Automated Setup of Multi-Cloud Environments for Microservices Applications

Gustavo Sousa, Walter Rudametkin, Laurence Duchien

Automated Setup of Multi-Cloud Environments for Microservices-Based Applications

Gustavo Sousa, Walter Rudametkin, Laurence Duchien
Université de Lille / CRIStAL UMR 9189 / École Centrale de Lille
Inria
Lille, France
Email: firstname.lastname@inria.fr

***Abstract*—Multi-cloud computing has been proposed as a way to reduce vendor dependence, comply with location regulations, and optimize reliability, performance and costs. Meanwhile, microservice architectures are becoming increasingly popular in cloud computing as they promote decomposing applications into small services that can be independently deployed and scaled, thus optimizing resources usage.**

However, setting up a multi-cloud environment to deploy a microservices-based application is still a very complex and time consuming task. Each microservice may require different functionality (e.g. software platforms, databases, monitoring and scalability tools) and have different location and redundancy requirements. Selection of cloud providers should take into account the individual requirements of each service, as well as the global requirements of reliability and scalability. Moreover, cloud providers can be very heterogeneous and offer disparate functionality, thus hindering comparison.

In this paper we propose an automated approach for the selection and configuration of cloud providers for multi-cloud microservices-based applications. Our approach uses a domain specific language to describe the application's multi-cloud requirements and we provide a systematic method for obtaining proper configurations that comply with the application's requirements and the cloud providers' constraints.

***Index Terms*—multi-cloud; microservices; cloud management; variability management; software product lines**

I. Introduction

Multi-cloud computing is the use of resources and services from multiple cloud providers where there is no agreement between providers to offer an integrated view of the system [1]. It is up to the customer or a third-party to integrate services from different providers. Multi-cloud computing is a way to avoid vendor dependence and to better exploit offerings in the cloud market by employing a combination of public and private cloud resources. Customers can build configurations that better fit their needs while reducing their dependence on any given provider.

The microservices architectural style [2] provides an approach for building scalable applications. In a microservices architecture, applications are composed of small services that run in separate processes and can be deployed to different environments. Besides the benefits from separating concerns and of modularization, this approach scales services independently, allowing better use of resources.

We argue that these two approaches have strong synergism. On the one hand, building a multi-cloud application involves deploying to different clouds. Having independent well-isolated services facilitates this. On the other hand running microservices across private and public clouds from different providers allows for improved scalability and reliability.

However, building a multi-cloud solution is complex. Customers must consider factors such as functional requirements, configuration options, pricing policy, data center location, availability levels, etc. Cloud providers have different management interfaces and use different concepts for their offerings, making them difficult to compare. In addition, each of the application's services may have different requirements. They may be written in different programming languages, using different application frameworks or databases, and may have different scalability and availability requirements. Thus, a cloud provider that supports one of the application's services might not support another. Also, local regulations may require data to be stored within a given region, imposing further constraints.

We propose an automated approach for the selection and configuration of cloud providers for multi-cloud microservices-based applications. Our approach relies on ontology reasoning [3] and software product line techniques [4] to get from a high-level description of multi-cloud requirements to a configuration for a multi-cloud environment. While some approaches deal with aspects of multi-cloud management [5], [6], [7], [8], [9], they do not take into account the rich variability that exists in cloud providers' configuration options, losing valuable insight. Work done to manage variability in the cloud [10] does not consider multi-cloud requirements. Our approach goes further than existing work by dealing with multi-cloud requirements for microservices-based applications, while considering provider configuration variability.

Our key contributions are:

- An approach to build multi-cloud environments that handles the intrinsic variability in existing providers.
- A method for translating high-level multi-cloud requirements to a set of provider specific feature model configurations.
- Domain specific languages for specifying an application's multi-cloud requirements.
- An experimental assessment of the approach using an example application and four popular cloud providers.

In Section II we discuss the motivation for multi-cloud computing and issues identified when setting up multi-cloud environments. In Section III we present our approach, and in Section IV we discuss its implementation. Finally, we discuss related work in Section V and conclusions in Section VI.

II. Motivation

Motivations for using multiple clouds have been discussed in recent research [11], [12]. A list of 10 reasons were identified in a survey conducted by Petcu [13]. We classify these motivations into three broad categories:

- *Reducing vendor dependence*. This includes protecting applications from cloud outages or failures, but also avoiding vendor lock-in.
- *Optimizing performance and costs*. Different cloud providers offer specific advantages. Combining them can improve QoS or costs. Multi-clouds can also be used to offload processing to other clouds when dealing with peaks, or to deploy services closer to the end user.
- *Complying with requirements, constraints or regulations*. In case no cloud satisfies all of the applications requirements, combining clouds becomes mandatory. Also, regulations may require sensitive data to be stored in a particular location, such as the user's country.

Multi-cloud applications exist for different reasons and approaches to manage them should take this into account.

To illustrate the complexity faced when building a multi-cloud system, we introduce an example scenario of an e-commerce application that requires the use of multiple clouds. This application is composed of the following services:

- a *product catalog* that keeps a record of product information and inventory;
- a *recommendation service* that records purchases and recommends new products for customers;
- a *user management* service that manages a *user database* and a *credit card database*;
- an *order management* service that tracks orders and their statuses;
- a *payment* service responsible for interacting with banking and credit card partners;
- as well as a *web frontend* and a *mobile API gateway* to provide access to end users' devices.

Each of these services may be developed independently, using different technologies. Therefore, each service has different functional requirements. For instance, while the *web frontend* is a web application written in Java, the *recommendation service* is a standalone Python application with an embedded database. Moreover, according to its development process, a service may also require extra tools for continuous integration, issue tracking, staging resources, etc.

In addition to functional requirements, this example scenario should also meet the following:

- instances of *web frontend* and *product catalog* should be deployed to at least two different providers to ensure essential services are always available, even in the case of a provider failure.
- data concerning European customers should be stored in a country that is a member of the European Union.
- the *credit card database* should be kept in a private cloud controlled by the company.

Apart from these requirements, there are no further restrictions on application services. Thus, when setting up a multi-cloud environment to deploy this application, we will be looking to optimize costs or improve the quality of service while complying with application requirements and constraints.

Even from this simple example we can see that taking all relevant factors into account for setting up a multi-cloud environment can be a very complex task, which calls for automation and supporting tools.

In this paper, we present an automated approach for the selection of cloud providers and generation of proper configurations that considers the following concerns:

Cloud providers heterogeneity: Cloud providers usually offer functionality as services at different levels of abstraction such as infrastructure (IaaS), platform (PaaS) and software (SaaS). Even at the same abstraction level, providers employ varying terminology and features which usually do not map directly those of competing providers. These differences complicate further the comparison between providers.

Cloud providers variability: Cloud providers may have complex rules concerning their configuration. According to the provider, some functionality may only be available in a given region or for a given user plan. Some features may have conflicts while other may have dependencies. These as well as other complex constraints can be found in cloud providers and should all be considered to obtain a proper configuration.

Multi-cloud requirements for microservices-based applications: Microservices can be developed by different teams, relying on different technologies and methodologies and may therefore require functionalities at different levels of abstraction. In a multi-cloud environment, microservices can be deployed across private and public clouds from different providers to implement scalability and redundancy mechanisms or to comply with location constraints.

III. Approach

Our approach to deal with the concerns identified in Section II relies on (i) a high-level domain-specific language for describing multi-cloud requirements; (ii) feature models to manage the variability in cloud providers; and (iii) ontologies for dealing with the heterogeneity across cloud providers.

Our approach, depicted in Fig. 1, makes a distinction between the roles of *cloud provider experts* and *developers*. A *cloud provider expert* is responsible for providing formal descriptions of cloud providers' offerings through feature models and ontology mappings. On the other hand, *developers* interact with the approach by providing requirements of a multi-cloud environment. These requirements are matched against providers' descriptions to generate a multi-cloud configuration,

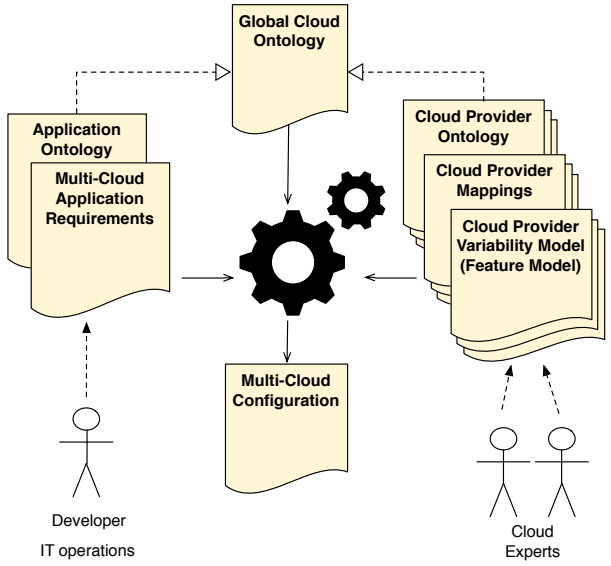

Fig. 1. Overview of the actors and artifacts involved in the proposed approach

which includes a selection of clouds providers and information on how to configure them for deploying the application.

A. A DSL for multi-cloud application requirements

We propose a language for defining multi-cloud requirements based on three main constructs: *service requirements*, *cloud variables* and *instance groups*. The *service requirements* are intended to describe the functionality required by a service to run. These can be virtual machines, runtime environments, databases, software management tools, or any other feature that is offered by a cloud provider. Fig. 2 depicts part of the requirements of the example given in Section II. For instance, the *web frontend* service requires a **`WebContainer`** with at least 2 GB of RAM that has a **`JavaWebContainer`** that implements version 2.5 of the Servlet API.

The concepts mentioned in service requirements are part of a global cloud ontology. If a concept is not in the ontology, the developer can use the application specific ontology to include new concepts. For example, one could add a new ontology class called **`HighPerformanceWebContainer`** and express that it is equivalent to a **`WebContainer`** that has more than 4 virtual CPUs. Once a new concept is added it can be used as part of a service requirements description.

The *cloud variables* describe conditions on the clouds where the services are to be deployed. This allows specifying requirements, such as, instances of a service should be placed in clouds maintained by different providers, or, that a given service should be deployed to a cloud located in a specific region. In Fig. 2, we have a condition that cloud *B* should be located in **`WesternEurope`** and that cloud *A* should be a different provider than *B*.

Finally, *instance groups* describe where to deploy services and how they will be instantiated. This construct lets developers specify constraints on the number of instances at the cloud or global levels. In our example, the *web frontend* service will

```
/* Service requirements */
service web_frontend {
  requires WebContainer {
    memory 2GB
    software JavaWebContainer {
      providesAPIs Servlet { version 2.5 }
    }
  }
}
service recommendation {
  requires VirtualMachine {
    memory 4GB
    operatingSystem Windows { version >= 7 }
  }
}
service user_db {
  requires PostgreSQL  { version >= 9 }
}
/* Cloud variables */
cloud A { self.provider <> B.provider }
cloud B { self.location = WesternEurope }
/* Instance groups */
instanceGroup  web_frontend {
  global { numInstances 1..20 }
  A { numInstances 2..19 options AutoScale }
  B { numInstances 1..5 options AutoScale }
}
instanceGroup user_db {
  B { role MasterDB }
}
instanceGroup recommendation {
  global { singleton }
}
```

Fig. 2. Sample multi-cloud requirements

have instances running in both clouds *A* and *B*; the *user db* database will be located in cloud *B*; and the *recommendation* service will have a singleton instance running in any cloud.

In essence, the proposed language enables developers to define requirements for each service individually, relying on concepts from the cloud ontology. The ontologies are extensible and allow defining complex requirements involving logical operators and composition rules. Finally, cloud variables and instance groups can be used to define scalability and redundancy rules across providers or regions, as well as location and colocation rules for services.

B. Feature models for managing cloud variability

Feature models are widely used to model commonalities and variability across a software product line or system family [14]. A feature model is usually depicted as a tree diagram whose nodes represent features that can be selected to build a software product. While the tree hierarchy describes a composition relationship between features, additional constraints allow for defining extra relationships between features.

In our approach, feature models are used to model the variability in the configurations of a given cloud provider. This means that a feature model describes what are the services and resources offered by a given provider, how they are related and what are the constraints between them.

Fig. 3 depicts part of a feature model elaborated to describe the variability in the OpenShift PaaS provider. OpenShift

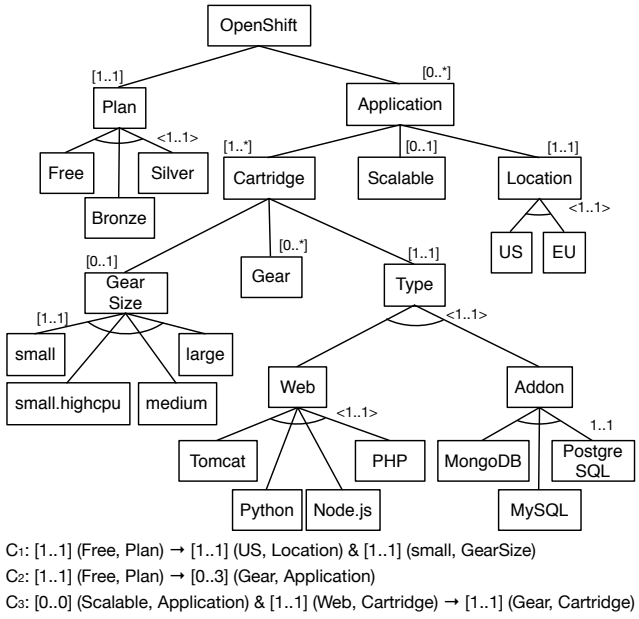

Fig. 3. Feature model for OpenShift PaaS

allows for configuring multiple *Application*s, which are composed by one or more *Cartridge*s. In the OpenShift PaaS, a *Cartridge* defines an executable container, which can be either of type *Web* or *Addon* and can have many *Gear*s (the running instances of the container) of a selected *Gear Size*. This feature model also describes additional constraints such as C_1, expressing that when the *Free Plan* is chosen all applications should be deployed in the *US* and use only *small* size *Gear*s.

The feature models employed in our approach, as the one shown, are extended with cardinalities and additional constraints. Therefore, we rely on previous developments done on cardinality-based feature models [15] and on modeling cloud providers with feature models [16]. For expressing relative cardinalities and the complex constraints found in cloud providers, we also rely on the results of our previous work on extending feature models with relative cardinalities [17].

A relative cardinality is the number of instances of given feature in relation to one of its ancestors in the feature model tree hierarchy. For example, in constraint C_2 of Fig. 3, [0..3] *(Gear, Application)* is the cardinality of feature *Gear* in relation to *Application* in the case of feature *Free* being selected. This means that the maximum number of *Gear*s in an *Application* is three when the *Free* plan is chosen.

Feature models have already been used in [18], [10] to manage the variability in a cloud provider's configuration. However, in their approach the feature models employed consider only configurations for single service applications. For instance, in the case of the OpenShift PaaS, it allows for defining a configuration for either a Java or a Python application, but it does not allows to define a configuration for an application composed of two services, written in different languages or deployed to different regions.

This limitation exists because their implementation of feature models does not consider that individual instances of a feature can have a different decomposition of its subfeatures. But it is also due to the limitation of existing feature modeling constructs, which could not describe the complex constraints involving relative cardinalities that surface once we consider configuring multiple services through feature models.

In a summary, by modeling cloud provider variability we guarantee that the configurations generated by our approach are valid according to providers' constraints. In addition, the use of feature models extended with relative cardinalities allows for modeling the complex constraints identified when service-based applications are considered.

C. Ontologies for managing cloud heterogeneity

Feature models can capture the variability in a cloud provider but they cannot describe the semantics of these features. In a feature model we can specify that choosing a *Free* plan implies using only *small* size *Gear*s, but we can not describe what is a *small* size *Gear*. For describing semantics of features we rely on the use of ontologies and mappings from features to ontology concepts. An ontology is a formal definition of concepts in a given domain and how they are interrelated. It is usually defined by a set of classes and objects that are related through properties.

In our approach, mapping a feature model to an ontology is done through a domain-specific language. Fig. 4 shows part of the mapping from the OpenShift PaaS feature model to the cloud ontology. In this example, we can see a mapping from feature *Tomcat* to ontology class **JBossEWS**. This mapping states that an instance of feature *Tomcat* represents an object of the ontology class **JBossEWS**. The concepts referenced in the mapping are also part of a global cloud ontology that includes the main concepts of the cloud computing domain.

However, while in some cases a feature may provide an instance of a cloud concept by itself, in other cases a set of features may be needed. For example, the *Cartridge* feature in the OpenShift PaaS (Fig. 3) represents an executable container that can be used to run a web application server or a database. In the cloud ontology, executable containers are represented by the class **AppContainer**, which has a set of properties such as CPU power, available RAM, disk size, the kind of software it can run, in which cloud it is deployed, etc. The properties of a given instance of feature *Cartridge* depend on how its subfeatures are selected. Therefore, an instance of feature *Cartridge* will have 1GB of RAM if it is of *medium* size, it will provide a JBoss EWS 2.0 as a servlet container if *Tomcat* feature is selected, and it will be deployed to Amazon US-East-1 cloud if the *US* feature is selected.

To deal with this scenario the mapping language allows describing properties that depend on the selection of features. The mapping for feature *Cartridge* in Fig. 4 illustrates how this can be done. In this case, the mappings work as a template for an ontology object, whose property values depend on the selection of features.

```
feature Cartridge provides AppContainer {
 isWeb value select Web
 memory value 512MB if select small
 vCPUs value 1 if select small
 memory value 512MB if select small.highcpu
 vCPUs value 2 if select small.highcpu
 memory value 1GB if select medium
 vCPUs value 2 if select medium
 memory value 4GB if select large
 vCPUs value 4 if select large

 deployedInCloud value AmazonUSEast1 if select US
 deployedInCloud value AmazonEUWest1 if select EU

 option value AutoScale if select Scalable
 option value AcrossAZs if [2..*](Gear, Cartridge)
 maxNumInstances value (Gear, Cartridge)
 software value from Tomcat
 software value from Python
}
feature Tomcat provides JBossEWS {
 version value 2.0
} /* ... */
```

Fig. 4. Mapping from feature model for OpenShift PaaS to cloud ontology

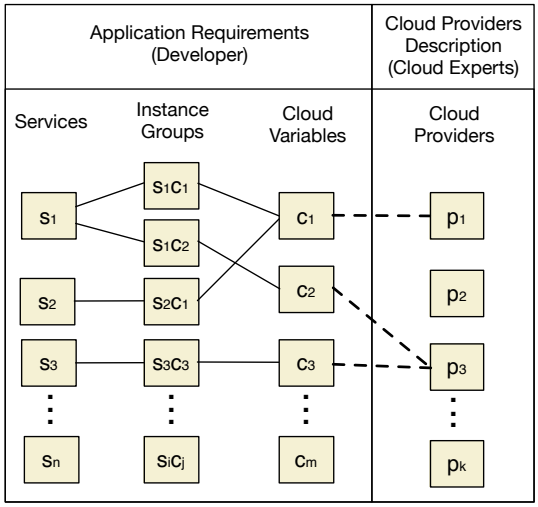

Fig. 5. Multi-cloud configuration as an assignment problem

Using ontologies has the advantage of allowing reasoning over concepts to find equivalence and specialization relationships between providers offerings and service requirements. For example, by reasoning upon ontologies we can discover that the *JBoss EWS 2.0* provided by OpenShift feature `Tomcat` can be matched to the requirements of *web frontend* service (see Fig. 2), which requires a Java Servlet Container implementing version 2.5 of the Servlet API.

To enable further extensibility of our approach, application specific and provider specific ontologies can be optionally defined by developers and cloud experts respectively. Developers can define application specific concepts, as well as their relationship to global concepts, which can then be used directly in the description of the application's requirements. Similarly, provider specific concepts can be used to simplify the description of feature model mappings.

In a summary, ontologies are used to bridge the gap between service requirements and cloud providers. By using ontologies we can describe service requirements that are as complex as an ontology class description and reason upon them to find cloud providers that may offer instances of these classes.

IV. Implementation

To evaluate the feasibility of our approach, we developed a set of modeling and processing tools for modeling to find a multi-cloud configuration that suits the application's requirements. The following subsections provide more information about the implementation of these tools and their evaluation.

A. Modeling

To model the artifacts required by our approach, we designed three domain-specific languages for describing (i) multi-cloud requirements; (ii) cloud provider variability; and (iii) cloud provider mappings. We then used the xText [19] framework together with Eclipse Modeling Framework [20] to provide an integrated editor for these languages. For describing the ontologies we use the standard OWL 2 Web Ontology Language [21], for which many modeling and reasoning tools are available [3], [22]. The developed modeling tools are available in the accompanying site[1].

B. Reasoning

The problem of finding a multi-cloud configuration that suits a set of application requirements can be seen as an assignment problem subject to constraints. As illustrated in Fig. 5, an instance of this problem is described by the requirements of an application and a set of available cloud providers. From application requirements we have a set of *services*, which are assigned by the developer to a *cloud variable* through a set of *instance groups* (see Section III-A). On the other side, we have a set of *cloud providers* that offer varying functionality. A solution to an instance of this problem is an assignment from *cloud variables* to *cloud providers*, which is depicted as a set of dashed lines.

Due to the number of providers, the search space for this problem can be very large, making it difficult to find a solution in a reasonable time. In addition, verifying the suitability of a solution is a complex and time consuming task as it requires reasoning on ontologies to convert application requirements to provider specific constructs and evaluating variability in feature models. These factors, when combined, may make this problem intractable. To deal with these difficulties we employ a strategy to reduce the search space gradually by evaluating less costly constraints first. Our algorithm relies on this strategy, as well as compiling and caching intermediate results, to improve the verification of candidate solutions, thus rending the problem more amenable.

Verification is done in three steps: 1) validating that assigned providers comply with cloud conditions; 2) checking that service requirements can be mapped to a set of features in

[1] http://researchers.lille.inria.fr/~sousa/mmcloud/

```
cloud B { self.location = WesternEurope }

Cloud that dataCenterLocatedIn some
           (partOfRegion value WesternEurope)
                          (a)

service web_frontend {
 requires WebContainer {
  memory 2GB
  software JavaWebContainer {
   providesAPIs Servlet { version = 2.5 }
  }
 }
}
instanceGroup web_frontend {
 B { numInstances 1..5 options AutoScale }
}

WebContainer that hasMemory some (DataAmount that
hasUnit value MB and hasValue some integer [>= 2048])
and hasSoftware some (JavaWebContainer that
providesAPIs some (Servlet that hasVersion value 2.5))
and deployedInCloud value AmazonEUWest1 and options
value AutoScale and maxNumInstances some integer[>= 5]
                          (b)

Cartridge { medium, Tomcat, EU, Scalable, 5 Gear }
                          (c)
```

Fig. 6. Sample translations from requirements, to ontology classes, to features selections

the assigned provider; and 3) verifying that there is a valid configuration where all services assigned to a provider can be deployed.

1) Validating cloud conditions: Cloud conditions are those described in application requirements as part of the cloud variables description (see Section III-A). In Fig. 2, for example, cloud variable B has as a condition that it should be assigned to a cloud located in **`WesternEurope`**. Evaluating these conditions is relatively simple and consists in checking if each assigned cloud variable complies with its constraint. However, this may still require some reasoning over ontologies to identify, for example, that a cloud located in **`France`** meets the criteria of being located in **`WesternEurope`**. To do so, we translate a cloud condition into an ontology class, which represents the class of clouds that complies with the condition, and then verify if the assigned provider cloud is an instance of this class. Fig. 6 (a) shows an example of cloud condition for cloud variable B (see Fig. 2) and the corresponding ontology class described in the OWL 2 Manchester Syntax.

2) Mapping service requirements to provider features: After verifying that the assigned providers comply with cloud conditions we still need to check if each selected provider supports all functionality required by the services assigned to it. To do so, we use the cloud provider mappings to check, for each assigned service, if its required functionality can be mapped to a set of features in the provider's feature model. First each service is converted into an ontology class, which defines the class of objects that provides service requirements. Fig. 6 (b) shows the requirements for *web frontend* service (see Fig. 2) and its corresponding ontology class. Then, we use ontology reasoning together with cloud provider mappings to find a selection of features that provides an instance of this required class. Fig. 6 (c) shows the result from mapping *web frontend* service to the OpenShift PaaS.

In this example, the *web frontend* service requires a **`WebContainer`**, but feature `Cartridge` in OpenShift provides just an **`AppContainer`** (see Fig. 4). However, OpenShift's specific ontology describes that a **`WebContainer`** is the same of an **`AppContainer`** with the data property **`isWeb`** set to true. OpenShift's mapping for feature `Cartridge` defines that the property **`isWeb`** is set to true if and only if the feature `Web` is also selected. Thus, an instance of feature `Cartridge` provides a **`WebContainer`** if its subfeature `Web` is selected.

The end result of this process is a *feature selection* defining a set of features that provide the requirements of a service. For the *web frontend* example, the expression obtained defines that an instance of feature `Cartridge` can provide service requirements if features `medium`, `Tomcat`, `EU`, `Scalable` and 5 instances of feature `Gear` are also selected. If a mapping from the required ontology class to a feature selection can not be found, then the assigned provider does not support the service requirements.

3) Verifying feature constraints and generating a complete configuration: After mapping service requirements to a selection of features, we know that the assigned providers support the services assigned to them. However, as there may be limits in the resources offered by a provider, conflicting features, or other constraints in cloud offerings, we still need to check if all services assigned to a provider can be deployed together and to generate a configuration for each provider. This process is done by evaluating the cloud providers' feature models against the feature selection expressions obtained in the previous step. This is a complex and time consuming task that involves translating the feature model to a constraint satisfaction problem and solving it to find a valid configuration that matches the requirements. This translation is based on existing work [23] and our previous developments [17] in cardinality-based feature models. If a solution to the corresponding constraint problem is found it means that the assignment is valid. Besides this, the output solution also defines for each selected provider a complete configuration of the corresponding feature model, detailing a hierarchy of feature instances that should be selected.

Our strategy for processing application requirements enables us to get from a set of requirements for microservices-based multi-cloud applications to a concrete set of selected features in each cloud provider. By decomposing this process into three steps we avoid doing more expensive calculations when not needed and we reuse intermediate results.

C. Experimentation

To validate the effectiveness of our approach and evaluate the performance of the developed tools we elaborated an example application based on the requirements described in Section II and measured the time consumed at each stage of processing, as well as the total time to get from requirements to a concrete multi-cloud configuration.

The application requirements were described using the domain-specific language for multi-cloud requirements and include 13 services, with 5 cloud variables and 15 instance groups. In addition, we elaborated feature models and ontology mappings for four popular cloud providers: OpenShift[2], Heroku[3], Google[4] and Jelastic[5]. Information from cloud providers was obtained from their documentation and through the use of their configuration tools.

We performed 20 executions of the developed tool to convert application requirements to a multi-cloud configuration. All experiments were run on a MacBook Pro Computer with a 2 GHz Intel Core i7 processor and 8GB of memory. From the obtained results, shown in Table I, we can see that translating from high-level application requirements to a concrete configuration took on average $6.62s$. For this set of four providers 32 valid solutions, in an universe of 1024 possible assignments, were found in an average time of $24s$.

Operation	Average time (ms)	Standard deviation (ms)
Validate cloud conditions	3.56	5.12
Translate a service to feature sets	1,796.82	2,215.69
Translate a feature model to CSP and solve it	3,350.47	827.34
Overall process	6,620.60	6,431.53

TABLE I
AVERAGE TIME FOR OPERATIONS IN THE GENERATION OF A MULTI-CLOUD CONFIGURATION.

To evaluate the performance in the presence of more providers we randomly generated extra 46 providers in a way that no generated provider would completely support the features required by the example application. With this new set of 50 providers, $52s$ were need to find the same 32 solutions. This indicates, that despite the time needed to process and generate a valid solution, invalid assignments can be discarded more quickly and reduce the search space.

D. Limitations

Our approach focus on the selection and configuration of cloud providers for a multi-cloud scenario and does not deal directly with the deployment of applications. Generation of deployment scripts from feature models has been discussed in the literature [18], and similar techniques could be integrated in our approach. We also still do not consider costs and quality of service for optimizing the selection as these depend heavily on application usage. Finally, for the conducted experiments, cloud provider feature models and ontology mappings were manually defined relying providers' documentation, and are thus susceptible to errors.

[2]http://www.openshift.com/
[3]http://www.heroku.com/
[4]http://cloud.google.com/
[5]http://jelastic.com/

V. RELATED WORK

Several recent works were proposed to deal with aspects of the management of multi-cloud systems. CloudMF [8] relies on a domain-specific language and models@runtime to deal with heterogeneity across providers and support provisioning and deployment of multi-cloud systems. We also employ domain-specific languages for modeling requirements and provider descriptions, but in addition we also support the selection of cloud providers and variability management. The mOSAIC [7] and soCloud [24] projects propose a multi-cloud PaaS for interoperability across multiple clouds. Instead of proposing developers a new platform service, our approach aims at supporting them to choose and combine existing platform, infrastructure and software services. *Wright et al.* [6] propose a constraints-based method for composing multi-provider environments. Like in our work, their goal is to find an assignment of services to cloud providers according to some constraints. However, requirements are defined as low-level constraints and services can use only infrastructure resources. CloudPick [9] is a framework that supports the selection of IaaS providers and the deployment of a network of virtual appliances (preconfigured virtual machine images). Cloud4SOA [25] also proposes the use of ontologies to provide interoperability across PaaS providers. Like in our approach, ontologies are used for semantically mapping requirements to providers' offerings. However, both approaches do not manage variability and constraints of cloud providers configurations.

Managing variability and heterogeneity in the cloud has also been identified as an important issue that is subject of substantial work. In [26] and [27], feature models are used to model variability in infrastructure resources (e.g. virtual machine, storage, network, etc) and support provider selection. Feature models are also used in a similar way in [28], but to optimize energy consumption. In our work, we consider not only variability in a resource or service configuration but across all the functionality offered by a provider.

Ontologies are also proposed as a way to deal with heterogeneity and the semantic gap between requirements and cloud providers offers [7], [29], [30]. However these approaches do not consider the variability in providers' offerings and apply reasoning upon predefined instances.

The SALOON [10] approach also relies on feature models to capture cloud provider configuration options and ontologies to map from requirements to cloud features. However, it does not consider applications that can be composed of multiple services and do not support multi-cloud requirements.

Our approach shares with existing work the use of ontologies to deal with heterogeneity across cloud providers and of software product line techniques for managing variability in cloud provider configurations. However, our work takes into account variability at the provider level, not only for a given resource type (virtual machine, storage, etc) and not just for one application service.

In addition, we also employ domain-specific languages and model driven-engineering techniques to achieve separation

of concerns between different user roles. We support the description of requirements for microservices-based applications and mechanisms to achieve multi-cloud requirements of location, scalability and redundancy. Overall, our approach differentiates from others by taking into account variability in cloud configuration options, multi-cloud requirements and service composition to build a suitable environment.

VI. Conclusion

Multi-cloud computing has the potential of reducing vendor dependence, increasing application reliability and optimizing resource usage. However, the wide number of available cloud providers, their high heterogeneity and their intricate configuration options, make it very complex to exploit these benefits. In this paper, we describe an approach to deal with this complexity, by supporting the selection and configuration of multi-cloud environments for microservices-based applications.

Our approach relies on a domain-specific modeling language for defining multi-cloud requirements and a combination of reasoning tools to obtain: (i) a valid assignment of application services to cloud providers and (ii) proper configurations for these providers. We implemented this approach as a set of tools for modeling the required artifacts and reasoning upon them. We conducted some experiments to show that valid configurations can be obtained in reasonable time.

For future work, we intend to include pricing and quality of service information as part of our analysis and investigate search algorithms and heuristics to deal with this as an optimization problem. This goes into the direction of our longterm goal, which is towards self-adaptive multi-cloud environments capable of identifying optimization opportunities as application requirements and cloud market evolve.

References

[1] N. Grozev and R. Buyya, "Inter-cloud architectures and application brokering: taxonomy and survey," *Software: Practice and Experience*, vol. 44, no. 3, pp. 369–390, 2012.

[2] J. Lewis and M. Fowler. (2014, Mars) Microservices: a definition of this new architectural term. [Online]. Available: http://martinfowler.com/articles/microservices.html

[3] E. Sirin, B. Parsia, B. C. Grau *et al.*, "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.

[4] K. Pohl, G. Bckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[5] D. Ardagna, E. Di Nitto, G. Casale *et al.*, "MODAClouds: A model-driven approach for the design and execution of applications on multiple clouds," in *Proc. International Workshop on Modeling in Software Engineering (MiSE '12)*, Zurich, Switzerland, 2012, pp. 50–56.

[6] P. Wright, Y. Sun, T. Harmer *et al.*, "A constraints-based resource discovery model for multi-provider cloud environments," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 1, no. 1, p. 6, 2012.

[7] D. Petcu, B. D. Martino, S. Venticinque *et al.*, "Experiences in building a mOSAIC of clouds," *Journal of Cloud Computing*, vol. 2, no. 1, pp. 1–22, 2013.

[8] N. Ferry, H. Song, A. Rossini *et al.*, "Cloud MF: Applying mde to tame the complexity of managing multi-cloud applications," in *Proc. IEEE/ACM International Conference on Utility and Cloud Computing (UCC'14)*, London, UK, 2014, pp. 269–277.

[9] A. V. Dastjerdi, S. K. Garg, O. F. Rana *et al.*, "CloudPick: a framework for QoS-aware and ontology-based service deployment across clouds," *Software: Practice and Experience*, vol. 45, no. 2, pp. 197–231, 2015.

[10] C. Quinton, D. Romero, and L. Duchien, "SALOON: a platform for selecting and configuring cloud environments," *Software: Practice and Experience*, vol. 46, no. 1, pp. 55–78, 2016.

[11] N. Ferry, A. Rossini, F. Chauvel *et al.*, "Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems," in *Proc. IEEE International Conference on Cloud Computing (CLOUD'13)*, Santa Clara, USA, June 2013, pp. 887–894.

[12] D. Petcu, E. Di Nitto, D. Ardagna *et al.*, "Towards multi-clouds engineering," in *Proc. IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Toronto, Canada, April 2014, pp. 1–6.

[13] D. Petcu, "Multi-cloud: Expectations and current approaches," in *Proc. International Workshop on Multi-cloud Applications and Federated Clouds ((MultiCloud '13)*, Prague, Czech Republic, 2013, pp. 1–6.

[14] L. Chen, M. Ali Babar, and N. Ali, "Variability management in software product lines: A systematic review," in *Proc. International Software Product Line Conference (SPLC'09)*, ser. SPLC '09, San Francisco, California, USA, Aug. 2009, pp. 81–90.

[15] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[16] C. Quinton, D. Romero, and L. Duchien, "Cardinality-based feature models with constraints: A pragmatic approach," in *Proc. International Software Product Line Conference (SPLC'13)*, Tokyo, Japan, Aug. 2013, pp. 162–166.

[17] S. Gustavo, W. Rudametkin, and L. Duchien, "Extending feature models with relative cardinalities," Inria Lille - Nord Europe, Tech. Rep., 2016.

[18] C. Quinton, D. Romero, and L. Duchien, "Automated selection and configuration of cloud environments using software product lines principles," in *Proc. IEEE International Conference on Cloud Computing (CLOUD'14)*, Alaska, USA, Jun. 2014, pp. 144–151.

[19] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[20] D. Steinberg, F. Budinsky, E. Merks *et al.*, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[21] S. Bechhofer, *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009, ch. OWL: Web Ontology Language, pp. 2008–2009.

[22] B. Motik, R. Shearer, and I. Horrocks, "Hypertableau Reasoning for Description Logics," *Journal of Artificial Intelligence Research*, vol. 36, pp. 165–228, 2009.

[23] R. Mazo, C. Salinesi, D. Diaz *et al.*, "Transforming attribute and clone-enabled feature models into constraint programs over finite domains," in *Proc. International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'11)*, Beijing, China, Jun. 2011.

[24] F. Paraiso, P. Merle, and L. Seinturier, "soCloud: a service-oriented component-based paas for managing portability, provisioning, elasticity, and high availability across multiple clouds," *Computing*, pp. 1–27, 2014.

[25] E. Kamateri, N. Loutas, D. Zeginis *et al.*, "Cloud4SOA: A semantic-interoperability paas solution for multi-cloud platform management and portability," in *Proc. European Conference on Service-Oriented and Cloud Computing (ESOCC'13)*, K.-K. Lau, W. Lamersdorf, and E. Pimentel, Eds. , Málaga, Spain: Springer Berlin Heidelberg, Sep. 2013, pp. 64–78.

[26] A. Ferreira Leite, V. Alves, G. Nunes Rodrigues *et al.*, "Automating resource selection and configuration in inter-clouds through a software product line method," in *Proc. IEEE International Conference on Cloud Computing, (CLOUD'15)*, New York, United States, Jun. 2015, pp. 726–733.

[27] J. García-Galán, P. Trinidad, O. F. Rana *et al.*, "Automated configuration support for infrastructure migration to the cloud," *Future Generation Computer Systems*, vol. 55, pp. 200 – 212, 2016.

[28] B. Dougherty, J. White, and D. C. Schmidt, "Model-driven auto-scaling of green cloud computing infrastructure," *Future Generation Computer Systems*, vol. 28, no. 2, pp. 371–378, Feb. 2012.

[29] F. Amato, A. Mazzeo, V. Moscato *et al.*, "A framework for semantic interoperability over the cloud," in *Proc. of IEEE International Conference on Advanced Information Networking and Applications Workshops (WAINA'13)*, Barcelona, Spain, Mar. 2013, pp. 1259–1264.

[30] G. Cretella and B. Di Martino, "A semantic engine for porting applications to the cloud and among clouds," *Software: Practice and Experience*, vol. 45, no. 12, 2015.