Author version of paper accepted at 2015 IEEE 8th International Conference on Cloud Computing:
http://dx.doi.org/10.1109/CLOUD.2015.14

Towards Automatic Application Migration to Clouds

Jorge Ejarque*, Andras Micsik† and Rosa M. Badia*‡
*Barcelona Supercomputing Center (BSC-CNS), Barcelona, Spain
†Computer and Automation Research Institute - Hungarian Academy of Sciences (SZTAKI), Budapest, Hungary
‡Artificial Intelligence Research Institute (IIIA), Spanish National Research Council (CSIC)
E-mail: {jorge.ejarque, rosa.m.badia}@bsc.es micsik@sztaki.hu

***Abstract*—Porting applications to Clouds is one of the key challenges in software industry. The available approaches to perform this task are basically either services derived from alliances of major software vendors and Cloud providers focusing on their own products, or small platform providers focusing on the most popular software stacks. For migrating other types of software, the options are limited to Infrastructure-as-a-Service (IaaS) solutions which require a lot of programming effort for adapting the software to a Cloud provider's API. Moreover, if it must be deployed in different providers, new integration procedures must be designed and implemented which could be a nightmare. This paper presents a solution for facilitating the migration of any application to the cloud, inferring the most suitable deployment model for the application and automatically deploying it in the available Cloud providers.**

***Index Terms*—Service Deployment, Cloud Migration, Cloud Computing, Platform-as-a-service, Cloud Interoperability.**

I. Introduction

The Cloud Computing [1] paradigm has become a revolutionary approach in distributed computing, providing computing and data resources in a dynamic and pay-per use model. This paradigm is becoming more attractive for different types of companies and institutions. We observe an increasing interest in exploring the potential benefits of moving partially or entirely their IT services and applications to Cloud infrastructures in order to decouple the provisioning and management of computing resources from their core business process in order to become more productive. However, migrating software to Clouds is not an easy task, since it requires a deep knowledge of the technology and the services offered by Cloud Providers. Among others, developers need to design how to partition the software into Virtual Machines (VMs), to develop how to build the VM images and how to deploy VMs and how to check the health status of the running VMs, and there are no easy solutions for this. During the last years, a new market of platform services has appeared to provide solutions to the aforementioned problems. The available platform services in the current Cloud market can be classified in two types: the ones derived from big alliances among the most important software vendors and major Cloud providers that offer software in an exclusive way; or small platform providers which focus just on well-known Model-View-Controller (MVC) frameworks. However, for other type of software, the offer is limited to simple platform services (such as simple batch job executions) or to basic infrastructure services which are complex to use. The adaptation of a tailored application to these services requires a lot of programming effort and a deep knowledge of Cloud technologies. Moreover, if the application requires to work with multiple providers, various integration procedures must be implemented, multiplying the effort. This could be a big burden for companies where IT management is not part of their core business.

The work presented in this paper aims to provide a platform that facilitates and automates the integration of applications in Cloud providers' infrastructures, lowering the barrier of Cloud adoption. To achieve this goal, our platform automatically finds the most suitable distribution and placement of software components on the different computing offerings. The key for achieving this goal is to define applications in a general-purpose and infrastructure-agnostic way, but providing the required information (e.g. communication links, quality, etc.) to automatically deduce how the different software components can be grouped, which component can be replicated or scaled and how they can be deployed to better adapt to the underlying heterogeneous distributed computing infrastructure. Once the suitable deployment is found, the platform automatically generates a plan to deploy it as a workflow of different Cloud providers API invocations, data transfers and process executions.

The paper is organized as follows. Section II presents the architecture of the platform describing the common ontology for application deployment and the different components in detail. Then, Section III provides a usage example to validate how a complex application can be described according the proposed ontology, and performs an evaluation in terms of the platform overhead and scalability. Section IV compares our approach with the related work and Section V draws the conclusions and propose guidelines for future work.

II. Architecture

The automatic migration of applications to Clouds is performed by the proposed system in two stages as depicted in Figure 1. In the first stage, given an application description, the system infers the most suitable deployment model and looks for the best placement according to the application properties and the available computing resources. The result of this stage provides a "what-if" scenario showing how the application will be distributed in heterogeneous clouds. Users can evaluate this solution, modify the application and repeat the process until a suitable solution is found. In the second stage, the system generates a workflow to provision the required resources and

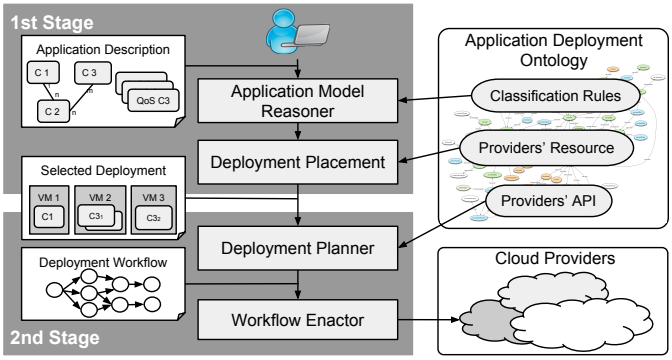


Fig. 1. Application Migration Framework Architecture

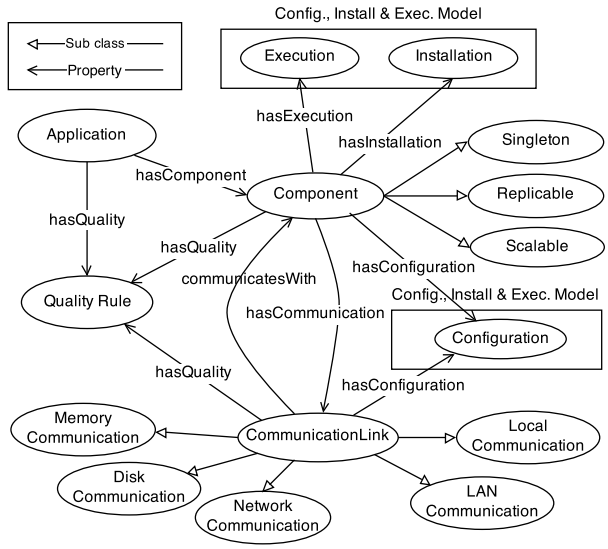


Fig. 2. Application Model

to deploy the application components according to the model and placement obtained in the first stage. The aforementioned migration processes are automatically performed by a combination of different Artificial Intelligence (AI) techniques detailed in the following paragraphs.

A. Application Deployment Ontology

The Application Deployment Ontology provides a common and machine-understandable model for sharing knowledge between the different components involved in the application deployment. This model is divided in two parts: a first part, which models the applications in an infrastructure-agnostic way (Figure 2); and a second part, which models Cloud providers describing their resources and the provided API to manage them (Paragraph II-A.II-A2).

1) Application Model: An application is mainly described as a component topology consisting of a set of components related by a set of links which describe how the components are intercommunicated. Each communication link is mainly defined by the type of communication (one-to-one, one-to-many, many-to-one or many-to-many) and the channel, which

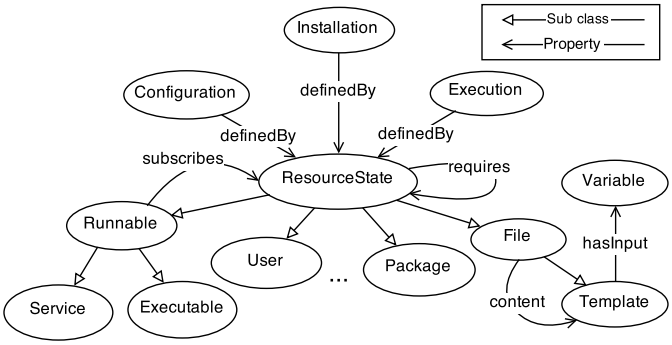


Fig. 3. Installation, Configuration and Execution Model

can be memory (e.g. libraries sharing objects, arrays, etc.), disk (e.g. processes which communicate by writing and reading files) or network (e.g. web services interchanging messages). Note, that communication links implicitly define a hierarchy, if a component depends to another one, it will be because there is component invocation by using one of the aforementioned communication channels.

In addition to the component topology, developers also have to define the required quality for the different components and communication links. These are described in the proposed Application Model as Quality Rules, where the required latency, bandwidth, processors, memory, storage or the number of component instances are inferred depending on the values of certain application or infrastructure metrics.

To finalize the application description, developers have to describe how the components and their dependencies are installed and configured. For describing this part (Figure 3), we have followed the resource state based model used by dev-ops tools such as Puppet [2] or LCFG [3](Large Scale Unix Configuration System). With this model, developers have to describe the installation, configuration and execution of a component as a set of resources (such as daemons, processes, files, packages, etc.) with a desired status (set of property-value pairs). They can also describe resource dependencies which force an execution order (with the *requires* property) or if a restart is required when a resource is updated (with the *subscribes* property). Section III will provide an example of how a complex application is described including the component topology, Quality Rules and the installation, configuration and execution descriptions.

In addition to the provided description, the Application Model also defines a hierarchy of components and communication link types and a set of description logic rules which enables the Application Model Reasoner to infer the best deployment model for an application. More details about this reasoning is described in Section II-B.

2) Infrastructure Model: Figure 4 gives an overview of the Infrastructure Model, which focuses on describing the computing resources offered by Cloud providers (VM types, storage, network). In the literature, we can find several models for describing computing resources such as [4], [5]. Therefore,

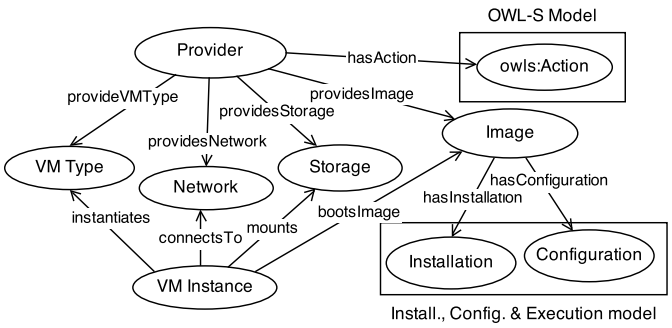


Fig. 4. Infrastructure Model

```
ec2:createInstance rdf:type owls:Action;
      owls:hasInput "?instanceType xml:String" ,
                    "?location xml:String";
      owls:hasOutput " ?instanceID xml:String" ;
      owls:hasPrecondition " ";
      owls:hasEffect " ?vm rdf:Type infra:VMInstance" ,
                     "?vm instantiates ?instanceType" ,
                     "?vm location ?location",
                     "?vm id ?instanceID",
                     "?vm status created";
```

Fig. 5. API Action Description Example

our contribution has focused on extending these models for including the description of images and the Cloud providers' APIs. In current approaches, these descriptions are provided in plain text which are hard to be processed by machines. For images, we propose to use the same resource state model used for the component installation and configuration. It provides a model for describing the current status of an image which can be easily processed by computers in order to check if an image totally or partially contains the resources required by the application components. On the other hand, we propose to model Cloud providers' API following OWL-S [6] concepts where different action are described by indicating the input and output parameters and the prerequisites and effects on the infrastructure state as variables and predicates. Figure 5 shows the description of the Amazon EC2 *createInstance* action with the defined parameters, preconditions and effects. This extended infrastructure model will be used by the Deployment Placer to compare the application requirements with the computing resources and by the Deployment Planner to describe the planning domain for the automatic generation of deployment plans. These processes will be explained in detail in Sections II-C and II-D.

B. Application Model Reasoner

Once a developer has described the application, the Application Model Reasoner classifies components and infers implicit affinity constraints. Applying description logic rules, the reasoner classifies the components as: *Singleton*, if the defined topology only allows a single component instance; *Replicable*, if the topology allows to have multiple copies, but the component and link configuration descriptions do not allow a run-time reconfiguration; or *Scalable*, if the topology allows multiple instances and a run-time reconfiguration is allowed. Once the components have been classified, the Application Model Reasoner applies the Quality Rules defined by inferring the communication quality for each link, the processing requirements for each component, as well as the number of instances in case the component is *Replicable* or *Scalable*. Finally, the reasoner applies another set of rules for inferring the component affinity constraints based on the inferred quality and the type of communication channels. These constraints will indicate which instances should be deployed in the same VM, in the same location, or which ones should share a disk. For instance, if the components have a memory communication or the required bandwidth and latency can be only achievable in a intra-host environment, the Application Model Reasoner includes a *sameVM* property between their components instances. In the case of required bandwidth and latency can be only achievable in a local environment, the Application Model Reasoner includes the *sameLocation* property and if the communication channel type is *Disk*, the Application Model Reasoner includes a shared disk and attach it to component instances. Finally, the Application Model Reasoner creates the deployment model for the application by grouping the component instances according to the *sameVM* and *sameLocation* properties and shared disks. This deployment model is passed to the Deployment Placer in order to assign these groups to the available provider resources.

```
PlacementSolution calculatePlacement (Set<Group> groups,
        Set<Providers> providers, long timeOutMillis) {
    long initTimeMillis = System.currentTimeMillis();
    PlacementSolution bestSolution = selectResources(groups, providers);
    PlacementSolution oldSolution, newSolution;
    boolean timedOut;
    do{
        Set<Group> nextEvalGroups;
        oldSolution = bestSolution;
        for( int i = 0; i<groups.length; i++){
            for( int j = i+1; j<groups.length; j++){
                Set<Group> newGroups = mergeGroups(i, j, groups);
                newSolution = selectResources(newGroups, providers);
                if (newSolution.cost <= bestSolution.cost){
                    bestSolution = newSolution;
                    nextEvalGroups = newGroups;
                timedOut = checkTimeOut(initTimeMillis, timeOutMillis);
            }
        }
        groups = nextEvalGroups;
    }while (newSolution != null && oldSolution.cost <= bestSolution.cost
            && !timedOut);
    return bestSolution;
}
```

Fig. 6. Placement algorithm code snippet.

C. Deployment Placer

The Deployment Placer aims at providing the best resource assignment for the application components according to the components processing requirements and the affinity constraints. In a first approach, we defined this problem as a constraint-satisfaction problem and using a state-of-the art solver we tried to get an optimal solution. However, the complexity of this solution tends to be exponential, so they are only tractable for a very small number of components and provider resources. For that reason, we decided to find the best sub-optimal solution which fulfills the application requirements but in an assumable period of time.

Figure 6 shows the code for calculating the placement

solution. First, the placer looks for a first placement solution by selecting: the cheapest VM for each atomic component group defined by the affinity constraints; the cheapest required disks and networks which have been derived from the communication links; and the suitable images available. This selection is solved in linear time by applying simple comparisons. Then, after getting the first solution, the placer tries to get a better solution by merging the groups and selecting a new VM, image, disk and network for this merge. If the merged solution has a lower cost it is assigned as best solution and will be the input for the next iteration. The same process is performed until we do not find a solution with a lower cost, no more VM groups can be merged or the search time-out has been reached. The complexity of this second phase is polynomial, because the maximum iterations in the loops is the number of VM groups. Once the Placer has provided a deployment solution, it can be evaluated by developers. In case they want to see the deployment with a different communication or configuration or quality level, they just need to modify the application description accordingly and repeat the first stage. Once a suitable deployment solution is found, it will be automatically deployed in the Cloud providers during the second stage.

D. Deployment Planner and Enactor

The Deployment Planner is in charge of finding a workflow to provision the required resources (VMs, shared disks and networks), configuring the communication links and installing, configuring and running the components on the provisioned resources. The deployment placement solution obtained in the first stage can be represented as a desired infrastructure state, and the providers' actions can be modeled as state transitions. Therefore, generating the deployment workflow can be obtained as a solution of a state-space search performed by AI planners. Applying this idea, the Planner generates a planning domain from the Infrastructure Model and a planning problem with an empty initial state and with the placement solution as goal state. Then, the domain and problem are introduced to a Partial Order Planner which performs the search and provides a sequence of actions which produces the goal state from the initial state. For the installation, configuration and execution of components, the Planner will inspect the resource status of the images assigned to each VM and compares it with the description of the components assigned to these VMs, generating a dev-ops manifest for automatically reaching the missing resource states. Finally, the Workflow Enactor executes the generated deployment workflow invoking the required actions of the Cloud providers' API for the resource provisioning and applies the manifests to install, configure and run the components on the cloud resources.

III. Evaluation

A working prototype of the described system has been implemented to validate the concepts presented in the paper. The Application Deployment Ontology has been described using OWL2 [7] and the inference rules has been described with SWRL [8]. Pellet [9] has been used to perform the ontology and rule reasoning described in the Application Model Reasoner. Planning4J [10] has been used to generate the planning domain and problem, which is solved using the FF planner [11]. The prototype has been deployed in an Intel i5 laptop with 8GB RAM and we have evaluated it in two parts. In the first part, we use the prototype to migrate the KOPI application to Amazon EC2 in order to validate the system capabilities and to illustrate the system behavior. In the second part, we evaluate the system overhead and its scalability.

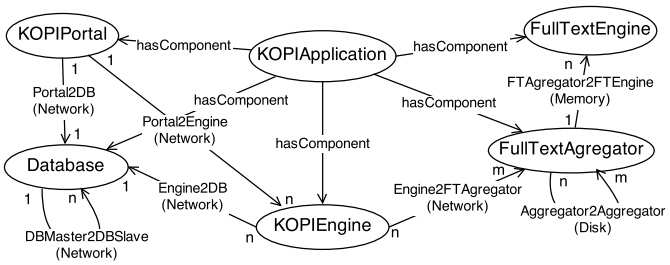


Fig. 7. KOPI Application Overview

A. Use Case

The KOPI application [12] is an On-line Plagiarism Search Portal developed by SZTAKI which implements an innovative cross-language plagiarism detection technique. It gives the opportunity to compare a reference document to other indexed collections of documents and search for potentially translated parts. This new technique is costly in terms of processing and data storage; therefore we sought the service is a potential candidate to be migrated to the cloud. Figure 7 shows the component topology of the KOPI application and Figure 8 shows a snippet of the application description. The application contains five components: the *KOPI Portal*, where users upload the documents to check; the *KOPI Engine*, which is in charge of managing the document checking life-cycle; a *Database* to store documents and plagiarism check results; and the *Fulltext Aggregator* and *Fulltext Engine* to perform various indexed search sub-tasks for *KOPI Engines*. For each of these components, developers have to define the communications with other components, the Quality Rules and the installation, configuration and execution.

Figure 9 shows some examples of the communications defined for the KOPI application. Each description includes the type of communication (one-to-one, one-to-many, etc.), its channel, the component which communicates with, and the Quality Rules.

Quality Rules define how to infer the quantity of resources consumed by a component or communication link according to different metrics. In this case, the KOPI Quality Rules are expressed as a function of the desired *CharsPerSecond* (document processing speed), *NumberOfDocs* (the number of simultaneous documents to be evaluated) and *IndexSize* (number of documents to be compared) metrics. To describe this functions, we have reused the approach described in [13]

```
:KOPIApplication rdf:type app:Application ;
    app:hasComponent :FulltextEngine , :FulltextAgreegator ,
                     :KOPIDatabase , :KOPIEngine , :KOPIPortal .
:KOPIPortal rdf:type app:Component ;
    app:hasCommunication :PortalDBCommunication ,
                         :PortalEngineCommunication ;
    app:hasConfiguration :PortalConfig ;
    app:hasInstallation :PortalInstall ;
    app:hasExecution :PortalExec ;
    app:hasQuality :PortalQuality .
:KOPIDatabase rdf:type app:Component ;
    app:hasCommunication :MasterSlaveCommunication ;
    app:hasInstallation :DBInstall ;
    app:hasConfiguration :DBConfig ;
    app:hasExecution :DBExec ;
    app:hasQuality :DBQuality .
:KOPIEngine rdf:type app:Component ;
    app:hasCommunication :EngineAgregatorCommunication ,
                         :EngineDBCommunication ;
    ...
:FulltextAgreegator rdf:type app:Component;
    app:hasCommunication :FTAgregatorAgregatorCommunication ;
                         :FTAgragatorEngineCommunication ;
    ...
:FulltextEngine rdf:type app:Component ;
    ...
```

Fig. 8. Component Description Snippet.

```
:MasterSlaveCommunication rdf:type :CommunicationLink ;
    app:numberDestinationInstances "many" ;
    app:numberSourceInstances "one" ;
    app:communicatesWithComponent :KOPIDatabase ;
    app:hasChannel :Network ;
    app:hasSourceConfiguration :MasterSlaveConfiguration ;
    app:hasQuality :DBToDBQuality .
    ...
:FTAgragatorEngineCommunication rdf:type app:CommunicationLink ;
    app:numberSourceInstances "one" ;
    app:numberDestinationInstances "many" ;
    app:communicatesWithComponent :FulltextEngine ;
    app:hasChannel app:Memory ;
    app:hasQuality :AggregatorToEngineQuality ;
    app:hasSourceConfiguration :AggregatorToEngineConfig .
    ...
:FTAgregatorAgregatorCommunication rdf:type app:CommunicationLink ;
    app:numberDestinationInstances "many" ;
    app:numberSourceInstances "many" ;
    app:communicatesWithComponent :FulltextAgreegator ;
    app:hasChannel app:Disk ;
    app:hasQuality :AggregatorToAggregatorQuality ;
    app:hasSourceConfiguration :AggregatorToAggregatorConfig ;
    app:hasDestinationConfiguration :AggregatorToAggregatorConfig .
    ...
```

Fig. 9. Component Communication Description Examples.

which embeds the mathematical formulas in RDF descriptions. Figure 10 shows a Quality Rule example for the *Fulltext Engine*, where the number of instances and their assigned hardware (cpu, memory, disk) are expressed as functions of the mentioned metrics. Following the same approach, users can define Quality Rules for communication links to describe bandwidth and latency requirements.

```
:FTEngineQuality rdf:type app:QualityRule ;
    app:hasMetric :CharsPerSecond, :IndexSize, :NumberOfDocs ;
    app:numberInstances "(:NumberOfDocs * :CharsPerSecond)/200" ;
    app:needsCores ":CharsPerSecond/10" ; ## numCores
    app:needsRAM "4 * (:IndexSize/100) * :CharsPerSeconds" ; ## MB
    app:needsDisk "5 * :IndexSize/1024 " . ## GB
:CharactersPerSecond rdf:type app:Metric ;
    app:source"file://etc/fulltext/config.property?property=cps" ;
    app:goalValue 20 .
:IndexSize rdf:type app:Metric ;
    app:source"file://etc/fulltext/config?property=indexsize" .
    app:goalValue 10000 .
:NumberOfDocuments rdf:type app:Metric ;
    app:source "http//:KOPIPortal[0].vm.ip/documents" .
    app:goalValue 100 .
```

Fig. 10. Component Quality Description Examples.

```
:DBInstall rdf:type app:Installation ;
    app:hasState :MySQLPackage, :DBFile .
:MySQLPackage rdf:type app:Package ;
    app:name "mysql" .
    app:ensures "installed" .
:DBFile rdf:type app:File ;
    app:requires :MySQLPackage" .
    app:location "/usr/mysql/dbs/" ;
    app:name "kopi.db".
    app:source "http://....";
    app:ensures "exists" .
:DBConfig rdf:type app:Configuration ;
    app:hasState :MySQLConfigFile .
:MySQLConfigFile rdf:type app:File ;
    app:location "file:///etc/mysql/" ;
    app:source "http://....";
    app:name "mysql_config" .
    app:ensures "exists" .
:DBExec rdf:type app:Execution ;
    app:hasState :MySQLService .
    app:requires app:MySQLPackage
    app:subscribes app:MySQLConfigFile
    app:ensures "started" .
```

Fig. 11. Component Installation, Configuration and Execution Description Example.

```
:MasterSlaveConfiguration rdf:type app:Configuration ;
    app:hasState app:MySQLConfigFile .
:MySQLConfigFile rdf:type app:File ;
    app:hasContent :MasterSlaveTemplate .
:MasterSlaveTemplate rdf:type app:Template ;
    app:hasInput :MasterDBVariable ,
                 :SlavesDBVariable .
:MasterDBVariable rdf:type app:Variable ;
    app:hasValue $:KOPIDatabase[0].host.ip$ ;
    app:name "master-node" .
:SlavesDBVariable rdf:type app:Variable ;
    :hasValue $:KOPIDatabase[1...].host.ip$ ;
    :name "slaves-nodes" .
```

Fig. 12. Communication Configuration Description Example.

The application description is finalized by providing the installation, configuration and execution states for component and communication links. Figure 11 provides an example of such descriptions for the *Database* component and its self-communication link. The Database component requires to have the *mysql* package and the database file installed, its configuration requires a *mysql_config* file defined, and its execution requires the *mysql* service started. For the case of the link configuration state (Figure 12), it requires to have a *mysql_config* file, whose content should match with a template that contains the master and slave addresses as input parameters. Note, that the *subscribed* property defined in the *DBExec*, indicates that the *mysql* service must be restarted each time the *mysql_config* file is updated. So, if the number of instances is modified, the file will be updated and the service will be restarted. This patterns are searched by the Application Model Reasoner to infer if a component is just *Replicable* or dynamically *Scalable*.

Once the application is defined, the user submits the application description in the system together with the desired supported values for *NumberOfDocuments, IndexSpace* and *CharactersPerSecond*. During the first step, the KOPI components are classified as described in Section II-B. As a result of the classification, the Application Model Reasoner classifies the *KOPI Portal* as *Singleton* because the topology only allows to have an instance. The rest can have several instances, but the Application Model Reasoner classifies the *Database* as *Replicable* because of the configuration-installation pattern

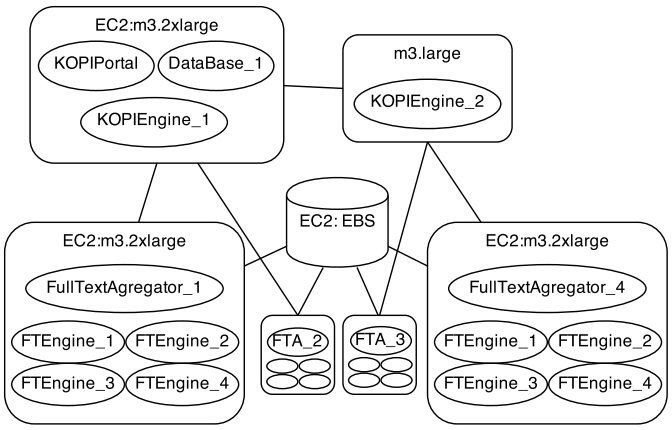


Fig. 13. KOPI Application Placement

```
sequence{
    action{ ec2:createVolume
        input{ size = 50; location = us-east;};
        output{ volumeID = ?ID_2;};};
    action{ ec2:createInstance
        input{ type = m3.2xlarge; location = us-east;};
        output{ instanceID = ?ID_4;};
    action{ec2:attachVolume
        input{volumeID = ?ID_2; instanceID= ?ID_4;};}
    action{ec2:startInstance
        input {instanceID= ?ID_4;};};
    ...}
```

Fig. 14. KOPI Deployment Workflow Snippet.

explained in the previous paragraph. As the rest of the components do not follow the same pattern, they are classified as *Scalable*. In this first step, the Application Model Reasoner also applies the Quality Rules for the desired metric values inferring the hardware, bandwidth and latency requirements as well as the number of instances per component. Evaluating these requirements and the communication channels types, the Reasoner deduces that *Fulltext Engines* must be located in the same VM as *Fulltext Aggregators* because of the memory channel; and *Aggregator* instances must be located in the same location because of the required bandwidth and latency of the disk communication link is not assumable by a wide-area network. In the second step, the Deployment Placer provides a solution for placing the different components in the Amazon EC2 resources as explained in Section II-C. As a result of this part, the system provides the placement depicted in Figure 13, and passing this placement to the Deployment Planner, it returns a plan for deploying the KOPI application in EC2. This plan is composed by a sequence of provider API calls to provision the computing resources (Figure 14), and a set of Puppet manifests for deploying the components in each VM.

B. Overhead and Scalability

Table I, shows the time spent by the system to get the solution for the KOPI use case, distributed as follows: 12 ms for the work performed by the Application Model Reasoner where is has evaluated the 6 components descriptions, 30 ms for getting a placement solution of the required component solution of 24 instances in 6 VMs and 1 Shared disk, and

TABLE I
TIME TO GET THE MIGRATION SOLUTION FOR KOPI APPLICATION

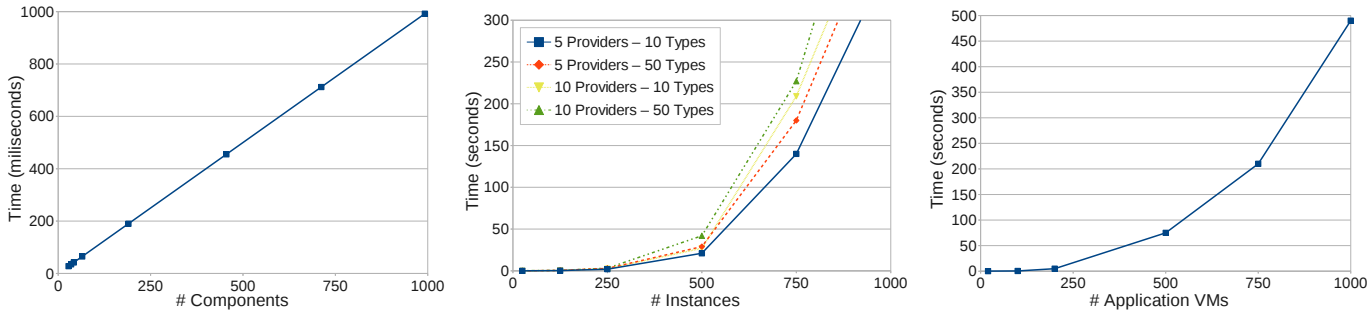
Main Action	**Time**
Application Model Reasoning	12 ms
Placement Solution	30 ms
Deployment Planning	65 ms
Total	112 ms

65 ms for getting the workflow which deploys the planning solution. To evaluate how the system performs for larger applications, we have generated synthetic application descriptions with different number of components, instances and providers with different number of resources, capabilities and costs. We have measured the time spent by our system in each phase. Figure 15(a) shows the time spent by the Application Model Reasoner for performing the reasoning with different number of components. As we can see in the image, the overhead grows linearly with the number of components, and the overhead is low even for large number of components and can be neglected compared to the overhead introduced by the Placer and Planner (Figures 15(b) and 15(c)). These Placer and Planner processing times grow polynomially with the number of instances and VMs which are closely related to the number of components defined in the application. Moreover, in the case of the Placer, the time grows slower or faster depending on the number of providers and VM types. In terms of time scale, inferring the deployment for large applications whose deployment requires hundreds of instances and VMs is obtained in several seconds or few minutes. This is acceptable for the users compared to the time for deploying applications in the Cloud.

IV. DISCUSSION AND RELATED WORK

In the current market place we can found several options to facilitate the application deployment at multiple providers. A first group of solutions are based on common interface, which could be used by developers, and a set of plug-ins which implements the access to different Cloud providers. There are several examples of this approach such as Apache jClouds [14] or OCCI [15]. These solutions work quite well for experienced developers migrating simple applications, but not for common software developers migrating complex applications. The learning curve for using IaaS is slow and a complex application deployment requires to learn how to do several provisioning and configuration tasks. Moreover, maintaining these solutions is also complex, because a change in the Cloud providers' API requires at least a change in the interface plug-ins and potentially a change in the common interface definition and all its plug-ins.

Another option to deploy applications are platform services, such as Heroku [16], Google App Engine [17] or Amazon CloudFormation [18]. These platform services have two limitations: first, they focus on a set of applications, traditionally MVC applications developed with J2EE, Django, Ruby on Rails, etc. If the application is not using any of the predefined stacks, the user is not able to use the platform; and



(a) Application Model Reasoner (b) Deployment Placer (c) Deployment Planner

Fig. 15. Overhead introduced by the system for different number of components, instances, and VMs

secondly, they also hide the underlying infrastructure, binding to work only with the Cloud provider that the platform has an agreement with. So, at the end, developers have again a provider lock-in but in another layer of abstraction. Regarding platforms for multi-cloud deployment, we can find two types of solutions: open platforms such as Cloud Foundry [19] or Cloudify [20], which provide a set of core capabilities and services to enable the deployment of applications to different Cloud providers; or a broker solution such as CloudPier [21], proposed by the Cloud4SOA project, which allows users to find platform services, which are compatible with their application stack. The common problem in both types of solutions is that, at the end, they only provide easy deployment and adaptation functionality for the same type of MVC applications as other platforms. In the open-platform cases, users could try to extend the platform to support their application stack but it will be similar than trying to use directly a common interface. This option also inherits the problem of having to change the implementation of the upper layer services as a consequence of a change in the Cloud providers' API.

Other solutions proposed to improve Cloud interoperability are based on extensible and machine-processable models to describe cloud resources and applications. The main idea behind it, is the use of these models to inter-operate with the cloud services and resources in the different phases of the application life-cycle. The benefits of this idea are that models are not bound to a specific implementation and can be easily extended and used to automate processes. The most popular Cloud application model is the Open Virtualization Format (OVF) [22]. This format is supported by several hypervisors and cloud middleware. It defines an application as a set of Virtual Systems which is very close to the infrastructure details. Therefore, it is attractive for experienced cloud developers and system administrators, but not productive for general software developers. The mOSAIC project [5] proposed an application model based on design patterns for parallel applications. For each pattern or application, developers must define a set of rules which map the application to infrastructure level resources, and these rules are used to deploy the application. The most extensive model for Clouds is the CloudML [23] which is developed in conjunction by the projects REMICS [24], MODAClouds [25] and PaaSage [26]. The difference with mOSAIC is that instead of matching the application with rules, they define a language to specify the deployment of the application in the cloud resources. The common drawback in those approaches is that developers must explicitly specify the details of how the application is deployed in the cloud according to the infrastructure model. Describing this deployment model is better than implementing the adaptations to the Cloud, but it is still too complex and remains as a main barrier to achieve a productive migration to Clouds.

Comparing these solutions with our system, it automates the full deployment process from a generic infrastructure-agnostic application description which supports any type of application or software stack. Developers do not need to specify the cloud deployment and placement, it is automatically inferred for any Cloud provider without user-intervention. So, it could be a perfect complement for model-based solutions such as CloudML, because it can generate the deployment model description from any application description. In addition, our proposal improves the current infrastructure models by including image descriptions as well as the component installation, configuration and execution descriptions in a machine-understandable way. These descriptions are then required to automatically infer how the provider VM images must be modified to include the application components. These are complex concepts and they are not tackled by the models mentioned above. Finally, in our approach, changes in the Cloud providers API will not require changing neither applications nor system services. As mentioned before, the deployment workflow is automatically generated on-demand according to the API description. So, a change in the provider services will be considered by the system the next time a deployment is requested, generating a new workflow which includes the new version of the API.

V. Conclusions and Future Work

We have presented an approach which combines different Artificial Intelligence techniques for performing automatic migration of distributed applications to the Cloud. Starting from a generic application model which defines components and their communication links, the system applies ontology

and rule reasoning to classify components and infer their affinity. After the first step, the system also searches for the best components-to-VMs assignment which fulfills the affinity constraints and resource requirements. Finally, from the deployment placement solution and the providers' API description, the system builds a planning problem whose solution will provide the workflow for provisioning the virtual resources as well as installing, configuring and executing the components assigned to each VM and setting up the communication links.

We have implemented a prototype and we have validated the concept with the KOPI application, which provides a portal for plagiarism detection. We have measured the overhead introduced by the system to perform the aforementioned tasks. We have seen the overhead is acceptable compared to the cloud deployment time and it is considerably smaller than the time spent by a developer to perform the same tasks. Finally, we have discussed the differences with the current deployment approaches and the benefits that our system provides compared to other solutions.

As future work, we see several points of improvement. For simplicity, we have made Quality Rule descriptions per component or communication, but it would also be interesting to support application-wide non-functional Quality Rules such as the availability and reliability which are closely related with the number of instances the application topology and the affinity constraints. Extending the current rules, we could infer the effect that a required availability and reliability has on the deployment model by setting extra affinity constraint or a minimum number of instances. Also regarding Quality Rules, developers have to currently provide the rules to infer number of instances and their computing requirements depending on a system load and quality metrics. However, some of these rules could be automatically deduced by applying machine learning methods, as we did in [27] and [28], to estimate the number of instances and computing requirements from historical data which monitored the quality metrics for different resource usage and system load.

Finally, another point of improvement is extending the automatic workflow creation. It is currently limited to provide the resource provisioning and to create the manifests for the component installation and configuration. However, recent cloud solutions also offer advanced services such as monitoring and scalability, image management and software licensing. The current Deployment Planner could be easily extended to include the workflow to setup these services on the deployment plan, enabling run-time adaptation and automatic management of software licenses.

VI. Acknowledgment

This work has been supported by the Spanish Government under contract TIN2012-34557 and grant SEV-2011-00067 and Generalitat de Catalunya under contract 2009-SGR-980.

References

[1] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: Towards a cloud definition," *ACM SIGCOMM Computer Communications Review*, vol. 39, no. 1, pp. 50–55, 2009.
[2] PuppetLabs, http://puppetLabs, last access May 2015.
[3] Large Scale Unix Configuration System, http://www.lcfg.org, last access May 2015.
[4] Distributed Management Task Force, "Common Information Model v.3.0," DSP0004, 2013.
[5] mOSAIC Project, http://www.mosaic-cloud.eu/, last access May 2015.
[6] D. Martin, et al, "OWL-S: Semantic markup for web services," *W3C Submission*, 2004.
[7] B. Motik, et al, "OWL 2 Web Ontology Language," W3C Recommendation, 2012.
[8] I. Horrocks, et al, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," W3C Submission, 2004.
[9] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical owl-dl reasoner," *Web Semantics: science, services and agents on the World Wide Web*, vol. 5, no. 2, pp. 51–53, 2007.
[10] Planning4J - A Java API for planning, http://code.google.com/p/planning4j, last access May 2015.
[11] B. Nebel, "The FF Planning System: Fast Plan Generation Through Heuristic Search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.
[12] A. Micsik, P. Pallinger, and D. Siklósi, "Scaling a plagiarism search service on the bonfire testbed," in *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science*, vol. 2, 2013, pp. 57–62.
[13] K. Wenzel and H. Reinhardt, "Mathematical Computations for Linked Data Applications with OpenMath," in *Proceedings of the 24th Workshop on OpenMath*, 2012, pp. 38–48.
[14] Apache jClouds, http://jclouds.apache.org/, last access May 2015.
[15] Open Cloud Computing Interface, http://occi-wg.org/, last access May 2015.
[16] Heroku, http://www.heroku.com/, last access May 2015.
[17] Google App Engine, http://appengine.google.com/, last access May 2015.
[18] Amazon Cloud Formation, http://aws.amazon.com/cloudformation/, last access May 2015.
[19] Cloud Foundry, http://cloudfoundry.org, last access May 2015.
[20] Cloudify, http://getcloudify.org/, last access May 2015.
[21] Cloud Pier, http://www.opencloudpier.org, last access May 2015.
[22] Distributed Management Task Force, "Open Virtualization Format Specification," DSP0243, 2013.
[23] Cloud ML, http://cloudml.org/, last access May 2015.
[24] Remics Project, http://www.remics.eu/, last access May 2015.
[25] MODAClouds Project, http://www.modaclouds.eu/, last access May 2015.
[26] PaaSage Project, http://www.paasage.eu/, last access May 2015.
[27] J. Ejarque, A. Micsik, R. Sirvent, P. Pallinger, L. Kovacs, and R. Badia, "Semantic resource allocation with historical data based predictions," in *Proceedings of the 1st International Conference on Cloud Computing, GRIDs, and Virtualization*, 2010, pp. 104–109.
[28] X. J. Collazo-Mojica, S. M. Sadjadi, J. Ejarque, and R. M. Badia, "Cloud application resource mapping and scaling based on monitoring of qos constraints." in *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering*, 2012, pp. 88–93.