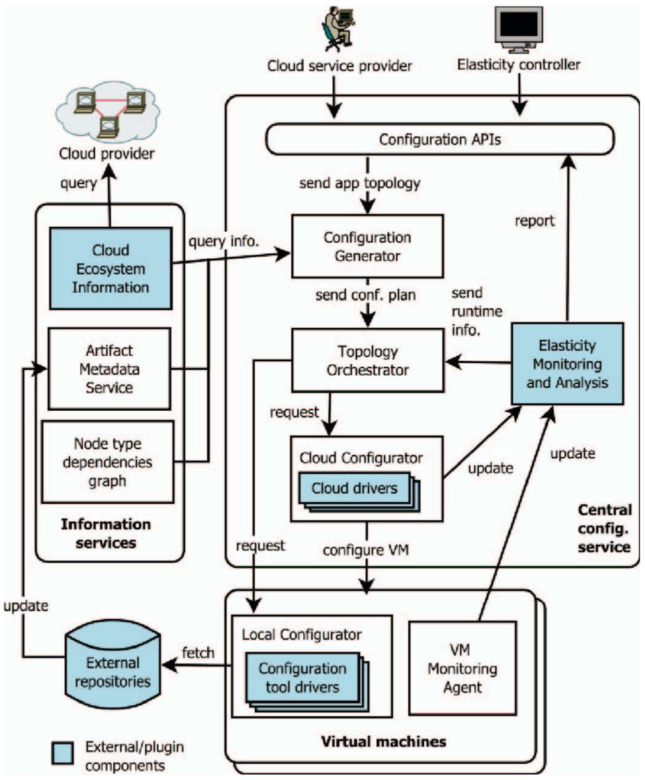


Fig. 8: SALSA's architecture

instance is in *configuring* state and the others are in *running* state, the whole service unit will be in the *allocating* state. If that service instance are hosted on another, the state of that hosting instance is set to *configuring*.

V. SALSA ARCHITECTURE

A. Overview

We have implemented SALSA[2] to support the multi-level configuration functionalities, of which the architecture is shown in Figure 8. SALSA comprises three main building blocks: (i) The Central Configuration Service is for orchestrating the configuration operations, (ii) Local Configurators perform tasks on top of deployment stacks by following the instruction from the Central Configuration Service, and (iii) the Information Services maintain the knowledge for generating the configuration plan and mapping with the external artifact repositories.

Users interact with the framework via *Configuration APIs*. To enrich configuration information, the *Configuration Generator* queries the *Information Services* to generate the full configuration details. As described in Section IV-A, it generates new nodes from the *Node Type Dependency Graph*, then adds the configuration capabilities and artifacts for software from *Artifact Metadata Service* and cloud resources form the *Cloud Ecosystem Information*. The artifact meta-information refers to real artifacts which are stored in *External Repositories* (e.g GitHub[3] and Chef Community[4]).

The *Topology Orchestrator* orchestrates configurations of multiple service units and service topologies using the greedy mechanism presented in Section IV-B. The *Cloud Configurators* and *Local Configurators* utilize different tools in order to deal with different cloud providers and application types. At the bootstrapping of a new spawned VM, we use cloud-init [35] to start the Local Configurator. The *VM Monitoring Agent* monitors different stacks of cloud services, which is integrated with the *Elasticity Monitoring and Analysis* [36].

B. Integration with multiple cloud infrastructures

Beside the core components, SALSA uses external tools and services to perform the actual configuration operations, increasing the extendability of the framework by exporting the configuration actions to different output formats and call the external tools.

The *Cloud Configurator* allows SALSA to plug in different cloud drivers for connecting to different cloud infrastructures. These cloud drivers also map the cloud specific APIs to the service unit's configuration capabilities depending to the services of the providers. We implemented the Openstack connector using JClouds [37] and some specific cloud APIs from cloud providers like StratusLab[5] and Flexiant[6].

The *Local Configurator* wraps the functionalities of different software configuration tools and maps them on the configuration capabilities of service units. By manipulating the VM environment, the Local Configurator uses the existing local package management tools (e.g. apt-get, yum, gem, pip) or setups and uses higher level tools (e.g. Chef). We also implemented a plug-in mechanism that allows the Local Configurator to work in different infrastructures. Depending on the nature of the tools, we have different ways to fetch the artifacts for them from External Repositories.

VI. ILLUSTRATING EXAMPLES AND EXPERIMENTS

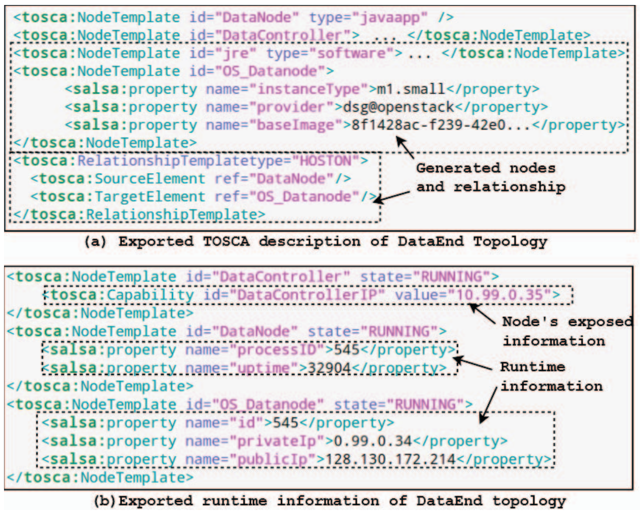
We use the M2MDaaS service described in Section II-A to show how SALSA is used to provide fine-grained configuration capabilities. We use following the cloud systems: (i) Our private cloud with OpenStack[7], (ii) the LAL site of StratusLab and (iii) the Flexiant public cloud infrastructure.

A. Simplifying complex configuration management

For starting a configuration process of a complex cloud service, the service provider defines the cloud service topology and its artifacts. We use TOSCA [12] to specify cloud service structures as it supports to define topology and different node types. Figure 9 shows an example of a *TOSCA-based* cloud service structure information. In this description, the

[2]Prototype and supplement materials: http://tuwiendsg.github.io/SALSA

[3]GitHub. https://github.com/
[4]Chef Community. http://www.getchef.com/community/
[5]StratusLab. http://stratuslab.eu
[6]Flexiant. http://www.flexiant.com
[7]OpenStack. http://www.openstack.org/



```
<tosca:NodeTemplate id="DataNode" type="javaapp" />
<tosca:NodeTemplate id="DataController"> ... </tosca:NodeTemplate>
<tosca:NodeTemplate id="jre" type="software"> ... </tosca:NodeTemplate>
<tosca:NodeTemplate id="OS_Datanode">
      <salsa:property name="instanceType">m1.small</property>
      <salsa:property name="provider">dsg@openstack</property>
      <salsa:property name="baseImage">8f1428ac-f239-42e0...</property>
</tosca:NodeTemplate>
<tosca:RelationshipTemplatetype="HOSTON">
  <tosca:SourceElement ref="DataNode"/>
  <tosca:TargetElement ref="OS_Datanode"/>
</tosca:RelationshipTemplate>
```

Generated nodes
and relationship
(a) Exported TOSCA description of DataEnd Topology

```
<tosca:NodeTemplate id="DataController" state="RUNNING">
    <tosca:Capability id="DataControllerIP" value="10.99.0.35">
</tosca:NodeTemplate>
<tosca:NodeTemplate id="DataNode" state="RUNNING">
    <salsa:property name="processID">545</property>
    <salsa:property name="uptime">32904</property>
</tosca:NodeTemplate>
<tosca:NodeTemplate id="OS_Datanode" state="RUNNING">
    <salsa:property name="id">545</property>
    <salsa:property name="privateIp">0.99.0.34</property>
    <salsa:property name="publicIp">128.130.172.214</property>
</tosca:NodeTemplate>
```

Node's exposed
information
Runtime
information
(b)Exported runtime information of DataEnd topology

Fig. 9: Example of input and runtime information in TOSCA

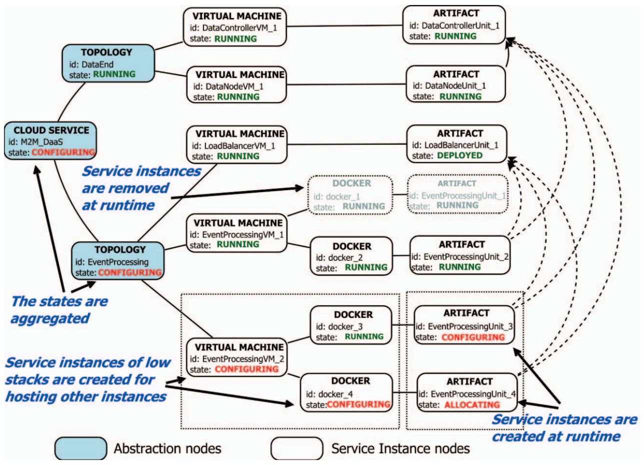


Fig. 10: Screenshot of configuration of the DaaS service

user defined two node templates: the *DataController* and the *DataNode*, which represent two service units.

To be able to deploy and configure the above cloud service, SALSA analyzed the user input, queried the missing configuration from the Information Services and produced a complete *service configuration*. For example, in order to host and run the *Data Node*, SALSA generated the dependency nodes, such as nodes for jre and Cassandra[8] packages, then a VM node which includes the provider name, the VM type and a base image (Figure 9(a)). The relationships connecting new nodes are generated. The configuration of the VM is determined by analyzing the properties of all the software on top of it. Some service units can expose their properties to be used during the configuration of other service units, such as the Data Controller exposed its IP (Figure 9(b)).

At runtime, service providers can interact with the cloud services via a user interface (shown in Figure 10) by triggering their configuration capabilities. The user interface also shows runtime information of service instances and the states of configuration operations.

[8]Cassandra. http://cassandra.apache.org/

B. *Dealing with configuration in multiple clouds*

1) Multiple clouds settings: When configuring cloud services on multi-cloud systems, we face the problem of heterogeneous environments, which limits the ability of configuring one artifact over multiple clouds. We examined several ways to configure an artifact: (i) one package that includes all dependencies and can only be deployed on a specific environment, (ii) a script that can be executed on any version of the same operating system, and (iii) a description that is defined using the model in Section III-C which SALSA can support to generate dependencies at deployment time (A, B and C in Table I).

Table I shows a comparison between these configuration possibilities. SALSA supports in choosing the low level dependencies regardless of the cloud environments, which will increase the options in configuring one service in heterogeneous cloud environments.

2) Evaluating configuration time of multiple service units: We use SALSA to configure a number of the sensor clients of M2MDaaS (Figure 1) on two cloud systems: (i) our private cloud of OpenStack with *m1.small* VM type (1 CPU, 2GB RAM) and (ii) Flexiant with *small* VM type (1 CPU, 1GB RAM). Each VM is set to host maximum 30 sensor clients. By using SALSA, we performed two tests. In the first test, we measure the configuration time of the whole stacks including spawning VMs, configuring environment and execute the sensor client artifacts. In the second test, we undeployed the sensor clients from the first test and keep the VMs running, then deployed the same number of sensor clients on top of existing VMs. We tested both cases with incremental number of sensors which the number of VMs is also increased. We defined the number of sensors to be deployed and undeployed, and SALSA determined number of VMs automatically.

Figure 11 shows the result of these tests. Because each test was executed on multiple VMs, we annotated the chart with the times of the last finished VM. Obviously, configuring only the application stack is faster than the whole stacks. While adding and removing VMs for dynamic services is slower, SALSA reserved the running VMs for the new sensors to be deployed later. Thus, the time for provisioning dynamic services is reduced.

We also notice that the number of instances have more impact on the private Openstack cloud and less on the Flexiant cloud. For the case of the configuration of the whole stacks, our OpenStack private cloud took a longer time to deploy more sensors because it needed more time to spawn VMs simultaneously, while Flexiant was better in spawning multiple VMs at a same time. In the second case, the time to deploy different number of sensors are more fixed as all the VMs received the same workload of 30 sensors. The time for OpenStack slightly increased due to the high workload of SALSA orchestration process, which did not happen for Flexiant as the Flexiant VMs completed tasks slower. Although the configuration time

Images		Provider's available images			Configuration possibilities		
OS	Ver.	(1)	(2)	(3)	(A)	(B)	(C)
Ubuntu	14.04	Yes	Yes	Yes	1	12	
	13.04	Yes	-	-			
	12.10	Yes	-	Yes			
	12.04	-	Yes	-			
Centos	6.5	Yes	Yes	Yes	1	8	
	6.3	-	-	Yes			
	6.2	-	Yes	-			
	6.0	Yes	-	-			
Total number of configurations					2	20	80

(1) OpenStack (2) Flexiant (3) StratusLab
(A) Package (B) Script (C) SALSA input

TABLE I: Comparison of config. possibilities on multiple clouds

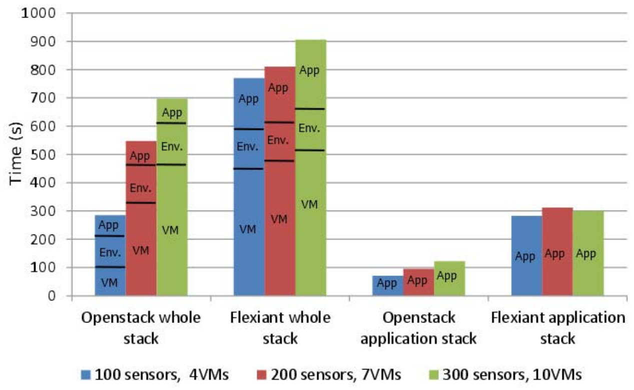


Fig. 11: Time for provisioning sensor clients

of Flexiant is longer, the stability shows that this cloud system is more suitable for IoT applications.

VII. Conclusions and Future Work

In this paper, we introduced a framework to support the automatic configuration of complex cloud services on multi-cloud environments. We show that our SALSA framework can simplify the complexity of multi-level configurations and deployments for cloud services.

In our future work, we will extend our framework with further optimization features. First, it is important that we improve the placement strategy for discovering suitable deployment targets. Second, we will be optimizing cloud resources selection at runtime. In order to achieve those, we need to analyze runtime information of cloud services and integrating some optimization algorithms to SALSA.

References

[1] Chef. http://www.getchef.com/.

[2] SlipStream. http://sixsq.com/products/slipstream.html.

[3] G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar, "Multi-level elasticity control of cloud services," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, S. Basu, C. Pautasso, L. Zhang, and X. Fu, Eds. Springer Berlin Heidelberg, 2013, vol. 8274, pp. 429–436.

[4] R. Han, L. Guo, M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2012*, May 2012, pp. 644–651.

[5] Google Cloud Deployment Manager. http://goo.gl/vksPCP.

[6] Puppet. http://puppetlabs.com/.

[7] R. Han, L. Guo, Y. Guo, and S. He, "A deployment platform for dynamically scaling applications in the cloud," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, Nov 2011, pp. 506–510.

[8] G. Juve and E. Deelman, "Automating application deployment in infrastructure clouds," *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pp. 658–665, 2011.

[9] P. W. Jacek Caa, "Automatic software deployment in the azure cloud," in *Distributed Applications and Interoperable Systems*, Amsterdam,Sumtech The Netherlands, Jun. 2010, pp. 155–168.

[10] H.-E. Yu, Y.-L. Pan, C.-H. Wu, H.-S. Chen, C.-M. Chen, and K.-Y. Cheng, "On-demand automated fast deployment and coordinated cloud services," in *5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2013)*, 2013.

[11] CFEngine. http://cfengine.com.

[12] T. Binz, G. Breiter, F. Leyman, and T. Spatzier, "Portable cloud services using tosca," *Internet Computing, IEEE*, vol. 16, pp. 80–85, May 2012.

[13] N. Ferry, F. Chauvel, A. Rossini, B. Morin, and A. Solberg, "Managing multi-cloud systems with CloudMF," in *Proceedings of the Second Nordic Symposium on Cloud Computing, 2013, Internet Technologies*, ser. NordiCloud '13. New York, NY, USA: ACM, 2013, pp. 38–45.

[14] C. Sofokleous, N. Loulloudes, D. Trihinas, and G. P. M. Dikaiakos, "c-eclipse: An open-source management framework for cloud applications," in *Europar*, 2014.

[15] Oracle Virtual Assembly Builder. http://goo.gl/neRgdg.

[16] K. M. Antonis Papaioannou, "An architecture for evaluating distributed application deployments in multi-clouds," in *5th IEEE International Conference on Cloud Computing Technology and Science, 2013*, 2013.

[17] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya, "The Aneka platform and qos-driven resource provisioning for elastic applications on hybrid clouds," *Future Gener. Comput. Syst.*, vol. 28, no. 6, pp. 861–870, Jun. 2012.

[18] K. Sledziewski, B. Bordbar, and R. Anane, "A dsl-based approach to software development and deployment on cloud," in *Advanced Information Networking and Applications (AINA)*, Apr. 2010, pp. 414–421.

[19] M. Belguidoum and F. Dagnat, "Dependability in software component deployment," in *2nd International Conference on Dependability of Computer Systems*, ENST Bretagne, Brittany, Jun. 2007, pp. 223–230.

[20] Juju. https://juju.ubuntu.com/.

[21] Brooklyn. http://brooklyncentral.github.io.

[22] M. Caballer, I. Blanquer, G. Molt, and C. de Alfonso, "Dynamic management of virtual infrastructures," *Journal of Grid Computing*, pp. 1–18, 2014.

[23] Ansible. http://www.ansible.com.

[24] Docker. https://www.docker.com/.

[25] Vagrant. http://www.vagrantup.com/.

[26] Karaf. http://karaf.apache.org/.

[27] Tomcat. http://tomcat.apache.org/.

[28] JBoss. http://jbossas.jboss.org/.

[29] N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg, "Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems," in *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, ser. CLOUD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 887–894.

[30] A.-F. Antonescu, P. Robinson, and T. Braun, "Dynamic topology orchestration for distributed cloud-based applications," in *Second Symposium on Network Cloud Computing and Applications (NCCA), 2012*, Dec 2012, pp. 116–123.

[31] H. Kim, Y. el Khamra, I. Rodero, S. Jha, and M. Parashar, "Autonomic management of application workflows on hybrid computing infrastructure," *Sci. Program.*, vol. 19, no. 2-3, pp. 75–89, Apr. 2011.

[32] F. Diaz, S. A. Zahr, and M. Gagnaire, "An exact placement approach for optimizing cost and recovery time under faulty multi-cloud environments," in *5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2013)*, 2013.

[33] F. Chang, R. Viswanathan, and T. Wood, "Placement in clouds for application-level latency requirements," in *Cloud Computing, 2012 IEEE 5th International Conference on*, 2012, pp. 327–335.

[34] F. Charrada, N. Tebourski, S. Tata, and S. Moalla, "Approximate placement of service-based applications in hybrid clouds," in *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2012 IEEE 21st International Workshop on*, June 2012, pp. 161–166.

[35] Cloud-Init. http://cloudinit.readthedocs.org/.

[36] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar, "Mela: Monitoring and analyzing elasticity of cloud services," in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, vol. 1, Dec 2013, pp. 80–87.

[37] JClouds. https://jclouds.apache.org.