Towards an SPL-Based Monitoring Middleware Strategy for Cloud Computing Applications

André Almeida¹,², Everton Cavalcante², Thais Batista², Frederico Lopes³, Flavia C. Delicato⁴, Paulo F. Pires⁴, Gustavo Alves², Nélvio Cacho²

¹ Federal Institute of Education, Science and Technology of Rio Grande do Norte – Parnamirim, Brazil
² Department of Informatics and Applied Mathematics, Federal University of Rio Grande do Norte – Natal, Brazil
³ School of Science and Technology, Federal University of Rio Grande do Norte – Natal, Brazil
⁴ Department of Computer Science, Federal University of Rio de Janeiro – Rio de Janeiro, Brazil
{andregustavoo, evertonranielly, thaisbatista, fred.lopes, fdelicato, paulo.f.pires, gustavoalvescc, neliocacho}@gmail.com

ABSTRACT
Cloud-based applications are composed of services offered by distinct third-party cloud providers. The selection of the proper cloud services that fit the application needs is based on cloud-related information, i.e. properties of the services such as price, availability, response time, among others. Typically, applications rely on middleware that abstracts away the burden of direct dealing with underlying mechanisms for service selection and communication with the cloud providers. In this context, in a previous work we already discussed the benefits of using the software product lines (SPL) paradigm for representing alternative cloud services and their properties, which is suitable for the process of choosing the proper services to compose the application. As most cloud-related information are dynamic and may change any time during the application execution, the continuous monitoring of such information is essential to ensure that the deployed application is composed of cloud services that adhere to the application requirements. In this paper we present an SPL-based monitoring middleware strategy to continuously monitoring the dynamic properties of cloud services used by an application.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems – Cloud Computing; D.2.13 [Software Engineering]: Reusable Software – Software Product Lines

Keywords
Cloud Computing, Software Product Lines, Monitoring Strategy

1. INTRODUCTION
The pay-per-use model and the elastic nature of service provisioning and de-provisioning of the Cloud Computing paradigm is very appealing to support complex applications, mainly those that deal with computational and data-intensive activities. However, the development of complex applications that rely on services provided by several underlying Cloud Computing platforms is a hard task due to the heterogeneity of cloud environments that hampers the development of applications using different cloud services. Thus, applications rely on middleware that abstracts away the burden of direct dealing with underlying mechanisms for service selection and communication with the cloud providers.

In this context, the use of the software product lines (SPL) [1] paradigm for developing Cloud Computing applications was presented in our previous work [3], in which the benefits provided by the use of SPL for the static generation of the application were described. SPL enable the creation of a family (product line) of similar products using a common software infrastructure to build and configure parts designed to be reused across the products. SPL use the concept of feature to represent characteristics that are relevant to the application domain stakeholders, thus describing similarities and variabilities of the products in a product line. This is especially interesting for the cloud applications context since services provided by different cloud platforms (alternatives services) can be catered as variabilities to core features and represented as alternative features of the service. In addition, an extended feature model with properties associated to each feature (cloud service) enables the representation of any cloud-related information such as pricing, availability, response time and others Quality of Service (QoS) parameters of the provided service. In our former work, these properties are used in the selection of the proper cloud services that fit the application needs. For instance, different cloud platforms offer a storage service each one with different values for price and availability. An application can specify its requirements in terms of price and availability of a storage service, and our strategy is able to choose the specific cloud platform that offers a service that fulfill the requirements.

However, as most cloud-related information are dynamic and can change any time during the application execution, the continuous monitoring of such information is essential to ensure that the deployed application is composed of cloud services that adhere to the application requirements. In fact, monitoring such information is a time-consuming, non-trivial, and error-prone task, as it is necessary to continuously observe and analyze such dynamic information. Hence, it is essential to automate this monitoring process. In this context, the use of an SPL approach is useful for (i) representing the distinct cloud platforms that provide the set of services that compose the application, (ii) defining the valid combination of the services, and (iii) annotating the cloud-related information as properties of each service, thus making it easier the identification of the dynamic information to be monitored.

In this paper, we present a monitoring middleware strategy to (i) continuously monitor the dynamic properties of the cloud services that are required/used by an application and (ii) trigger a dynamic adaptation process upon the detection of changes that affect the requirements of the deployed application. This work
focuses on the monitoring strategy; the description of the adaptation process is out of the scope of this paper. Our monitoring strategy provides facilities to enable the user to specify constraints that the cloud services must fulfill. In order to implement the proposed strategy, we use a flexible monitoring component designed to be extensible, so that users can implement their own measurement techniques and aggregate them to the already provided techniques.

This paper is structured as follows. Section 2 briefly presents the background on SPL and our running example. Section 3 details our monitoring strategy. Section 4 discusses the implementation. Section 5 presents an analysis of our approach. Section 6 discusses related work. Finally, Section 7 presents our final remarks.

2. BACKGROUND

2.1 Software Product Lines

The SPL approach is driven by a planned reuse of software artifacts. Therefore, it arises as a paradigm that enables organizations to achieve a massive reuse of core artifacts by exploiting the similarities between products. Among the benefits achieved with this approach [1], it can be highlighted the reduction of the production time, greater flexibility, increased quality of the developed products, ability to perform customization, etc. SPL approaches usually identify commonalities (similarities) between all members of the family as well as characteristics that vary among them, the variabilities. In this perspective, the members of a family have a basic set of common functionalities (cloud services, in case of a cloud-based application) and associated variations (the different cloud platforms that offer each service) that individualize each of these members. Typically, commonalities and variabilities between products of a family are modeled in terms of features. A feature is a concept that is visible to any stakeholder involved in the application development and may be a requirement, a function, or a non-functional feature depending on the interest of such stakeholder. Features are organized in feature models [1] that represent similarities, variabilities, and constraints related to the variations between features and their relationships. Feature models often have a tree structure in which features are represented by tree nodes and the variations between features are represented by edges and feature groups, so that the hierarchical organization of the diagram describes the key concepts from more general to more specific concepts as they descend the tree. Features can be [1]: (i) mandatory, i.e. the feature must be included in a product; (ii) optional, i.e. the feature may or may not be included if the feature from which it derives is selected; (iii) or-inclusive, i.e. among the set of related features at least one of them must be selected, and; (iv) alternative, i.e. among the set of related features exactly one of them must be selected.

2.2 Running Example: Health Watcher

Health Watcher (HW) [2] is a Web-based system that enables citizens to register complaints and consult information related to the public health system of a city. This system was selected to serve as running example of this paper because it is a real, non-trivial application, thus having quality requirements found in several information systems, e.g. Web user interface, persistence, concurrency, distribution, and implementation technologies such as Java servlets, JDBC (Java Database Connection) and RMI (Remote Method Invocation).

2.2.1 Modeling Similarities and Variabilities in HW

In our previous work [3], we proposed a seamless adaptation of the SPL-based development to support specificities of cloud-based applications by adopting an extended feature model in order to introduce attributes to the features, in which an attribute is any characteristic of a feature that can be measured [12]. Similarly, we also ground on this idea of introducing attributes to features in the feature model with the notion of properties, which have the form of "<name, type, value" triples regarding a feature. Thus, a property can represent cloud-related information such as pricing, elasticity support, QoS parameters, among others. It is noteworthy that this idea of including properties is very flexible and goes beyond of simple attributes but actually augments the feature model encompassing any sort of information that may be useful in such model, so that it is possible to use these properties regarding the features to make inferences about the products generated from the SPL. In addition, the feature model becomes more expressive in order to represent important characteristics of the cloud services such as the pricing model (since users pay for the use of the services), availability, and response time.

Furthermore, we presented HW-CSPL, an SPL developed from the original HW system. The commonalities observed in HW-CSPL were proposed from the requirements and features in the original HW system and the different service facilities provided by cloud platforms led to the features that represent the variabilities in the SPL. Figure 1 illustrates the HW-CSPL extended feature model. This model contains mandatory features representing commonalities, such as: (i) Deployment defines which cloud platform is used to deploy the application in; note that the feature represents two alternatives for this service, namely Amazon Web Services or Google App Engine, and the relevant cloud-information for this service is price, which is represented as a property; (ii) Persistence, which defines the persistence mechanism of the application; (iii) Login System, which defines the infrastructure used for the authentication process, and; (iv) Log System, which defines the format for storing log information. This model also contains one optional feature, File Storage, which defines how files (e.g. images related to the application data) are managed in the application. Each one of these top-features has some properties regarding the services represented by their alternative feature groups. For instance, the Persistence feature has three dynamic properties (price, availability, and responseTime) and offers two options for application's data persistence, relational or non-relational, respectively represented by the Relational Persistence and Non-Relational Persistence. In turn, Non-Relational Persistence feature offers two additional options for non-relational persistence.

2.2.2 Cloud platforms and services in HW-CSPL

In our previous work [3], we have implemented several functionalities of the HW system using two Cloud Computing platforms, namely Amazon Web Services (AWS) [4] and Google App Engine (GAE) [5], in order to develop HW-CSPL. Besides these platforms are very well-known solutions on the market, they offer a wide range of services, a good support for the development of cloud applications through a well-defined API, and their provided services can be viewed as complementary services.
AWS Cloud Computing platform is widely used by companies of various sizes and domains and provides computational power, storage facilities, and other functionalities that allow companies to deploy applications and services at low cost, with great flexibility, scalability and reliability. Among these services, we can highlight: (i) Amazon EC2, which offers elastic computational resources by creating virtual machine instances to host applications; (ii) Amazon SimpleDB, which implements a simple non-relational database mechanism; (iii) Amazon S3, which allows file storage on the cloud, and; (iv) Amazon RDS, which enables creating relational database instances, such as MySQL and Oracle.

GAE platform focuses on supporting Web applications hosting. Virtualization and elasticity clearly observed in AWS are practically imperceptible in GAE since virtualization management and elasticity are automatically performed by the platform. Compared to AWS, it restricts the possibilities of configuring the application execution environment. While in AWS we have many possibilities for configuring applications, in GAE developers are constrained by pre-established rules applied inside the sandbox, which controls the access to the GAE resources through rules related to the use of multiple threads, sockets and access to the file system. Among the services provided by GAE, the following can be highlighted: (i) facilities for deploying a cloud application using the GAE infrastructure; (ii) DataStore, a text-based, non-relational persistence service for storing application’s data; (iii) Blobstore, which allows storing objects up to 2 GB, and; (iv) Log Service, which stores application logs into an internal file.

3. OUR MONITORING APPROACH

As discussed in Section 2.2.1, in our previous work [3] we proposed an SPL approach for developing Cloud Computing applications. In the mentioned work, the product generation is made at design time and the deployed product cannot be changed after deploying it. As pointed out in this work, the use of an extended feature model by adding properties to the features regarding QoS parameters, for example, can give more powerful information for generating a product to be deployed. However, QoS parameters and other dynamic-kind information may change over time. For instance, a persistence service provided by Amazon RDS may experience low levels of availability or high response times for a request, so that the deployed application would be affected by it. Therefore, it is necessary to constantly monitor such information and then take reactive actions when such changes happen since they affect the deployed application performance.

In this perspective, we propose a strategy to mitigate such drawbacks by extending our previous work [3] to face a more dynamic scenario. At design time, we must build the feature model and include in its features such dynamic properties, which provide information for dynamic monitoring and further adaptation. The defined properties must be monitored, at runtime, in order to select the product to be deployed based on user-defined criteria. For a first deployment, we use the current monitored values of the properties. Once the product (i.e. the application) has been deployed, a time interval is established to gather updated values of these properties. If there is any change on these values that affects the product already deployed, it is necessary to provide a dynamic way to make the redeployment. The description of the dynamic process is out of the scope of this paper.

Figure 2 illustrates our proposed monitoring approach, whose elements are conceptually described in the Sections 3.1 and 3.2.

3.1 Feature Monitoring Agent

The Feature Monitoring Agent was designed to gather values regarding to dynamic or static properties. Since we can have dynamic properties (e.g. response time and availability) and static properties, we defined a system that periodically gathers the runtime values and also supports the inclusion of static values (if it is necessary to consider them in the product definition). This component is based on QoMonitor [6], a monitoring system originally designed to monitor metadata in Ubiquitous Computing applications. For monitoring dynamic properties it is necessary to: (i) write a Web service that invokes a service on a specific cloud platform; (ii) register such Web service in the Feature Monitoring Agent, and; (iii) define the time interval between information gathering and thus applying the defined measurement rules for each dynamic property. These rules are user-defined and implemented using an interface provided by the monitor, called IMeter, which defines a method called measure to be implemented by the new measurement technique. There are several built-in measurement rules for dynamic properties, such as error rate, response time, mean time between failures (MTBR), availability, etc., but other properties can be easily added to the monitor by implementing the interface pro-
vided by the monitor. When the Web services that invoke cloud services are registered in the monitor, it is necessary to specify which SPL feature and its respective properties are associated to that Web Service. For instance, in HW-CSPL (see Figure 1) the Persistence feature has three properties (price, availability, and responseTime) and three associated variabilities, namely DataStore, Memory, and Relational Persistence. Since DataBase and Relational Persistence are variabilities respectively associated to GAE and AWS cloud platforms, it is necessary to write two Web services, one for each platform, and register them in the Feature Monitoring Agent. The measured properties are stored in a database (Feature Property Database in Figure 2) and are made available to the Adaptation Trigger component, which is responsible for triggering the dynamic adaptation based on such values and the Product Selection Criteria (see Section 3.3), which represents the user-defined function to select the best product to be deployed.

3.2 Feature Implementation
The Feature Implementation represents the artifacts (piece of code or class(es)) associated to the respective feature in the feature model. Following the principles of the feature-oriented software development [7], it is necessary to provide an implementation for each feature in the feature model, so that this implementation can be assigned to the deployed application (i.e. a generated product). Since we focus on deploying and monitoring an application that use services of several cloud platforms, each leaf feature (i.e. variability) represents a service regarding a specific cloud platform and the implementation to use such service. For instance, the Login System feature has two possible variabilities, namely Google Authentication and Database. For the Google Authentication variability, it is necessary to provide a set of classes that can interact with the Google Authentication cloud service and then perform the authentication. This is done for each variability in the feature model, which was designed to represent the possibilities of the deployed application to use services of different cloud platforms.

3.3 Product Selection Criteria
Since our goal is to deploy the best product (application) based on constraints defined by the user (user-defined criteria), we must provide a way to users describe such constraints. For this purpose, we define a Product Selection Function, which provides a specific function to specify the constraints used to determine the best product that suits such user-defined constraints. These constraints are expressed by using a set of operations and take as operands the feature properties. The Product Selection Function has two parts: (i) a Product Constraint Function, which is used to evaluate each product in order to determine if such product satisfied the established constraints, and; (ii) a Selection Function, which selects the best product based on the criteria defined on the function. Our product selection criteria support comparison, logical, arithmetic operators, and the maximum and minimum functions. For example, considering HW-CSPL and user-defined criteria that state that (i) the response time must be smaller than ten seconds regarding the Log System feature and (ii) the price regarding the Persistence feature must be the lowest one, this set of constraints for the Product Selection Function can be written as:

\[ \min(\text{Persistence.price}) \&\& (\logSystem.responseTime < 10) \]

Within the first brackets, the Selection Function specifies that the first criterion is the lowest price regarding the Persistence feature. In the second brackets the Product Constraint Function specifies the second criterion that is the response time regarding the Log System feature must be up to ten seconds.

In order to handle situations when more than one product fulfill the established criteria (for instance, two products that have the lowest prices and meets the availability criteria), it is necessary to refine these criteria. For example, it could be specified that the user is interested not only in the lowest cost associated to the Persistence feature, but also in the best availability associated to the Log System feature. Then, it is necessary to change the first part of the product selection function to express this new criterion:

\[ \min(\text{Persistence.price}) \&\& \max(\logSystem.availability) \&\& (\logSystem.responseTime < 10) \]

3.4 Adaptation Trigger
The Adaptation Trigger component is responsible for evaluating the product selection criteria and then triggering a dynamic process in order to embed the respective feature implementation code into the deployed application. The product is selected based on the values gathered by the Feature Monitoring Agent and the specified product selection criteria through an algorithm that generates all possible products and evaluates the product selection criteria using the gathered values regarding the feature properties. This algorithm is depicted in Figure 3.

Input:
FM – Feature Model
PCF – Product Constraint Function
SF – Selection Function
BProduct – Best Selected Product

Output: BProduct – best available product
1: LP = GenerateProductList(FM)
2: for each product p ∈ in LP do
3: if PCF(p) then
4: productTemp ← productTemp ∪ {p}
5: end if
6: end for
7: BProduct ← SF(productTemp)

Figure 3. Algorithm for selecting the best product

The algorithm depicted in Figure 3 takes as input three parameters: (i) the feature model (FM) described as an XML file; (ii) the Product Constraint Function (PCF), and (iii) the Selection Function (SF), described in Section 3.3. The output is the best available product. In line 1, a list containing all possible products (LP) is generated based on the feature model. From lines 2 to 6, the defined Product Constraint Function is applied for each generated product. If the current product being evaluated fits these constraints, then it is put in the productTemp list. In line 7, the Selection Function is applied to the products contained in productTemp, so that the best product is determined.

For this algorithm, it is necessary an XML description of the feature model (see Section 4.1) and also the product selection function. Since the Product Selection Function requires the values of properties of several features, the Feature Monitoring Agent must provide such values. After selecting the best product, the dynamic adaptation mechanism is triggered to weave the associated feature implementation into the application.

The Adaptation Trigger is composed of the following four elements. The Feature Monitoring Bean represents a class that accesses the properties’ values stored in the database within the Feature Monitoring Agent. The Parser component is responsible for parsing the XML representation of the feature model and generating all possible products, and also for parsing the Product Selection Function, represented by a single string. The Interpreter receives the list of products and applies the algorithm described in Figure 3. Although the Interpreter is aware whether one of the monitored properties has changed, the product selection algorithm will be executed only if the changed properties are used in the product selection criteria. Finally, once the Interpreter has applied the algorithm, the Application Adapter com-
ponent is notified to start up the dynamic adaption of the deployed application. The Application Adapter is responsible for adapting the running application. The description of this element is not in the scope of this work.

4. Implementation

In this section we describe the implementation of the main elements that compose our approach.

4.1 Feature Model

The feature model was described using an extended version of a feature model generated by the Feature IDE [8], which is a tool framework for generating XML representations of a feature model. Feature IDE is deployed as an Eclipse IDE plug-in and can be integrated with several tools to generate products. Since we are dealing with an extended feature model with annotated features [12], it was necessary to modify the originally generated XML file. Therefore, two new XML tags were introduced, namely properties and property. The properties tag starts a section in which each property associated to a specific feature is described through the property tag. For each variability, it is necessary to indicate a reference to a Feature Monitoring Bean, which is responsible for gathering the values regarding the described properties. The Adaptation Trigger component uses this reference to instantiate such bean and then analyze the generated products using the Product Selection Criteria. Figure 4 shows a fragment of the XML representation for the Persistence feature in HW-CSPL.

4.2 Feature Monitoring Agent

The QoMonitor solution [6] is used as the Feature Monitoring Agent in order to provide dynamic values of feature properties. For instance, to measure the responseTime and availability QoS properties from the Relational Persistence feature, we implemented a Web service hosted on an Amazon EC2 instance that inserts one line in a table on the Amazon RDS cloud service and then retrieves a line from such table based on a column ID. The response time is the time elapsed from the instant in which a method is called to its return. In several calls, we measure if there is any error in order to calculate the error rate. The availability is calculated by the overall operating time of such service. Since some cloud providers charge based on the number of requests, data storage, and so on, it is necessary to define the time between calls when the monitor is started. Without such constraint, the budgets and applications may quickly be overcharged. In a future work, we will discuss how we can optimize such measurement techniques.

4.3 Adaptation Trigger

The Adaptation Trigger component parses the XML representation of the feature model and applies the defined criteria to select the best product based on the information provided by beans that retrieve the values of the properties and the Product Selection Criteria. Since our implementation uses the Java programming language, we used the standard Java API to handle XML document for parsing purposes. After parsing the XML document, the algorithm shown in Figure 3 is applied. In an attempt to reduce the number of products that must be analyzed, we tried to use only the products that have variabilities associated to the selection criteria. One of our future challenges is to optimize the proposed algorithm since if we have an SPL with a high number of possible products then the time for generating the best product can compromise the adaption process. Based on the best product, we must trigger the adaptation process, which is not in the scope of this work that focuses only on the monitoring strategy.

5. Analysis

For a brief analysis purpose and considering the running example described in Section 2, we defined the following Product Selection Criteria to select the product by comparing the measured results against all possible products of HW-CSPL:

\[
\text{min}(\text{Persistence.price}) \text{ and } \text{min}(\text{FileStorage.responseTime}) \text{ and } \\
\text{min}(\text{Deployment.price}) \text{ if } \text{Persistence.responseTime} > 10
\]

In Table 1 we present the possible HW-CSPL products. The first column labels the HW-CSPL generated products. The next three columns indicate which cloud service is being used (if it is provided by AWS or GAE cloud platform) for Persistence, File Storage, and Deployment features. For each feature/platform (PL) we show the values monitored for price (P) in dollars and response time (RT) in seconds used in our product selection criteria. In our running example, which has run for four hours, we have established that a measurement for such feature properties was made every five minutes. The values regarding the price were defined based on the billing sheets provided by AWS and GAE and the values regarding the response time property. We do not consider yet how our Feature Monitoring Agent impacts on the project budget for product selection purposes. Based on the product selection criteria above specified and the last assessment of the measured properties, the selected product was HW5 (as highlighted in Table 1). In regards to the HW5 product, all features use the GAE cloud platform, which provides a better response time and offers a lower cost for deployment and use of the services. More details about the results of this preliminary experiment can be found at the following URL: http://www.dimap.ufrn.br/splmonitoring/

<table>
<thead>
<tr>
<th>HW-CSPL Prod.</th>
<th>HW-CSPL Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Persistence</td>
</tr>
<tr>
<td>PL</td>
<td>P</td>
</tr>
<tr>
<td>HW1</td>
<td>AWS</td>
</tr>
<tr>
<td>HW2</td>
<td>AWS</td>
</tr>
<tr>
<td>HW3</td>
<td>GAE</td>
</tr>
<tr>
<td>HW4</td>
<td>GAE</td>
</tr>
<tr>
<td>HW5</td>
<td>GAE</td>
</tr>
<tr>
<td>HW6</td>
<td>GAE</td>
</tr>
<tr>
<td>HW7</td>
<td>AWS</td>
</tr>
<tr>
<td>HW8</td>
<td>AWS</td>
</tr>
</tbody>
</table>

6. RELATED WORK

Although cloud platforms provide monitoring mechanisms for specific resources and services, they typically concentrate on gathering and reporting usage rates, performance and other information about their own services. In addition, the measurement techniques are proprietary and can be significantly different regarding distinct cloud platforms. In contrast, our work focuses on monitoring information related to alternative cloud services potentially used by a cloud application. To the best of our knowledge, this is this first work that presents a strategy to monitor SPL variabilities (cloud services) used by an application to be further used by a dynamic adaption process. There-
fore, we will compare this strategy with some proposals in the literature that aim to monitor different cloud infrastructure services and are related to our proposed direction.

Nahiju et al. [10] present Q-Clouds, a system designed to ensure that the performance experienced by the applications is independent of whether the Q-Clouds system is consolidated with other workloads associated to other deployed applications. Designed to be used by cloud providers, it constantly monitors QoS requirements defined by SLAs (Service-Level Agreements), and whenever any quality degradation is detected, the infrastructure is reorganized to meet the specified agreement. In our work we focus on the application itself by monitoring several cloud services provided by different cloud platforms and enabling the application to be adapted based on the monitored data since we do not have full control of the infrastructure, as can be observed in the use of public cloud platforms.

Chaves et al. [11] present a three-layered architecture for monitoring private clouds. Since the work is focused on private clouds and is integrated to a private cloud provider, the monitoring is done by analyzing the nodes that compose such private cloud, thus retrieving information for applications or administrator users. It introduces a monitor in every virtual machine that is started and sends the monitored information to an intermediate layer that gathers that information.

It is important to mention that the previously described works focus on the cloud infrastructure (i.e. adopt approaches that run in the provider side of the cloud) and rely on monitoring virtual machines, thus requiring complete knowledge of the virtualization process. Such approaches are not compatible with the use of public clouds. Unlike these works, our proposal is not restricted to be used in private cloud environments since it focuses on monitoring the cloud services used by a cloud application.

7. FINAL REMARKS

The development of cloud-based applications that are composed of services offered by distinct third-party cloud providers is a hard task due to the inherent heterogeneity of cloud environments. The selection of the proper cloud services that fit the application needs is based on cloud-related information. Our previous work [3] discussed the benefits of using SPL for developing Cloud Computing applications in terms of representing alternative cloud services and their properties. However, these properties are typically dynamic and may change any time during the application execution, so that it is necessary to continuously monitor such information to ensure that the deployed application is composed of cloud services that adhere to the application requirements. To address this issue, this paper proposes a monitoring middleware strategy to (i) continuously monitor the dynamic properties of the cloud services that are required/used by an application and (ii) trigger a dynamic adaptation process upon the detection of changes that affect the requirements of the deployed application. Considering that this work is focused on the monitoring strategy, we implemented an extensible monitoring component called Feature Monitoring Agent that enables to continuously monitor QoS parameters and other properties regarding cloud services. In addition, we proposed facilities to enable the user to specify constraints (generally called Product Selection Criteria) that the cloud services must fulfill. When combining these two elements, the strategy is able to evaluate and generate the best product to be deployed or even redeployed, if the application adaptation process is triggered due to changes in these dynamic properties. Our preliminary experiments showed that our monitoring strategy is effective in choosing the best services based on the user-defined criteria. In order to optimize our monitoring approach and to avoid the increase of cost generated by constant services monitoring, we are investigating the use of monitoring capabilities already available within the Cloud platforms. In the current state of our work, the dynamic adaptation process is under implementation. We are now studying available solutions for adaptation, such as the proposed by Villazón et al. [9], which enables dynamic weaving of pre-compiled code. Other issues must be addressed, such as situations in which the adaptation process changes the persistence mechanism, for example, changing from Google DataStore to Amazon RDS. In this case, how is the integrity of the previously stored information maintained? This is one of the questions that are being addressed in our ongoing research. Associated to the adaptation process, we also have to analyze the impact of triggering this process too many times, which can lead to an instability problem. Another question to be investigated is the scalability of our approach in case of monitoring several cloud platforms, each one providing several services. At last, we will address the ideal adaptation triggering interval and the adaptation impact on application performance.

8. ACKNOWLEDGMENTS

This work was partially supported by the Brazilian Academic and Research Network (RNP) through the AltoStratus Project, by FAPERJ grant E-26/102.961/2012 for Flavia Delicato, and by CNPq through the grants 307269/2010-8, 485935/2011-2, 311363/2011-3, and 311515/2009-6.

9. REFERENCES