Lifecycle Management of Service-based Applications on Multi-Clouds: A Research Roadmap

George Baryannis, Panagiotis Garefalakis, Kyriakos Kritikos, Kostas Magoutis,
Antonis Papaioannou, Dimitris Plexousakis and Chrysostomos Zeginis
Institute of Computer Science
Foundation for Research and Technology - Hellas
Vassilika Vouton, P.O. Box 1385, GR 711 10, Heraklion, Greece
{gmparg,pgaref,kritikos,magoutis,papaioan,dp,zegchris}@ics.forth.gr

ABSTRACT

In this paper we identify current challenges in the deployment of complex distributed applications on multiple Cloud providers and review the state of the art in model-driven Cloud software engineering. Challenges include lack of support for heterogeneous Cloud providers; limited matchmaking between application requirements and Cloud capabilities; lack of meaningful cross-platform Cloud resource descriptions; lack of lifecycle management of Cloud applications; inadequate cross-layer monitoring and adaptation based on event correlation; and others. In this paper we propose solutions to these challenges and highlight the expected benefits in the context of a complex distributed application.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*; H.3.5 [**Information Storage and Retrieval**]: Online Information Systems—*Web-based services*

Keywords

Cloud Computing, Multi-Clouds, Service Management

1. INTRODUCTION

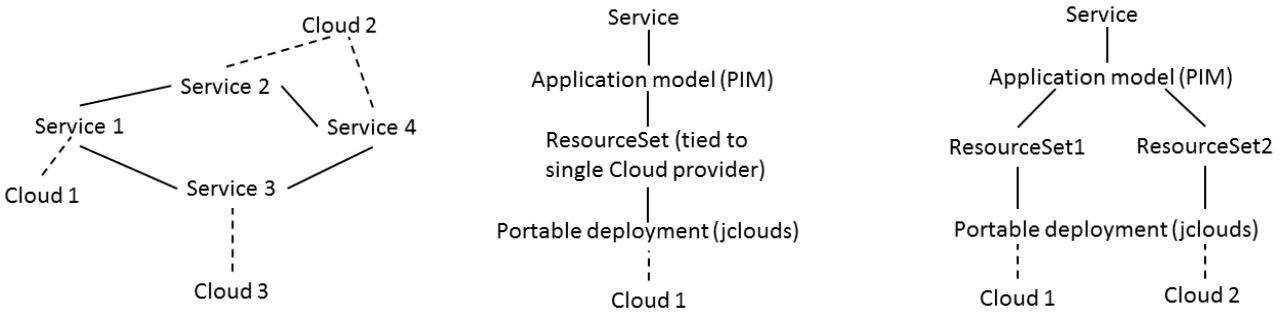
Cloud computing is having a transformational effect on enterprise IT operations, software engineering, service-oriented computing, and society as a whole. Despite its undisputed market traction, to this day there are a number of issues that require solutions for Cloud computing to develop further [8]. In this paper we specifically focus on the need to break the current lock-in experienced by application developers on the Cloud provider they design for and deploy on, and to allow them to simultaneously use (and seamlessly arbitrate between) several Cloud providers. We refer to this goal as the *Multi-Cloud* approach. Our focus is holistic: namely, we describe an end-to-end lifecycle management approach for Service-Based Applications (SBAs) and a research roadmap for realizing it.

Application development for Cloud platforms today follows two main (often complementary) approaches: Composition and use of Software-as-a-Service (SaaS) instances exported by providers such as Salesforce (CRM and ERM applications), Google (Google Apps), etc.; and, development of the application over middleware offered via Platform-as-a-Service (PaaS) providers (such as Amazon Elastic Beans) or at a lower level of abstraction, over Infrastructure-as-a-Service (IaaS) providers (such as Amazon EC2 or Microsoft Azure). In the former approach (Figure 1(a)), each SaaS instance can be implemented and deployed over a different Cloud provider, naturally supporting heterogeneity (although at a fairly coarse level of granularity). In the latter approach (Figure 1(b)), a SaaS instance is typically developed using model-driven software engineering methodologies targeting individual Cloud providers. A problem with current methodologies is that –although addressing portability via the use of generic platform APIs such as jclouds – they tie all Cloud resources (referred to as ResourceSets) to a single Cloud provider and thus preclude deployment of the application on multiple Cloud providers.

A key contribution of this paper is proposing extensions to the above approaches to allow deployment of applications over multiple, heterogeneous Cloud providers (as depicted in Figure 1(c)). While offering more flexibility, certainly not all applications are expected to benefit from it: applications whose ResourceSets exhibit significant cross-talk could suffer from excessive Internet charges and performance issues (high latency, low bandwidth across Cloud providers). On the other hand, decomposing complex applications at increasingly coarser boundaries may eventually reach the service interfaces between organizations, resembling the approach of Figure 1(a). We thus believe that one needs to make a careful decomposition of application components, at a lower level than the service interfaces between organizations but higher than software component interfaces involving tightly-coupled resources in order for the design of Figure 1(c) to make sense (more on that in Section 2). Reasoning about effective Multi-Cloud deployment at this level requires capturing component dependencies in application models.

There are several reasons justifying deployment of complex applications on Multi-Clouds. For instance, an application may have dependencies on software components or services offered by different Cloud providers. In addition, different components of an application may have different

MultiCloud'13, April 22, 2013, Prague, Czech Republic.



(a) SaaS composition (b) Single-Cloud deployment (c) Multi-Cloud deployment

Figure 1: Cloud application development approaches

resource requirements that are best satisfied by different Cloud providers. For instance, provider A may offer specialized VMs of a certain kind –e.g., featuring graphics accelerators, solid-state storage devices, etc.– while provider B may specialize in another –e.g., higher core-count or dynamically reconfigurable VMs–. Cloud providers may also differentiate on their offered cost for different types of resources (e.g., CPU is cheaper on provider A while provider B delivers cheaper I/O throughput). Two components of the same application may need to be deployed on different geographical zones for proximity, to stay local to these geographies –data, sensors, etc.– or to minimize chances of catastrophic failure [26]. In the latter case, an application provider typically deploys redundant parts of the application to different Cloud providers. Generally, decomposition of a complex application on Multi-Clouds may be either *functional* (different parts of the application logic placed on different providers) or *data-driven* (redundant functionality with state/data split to different providers) or a combination of them.

The paper is organized as follows. Section 2 motivates our work through a specific Multi-Cloud management scenario, while also identifying the related challenges and research issues. Section 3 offers a critical overview of the state of the art for each activity of the model-driven Cloud-based SBA lifecycle. Section 4 analyzes our proposed solution and Section 5 concludes.

2. MOTIVATION

To motivate our work, we use the example of a service-based application that predicts and controls traffic in urban areas. The application consists of three tasks (services) shown in Figure 2: (a) the monitor/check task T_M responsible for collecting air pollution and traffic data from environmental sensors deployed in urban areas, while also taking into account the timing of special events such as high traffic hours, etc.; (b) the assessment task T_A gathers information produced from T_M and assesses the current and future situations and plans appropriate actions to manage traffic; (c) the device configuration task T_D is responsible for receiving actions from T_A and (re-)configures traffic control devices.

Tasks T_M, T_A, and T_D are mostly self-contained. Information is exchanged periodically (e.g., every hour) and low response time in cross-task communication is not a priority. T_A requires high computation, availability and throughput, whereas T_M and T_D do not. T_M and T_A require storage capacity while T_D does not. T_A has moderate storage throughput capabilities since, while a high amount of information is exchanged, this happens rather infrequently. Finally, T_M and T_D should be deployed geographically close to the sensor infrastructure (the urban areas) while T_A can be deployed anywhere. Based on these characteristics, a Multi-Cloud deployment can be created that places T_M and T_D on a private/municipal Cloud (different instances of them in different cities) close to the sensor infrastructure. The VMs chosen can be of low computational power; T_M however should be coupled with a storage service. T_A can be placed in any Cloud that can provide high-computation VMs with storage capacity at the best reliability/price ratio.

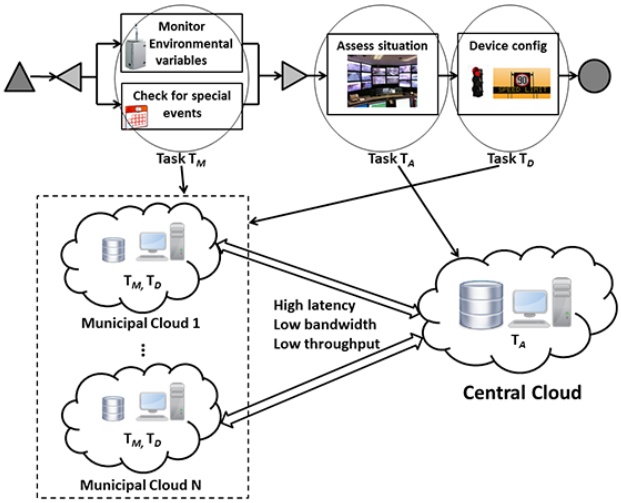


Figure 2: Multi-Cloud Traffic Management

The process of mapping application requirements to the appropriate Cloud infrastructures outlined above is also referred to as *matchmaking*, depicted in the diagram of Figure 3. In a Multi-Cloud scenario, keys to effective matchmaking are the *specification* of application *requirements*, uniform (cross-Cloud) *resource descriptions*, and the exploitation of *semantics*. The latter in particular can improve the accuracy of matchmaking compared to syntactic, structural description-based approaches [23]. Besides the initial SBA deployment to multiple Cloud platforms, there is a need to manage the application throughout its lifecycle via *cross-*

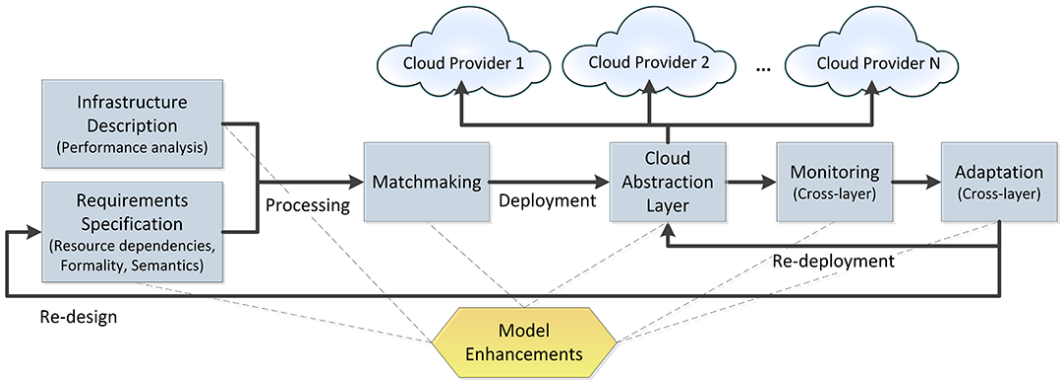


Figure 3: Lifecycle of Service-based Applications deployed on multiple Cloud infrastructures.

layer monitoring and adaptation. The rest of this section describes our motivation to address problems in this space.

Requirements specification. A Multi-Cloud deployment of an application (as shown in Figure 1-(c)) is possible when the *resource requirements* as well as the *dependencies* between resources are clearly expressed. For example, deploying all three tasks of our example application onto a private/municipal Cloud is a feasible scenario; however, the availability of cheap high-computational power VMs in a different Cloud is a preferable solution for deploying T_A. This Multi-Cloud solution is made possible not just by declaring T_A's computational requirements but also by expressing the low-bandwidth, high-latency dependency between T_A and the other two tasks. Thus, *capturing dependencies* (such as the amount of communication between components, the type of resource requirements, etc.) between application components is key to enabling a Multi-Cloud setting.

Infrastructure Description. *Heterogeneity* in Multi-Clouds requires that resources (e.g., VM types, storage, etc.) across different Cloud infrastructures are *expressed in a uniform, platform-independent manner*. For example, the relation between a provider's *"medium"* CPU instance to another's *"medium"* instance is not clear from the perspective of our application. Modeling thus needs to go beyond simple properties such as CPU speed offered by a VM, to a deeper description of performance capabilities that make more sense to applications while also allowing for comparisons across Clouds. A few Cloud providers have defined their own metrics (Amazon EC2 units, Google Compute Engine units, etc.); however these cannot readily extend to Multi-Cloud units. There is a stronger disparity of descriptions across vendors regarding quality of service (QoS) metrics, raising the need for *cross-platform specification of QoS capabilities*.

Matchmaking. Current approaches to matching application requirements with infrastructure offerings at application deployment time face several challenges. They are either completely manual, exposing a human administrator to a potentially unmanageable space of possibilities, or when attempting to automate a solution, they tend to oversimplify the problem by degrading the specification of application requirements to the level of resource descriptions. To achieve a breakthrough, we need to use application requirements described at a higher level than infrastructure offerings (e.g. not resource-related but application-related conditions) and to come up with cross-platform resource descriptions and a general classification of them in a Multi-Cloud setting. To achieve both, we need to introduce *formal specifications*: formality in specifications, especially if combined with *semantics* when necessary, is indispensable when attempting to answer whether properties of a specification hold, in an automated way. For instance, formalizing requirements as a set of application-related conditions and following a similar procedure for describing what a provider offers, one can use verification techniques, constraint optimization, or mixed-integer programming techniques, to answer whether a particular offering satisfies a requirement.

Monitoring and Adaptation. An important concern for any Cloud-based application is to maintain its desired level of service along its entire lifecycle. This can be achieved by either setting up appropriate service-level agreements (SLAs) with Cloud providers or by explicitly managing their level of service through monitoring of the underlying infrastructure and adapting to changes to it. Leading Cloud providers are just starting to roll out solutions in both areas (see Section 3 for existing approaches), while most of the industry is still lagging behind. Looking forward, applications designed for Multi-Cloud environments will be facing challenges stemming from the lack of uniform (cross-platform) support for monitoring and adaptation solutions. Assuming that these challenges are eventually met, a number of other problems still need attention: (1) cross-layer (IaaS, PaaS and SaaS) monitoring and alignment of the monitored events; (2) cross-layer coordination of adaptation actions; (3) proactive as well as reactive adaptation policies.

3. RELATED WORK

In this section we review related work following the general structure of Figure 3. Several related approaches ascribe to *model-driven software engineering* (MDE) principles. MDE has received significant attention in recent years, including in the context of Cloud computing [4, 22]. It offers indisputable advantages for software development but challenges as well. As pointed out in [18], MDE allows developers to work at higher levels of abstraction, improving software quality, reducing complexity and facilitating reuse. However, it is prone to redundancy issues and the fact that updates in one model may affect other inter-related models (the *round-trip* problem). In the context of Cloud computing, MDE involves defining Platform Independent Models (PIMs) to describe the deployment of an application in the Cloud, independent of specific Cloud realizations. Model

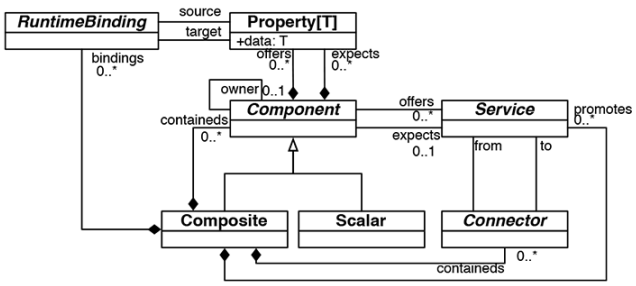


Figure 4: PIM4Cloud DSL Metamodel [11].

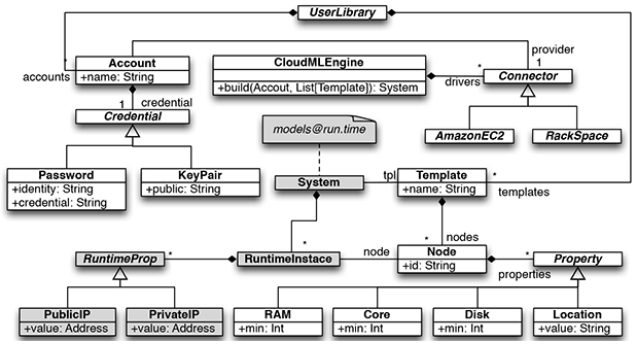


Figure 5: CloudML Metamodel [12].

transformations can then result in Platform Specific Models (PSMs) that capture the details of actual deployment environments.

With the advent of Cloud computing, the Software-as-a-Service (SaaS) delivery model gained significant traction in the software market and with it, the Service-Oriented Architecture (SOA) discipline received a significant boost. Effective development with SaaS and SOA technologies is driven by modeling frameworks such as SoaML [6], a UML-based language for specifying Service-Oriented Architectures, namely defining components and their capabilities as well as component dependencies at the business and service levels.

Requirements specification. Model-driven development of service systems for Cloud platforms requires that high-level specifications (e.g. SoaML) are transformed to more concrete application models, independent of specific Cloud platforms. Recent research in abstracting Cloud platforms to PIMs has resulted in three interrelated metamodels and languages, namely PIM4Cloud DSL [11], PIM4Cloud [5] and CloudML[1] [12]. PIM4Cloud DSL provides a way to model software applications and their deployment requirements by employing a component-based approach (Figure 4). While addressing basic deployment requirements, it does not provide the ability to express formal, detailed resource requirements (such as, minimum performance required) nor semantic annotations; it also fails to capture *dependencies between resources* (e.g. amount of communication).

Infrastructure Description. PIM4Cloud is another platform independent language devoted to the modeling of both private and public Cloud infrastructures through the description of the resources exposed by these infrastructures and required by a specific application. PIM4Cloud enables the expression of the intent of a service model without capturing its realization in a runtime framework. It can be exploited in matchmaking scenarios involving the discovery of a Cloud that offers the resources required by a specific application. A limitation of PIM4Cloud however is that it ignores other phases of an application's lifecycle (Figure 3). Additionally, its infrastructure descriptions are too low-level (e.g., CPU frequency). Other approaches that model Cloud infrastructure at the resource level are Cloudify [3] and Amazon's CloudFormation [1]. Cloudify offers advanced support for application lifecycle management, while CloudFormation is platform-specific. None of the aforementioned modeling efforts support information inference via the use of formal logic, which is a powerful tool for automated matchmaking.

[1]Note that CloudML is also the name of the Cloud modeling language that is the focus of standardization efforts by several new FP7 projects, such as PaaSage and MODAClouds.

Matchmaking. CloudML is a domain-specific language that supports matchmaking between deployment requirements and infrastructure descriptions. CloudML includes a runtime that performs provisioning actions on the Cloud provider deemed most suitable. As depicted in Figure 5, application requirements are modeled as templates consisting of nodes that need to be provisioned in the Cloud. During the provisioning process, an engine creates instances for each node and attempts to find Cloud resources for each one of them. Runtime instances follow the models@run.time [9] approach, which allows the user to query their status or any other runtime-related information. Once provisioning is completed and all runtime instances have started, deployment can be performed. Just like PIM4Cloud DSL, CloudML does not support Multi-Cloud provisioning: a single application cannot be spread across multiple providers. Finally, CloudML node models are characterized by simple textual descriptions of RAM, CPU, disk and location properties and thus cannot support advanced reasoning for Multi-Cloud cross-platform provisioning decisions.

An interesting approach to matchmaking Cloud services based on application requirements, focusing solely on Cloud storage services has been proposed by Alvarez et al. [24]. This work proposes an XML-based language to describe the storage capabilities of Cloud services and requirements of user applications. Apart from the selection of Cloud storage services, this work shows how to estimate the cost savings from switching between different Cloud storage services as well as the evolution over time of cost and storage performance. Garg et al. [17] propose a framework for ranking Cloud services based on their performance on QoS properties and the weights given to these properties, by exploiting an Analytical Hierarchy Process (AHP)-based algorithm. Zeng et al. [31] propose a Wordnet-based matching algorithm that considers the semantic similarity of the concepts mapping to the I/O parameters of the services. Finally, Buyya et al. [13] propose the federated Cloud computing environment to match user-provided QoS targets (as well as energy management goals) to appropriate Cloud services. Matchmaking and selection of PaaS services is proposed by D'Andria et al. [14]. Their algorithm searches among available PaaS offerings and ranks them based on the number of satisfied user preferences. The ranking technique employed currently does not take into consideration service semantics or weighting of the search criteria, returning all compatible platforms to the end user. García-Gómez et al. [16] perform matchmaking via the use of *blueprints*. In their approach, requirements con-

tained in an input blueprint are compared against existing blueprints in a repository, producing a set of alternative Abstract Resolved Blueprints (ARBs). Each ARB is a possible combination of blueprints constituting a Cloud application.

Cloud Abstraction Layers. Employing a Cloud abstraction layer eases application portability across Cloud platforms by allowing users to manipulate virtual machines as objects. A variety of libraries such as jclouds (www.jclouds.org) and Apache libcloud (libcloud.apache.org) can abstract away the differences among Cloud providers. They provide a Cloud abstraction API that works as a wrapper around a number of Cloud provider APIs. Jclouds and libcloud are the most popular solutions, each supporting over 30 different Cloud providers and software stacks. They consist of multiple, relatively self-sustained components that expose a simple API. There is significant similarity in the terminology and models used across these libraries.

Monitoring and Adaptation. A prime example of Cloud monitoring services is Amazon's CloudWatch, which provides support for Amazon Web Services (AWS) production services as well as custom data. Production services automatically push metrics to CloudWatch; in addition, users can manually define custom metrics and calculate statistics. The available metrics can be considered adequate for offering applications a view of their operation. The custom metrics, possibly derived from available CloudWatch metrics, can also cater for the measurement of important missing parameters in application requirements. For example, service availability can be calculated based on the number of successful invocations. CloudWatch was recently enriched with an *alarm* feature that informs users of metric violations. Unfortunately, the current solutions in the Cloud space do not provide any uniform way for measuring metrics. While some providers (e.g., Eucalyptus, GreenQloud) exploit the Amazon API, there are various others using different metrics and/or metric measurements, creating the need for *metric alignment and mapping*. PaaS monitoring discussed in [14] considers latency and throughput metrics. Adaptation policies are currently mostly handled explicitly by applications. When this is not an option, applications can opt for guaranteed service levels via Service-Level Agreements (SLAs) with Cloud providers. Unfortunately, SLA management in Cloud computing is still at a primitive stage and supported by few providers. Thus, Cloud abstraction layers are far from including SLA management among their operations.

Application Lifecycle. A modeling approach must address the complete Cloud application lifecycle through appropriate processes and plans. TOSCA [10] is such a recent specification language, derived from a long line of work in software provisioning, deployment and management of distributed services. TOSCA is a middle-level language for the specification of the topology and orchestration of an IT service in the form of a service template. The language focuses on the semi-automatic creation and management of an IT service independent of any Cloud providers. The management of an IT service is achieved through the specification and execution of *process models* (BPMN, BPEL), which define an orchestration of services. Three types of plans are envisioned: build and termination plans associated to the deployment and termination of an IT service as well as modification plans associated to the IT service management. In a Cloud computing context, TOSCA requires mapping to models expressing Cloud notions. Such a mapping could be performed through a model transformation to a domain-specific Cloud-based language, such as PIM4Cloud. Issues such as matchmaking resources to application requirements and supporting non-functional services aspects are considered as falling under the domain-specific model that TOSCA maps to and thus are not explicitly modeled by it.

Another approach to support the lifecycle of SBAs over different Cloud providers is brokerage. Existing Cloud brokers include CompatibleOne (www.compatibleone.org), JamCracker (www.jamcracker.com/jamcracker-platform), and DuraCloud (www.duracloud.org). While these brokers support the deployment of applications in a variety of Clouds, they either do not support multi-Cloud deployments or concentrate on specific application types or services (e.g., storage).

4. PROPOSED SOLUTIONS

Application Lifecycle. To address the full lifecycle of complex SBAs we propose to unify TOSCA topology and orchestration specifications with existing Cloud service description languages. This integration will combine the generality of addressing multiple Cloud domains with the ability to support complete Cloud services lifecycle management via BPMN/BPEL workflows. We propose that the unified approach be *logic-based*, expressing requirements and capabilities as a set of predicate constraints that impose conditions on the application deployment. The names for predicate constraints and attributes could be derived from a common vocabulary, defined as a set of *ontologies* for Cloud-based deployment. Semantic annotation through ontologies provides a shared understanding of domain concepts [7] and facilates the expression of requirements for the wide range of SBAs that can be deployed in the Cloud and the description of different infrastructure solutions and their capabilities.

Infrastructure description. We aim for modeling and analyzing the performance of a Cloud-provided VM at a level of description higher than the bare-hardware, enabling *cross-platform comparisons of Cloud resources*. Our motivation is similar to that behind the traditional use of *benchmarking* to set a baseline for comparing computer architectures [2]. Thus we use a broad set of benchmarks to build a multi-dimensional profile of the performance of a VM. We focus on four areas: CPU, disk, memory as well as overall system performance. Each benchmark is classified into a group associated with one of the areas above. We create a *vector-based performance profile* of each VM, drawing from previous work in the area of application-specific benchmarking [29]. The VM profile consists of four vectors used to represent the set of results of each benchmark group. We believe that this vector-based technique is key to providing a meaningful way of comparison between VMs.

The first group of benchmarks focuses on CPU performance. A parameter in such benchmarks is their stress on parallelism (ability to utilize multiple cores). While we assign a higher weight to benchmarks that exploit parallelism, we note that their applicability depends on the degree of concurrency available in the application. The second set of benchmarks focuses on disk I/O performance for both remote and local storage options, where available. In most cases, remote storage systems provide more durability and availability of data as they can survive a VM failure. In contrast, data in a local storage system live as long as the VM is running, but such solutions offer higher throughput at lower cost. We focus on measuring the throughput and

latency characteristics of the underlying storage system in all cases. The third group of benchmarks focuses on memory I/O, a metric of interest for applications that involve heavy data movement. While memory size is a key aspect, memory throughput can also affect system performance. The fourth group consists of higher-level application benchmarks such as SPECjvm2008 [28] and Unixbench [27] that can characterize the overall system performance. These benchmarks combine multiple tests that examine various aspects of system performance, in particular CPU and memory properties.

To allow a cross-platform categorization of resources to different classes of service (*"small"*, *"medium"*, *"large"*, and possibly others) for different resource types, we use clustering techniques to separate VM instances of different providers. We use the *k-means* [20] clustering method for the benchmark results of each VM aspect. Alternatively, we offer a *cost-normalized* view of resources by taking monetary cost into account. Based on the VM performance profiles derived, we use a logic-based formal specification language to describe infrastructure offerings. For instance, a VM supporting multi-core processing, offering high CPU performance, and employing a remote storage system is described as:

$$CPUCores(VM, many) \wedge CPUPerformance(VM, high) \wedge Storage(VM, remote)$$

Requirements specification. We propose to extend existing application deployment models (Section 3) in two directions. First, we consider the explicit modeling of component and resource *dependencies* as key to reasoning about Multi-Cloud deployments. These dependencies can be either described by an expert or discovered by a dependency discovery system [21]. We also propose incorporating in the models certain aspects of component behavior (such as use of concurrency, need for persistence, etc.). Second, we propose to align application requirements with infrastructure descriptions by expressing the former as a set of *predicate constraints* that impose conditions on the application deployment. Table 1 depicts an encoding in logic of the requirements expressed for the Traffic Management example of Section 2. Requirement 1 expresses the fact that T_A has high computation demands. Optionally, we could express the fact that the task is highly concurrent, via $Parallelism(T_A, high)$. Requirements 2 and 3 refer to a software component (an Enterprise Java Bean) of a task. References to the quality of resources required, expressed as *"high"*, *"medium"*, *"low"*, etc., imply a cross-platform mechanism to classify resources as such, as was described for our infrastructure description methodology. Another way to assign a meaning to *"high"* is by comparing it to a deployment history (i.e., *"higher"* compared to the resource used in execution of "1-FEB-2013-12:47") of the same application or of applications that are deemed *similar based on some defined patterns* [15]. Finally, under an "expert" mode, a developer should be allowed to express the exact requirements of their application, in relation to a particular benchmark, e.g.

$$Throughput(T_A, EJB_x, > 80 \text{ in SPECjvm2008-Derby}).$$

It should be stressed that the requirements expressed here are closely related to the infrastructure description approach described earlier and at a lower level than goals defined in Service-Level Agreements (SLAs). Such requirements can be expressed directly by the application developer and also derived indirectly from application attributes (e.g. concurrency or functional characteristics).

Table 1: Traffic Management requirements

#	Task	Requirements	
		Nat. Language	**Logic**
1	T_A	high computation	$CPUPerformance(T_A, high)$
2	T_A	high availability	$Availability(T_A, EJB_x, high)$
3	T_A	high throughput	$Throughput(T_A, EJB_x, high)$
4	T_M	low computation	$CPUPerformance(T_M, low)$
5	T_D	low computation	$CPUPerformance(T_D, low)$
6	T_A	persistent storage	$DiskStorage(T_A, persistent)$
7	T_A	moderate disk throughput	$DiskThroughput(T_A, medium)$
8	T_M	non-persistent storage	$DiskStorage(T_M, nonpersist)$
9	T_M T_D	geographically close	$Proximity(T_M, T_D, high)$

Table 2: Traffic Management deployment rules

#	Rule
1	$Parallelism(T_x, high) \Leftarrow Deploy(T_x, VM) \wedge CPUCores(VM, multiple)$
2	$DiskStorage(T_x, persistent) \Leftarrow Deploy(T_x, VM) \wedge Storage(VM, remote)$
3	$DiskStorage(T_x, nonpersist) \Leftarrow Deploy(T_x, VM) \wedge Storage(VM, local)$
4	$Proximity(T_x, T_y, high) \Leftarrow Deploy(T_x, VM_1) \wedge Deploy(T_y, VM_2) \wedge Host(VM_1, Cloud_x) \wedge Host(VM_2, Cloud_x)$
5	$Throughput(T_x, EJB_y, high) \Leftarrow Deploy(T_x, VM) \wedge CPUPerformance(VM, high)$

Matchmaking. Combining formal application requirements with the infrastructure descriptions/capabilities proposed is the basis for a powerful matchmaking process, as shown in Figure 6. A detailed specification of requirements (such as the one presented earlier) is given as input to the matchmaker engine, which also accesses infrastructure descriptions through a knowledge base. *Constraint satisfaction rules*, stored in a rule base, are employed in order to match requirements with existing infrastructure offerings, resulting in one or more proposed plans for deployment. Table 2 shows some example rules that guide the matchmaking process for the example of Figure 2. Rule 1 expresses the fact that a highly-concurrent task should be deployed in a VM supporting multiple cores. Rules 2 and 3 choose remote or local storage systems depending on the persistence requirements of a task. Rule 4 deploys two tasks in VMs hosted by the same Cloud provider, when they are required to be tightly coupled (such as T_M and T_D in our example).

Such rules can either be expressed by deployment experts or result from learning processes based on the deployment history of the application (or similar applications). Based on the use of such rules and due to the need for solving an optimization problem, we follow a constraint logic programming approach to matchmaking which simultaneously considers the optimization of many objectives and provides the best (deployment) solution even when the requirements posed are over-constrained. To enable the most suitable and fair ranking of the deployment solution we will consider exploiting the AHP process to indicate the relative importance

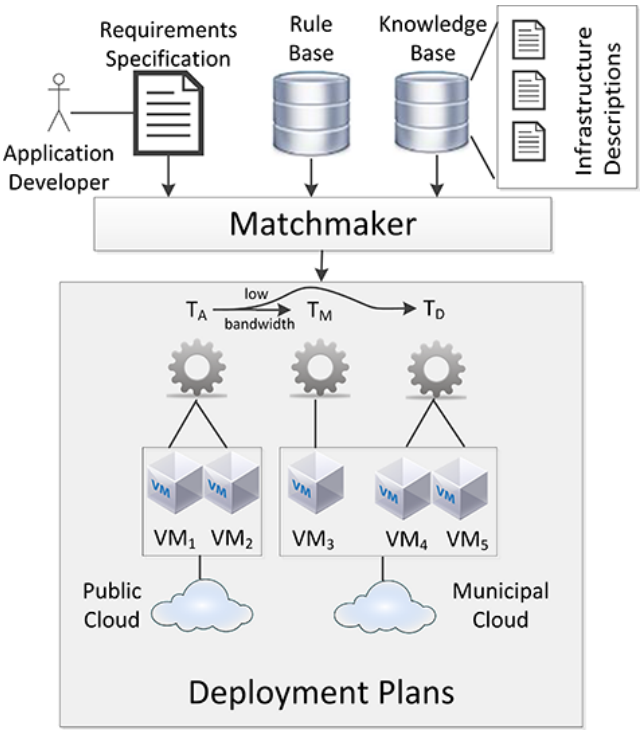


Figure 6: Matchmaking process.

of the optimization criteria as well as particular utility functions for uniformly normalizing the values of these criteria.

Rule-based reasoning techniques are often associated with high computation complexity or issues of looping and/or contradicting rules. We expect the computation complexity of our approach to be moderate, given the fact that rules correspond to the limited number of dimensions expressed in infrastructure descriptions. Looping cases are improbable, given the fact that the heads and bodies of rules almost always would refer to different entities (application requirements and infrastructure features respectively). Finally, contradiction can only be imagined when there is a cost-related rule (e.g. low cost requirement contradicting high CPU performance), which can be solved by imposing an ordering between rules based on the user's preferences.

Monitoring and Adaptation. We plan to extend previous work on cross-layer event-based monitoring and adaptation of SBAs [30] to all three layers (SaaS, PaaS, IaaS) of a Multi-Cloud environment. This framework guarantees the correct and timely ordering of the delivered events, by exploiting distributed time protocol algorithms. Events span both functional and non-functional service aspects and can lead to the detection of a particular root problem through the exploitation of component/service dependencies and *metric derivation trees* (MDTs). MDTs explain how a high-level metric (e.g., the parent) is measured through mathematical formulas applied to lower-level metrics (MDT children) which can in turn be measured based on further lower level metrics. In this way, the root of an MDT is eventually measured based on all metrics residing at the leaf level. Besides the measurement of QoS parameters and subsequent evaluation of application requirements, they can also be used to detect which low-level metrics are to blame for the violation of a specific requirement. MDTs can also be used to define and measure (previously non-existing) QoS metrics through existing lower-level metrics in a specific infrastructure. For instance, if the unavailable metric A can be computed from metrics B and C, while the unavailable metric C can be computed from the available metrics D and E, then eventually metric A will be computed based on metrics B, D, and E. We plan to extend this framework to define service/component dependencies and MDTs in Multi-Clouds.

The framework forms event-to-action correlation rules [25] by capturing event patterns and mapping them to particular adaptation actions using rule derivation techniques. This approach to adaptation can be exploited in Multi-Clouds by mapping *cross-layer and cross-platform* event patterns (e.g., an event at one IaaS provider may cause an event at a different PaaS provider) to suitable adaptation strategies. For instance, an event detected by monitoring indicates that the storage volume on which T_A depends performs at a lower data rate than the resource's description. This change means that task T_A's throughput requirement $Throughput_{disk}(T_A, medium)$ is no longer satisfied. Based on our event history, we anticipate that such a violation reduces the availability of the EJB component (increase in failed/aborted transactions) of the task. A pattern is thus detected, comprising these two events and immediately after the occurrence, a failover with minimum downtime is performed (e.g. switch T_A to a backup VM). This is an example of proactive adaptation to reduce the unavailability of a higher-level service. This behavior is encoded in the following Condition-Action rule:

$\neg DiskThroughput(T_A, medium) \wedge \neg DiskThroughput(T_A, high)$
$\wedge \neg Availability(T_A, EJB_x, high) \rightarrow Failover(T_A)$

By moving T_A to a healthy VM, we reduce the probability that the high availability requirement of the EJB is violated. Our proposed monitoring and adaptation actions for Multi-Clouds can be described through process plans similar to the ones supported by TOSCA. In this way, our framework uniformly covers provisioning and adaptation of SBAs throughout their lifecycle.

Besides adaptation, the need to re-define application requirements (as they may have been underestimated) or to re-provision resources (as the initial provisioning may not be meeting requirements) is a key lifecycle management task, as depicted in Figure 3. For example, if a throughput requirement is frequently violated, there is a need to either increase application requirements or upgrade resources towards higher performance. Our adaptation framework supports re-design by maintaining the history of adaptation actions and informing the user when their frequency exceeds a specific threshold or when there is a change in the Cloud service space (e.g., a storage service satisfying the user-provided requirements is not available any more) which may lead to repeated execution of particular adaptation actions.

Finally, in a Multi-Cloud monitoring system, in addition to establishing a common vocabulary using ontologies, it is necessary to correlate different but related terms used in specifications of requirements and capabilities. For example, differently defined but equivalent or related QoS metrics may be measuring the same QoS parameter. In the former case, we plan to use QoS *metric matching* algorithms [19] to align requirement and capability specifications. For related QoS metrics, such as metrics of average and minimum availability, we can define a *comparison operator* indicating

that minimum availability is less than equal to the average one. Such relations, expressed through rules, establish connections between specifications containing such metrics, thus enabling previously not possible matchmaking.

5. CONCLUSIONS

In this paper we described a research roadmap for addressing current challenges in the lifecycle management of service-based applications on Multi-Clouds. Key elements of this roadmap are: extensions to current Cloud models to capture dependencies and behavioral attributes; use TOSCA to tie several Cloud-specific models together and to connect with workflows; using a logic-based approach expressing requirements and capabilities as a set of predicate constraints; supporting matchmaking via a constraint logic programming approach; and realize cross-layer and cross-platform monitoring and adaptation. We believe these are key challenges to address on the road to realize Multi-Cloud platforms.

6. ACKNOWLEDGMENTS

We thankfully acknowledge the support of the PaaSage (FP7-317715) EU project.

7. REFERENCES

[1] AWS CloudFormation. http://aws.amazon.com/cloudformation.

[2] Cloud Harmony. http://cloudharmony.com/benchmarks.

[3] Cloudify. http://www.cloudifysource.org/.

[4] *CloudMDE 2012 Workshop (collocated with ECMFA'12, Copenhagen)*, 2-5 July, 2012.

[5] REMICS Deliverable D4.1: PIM4Cloud. http://www.remics.eu/system/files/REMICS_D4.1_V2.0_LowResolution.pdf, 2012.

[6] Service Oriented Architecture Modeling Language (SoaML). http://www.omg.org/spec/SoaML/, 2012.

[7] G. Antoniou and F. van Harmelen. *A Semantic Web Primer*, chapter 1, pages 10–13. MIT Press, Cambridge, MA, 2008.

[8] M. Armbrust et al. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[9] U. Aßmann, N. Bencome, B. H. C. Cheng, and R. B. France. Models@run.time (dagstuhl seminar 11481). Technical Report 11, Dagstuhl Reports, 2011.

[10] T. Binz, G. Breiter, F. Leymann, and T. Spatzier. Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, 16(3):80–85, 2012.

[11] E. Brandtzæg, M. Parastoo, and S. Mosser. Towards a Domain-Specific Language to Deploy Applications in the Clouds. In *Third International Conference on Cloud Computing, GRIDs, and Virtualization(CLOUD COMPUTING)*, pages 213–218, Nice, 2012.

[12] E. Brandtzæg, M. Parastoo, and S. Mosser. Towards CloudML, a Model-based Approach to Provision Resources in the Clouds. In H. Störrle, editor, *8th European Conference on Modelling Foundations and Applications (ECMFA)*, pages 18–27, 2012.

[13] R. Buyya, R. Ranjan, and R. N. Calheiros. InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. In *ICA3PP*, LNCS, pages 13–31. Springer, 2010.

[14] F. D'Andria, J. Gorroñogoitia Cruz, J. Ahtes, S. Bocconi, and D. Zeginis. Cloud4SOA: Multi-Cloud Application Management Across PaaS Offerings. In *MICAS*, 2012.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Cambridge, MA.

[16] S. García-Gómez et al. Challenges for the comprehensive management of Cloud Services in a PaaS framework. *Scalable Computing: Practice and Experience*, 13, 2012.

[17] S. K. Garg, S. Versteeg, and R. Buyya. SMICloud: A Framework for Comparing and Ranking Cloud Services. In *UCC*, 2011.

[18] B. Hailpern and P. L. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–462, 2006.

[19] K. Kritikos and D. Plexousakis. Semantic QoS Metric Matching. In *ECOWS*, pages 265–274, Zurich, Switzerland, 2006. IEEE Computer Society.

[20] J. MacQueen. Some Methods for Classification and Analysis of Multivariate Observation. In *Proc. of 5th Berkeley Symposium on Mathematical Statistics and Probability*. University of California Press, 1967.

[21] K. Magoutis, M. Devarakonda, N. Joukov, and N. Vogl. Galapagos: Model-Driven Discovery of End-to-End Application-Storage Relationships in Distributed Systems. *IBM Systems Journal*, 52(4/5):367–378, 2008.

[22] B. Nagel et al. Model-driven Specification of Adaptive Cloud-based Systems. In *Proc. of 1st Workshop on Model-Driven Engineering for High Performance and Cloud Computing*, 2012.

[23] P. Plebani and B. Pernici. URBE: Web Service Retrieval Based on Similarity Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 21(11):1629–1642, 2009.

[24] A. Ruiz-Alvarez and M. Humphrey. An Automated Approach to Cloud Storage Service Selection. In *ScienceCloud*, San Hose, CA, USA, 2011. ACM.

[25] S. Sarkar, R. Mahindru, R. A. Hosn, N. Vogl, and H. V. Ramasamy. Automated incident management for a platform-as-a-service cloud. In *Hot-ICE*, 2011.

[26] J. Schectman. Netflix Amazon Outage Shows Any Company Can Fail. http://blogs.wsj.com/cio/2012/12/27/netflix-amazon-outage-shows-any-company-can-fail.

[27] I. Smith. UnixBench. http://code.google.com/p/byte-unixbench/.

[28] SPEC. Java virtual machine benchmark 2008. http://www.spec.org/jvm2008/.

[29] Xiaolan Zhang. *Application-Specific Benchmarking*. PhD thesis, Harvard University.

[30] C. Zeginis, K. Konsolaki, K. Kritikos, and D. Plexousakis. Towards Proactive Cross-Layer Service Adaptation. In *WISE*, pages 704–711, 2012.

[31] C. Zeng, X. Guo, W. Ou, and D. Han. Cloud Computing Service Composition and Search Based on Semantic. In *CloudCom*, LNCS. Springer, 2009.