Mantus: Putting Aspects to Work for Flexible Multi-Cloud Deployment

Alex Palesandro*§ Marc Lacoste* Nadia Bennani † Chirine Ghedira Guegan ‡ Denis Bourge*
* Orange Labs, France
{alex.palesandro, marc.lacoste, denis.bourge}@orange.com
† Université de Lyon, CNRS, INSA-Lyon, LIRIS, UMR5205, F-69621, France
nadia.bennani@insa-lyon.fr
‡ Université de Lyon, CNRS, IAE - Université Lyon 3, LIRIS, UMR5205, France,
chirine.ghedira-guegan@univ-lyon3.fr
§ Magellan, IAE, Université Lyon 3, France

***Abstract*—Cloud provider barriers still stand. After a decade of cloud computing, customers struggle to overcome the challenge of crossing multi-provider clouds to benefit from fine-grained resource distribution, business independence from CSPs and cost savings. Although increasingly popular, most adopted IaaS intercloud solutions are generally limited to specific public cloud providers or present maintainability issues. Remaining hurdles include complexity of management and operations of such infrastructures, in presence of per-customer customizations and provider configurations. The Infrastructure as Code (IaC) paradigm is emerging as key enabler for IaaS multi-clouds, to develop and manage infrastructure configurations. However, due to complexity of the infrastructure life-cycle, to heterogeneity of composing resources and to user-customizations, this approach is far from being viable. In this paper, we explore an aspect-oriented approach to IaC deployment and management. We propose Mantus, a IaC-based multi-cloud builder composed of an aspect-oriented Domain-Specific Language called TML, or TOSCA Manipulation Language, and a corresponding aspect weaver to inject flexibly non-functional services in TOSCA infrastructure templates. We show the practical feasibility of our approach, with also good results in terms of performance and scalability.**

I. Introduction

After a decade of existence, cloud infrastructures are the omnipresent foundation of IT services. Cloud consumers may now orchestrate fine-grained, easy-to-scale, micro-service-built applications in a homogeneous domain. This may also include the network edge, overcoming the geographical barrier of data centers. However, cloud consumers seldom accept – and only when strictly needed – the challenge of crossing Cloud Service Provider (CSP) domains [1].

Interconnection of multiple provider resources promises important benefits [1, 2] compared to the single-CSP model: (1) *finer-grained distribution of resources* across multiple countries, improving QoS; (2) *business independence from a single CSP*; and (3) *cost savings*, optimizing expenditures through dynamic price comparisons between providers (e.g., for Spot instances).

Unfortunately, it seems that no current generic approach really fully satisfies customer requirements [1]. Provider-centric architectures (e.g., hybrid clouds, cloud federations) have the great benefit of resource location transparency from the user perspective but lack effective market-adoption by providers [3]. Hybrid clouds [4, 5] provide a seamless extension of private clouds but are limited to a single CSP. Client-centric approaches (e.g., library-based multi-clouds) are gaining popularity [6, 7]. However, they remain limited to a least common denominator of CSP features, without providing full control over the infrastructure, with also some maintainability issues [1]. In addition, broker-oriented approaches [8] have not gained the expected popularity so far [9].

To address those challenges, we designed ORBITS (ORchestration for Beyond InTercloud Security), an overlay intercloud architecture providing simultaneously flexible application provisioning across multiple providers with a homogeneous service abstraction across multiple clouds enforced at the IaaS level [10]. ORBITS conciliates: (1) flexible provisioning requirements of microservice-based applications, handling placement, elasticity and availability; and (2) infrastructure homogeneity over generic CSPs through the Infrastructure as Code (IaC) paradigm [11]. IaC enables to develop and manage infrastructure configurations, notably for cloud infrastructures, automating resource provisioning in addition to software deployment.

However IaC-based deployment and management of a generic IaaS multi-cloud requires to overcome a number of roadblocks. Traditional infrastructure user-customization should notably be coupled with the heterogeneity of underlying providers. This requires to flexibly (1) inject/remove non-functional services, (2) reduce at a minimum the multi-provider overhead and (3) orchestrate in an interoperable manner resources over multiple CSPs.

Introduced in [12], Aspect-Oriented Programming aims to regroup in a single place code fragments implementing a non-functional property using the aspect programming abstraction. Leveraging a "weawing" process, the aspect code will be scattered across the overall program at compilation or execution time. We adapt this approach to declarative-based Infrastructure-as-Code systems, in order to flexibly and dynamically inject non-functional services, decoupling those from the rest of the system.

In this paper, we present Mantus, an aspect-oriented

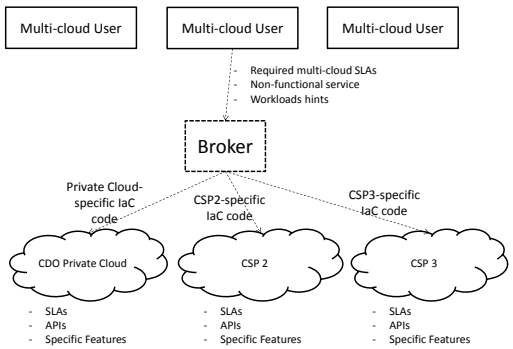

Fig. 1: Cloudbursting Use-Case

multi-cloud builder, designed to generate the IaC template to deploy an overlay infrastructure-oriented multi-cloud. At a cost of an acceptable overhead, aspect-oriented weaving separates cross-cutting concerns, fostering reuse of extra code and infrastructure life-cycle automation.

The contributions of this paper are as follows. First, we propose an aspect-oriented approach to IaC multi-cloud deployment and management. Second, we propose the TML (TOSCA Manipulation Language) aspect specification language and the MANTUS aspect weaver to realize an aspect-oriented IaC multi-cloud builder, injecting flexibly non-functional services in TOSCA infrastructure templates. Third, we validate a first prototype in terms of performance, scalability, and dynamicity, showing the practical feasibility of our design approach.

This paper is organized as follows. Section II first analyzes requirements for an IaC multi-cloud, and identifies a number of key guiding principles. Section III then introduces the aspect-oriented IaC approach, and presents TML, with some examples. Sections IV and V describe a prototype implementation of the aspect weaver and validation results. Section VI discusses related work. Finally, Section VII concludes.

II. TOWARDS IAC-BASED MULTI-CLOUDS

We start from a sample use-case to motivate the need for IaC-based multi-clouds. We then detail requirements and key design principles to realize an IaC-based multi-cloud builder.

A. Motivating Use-Case Example

We consider a generic cloud-bursting use-case, shown in figure 1: an organization wishes to enhance the resources available on its private cloud, embracing simultaneously more than one public cloud provider.

We take the example of a Care-Delivery Organization (CDO) managing different classes of workloads, ranging from telemedicine, e.g. remote healing, to data mining on sensitive data, e.g., electronic health records (EHR). Cloud adoption may bring some benefits. However, requirements in terms of Quality of Service (QoS) or Quality of Protection (QoP) may overwhelm the SLA guarantees a single CSP may offer.

On the one hand, telemedicine services have severe constraints hardly satisfied by a single CSP [13]. In particular, CDOs must face stringent law prescriptions for geographical data location, replication and service availability. Therefore, more than one CSP is often needed. On the other hand, a CDO may also need to perform analysis on medical data, possibly using records coming from other institutions. Handling such class of records usually requires a lot of effort to avoid unwanted disclosures, using techniques such as Multi-Party Computation (MPC) [14]. The resource overhead imposed by such protection techniques could benefit from cloud economies of scale. However, a single CSP does not fulfill the security model requirements of MPC algorithms that are based on secret-sharing techniques [15].

B. Design Requirements (DRs)

We already defined a first overlay infrastructure completely managed through IaC templates [10]. This infrastructure extends the idea of a client-centric virtual infrastructure layer [16, 17] for enhanced resource control and multiple CSP support. A full text-based description of software and hardware infrastructure resources is key to control the complete infrastructure life-cycle, which has proven tough to manage even in single cloud provider settings [18]. However, several challenges remain open for this to be possible.

A IaC-based virtual infrastructure layer should meet three requirements:

1) **DR1: Non-functional extensibility** Desired cloud infrastructures may differ a lot depending on the functional services deployed (e.g., Cloud Management System, SDN Controller). However, those basic infrastructure should be manipulated in order to add complementary non-functional services (e.g. monitoring, auditing). This makes easily complexity explode and does not allow to dynamically inject services inside basic templates.
2) **DR2: CSP-aware feature selection** To compete on the market, CSPs provide a number of differentiating services (e.g., DBMS-as-a-Service (DBMSaaS), Firewall-as-a-Service (FWaaS)) additional to traditional resource provisioning (e.g., VMs, Object Storage). CSPs also propose differently-flavored resources (e.g., high I/O VM types, accelerated NICs) to better satisfy specific workloads. To effectively leverage CSP features, the interoperable virtual infrastructure should overcome simple "least common denominator" limitations to adapt instantiations to specific CSPs.
3) **DR3: Multi-layer interoperability** The multi-cloud should also satisfy the following sub-requirements to achieve the desired level of interoperability:
   a) The user should be able to describe its service specification (functional, non-functional, SLA) without knowing in advance on which CSP the multi-cloud will be deployed, and how it will be actually implemented. In addition, the virtual infrastructure layer should be *interoperable*, i.e., the same abstract definition may be deployed on different CSPs, being "understood" by the different CSP orchestration engines.

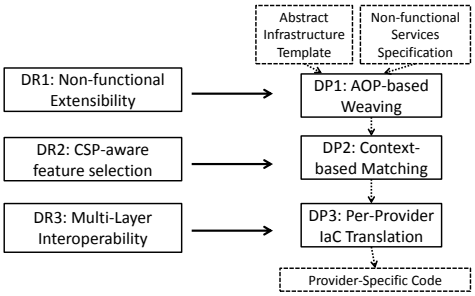

Fig. 2: Mapping Design Principles to Design Requirements

b) The virtual infrastructure layer should guarantee *portability* of Execution Environments (EEs) where service components are executed on any CSP part of the multi-cloud. Deployed services should work similarly and expose the same APIs, requiring little adaptions from existing applications.

c) The architecture should provide *a single point of orchestration* of services (e.g. MPC, High-Availability Web Services) over the distributed multi-cloud infrastructure. This entry-point should be similar to widely-used frameworks (Mesos, Swarm, Kubernetes) that integrate seamless extensions for multi-cloud enabled CSPs.

The deployment result is an *interoperable software layer* over a selected set of CSPs, coordinated by an *orchestration layer* having multi-cloud awareness without dealing with hetereogeneity.

C. Design Principles (DPs)

To meet such requirements, we propose ORBITS, a multi-cloud architecture overcoming limitations of application-specificity of traditional multi-cloud libraries [10]. The ORBITS design is based on the following principles:

1) **DP1: AOP-based weaving** To satisfy the non-functional extensibility requirement, we adopt an Aspect-Oriented Programming (AOP) [12] design approach. AOP is a programming paradigm that increases modularity by separating cross-cutting concerns. Adding features does not require persistent modifications to the base code. Additional behaviors (advices) need to be applied to the existing code, tracing the modifications. This approach is not only applicable to user-specified services but also to services whose aim is to optimize the infrastructure life-cycle management.
2) **DP2: Context-based matching** To satisfy the CSP-aware feature selection requirement, we leverage a flexible form of matching derived from the TOSCA *Substitution Matching*. In TOSCA, the abstract definition of resources is separated from their actual implementations. According to inputs, resources may be matched with a different equivalent implementations of the same service. The best implementation to fit a particular context (e.g., user-specified workload, selected CSP) should be selected. We thus inflected this mechanism to our setting, introducing different forms of matching in TOSCA [19].
3) **DP3: Per-provider IaC translation** To satisfy the interoperability requirement, we defined a model of a cloud infrastructure, capturing the distributed aspect of multi-clouds, and the need of coordination between different services. This model was specified using the TOSCA modelling language [20]. We also implemented a translator to transform a generic TOSCA description into code of a specific cloud provider.

Figure 2 maps DRs to steps of a simple workflow description, showing how DPs handle different DRs. In the sequel, we focus on DP1 and how our approach can satisfy the DR1 requirement, in terms of feasibility, scalability, and dynamicity.

III. Aspect-Oriented Weaving for IaC

We now introduce the principles of aspect-oriented approach for IaC. We start with some background on the TOSCA specification language. We then introduce TML for TOSCA templates.

A. TOSCA Background

The OASIS TOSCA [20] format is designed for providing interoperability between templates of cloud applications, leveraging an object-oriented resources modeling TOSCA templates are composed of a graph of by *nodes_template* instances, which reifies abstract *node_types*. Those "equivalent" classes not only described interactions between different parts of the service but also their life-cycle.

TOSCA nodes are described by a set of 4 classes of attributes: (1) capabilities that they are able to provide to other nodes (*Caps*), (2) requirements that they need to run correctly (*Reqs*), (3) properties (*Props*) and (4) interfaces (*Ints*). In TOSCA, an *interface* models a way to modify the resource dynamically at run-time, supporting the mapping of (5) operations.

Such attributes model the life-cycle of the components and may be provided through concrete artifacts [1].

A TOSCA *ServiceTemplate* is mainly composed of: (1) a TopologyTemplate, a directed graph-oriented definition of services; (2) *NodeTypes*, a list of definitions of nodes composing the *TopologyTemplate*, and (3) *RelationshipTypes*, a list of edge types for the TopologyTemplate, modelling custom links between resources. In addition, a ServiceTemplate usually includes: (4) a set of *BoundaryDefinitions* that specify which capabilities and requirements have to be exported outside of the ServiceTemplate; and (5) Plans, which specify how the node operations should be executed to manage the service life-cycle.

TOSCA fosters reuse of defined components and interoperability through a mechanism of matching abstract NodeTypes with ServiceTemplate implementations. TOSCA Orchestrators match concrete implementations through a *matching process* [19, 20]. The TOSCA templates get interpreted and instantiated in a set of distinct resources called a *topology instance*. Listing 1 presents a sample ServiceTemplate, which

[1] An artifact is a named and typed file used to implement deployment and other interface operations. (e.g., build or configuration scripts).

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: Main demo file
topology_template:
  inputs: # [...] Omitted for brevity

  node_templates:
     controller_node:
      type: orbits.nodes.ComputeBox
      properties:
          name: Controller
          flavor: ml.xl
        # [...] Omitted for brevity
     control_port:
      type: orbits.nodes.network.Port
      properties:
       order: 0
      requirements:
      - binding: controller_node
      - link: control_network
     control_net:
      type: orbits.nodes.network.Network
      # [...] Omitted for brevity
     internal_port:
      type: orbits.nodes.network.Port
      properties:
       order: 1
      requirements:
      - binding: controller_node
      - link: internal_network
     internal_net:
      type: orbits.nodes.network.Network
      # [...] Omitted for brevity
     router:
      type:  orbits.nodes.network.Router
      properties:
       external_network_name: ext-net
     router_interface:
      type: orbits.nodes.network.RouterInterface
      requirements:
      - routable: control_network
      - router: router
substitution_mappings:
    # [...] Omitted for brevity
```

Listing 1: TOSCA template with sample TopologyTemplate (extract)

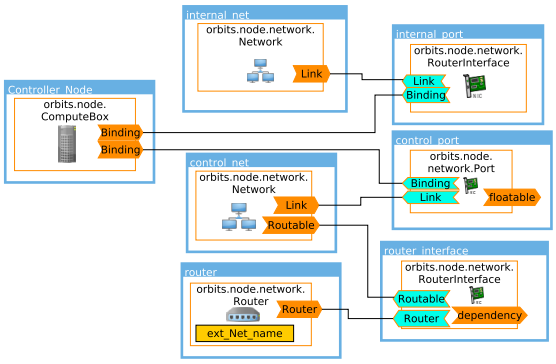

Fig. 3: Graphic Representation of template resources

is graphically presented in figure 3 composed of one VM (ComputeBox) with two NICs, two virtual networks (31-33, and 21-23) and a virtual router (35-37) and a router interface (38-42), with annex interface connected to control network (lines 45-48). Resources and their requirements compose the set of vertex and edges of the TOSCA graph.

B. AOP-Based Weaving

Mainstream cloud infrastructures do not always rely on identical functional services to provide on-demand resources. Two classes of extensibility challenges may be identified in a multi-cloud setting with a continuously evolving infrastructure.

First, a number of non-functional services (e.g., security middleboxes, monitoring services, network applications, OS hardening profiles) which are not present in the description of the basic infrastructure, are often required by users when extending their private clouds to public CSPs. Those services may offer a better integration with legacy, and enrich the set of features available for the user. This may introduce an interesting added-value for cloud brokers – beyond basic resource brokering, thus enlarging the so far narrow market share of the cloud brokering model [1].

Second, infrastructure templates are composed by a set of functional services (e.g., cloud management system) which must be debugged, tested and monitored through additional non-functional services, which may change across the entire life-cycle (development, auditing, production). The ability to inject auxiliary services for different life-cycle steps may thus represent a straightforward approach to simplify deployment and operations. As part of this challenge are notably: (1) the easiness of non-functional service injection/eviction (e.g., to isolate the cause of a misbehavior); (2) possible code re-use across different base templates (e.g., required by different customers).

AOP [12] enables to separate cross-cutting concerns to enhance modularity. Additional "behaviors' to the base code are described without having to modify the original code itself. Code composition, or *weaving*, is behavior-oriented. It relies on the concepts of *point-cut* and *advice*, which are respectively the extra-code, and the code location where it should be applied. We apply the "aspect" definition to "Infrastructure as Code", considering non-functional properties that are normally scattered across several resources (e.g. Network topology, OS configuration) implementing a common non-functional service (e.g. Monitoring, Auditing). For example in case of monitoring, we may have multi-layer agents inside user VMs and alerts managers as standalone [21].]. Decoupling non-functional and functional services results in (1) enhanced reusability on different base TOSCA Templates and (2) dynamic injection upon events.

To address such extensibility challenges, we designed TML, or TOSCA Manipulation Language. TML is an AOP-oriented Domain Specific Language (DSL) to manipulate TOSCA templates. Code weaving does not operate on imperative code, but on the declarative graph in TOSCA files that are parsed, analyzed and modified according to external TML scripts.

The objectives of TML are the following:

- **Service weaving / un-weaving** This means to inject / evict flexibly non-functional services in / from a TOSCA template.
- **Semantic checking** This means exploring the graph of NodeTemplates in a ServiceTemplate for early detection (i.e., before deployment) of semantic errors (connecting to the wrong resource, security constraint violation), beyond simple syntactic checking (missing connections, wrong references).

C. The TOSCA Manipulation Language (TML)

TML scripts enable to modify a TOSCA graph, composed by the resources specified in the *node_templates* fields. This

graph is formed of a set V of vertices, capturing the set of resources with their attributes (e.g., properties, interfaces with annex artifacts), and of a set E of edges, capturing the set of requirements for interconnection.

The structure of a TML script is the following:

- **Filters** They enable to navigate within the graph, and to specify "anchor" resources (i.e., point-cut) from which rules in the advice will be triggered. A filter is composed of a name, a *root* field which identifies the type of resource to be used as anchor, and a *rule* which defines the condition to be verified to select the node when exploring the graph. The operators used to build a rule are is presented in table **??**.
- **Actions** They represent the core of the advice and define the list of modifications to apply to the graph, leveraging filtered resources as arguments. An action is composed of an action type, and of arguments. Following a declarative approach, their execution order does not correspond to their specification order on how actions are specified. A priority graph allows to create resources before they have to be manipulated or connected with the others.
- **Checks** After executing actions, a list of statements is run to validate the obtained template. In case of an empty actions section, the advice simply attempts to verify some properties on the template graph.

Weaving of a filter advice is straightforward. Starting from resources of a specified type, the *filters* rules select resources having specific requirements. The mandatory rules in the filter act as trigger condition for the advice: when all mandatory filter conditions are able to retrieve a resource, the *actions* section is executed. Advices are atomic. Therefore, the TOSCA template should be in a valid state after application of all actions. If check statements are specified, they should be verified to commit the template and accept the weaving. Developers may also define an optional *inputs* section to customize TML script inputs.

Actions may modify the graph according to TOSCA type specifications, but cannot violate imposed constraints (e.g., introducing new requirements or capabilities). Actions cannot use filters on temporary states of the graph as filters are computed before execution – similarly for filters "embedded" inside actions (e.g., for the select operator), and for checks after weaving of any action.

D. An example for weaving from a TML advice: Injecting a Floating IP within a VM

During infrastructure debugging, it may be useful to have external IP addresses to inspect the status of all VMs – which on the contrary should not be accessible in a production deployment. Listing 2 shows a simple TML advice to selectively inject in a VM a *floating IP* (also called elastic IP), or public Internet-accessible IP, for each network port connected to the Internet. However, not all ports should receive a floating IP, as most are not connected to the Internet, but only to internal virtual networks. This script may be useful in the Continuous Integration (CI) chain to easily collect information about errors while testing.

```
1  inputs: #No inputs
2  filters:
3   ports_internet_without_fip:
4    root: orbits.nodes.network.Port
5    rule:"R.link[].routable<>.rou_int[] && !R.floatable<>"
6    mandatory: true
7
8    \end{document}
9  actions:
10  create_fip:
11   type: create_and_connect
12   args:
13    for_each: [ports_internet_without_fip,"R"]
14     name: compute_fip
15     tosca_type: orbits.nodes.network.FloatingIP
16     capability: floatable
17     properties:
18      net_name:{sel_cur:"R.link[].routable<>.router[].
       net_name"}
19 checks:
20     no_ports_internet_without_fip:
21      root: orbits.nodes.network.Port
22      rule:"R.link[].routable<>.rou_int[] && !R.floatable
       <>"
23      mandatory: true
```

Listing 2: TML Floating IP injection Script

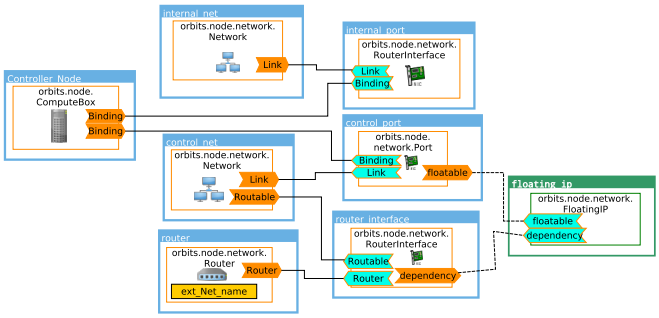

Fig. 4: The template sample in figure 3 weaved with Floating_ip script

The *filters* rule starts looking-up and selecting Ports in the graph, with the following point-cut conditions (line 4): (1) selects all ports which have links connected to networks connected to routers connected to Internet; and (2) avoid interfaces already having a floating IP.

In the actions section (lines 8-17), the *create_fip* item will create the new resources for each selected port. It defines an iterator (*R*) (line 12) to specify the set of new resources and to complete the other resource parameters (*net_name*). For every matching resource, it will create and connect a new floating IP resource, through the "floatable" capability defined in the type definition of *orbits.nodes.network.Port*.

Weaving the TML script to the template shown in Section III-A (listing 1) yields the result shown in figure 4. When run, the TML advice will create and connect a new floating IP resource for the unique port connected to the Internet, *control_port*.

It is worth noticing that the TML aspect is completely decorrelated from the base template, and may be applied to any TOSCA ServiceTemplate. Moreover, if the ServiceTemplate has more than one port connected to an external network, multiple resources will be injected without requiring more generated code.

In this paper, for sake of simplicity, we assume the absence of overlap between TML scripts in case of multiple

weaving, not focusing yet on idempotency and conflicts issues. However, more sophisticated conflict resolution strategies have already been explored, notably to satisfy an idempotency property [22], and their integration in MANTUS represent important direction for future work.

IV. IMPLEMENTATION

We now present a prototype of the MANTUS multi-cloud builder, focusing on the TML weaver component. A MANTUS *implementation workflow* includes the following phases, also used for weaver benchmarking:

- **Matching** A ServiceTemplate is created, composed of concrete nodes starting from an abstract specification. We adopted the standard TOSCA plug-in matching developed by Brogi et al. [19].
- **Fusion** The matched file is generated, reconnecting different branches of matching.
- **Weaving** Non-functional services are injected by applying TML scripts against the TOSCA ServiceTemplate as described in the previous section.
- **Translation** Finally, the resulting TOSCA templates are translated to the Heat Orchestration Language (HOT) to be deployed on an OpenStack CSP on an OpenStack CSP as a target example of CSP.

We implemented the MANTUS TML weaver in Python. MANTUS first builds the expression abstract syntax tree for resources in the TOSCA graph using the pyPlus[2] LR-parser. We used the tosca-parser[3] library to manipulate TOSCA ServiceTemplates for fully compliant TOSCA generation. We also developed a driver-based small TOSCA translator supporting HOT and CloudFormation back-ends. The translator was developed from scratch to keep it minimal and to put emphasis on polymorphism for multi-CSP support.

We introduced a hierarchy of TOSCA resource types to model infrastructure services, focusing on computing and networking resources. Such types were then applied to a full-fledged cloud infrastructure based on OpenStack and Mesos, using Xen, KVM and LXC as core virtualization technologies: infrastructure components were described as TOSCA templates using our custom nodeTypes. We also defined an encapsulating Overcloud type to foster template reuse for multi-cloud environments. Overcloud templates were then refined using the MANTUS workflow to be finally translated to the HOT format. We defined the infrastructure software life-cycle through a interface *Standard*, with annexed operations events (e.g.; Install, Start, Stop) with respective configuration artifacts.

V. VALIDATION

In this paper, we presented an AOP-based approach to meet the non-functional extensibility requirement for a multi-cloud deployment context (DR1). We now validate our approach through several benchmarks. Non-functional extensibility requires three main properties to be satisfied:

[2]https://github.com/erezsh/plyplus

[3]https://github.com/openstack/tosca-parser

TABLE I: Benchmark Variables

Notation	Meaning
T	Number of initial Template Nodes
F	Number of filters per TML Script
R	Number of injected resources per TML script
E	Number of injected "edges" per TML script
S	TML script characterized by S[R;E;F]
W	Weaving operation W(T,S)
A	Coefficient of replication

- **Dynamicity**: The support of dynamic reconfiguration of service, in particular "incremental injections" to be able to add or drop services during execution.
- **Scalability**: The scalability of the system in terms of number of injection and size of template.
- **Performance**: Incremental or reactive extension requires service injection to be performant, supporting timely weaving.

We used, as input template, the basic OpenStack-based ORBITS template, introduced in the previous section, as the testing input for all the benchmarks. This template is an example of "overlay infrastructure layer, composed of a "controller" node and 3 different group instances of "compute nodes" (Xen, KVM, LXC), with two virtual networks (internal and external access (Internet)) and relative network resources (ports, routers). The resulting template used for running benchmarks is a graph of 33 nodes after matching and fusion phases in the MANTUS workflow, starting from a single Overcloud component.

We evaluated the non-functional extensibility with different criteria: (a) influence of aspect complexity to verify scalability; (b) cost of incremental weaving during infrastructure life-cycle to evaluate dynamicity; (c) weaving overhead in overall deployment, and (d) compositional efficiency (multiple weavings) to assess performance.

A. Scalability

1) Compositional Weaving Efficiency: First, we analyzed scalability in terms of TML script complexity. A first complexity metric is the number of injected resources (R) and relationships (E) in the action section (resp. vertices and edges of the graph). We thus evaluated weaver efficiency w.r.t. the number of actions ($A \rightarrow [1-100]$) in weaved script, adding different amounts of resource nodes ($A * R$ at each iteration) and edges per script ($A * E$).

In addition, complexity may also be captured by increasing the number of rules in the filters and checks sections (*Filter-Rules* curve, $F * A$). The benchmarks were performed with $T = 1, S = 1$. Each curve is characterized by a different configuration of S w.r.t. R, E, F, we obtain crafting ad hoc TML scripts. As shown in figure 5-a, weaving time increases but remains linear with the amount of actions and resources injected, which gives a good indication of scalability avoiding combinatorial explosion. Those results indicates that look-up operations in graph scales well and and should encourage TML scripts developers to add extra-rules condition (e.g.; to verify semantic properties after the weaving process).

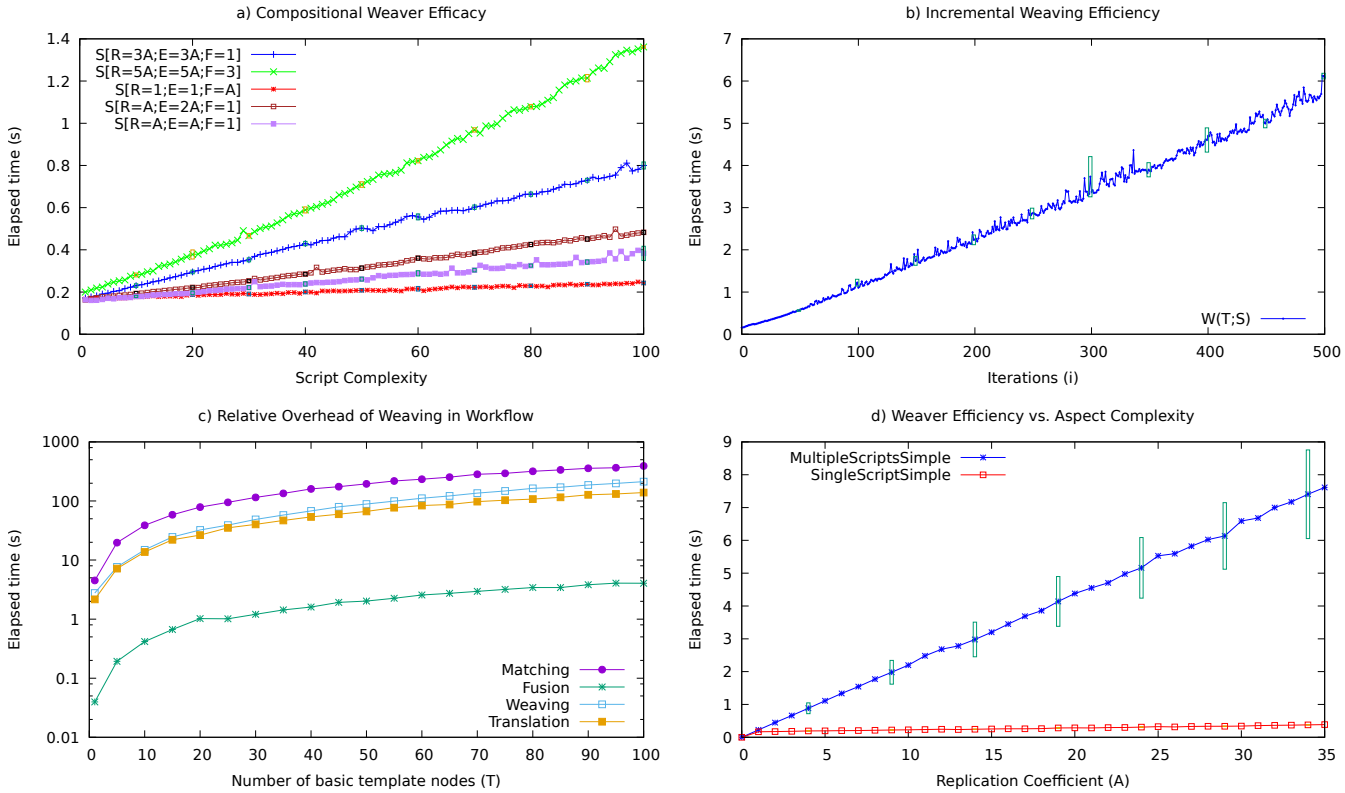

Fig. 5: The tests were performed on a Intel Xeon E5-2650 Haswell at 2.60GHz with 64 GB of RAM.

B. Dynamicity

1) Incremental Weaving Efficiency: In figure 5-b, we analyze the weaving marginal cost, also showing standard deviation of samples. Starting from a base template ($T = 1$), we performed incremental weaving (W_i), iteratively adding one additional TML script to a growing template $T\_i = W(T_{i-1}, S)$ with $T_0 = T; A = 1; R = 3; E = 3; S = 1$. This benchmark suggests that the possibility weaving incrementally new scripts may be affordable even with bigger "previously-weaved" templates, enabling the possibility to weave new resources as reaction to specific events (e.g.; intrusion detection) validating the "dynamicity" property of DR1.

C. Performance

1) Relative Overhead of Weaving in Workflow: We first studied the impact of the weaving phase in the overall MANTUS workflow, to validate the performance of AOP-weaved approach. We performed a full deployment workflow, weaving 10 independent TML scripts of size ($A = 1, E = 3, R = 3, S = 10$), for an increasing number of starting OverCloud nodes($T \rightarrow [0 - 100]$). The TML script simply injects one resource per compute node, with conditions which are always verified and therefore each per-script iteration weaved. Results are shown in figure 5c (log scale): for all phases, the execution time is linear (logarithmic in log scale) w.r.t. the graph size, the weaving phase overhead being less than that of matching. Therefore, we can consider that the weaving operation remains acceptable even with bigger templates.

2) Weaver Efficiency vs. Aspect Complexity: In figure 5-d, to identify a valuable strategy to conceive and develop TML scripts, we compared weaving overheads of multiple weaving ($(W(T, S * A)$ with $S[R; E; F]$) injecting a single resource and of a single script injecting multiple resources($(W(T, A)$ with $S[R*A, A*E, F]$), according to the number of resources. Due to the cost of YAML parsing and graph reconstruction, the single-script approach show more efficiency compared to using multiple scripts. This trend may lead to an interesting direction for further work, somehow also overlapping the conflict detection issue discussed in section III-C: detecting and avoiding conflicts between TML scripts opens the door to weaving performance optimizations, resulting in script size increase, but reducing the number to be weaved.

VI. Related Work

An increasingly popular way to deploy infrastructure in traditional mono-cloud settings is to rely on Infrastructure-as-Code (IaC) paradigms. For instance, in the context of the OpenStack project, several approaches use IaC to simplify the life-cycle management of a cloud infrastructure.

Library-based (e.g.; jclouds,libcloud) approaches provide the best flexibility, supporting many different provider APIs. Libraries are useful to express resource-oriented workloads requirements, communicating with different APIs with a driver mechanism. However, they fail to provide a global transparent vision of the multi-cloud, and do not allow to easily control resources (e.g., networking) [1].

Extending and simplifying TOSCA templates writing, Elastic-TOSCA [23] introduces support for dynamic scaling for TOSCA templates. Cloudify [24] extends the basic standard so that user can define application "blueprints", proposing a "driver" oriented type ontology for multi-cloud application deployment. However, Cloudify does not approach the challenge of non-functional extensibility. CloudMF [25] provides a model-oriented framework to deploy and manage multi-cloud applications leveraging provider-independent and provider-dependent code. In CloudMF, application component definitions are agnostic from the specificity of a single CSP and are then adapted to a specific implementation. CloudMF fosters re-usability and adaptability but without addressing infrastructure issues (e.g.; non-functional injections).

Concerning the provider-centric approach, hybrid clouds [4, 5] usually provide a seamless extension of a private cloud, supporting bidirectional workload migration. However, this extension is limited to a single provider which normally belongs is the same enterprise or a partner of the editor of private cloud Cloud Management Systems.

VII. Conclusion and Future Work

In this paper, we presented MANTUS, an aspect-oriented multi-cloud builder, designed to generate an overlay infrastructure-oriented multi-cloud. MANTUS flexibly weaves non-functional services, translating the outcome to a provider-specific language. At a cost of an acceptable overhead, aspect-oriented weaving separates cross-cutting concerns, fostering reuse of extra code and infrastructure life-cycle automation. We validated the suitability of aspect-oriented approach for non-functional service injection evaluating scalability, dynamicity and performance.

Future work will be oriented in three main directions. First, we will explore idempotence and conflicts when weaving multiple scripts. This axe of work is particularly interesting due to the result of 5-d, where the weaver showed better performance with few huge scripts compared to a multitude of tiny scripts. A method to reduce in a single-script non conflicting multiple-scripts would speed up significantly the weaving process. Second, we will perform a complete set of benchmarks, comparing the overall MANTUS approach to existing baselines and measuring the gain in terms of dynamic reconfiguration. Third, we would evaluate the complexity of the filter rules language, elaborating more complex rules and evaluating the performance w.r.t. the rule complexity and not only their number.

Acknowledgments

This work is partially supported by the by the French Ministry of Education and Research (CIFRE grant) and European Union SUPERCLOUD Project (Horizon 2020 Research and Innovation Program, grant 644962).

References

[1] Yehia Elkhatib. "Mapping Cross-Cloud Systems: Challenges and Opportunities". In: *HotCloud*. 2016.

[2] *Ever heard of 'multi-cloud'? Get with cool kids – it's the New Big Thing™*. URL: goo.gl/KMtdbl.

[3] Sixto Ortiz. "The problem with cloud-computing standardization". In: *Computer* 44.7 (2011), pp. 13–16.

[4] *VMware vCloud*. URL: http://vcloud.vmware.com/.

[5] *Microsoft Azure Hybrid Cloud*. URL: https://www.microsoft.com/en-us/cloud-platform/hybrid-cloud.

[6] *jClouds*. URL: https://jclouds.apache.org/.

[7] *Fog Cloud Library*. URL: https://github.com/fog/fog.

[8] B. Satzger et al. "Winds of Change: From Vendor Lock-In to the Meta Cloud". In: *IEEE Internet Computing* 17.1 (2013), pp. 69–73.

[9] Adam Barker, Blesson Varghese, and Long Thai. "Cloud services brokerage: A survey and research roadmap". In: *IEEE Cloud Computing (CLOUD) 2015*.

[10] Alex Palesandro et al. "Overcoming Barriers for Ubiquitous User-Centric Healthcare Services". In: *IEEE Cloud Computing* 3.6 (2016), pp. 64–74.

[11] *Infrastructure as Code*. URL: https://www.thoughtworks.com/insights/blog/infrastructure-code-reason-smile.

[12] Gregor Kiczales et al. "Aspect-oriented programming". In: *ECOOP*. Springer. 1997, pp. 220–242.

[13] Zhanpeng Jin and Yu Chen. "Telemedicine in the cloud era: Prospects and challenges". In: *IEEE Pervasive Computing* 14.1 (2015).

[14] Andrew Chi-Chih Yao. "How to generate and exchange secrets". In: *Foundations of Computer Science, 1986., 27th Annual Symposium on*. 1986.

[15] Dan Bogdanov et al. "High-performance secure multiparty computation for data mining applications". In: *International Journal of Information Security* 11.6 (2012).

[16] Kaveh Razavi et al. "Kangaroo: A Tenant-Centric Software-Defined Cloud Infrastructure". In: *IEEE International Conference on Cloud Engineering*. 2015.

[17] Alex Fishman et al. "HVX: Virtualizing the Cloud". In: HotCloud (2013).

[18] *OpenStack Deployment, Complexity Concerns Persist*. URL: goo.gl/IwDBeU.

[19] Antonio Brogi and Jacopo Soldani. "Matching cloud services with TOSCA". In: *ESOCC*. 2013, pp. 218–232.

[20] *OASIS Tosca*. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca.

[21] A. Wailly, M. Lacoste, and H. Debar. "VESPA: Multi-layered Self-protection for Cloud Resources". In: ICAC'12.

[22] Waldemar Hummer et al. "Testing idempotence for infrastructure as code". In: *MIDDLEWARE*. 2013.

[23] Rui Han, Moustafa M Ghanem, and Yike Guo. "Elastic-TOSCA: Supporting elasticity of cloud application in TOSCA". In: *CLOUD COMPUTING 2013*.

[24] *Cloudify*. URL: https://getcloudify.org.

[25] Nicolas Ferry et al. "Managing multi-cloud systems with CloudMF". In: *Second Nordic Symposium on Cloud Computing & Internet Technologies*. 2013.