# Automated Selection and Configuration of Cloud Environments Using Software Product Lines Principles

Clément Quinton, Daniel Romero, Laurence Duchien

# Automated Selection and Configuration of Cloud Environments Using Software Product Lines Principles

Clément Quinton, Daniel Romero, Laurence Duchien
*INRIA Lille - Nord Europe, LIFL UMR CNRS 8022, University Lille 1, France*
*firstname.lastname@inria.fr*

***Abstract*—Deploying an application to a cloud environment has recently become very trendy, since it offers many advantages such as improving reliability or scalability. These cloud environments provide a wide range of resources at different levels of functionality, which must be appropriately configured by stakeholders for the application to run properly. Handling this variability during the configuration and deployment stages is a complex and error-prone process, usually made in an *ad hoc* manner in existing solutions. In this paper, we propose a software product lines based approach to face these issues. Combined with a domain model used to select among cloud environments a suitable one, our approach supports stakeholders while configuring the selected cloud environment in a consistent way, and automates the deployment of such configurations through the generation of executable deployment scripts. To evaluate the soundness of the proposed approach, we conduct an experiment involving 10 participants with different levels of experience in cloud configuration and deployment. The experiment shows that using our approach significantly reduces time and most importantly, provides a reliable way to find a correct and suitable cloud configuration. Moreover, our empirical evaluation shows that our approach is effective and scalable to properly deal with a significant number of cloud environments.**

## I. Introduction

Cloud computing has recently emerged as a major trend in distributed computing. This layered model, as defined by the NIST [1], enables the configuration of many computing resources that can be provisioned to support the deployment of applications, provided as *Software-as-a-Service* (SaaS) [2], [3]. Many cloud providers, either at *Infrastructure* (IaaS) or *Platform* (PaaS) level, propose different services and pricing models. Due to this variability, developers face three key challenges when deploying an application to the cloud.

The first challenge is to select a cloud environment that complies with both *functional* and *non-functional* requirements [4]. Among the plethora of cloud providers, developers have to *(i)* find the ones that provide all functionalities required by the application to run properly, *e.g.*, the correct type of application server and database, and then *(ii)* select the one that proposes the correct quality for these functionalities, *e.g.*, a solution with at least 4 GB of RAM with as much CPU power as possible. The second challenge is to define a proper configuration. Dealing with clouds variability leads to complex and error-prone configuration choices that are usually made in an *ad hoc* manner. Moreover, developers' knowledge is not exhaustive and the way a cloud environment is configured can lead to inconsistencies between cloud services when running the application, whether this configuration is done by hand or using dedicated commands provided by cloud environments. The third challenge is to deploy in a reliable way. Once a cloud environment is selected and a configuration is defined, developers have to avoid errors in the deployment process, in particular when defining environment configuration files and executing deployment scripts, to ensure the application will be deployed in a reliable way.

To address these challenges, we propose to use a *Software Product Line* (SPL) based approach [5], [6]. SPLs are dedicated to automate the configuration and the *derivation*, *e.g.*, composition and/or generation, of software products with high variability. It provides means to *(i)* capture the common and variable artifacts of the handled software in a variability model and *(ii)* reuse those artifacts to automatically derive the software product, thus reducing development costs while increasing reliability. The contribution of this paper is threefold. First, we propose to use *Feature Models* (FMs) [7] extended with cardinalities and attributes as variability models to describe cloud environments. In particular, our approach automatically handles constraints over these cardinalities and attributes, which is not supported in existing feature modeling approaches but required to deal with cloud configuration. Second, artifacts are reified as configuration files and execution scripts to automate the cloud configuration. Finally, we propose to map these variability models with a domain knowledge model to deal with clouds heterogeneity. To illustrate the practical applicability of our approach, we conduct experiments and provide a tool support implemented in the Saloon framework [8], [9].

The paper is organized as follows. We present in Section II our SPL-based approach and we discuss different concerns regarding its validity. In Section III, we explain the evaluation we did to assess our approach. We then describe in Section IV close-related work. Finally, Section V concludes the paper and presents the perspectives.

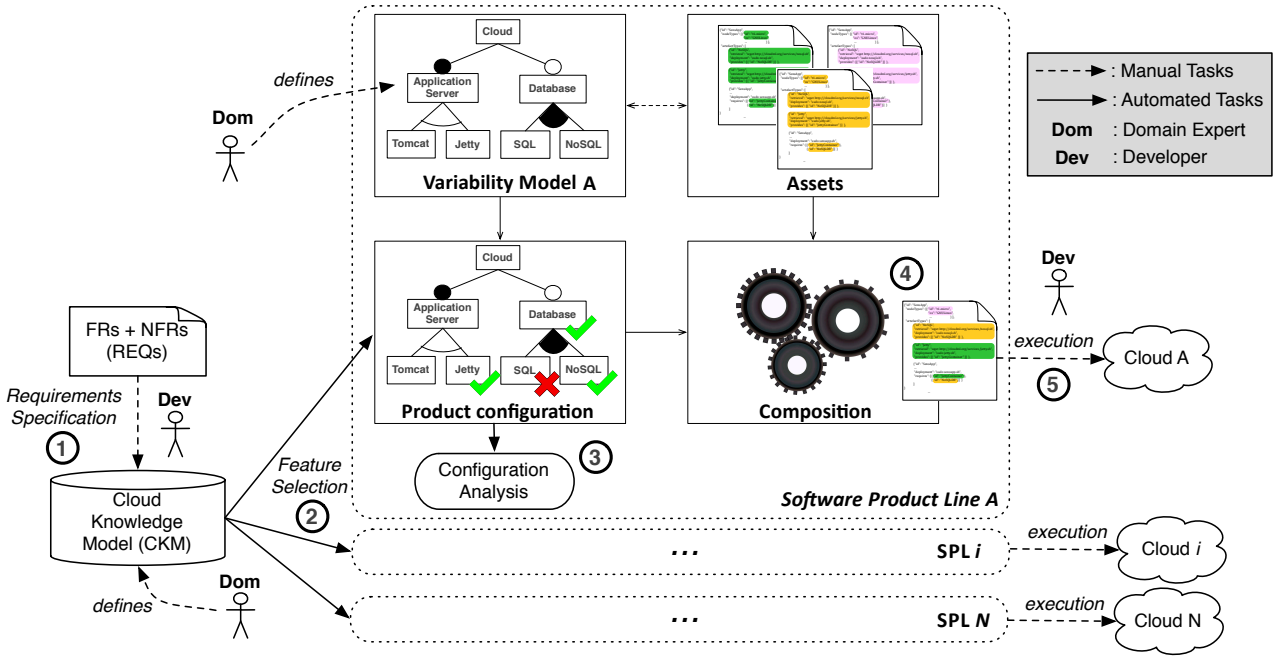



Figure 1: Approach Overview: From Requirements Specification to Cloud Deployment

## II. Software Product Lines for Cloud Selection and Configuration

SPL engineering aims at building software while ensuring quality, reliability and reduction of cost, efforts and time-to-market [5], [6]. The building process relies on the definition and the composition of a set of software artifacts, *e.g.*, piece of code, model, component or aspect, defined as *assets*. Some of these assets are mandatory and will be part of all the built software (commonalities), while other assets define the way software differ from each other (variabilities). The definition of variabilities and commonalities, known as variability modeling, is a central activity in SPL engineering and relies on *variability models*. In these models, assets are abstracted as *features*. The developer then selects or deselects features to get a combination of features. We refer to this process as *feature selection*, where a product is a valid combination of features. This valid *product configuration* is then given as input together with the related assets to the *composition* tool that yields the software product.

To tackle the challenges described in Section I, we propose an SPL-based approach, depicted in Fig. 1, which provides the following three features:

$(i)$ the description of cloud environment variability, *i.e.*, commonalities and variabilities, as feature models [7] extended with cardinality [8], attributes [10], and constraints over them. One feature model is used to describe one cloud environment. There is thus one SPL per cloud.

$(ii)$ the reification and gathering of cloud environment provided functionalities into a *Cloud Knowledge Model*, mapped to each cloud FM to automate the feature selection process.

$(iii)$ the configuration analysis of these FMs, including complex constraints over attributes and cardinalities, as well as the generation of the related software products as deployment scripts, both processes being automated.

Our approach distinguishes between two roles, *domain experts* and *developers*. The formers are cloud computing experts involved in the definition of the architecture models. They first describe clouds variability and commonality points into FMs, one per SPL. Then, they gather their cloud knowledge to define the Cloud Knowledge Model. On the other hand, developers are all stakeholders involved in cloud configuration and deployment who are using the proposed approach. The developer specifies its requirements using the Cloud Knowledge Model (Fig. 1 ①). Then, features and attributes of each FM are selected according to the mapping between this cloud model and the FMs ②. Each FM `configuration` is then checked ③ to be used as input, if valid, by the `composition` tool that yields the related configuration files and/or deployment scripts ④, executed by the developer ⑤. We describe in details in the following sections the different concerns of our approach.

### A. Cloud Environments as Feature Models

SPL engineering begins with the description, management and implementation of the commonalities and variabilities existing among the members of the same family of software

products [5], [6]. A well-known approach to variability modeling is by means of FMs [7], where FMs describe the way software artifacts are configured and reused to yield software products that satisfy a set of defined constraints. In these FMs, known as boolean FMs, a feature is either present or absent in the final product according to the configuration and the involved constraints.

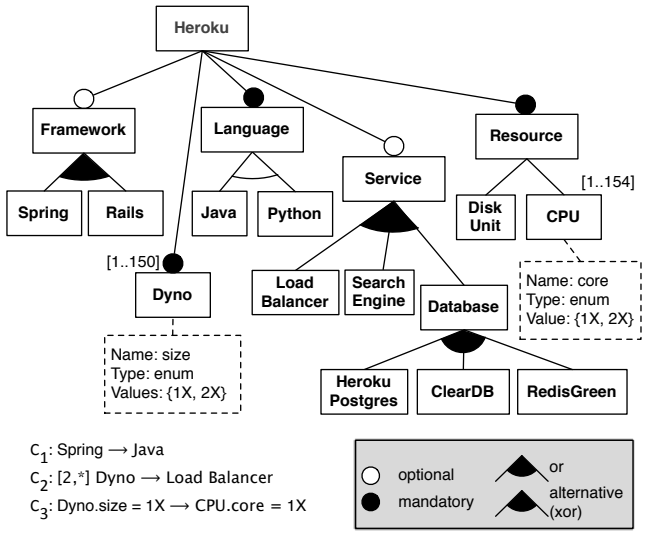



Figure 2: The Heroku Platform as Feature Model (excerpt)

FIG. 2 depicts the FM that we have defined to describe the variability of the Heroku PaaS[1]. When configuring this PaaS, some services must be selected and are represented in the FM as *mandatory* features, *e.g.*, the wished **`Language`** support or the required **`Resources`**. Others services may be part of the final configuration, and are thus depicted as *optional* features, *e.g.*, the **`Framework`** support. We assume the FMs to be correct *w.r.t.* the cloud specifications.

In our approach, we extend boolean FMs with cardinalities [8] and attributes [10]. Features with cardinality, *e.g.*, **`Dyno`**, describe a service or functionality that can be instantiated several times. A **`Dyno`** is the unit of computing power on Heroku, providing 512 MB of memory and one CPU Share in its default configuration "1X". These values double when considering the "2X" configuration. Adding cardinality to features introduces variability, since the total number of possible configurations increases proportionally. Feature attributes are used to add information to the feature. In our approach, we use it to specify the non-functional properties of the related feature. In the Heroku example, the **`Dyno`** feature has an attribute, named *size*, indicating that **`Dynos`** are available in 1X or 2X size. Our approach supports feature attributes whose type is either integer, real, boolean or enumeration, which is a set of values, *e.g.*, the *core* attribute.

[1]https://www.heroku.com

Cardinality and attribute-based constraints, which are not supported in existing feature modeling approaches but are required to deal with cloud configuration, are automatically handled in our approach. In boolean FMs, declarative constraints are used to specify if the selection of a feature *implies* or *excludes* the selection of another one, *e.g.*, $C_1$ in FIG. 2 indicating that using the **`Spring`** framework implies the **`Java`** support to be configured. Regarding constraints dealing with cardinalities, we rely on the syntax and semantics we introduced in our previous work [8]. For example, constraint $C_2$ is a cardinality-based constraint. It describes the fact that if there are at least two instances of **`Dyno`** configured, then a **`Load Balancer`** must also be configured. In this paper, we introduce in addition constraints over feature attributes, defined as attribute-based constraints.

**Definition 1.** (ATTRIBUTE-BASED CONSTRAINT)
An *attribute-based* constraint $Attr_{cons}$ is written $F_{from}.attr_{from} = val_{from} \rightarrow F_{to}.attr_{to} = val_{to}$, where

- $F_{from}, F_{to} \in \mathcal{F}$ where $\mathcal{F}$ is the non empty set of features of the FM;
- $attr_{from} \in \mathcal{A}_{from}$ and $attr_{to} \in \mathcal{A}_{to}$, where $\mathcal{A}_{from}$ and $\mathcal{A}_{to}$ are the sets of attributes of features $F_{from}$ and $F_{to}$ respectively;
- $val_{from}$, $val_{to}$ are values given to $attr_{from}$ and $attr_{to}$ respectively;

Then $Attr_{cons}$ is satisfied if

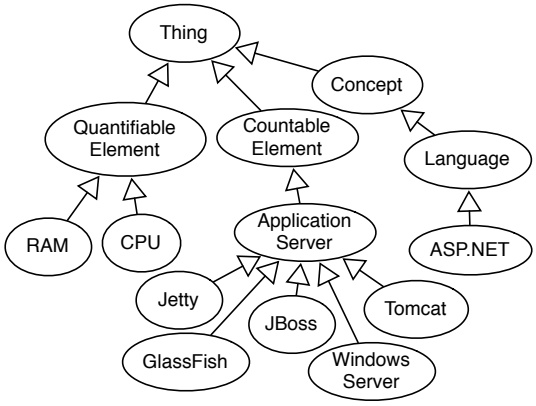
$$attr_{from} = val_{from} \Rightarrow attr_{to} = val_{to}$$

In the Heroku FM, the constraint $C_3$ describes that if the *size* of the **`Dyno`** is 1X, then the **`CPU`** *core* must also be 1X. Once features are selected, boolean, attribute and cardinality-based constraints must hold for the configuration to be valid. We use on an off-the-shelf *Constraint Satisfaction Problem* (CSP) solver to reason on these configurations and check whether they are valid or not [9], as we relied on it in our previous work regarding cardinality-based FMs [8].

*Summary.* Combining in one hand cardinalities and attributes with in the other hand constraints over them allows our approach to define in a reliable way the elements required for the configuration of cloud environments, which was not feasible with existing feature modeling approaches. Moreover, using such extensions to FMs, it is possible not only to define configurations that hold regarding a given set of functional requirements, but also to specify non-functional requirements over these configurations, *e.g.*, to find a configuration for this cloud with *a PostgreSQL support and at least 1 GB of RAM*.

### B. The Cloud Knowledge Model

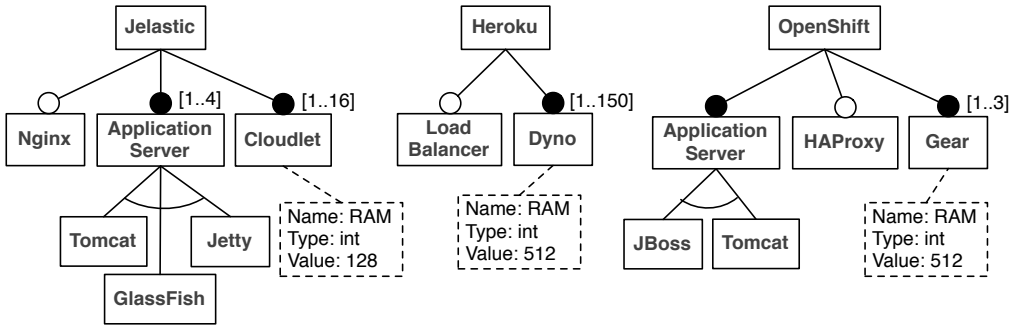
Our SPL-based approach relies on the reification of cloud environments as FMs to check the validity of their configuration in an automated way. In a typical SPL engineering process, the selection of the required features is done by hand. Applied to our approach, this means selecting features in each




(a) The Cloud Knowledge Model

**RAM:**
Jelastic.Cloudlet.RAM | Heroku.Dyno.RAM | OpenShift.Gear.RAM
**Load Balancer:**
Jelastic.Nginx | Heroku.Load Balancer | OpenShift.HAProxy
**Application Server:**
Jelastic.Application Server | OpenShift.Application Server
**Tomcat:**
Jelastic.Tomcat | OpenShift.Tomcat
**GlassFish:**
Jelastic.GlassFish
**Jetty:**
Jelastic.Jetty
**JBoss:**
OpenShift.JBoss

(b) The mapping relationships




(c) The feature models

Figure 3: The Cloud Knowledge Model, the feature models and the related mappings

FM, one FM after the other, which is a tedious and error-prone task since there are currently tens of cloud environments available. To cope with this issue, our approach relies on a cloud model, the Cloud Knowledge Model, describing the domain the SPL has been built for (here the one of cloud environments). The Cloud Knowledge Model defines formally all the concepts relevant to the domain, and thus gathers every FM features, reified as concepts. In our approach, we rely on the ontology formalism [11] to represent the Cloud Knowledge Model, so that it provides a common vocabulary in a machine readable format. We assume that the Cloud Knowledge Model is defined by cloud experts (each expert of its cloud environment) by adding the cloud functionalities in the model for it to be exhaustive.

There are three kinds (FIG. 3.a) of concepts in the Cloud Knowledge Model [9]. *Concept* is used to define basic concepts, *e.g.*, *Language*, which defines the language the application to deploy has been developed with. *CountableElement* captures concepts whose the required number of instances can be specified, *e.g.*, four *Application Server* instances. Finally, *QuantifiableElement* are used to describe concepts whose required quantity and unit can be specified, *e.g.*, 500 MB of *RAM*. Some constraints are also defined over these concepts, *e.g.*, $C_4$: ASP.NET $\rightarrow$ Windows Server in FIG. 3 (a), meaning that if the application to deploy is written in ASP.NET, then the cloud environment must provide the Windows Server support to host it.

The mapping relationships link concepts from the Cloud Knowledge Model with features in the FMs. FIG. 3 depicts how the mapping works with excerpts of the different models and relationships. As previously described, the Cloud Knowledge Model (FIG. 3.a) gathers all concepts that can be found in the FMs (here, an excerpt of three of them, FIG. 3.c). The mapping relationships (FIG. 3.b) link them together. These relationships can be either *1-to-1* or *1-to-** relationships. For example, **`JBoss`** mapped to **`OpenShift.JBoss`** is a *1-to-1* relationship and **`Tomcat`** linked to **`Jelastic.Tomcat`** and **`OpenShift.Tomcat`** is a *1-to-** relationship.

Two kinds of mapping relationships exist, either from concept to feature or from concept to attribute. Let us now consider as an example the set of requirements $\text{REQ}_1$: {Tomcat, 1 GB RAM}. Regarding the models and mapping relationships depicted in FIG. 3, countable element **`Tomcat`** is mapped to features **`Tomcat`** in the Jelastic and Openshift FM,

while the quantifiable element **RAM** is mapped to attributes **RAM** for the same FMs. For **Tomcat**, the related features are selected and a value may be given to the feature cardinality if several instances are required. For **RAM**, *(i)* the attribute parent features are selected and *(ii)* a value alignment algorithm taking units into account is processed that may affect feature cardinality, as described in [9]. For example, for $\text{REQ}_1$ to be satisfied, the cardinality of the **Cloudlet** feature must be set to 8 in the **Jelastic** FM, since 8*128 MB $\geq$ 1 GB.

Using such a mapping between the Cloud Knowledge Model and the FMs has three main benefits. First, it automates the feature selection process (and consequently, the configuration validity checking process). The developer thus does not have to select features by hand in every FM, which is considerably error-prone, but simply defines its requirements once in the Cloud Knowledge Model. Second, it bridges the semantic gap between cloud environments by mapping Cloud Knowledge Model concepts to features in different FMs with the same semantics. For example, features **Nginx**, **Load Balancer** and **HAProxy** are mapped to the same Cloud Knowledge Model concept **Load Balancer**, since they are semantically equivalent even if their names differ. Finally, it reduces the range of FMs to be configured by acting like a filter. Indeed, it avoids checking the validity of certain FMs whose configuration can not cope with the requirements set. For example, if **Tomcat** is part of the functional requirements, then this concept cannot be mapped to FMs which do not provide this application server support, *e.g.*, **Heroku** (regarding FIG. 3, not for real). Thus, these FMs are not considered for the rest of the configuration process, since the related cloud environment is unsuitable. Constraints defined in the Cloud Knowledge Model, *e.g.* $C_4$ in FIG. 3 (a), are also used to avoid configuring unsuitable FMs[2].

Even if the selection of features in the different FMs is automated regarding the defined mapping relationships, the developer still has to select the final cloud environment. Indeed, several cloud FM configurations may be valid regarding the given requirements. In such a case, the developer selects the one that best fits his/her requirements. This choice is driven by the way the solver is configured, since weights can be given to the most important requirements and an optimal configuration can be found regarding those requirements.

### C. Configuration Files and Execution Scripts as Assets

As described at the beginning of this section, features hold as assets software artifacts that are put together to yield the final product. Thus, reasoning on feature combinations to find a valid configuration means searching for a proper way to compose concrete software artifacts (*i.e.*, assets such as code snippets, aspects or model fragments) to yield the software product. In our approach, we define assets as *(i)* commands that can be executed in a command line interface or a dedicated environment and *(ii)* configuration files. A feature can hold none, one or several assets, while an asset can be shared among several features. FIG. 4 depicts those situations, with the Heroku PaaS as an example. The **Java** feature holds as asset the *pom.xml* file. It is required by Heroku for every **Java** application. The *system.properties* file is added to the configuration to specify which Java JDK is required, either 1.6 or 1.7. If **Python** is selected, then Heroku requires a *requirements.txt* file. The *Procfile* is a text file placed in the root of the application, that lists which processes are run by the application, *e.g.*, the main class for a Java application. This file is thus held by the **Heroku** feature since it is required for each configuration, whatever the selected features. However, these selected features may have interactions with this file, *e.g.*, to specify the language of the application to deploy and the main class or script to be run.

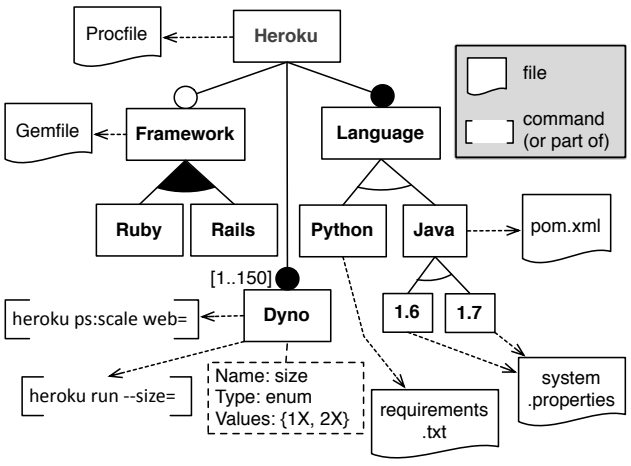



Figure 4: The Heroku FM and its Assets (excerpt)

Attributes and cardinalities also interact with these assets. For example, the **Dyno** feature holds as assets command parts, which are completed by the required amount of **Dyno**, *e.g.*, *heroku ps:scale web*=8, or by the value given to the *size* attribute to specify whether the configuration should use a 1X or 2X **Dyno**, *e.g.*, *heroku run --size=2X*. When several commands are required to configure the cloud environment, they are gathered in a single shell script, which can then be executed in a command line interface. As each cloud environment relies on its own commands, our approach presupposes that the correct set of libraries and SDK are present when executing the commands, *e.g.*,

[2]At this point, the reader may wonder about the difference between FMs and the Cloud Knowledge Model, and why the automated configuration can not be properly handled at the Cloud Knowledge Model level. Constraints defined in the Cloud Knowledge Model are constraints that are not cloud-specific, *e.g.*, $C_4$. Thus, these constraints are shared among every cloud environment, *e.g.*, if ASP.NET is required, any cloud environment that does not provide a Windows Server support is not well-suited and it is unnecessary to configure the related cloud FM. Constraints defined in the FMs are cloud-specific, and thus can not be defined in the Cloud Knowledge Model.

Git[3] and the Heroku client[4]. To yield the configuration files and shell scripts, we rely on templates and code generation techniques, together with merging mechanisms. Command generation can be properly ordered using existing composition process [12]. These mechanisms are classic in model-driven engineering [13], and the description of such algorithms is out of the scope of this paper.

### D. Threats to Validity

We have presented in the previous sections the main functionalities of our approach. We now discuss some concerns regarding its validity. The main concern in our approach that may constitute a threat to validity are the models it relies on. Indeed, for an existing configuration to be found, the provided models must be correct and exhaustive. The FMs used in this paper have been manually described for illustration purpose, based on our experience in cloud services configuration and deployment. We thus had to limit our feature modeling to features which are explicitly released by cloud providers, since constraints finding and modeling for implicit features are far more complex. Moreover, due to the evolutive nature of cloud computing, *e.g.*, cloud providers that appear/disappear or existing environments evolving, the FMs described in this paper might not be valid anymore over the long term. One possible solution to tackle these challenges is to reverse-engineer cloud FMs from their web configurator [14]. Let us now consider the generation of configuration files and deployment scripts as described in SECTION II-C. Although the presented example deals with the Heroku PaaS example, the approach proposed here is not specific to this cloud layer. Indeed, if the targeted cloud environment is a IaaS, our approach can be used as input to configure and manage virtual machines [15].

## III. EVALUATION

The concepts described in this paper were implemented in the Java-based SALOON framework [8], [9]. In this section, we describe the experiments we conducted with SALOON to evaluate our approach. The intent of this evaluation is to answer the following research questions:

**R1:** *Practicality*. Is our SPL-based approach well-suited to support developers in the configuration and deployment of cloud environments?

**R2:** *Scalability*. We describe in this paper an example with three cloud environments, but is our approach still performing well when handling tens of cloud models?

The first experiment regards practicality. Its purpose is to evaluate the effectiveness and efficiency of our approach, compared to a manual configuration and deployment process. This experiment was conducted with a group of 10 participants, either Ph.D students or engineers, whose experience in cloud configuration and deployment spreads from beginner to experienced (1 to 4 respectively in Table. I). Each of these participants was given the same task: *Configure a Heroku environment, upload a web application, then add a PostgreSQL support.* The prerequisite is that Git and Eclipse must be installed on every participant computer, while we provided the web application (a basic *HelloWorld* application as .war file). They were then free to select the way they proceed, either using Git (G), the Eclipse plugin (P) or the web interface (W), but they had to time their experiment. Table. I describes the results of these experiments.

*Result Analysis.* The task was rather simple (adding support for a PostgreSQL database is straightforward, the web application does not have to be connected to the database) but it takes at least 19 minutes to be manually completed by an experienced participant (with a running application). One of them (#8) even gave up after several failed attempts. Moreover, the results show that whatever the way used to deploy, it can be very long to achieve the task, *e.g.*, participant #5 with a high level of experience and a dedicated plug-in. The last row of the table indicates whether the application is running or not at the end of the deployment. Indeed, an environment can be created and incorrectly configured, which may prevent the application from running properly. During a debriefing session, the participants explained that the main problem they met was to find out that a *Procfile* was required and/or what should be written in this file. They thus argued that if it could be automatically generated, they would have saved time. The need for an automated support is thus obvious, especially if the configuration is more complex. Moreover, in our experiment, we only consider one cloud provider but there are tens of them to be taken into account when considering deploying an application. To compare with our approach, we then asked participants to use SALOON to execute the same task. All of them are unexperienced with SALOON. They all selected *Java* and *PostgreSQL* in the Cloud Knowledge Model, and automatically retrieved the two generated files: the related *Procfile* and a file containing the commands required for this task. It then takes within a minute for them to be completed, where the main part is due to uploading files time (see [9]).

| Participant | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Time (min) | 26 | 19 | 32 | 26 | 48 | 60 | 17 | - | 23 | 28 |
| Method | G | G | G | P | P | G | G | G | W | G |
| Experience | 2 | 4 | 2 | 3 | 3 | 1 | 4 | 1 | 3 | 2 |
| App running | ✓ | ✓ | - | ✓ | ✓ | - | - | - | ✓ | - |

Table I: Configuring Heroku and deploying the application

[3]http://git-scm.com/

[4]https://toolbelt.heroku.com/

We then evaluate *(i)* the time needed to find a configuration for a real-world cloud system and *(ii)* the number of cloud environments that can be modeled in our framework while still running properly, to evaluate whether this time is a threat to scalability. We perform our evaluation on a MacBook Pro with a 2,6 GHz Intel Core i7 processor and 8 GB of DDR3 RAM. For the first evaluation, we select three cloud environments: *Heroku*, *Google App Engine* (GAE) and *Windows Azure* (WinAz). This evaluation aims at showing that the time required by SALOON to find a configuration is not a threat to scalability. These three clouds are selected as a representative panel of cloud environments, since they cover both IaaS and PaaS clouds with different sizes. Their related model, once translated to CSP, contains 67, 36 and 76 variables and 134, 74 and 162 constraints respectively. We then run SALOON with a set of requirements as input and measure the configuration time for each cloud FM. Each model configuration run is repeated 20 times, with a different set of requirements each time, and we compute the average configuration analysis time for each model. The average time we get from the experiment is 12, 20 and 22 milliseconds for the Heroku, GAE and WinAz model respectively.

| # models | 10 | 50 | 100 | 200 |
|---|---|---|---|---|
| Time (s) | 1,3 | 2,8 | 3,3 | 4,4 |

Table II: Feature selection and configuration analysis time

Finally, we compute the time taken by SALOON to configure an entire FM set. The aim of this evaluation is to evaluate whether the total configuration time is always reasonable, no matter what cloud environment is used in SALOON. We thus developed an algorithm that, given the number of features and constraints, generates a random cardinality-based FM with attributes and constraints over them, as described in SECTION II-A. We then generate random FMs to use them in our evaluation. Table. II presents these results, where the time is the average value computed over 50 different runs.

*Result Analysis.* The time required to execute this task is mainly due to the loading of the FMs and mapping models, which explains why it takes more than one second for 10 cloud models. Otherwise, solving 10 models is done within a few milliseconds. Overall, as our empirical evaluation shows, we observe that SALOON is well-suited to *(i)* handle an important number of cloud environments and *(ii)* deal with realistic cloud FMs, with a substantial number of features and constraints, on features, attributes or cardinalities. The time required to find a configuration is negligible (a few milliseconds) while 200 cloud FMs can be handled in less than 5 seconds.

## IV. RELATED WORK

Several cloud environment variability modeling and configuration approaches have been proposed in recent works.

In 2010, Van der Aalst [16] showed that handling variability is one of the main challenges to support configurable cloud services, and proposed configurable models to support cross-organizational processes mining. Calheiros *et al.* [17] developed the CloudSim framework for modeling and simulating cloud infrastructures. Clouds are described as abstract classes or interfaces at code level, which can then be implemented. This approach is well suited to simulate IaaS clouds but misses an abstraction level to handle properly both cloud selection and configuration. Ruiz-Alvarez *et al.* [18] use an XML schema format to describe cloud storage services and find which one is the best suited for a given dataset, relying on a specifically developed application. Our approach also supports that, and provides additionally a means of configuring automatically these services and expressing constraints between them using FMs. Some authors [19], [20] proposed a survey on existing approaches to model variability in cloud environment.

Moreover, FMs have been used in recent work to describe cloud services. Wittern *et al.* [21] present a cloud service selection process based on variability modeling. They rely on FMs to describe cloud services, but they handle neither cardinalities nor constraints over cardinalities and attributes. Galán *et al.* [22] propose to use an SPL-based approach to configure the Amazon IaaS. They describe Amazon EC2, EBS, S3 and RDS services as FMs and rely on off-the-shelf solvers to find a suitable configuration. The approach we propose in this paper goes in the same direction, but we go further in the SPL process. Our FM analysis is not limited to boolean FMs and thus handles properly the whole configuration. We also provide a tool to yield the related software artifacts. Schmid *et al.* [23] combine SPL engineering with service-oriented computing to deal with the variability of service platforms, *e.g.*, cloud platforms. Their paper explains how SPLs could help in such a case, but remains at a theoretical level, since no concrete example or validation is provided. Dougherty *et al.* [24] explain how virtual machine (VM) configurations can be captured by feature models. They also use attributes to define the energy consumption of a feature, in order to find a configuration that meets the requirements with the least energy consumption. Although this approach is closely related to ours, it does not provide means to reason about attributes and cardinalities, and does not automatically derive the VM configuration.

## V. CONCLUSION

Developers involved in cloud environment selection and configuration have to deal with a wide range of resources at different levels of functionality among available cloud solutions, leading to complex choices which are usually made in an ad hoc manner. In this paper, we describe an approach that addresses these issues, providing a reliable way to select a cloud environment, define a configuration for this environment and deploy the application. Our approach

relies on a combination of Software Product Lines (SPLs) and a domain model, enabling the developer to automatically *(i)* select a cloud environment that fits a set of requirements and *(ii)* get the description files and executable scripts to configure the related cloud environment. To evaluate our approach, we conducted experiments showing that configuring one cloud environment, even for a classic *HelloWorld* application, is not straightforward and leads to configuration errors in 50% of cases. Using an automated approach can thus lead to significant benefits when considering selecting among tens of cloud environments and deploying more complex applications.

For future work, we plan to take into account cloud environment evolution. As the cloud market evolves constantly, changes can occur that require the application environment to be reconfigured, *e.g.*, a non-functional requirement is violated or a new cloud provider is available. To deal with such changes, the evolution of SPLs based on FMs extended with attributes and cardinalities must be taken into consideration.

## Acknowledgments

We sincerely thank the SPIRALS participants for helping us with the experiments. This work is partially supported by the EU FP7 PaaSage project.

## References

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *National Institute of Standards and Technology*, vol. 53, no. 6, p. 50, 2009.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, Berkeley, Tech. Rep., Feb 2009.

[3] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility," *Future Gener. Comput. Syst.*, vol. 25, pp. 599–616, June 2009.

[4] M. Glinz, "On Non-Functional Requirements," in *15th IEEE International Requirements Engineering Conference, 2007. RE'07.*, 2007, pp. 21–26.

[5] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*, 2001.

[6] K. Pohl, G. Böckle, and F. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, 2005.

[7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) - Feasibility Study," The Software Engineering Institute, Tech. Rep., 1990.

[8] C. Quinton, D. Romero, and L. Duchien, "Cardinality-based Feature Models with Constraints: A Pragmatic Approach," in *Proceedings of the 17th International Software Product Line Conference*, ser. SPLC'13, 2013, pp. 162–166.

[9] http://researchers.lille.inria.fr/~cquinton/cloud/cloud2014.html.

[10] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated reasoning on feature models," in *Proceedings of the 17th international conference on Advanced Information Systems Engineering*, ser. CAiSE'05, 2005, pp. 491–503.

[11] O. Corcho, M. Fernández-López, and A. Gómez-Pérez, "Ontological Engineering: Principles, Methods, Tools and Languages," in *Ontologies for Software Engineering and Software Technology*, 2006, pp. 1–48.

[12] C. Parra, A. Cleve, X. Blanc, and L. Duchien, "Feature-based Composition of Software Architectures," in *Proceedings of the 4th European Conference on Software Architecture*, ser. ECSA'10, 2010, pp. 230–245.

[13] F. Fleurey, B. Baudry, R. France, and S. Ghosh, "Models in software engineering," H. Giese, Ed., 2008, ch. A Generic Approach for Automatic Model Composition, pp. 7–15.

[14] E. K. Abbasi, M. Acher, P. Heymans, and A. Cleve, "Reverse Engineering Web Configurators," in *17th European Conference on Software Maintenance and Reengineering (CSMR)*, 2014.

[15] C. Quinton, R. Rouvoy, and L. Duchien, "Leveraging Feature Models to Configure Virtual Appliances," in *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*, ser. CloudCP'12, 2012, pp. 2:1–2:6.

[16] W. Aalst, "Configurable Services in the Cloud: Supporting Variability While Enabling Cross-Organizational Process Mining," in *On the Move to Meaningful Internet Systems: OTM 2010*, ser. LNCS, 2010, vol. 6426, pp. 8–25.

[17] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms," *Softw. Pract. Exper.*, vol. 41, no. 1, pp. 23–50, Jan. 2011.

[18] A. Ruiz-Alvarez and M. Humphrey, "An Automated Approach to Cloud Storage Service Selection," in *Proceedings of the 2Nd International Workshop on Scientific Cloud Computing*, ser. ScienceCloud'11, 2011, pp. 39–48.

[19] H. Eichelberger, C. Kröher, and K. Schmid, "Variability in Service-Oriented Systems: an Analysis of Existing Approaches," in *Proceedings of the 10th international conference on Service-Oriented Computing*, ser. ICSOC'12, 2012, pp. 516–524.

[20] A. Benlachgar and F.-Z. Belouadha, "Review of Software Product Line Models Used to Model Cloud Applications," in *ACS International Conference on Computer Systems and Applications (AICCSA)*, 2013, pp. 1–4.

[21] E. Wittern, J. Kuhlenkamp, and M. Menzel, "Cloud Service Selection Based on Variability Modeling," in *Proceedings of the 10th international conference on Service-Oriented Computing*, ser. ICSOC'12, 2012, pp. 127–141.

[22] J. García-Galán, O. F. Rana, P. Trinidad, and A. Ruiz-Cortés, "Migrating to the Cloud: a Software Product Line based analysis," in *3rd International Conference on Cloud Computing and Services Science (CLOSER)*, 2013, pp. 416–426.

[23] K. Schmid, H. Eichelberger, and C. Krher, "Domain-Oriented Customization of Service Platforms: Combining Product Line Engineering and Service-Oriented Computing," *Journal of Universal Computer Science*, vol. 19, pp. 233–253, 2013.

[24] B. Dougherty, J. White, and D. C. Schmidt, "Model-driven Auto-scaling of Green Cloud Computing Infrastructure," *Future Gener. Comput. Syst.*, vol. 28, no. 2, pp. 371–378, 2012.