Exploiting Software Product Lines to Develop Cloud Computing Applications

Everton Cavalcante¹, André Almeida¹,², Thais Batista¹, Nélio Cacho¹, Frederico Lopes³, Flavia C. Delicato⁴, Thiago Sena¹, Paulo F. Pires⁴

¹ Department of Informatics and Applied Mathematics, Federal University of Rio Grande do Norte – Natal, Brazil
² Federal Institute of Education, Science and Technology of Rio Grande do Norte – Parnamirim, 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

{evertonranielly, andregustavoo, thaisbatista, neliocacho, fred.lopes, fdelicato, diegodimap, lord.sena, paulo.f.pires}@gmail.com

ABSTRACT
With the advance of the Cloud Computing paradigm, new challenges in terms of models, tools, and techniques to support developers to design, build and deploy complex software systems that make full use of the cloud technology arise. In the heterogeneous scenario of this new paradigm, the development of applications using cloud services becomes hard, and the software product lines (SPL) approach is potentially promising for this context since specificities of the cloud platforms, such as services heterogeneity, pricing model, and other aspects can be catered as variabilities to core features. In this perspective, this paper (i) proposes a seamless adaptation of the SPL-based development to include important features of cloud-based applications, and (ii) reports the experience of developing HW-CSPL, a SPL for the Health Watcher (HW) System, which allows citizens to register complaints and consult information regarding the public health system of a city. Several functionalities of this system were implemented using different Cloud Computing platforms, and run time specificities of this application deployed on the cloud were analyzed, as well as other information such as change impact and pricing.

Categories and Subject Descriptors

General Terms
Design, Experimentation, Standardization.

Keywords
Cloud Computing, software product lines, cloud platforms, services, Health Watcher system.

1. INTRODUCTION
Cloud Computing [1, 2, 3] is a new computing paradigm that enables ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, applications, and servers) that can be rapidly provisioned and released with minimal management effort or interaction with the service provider [4]. Among the benefits commonly attributed to the adoption of this computing model, it can be highlighted the low initial investment in terms of physical infrastructure, reduction and greater control of operational costs, and high scalability. Although this paradigm provides benefits absent in current technologies, such as a pay-per-use model, and the elasticity of the applications (the capability of quickly increasing or decreasing the use of the computational infrastructure without incurring unnecessary costs with idle or underutilized resources), the development of Cloud Computing is still in its infancy [5]. In fact, building cloud-based applications is a challenging task as they are significantly more complex due to the intrinsic complexity of using third-part cloud providers. The major difficulties encompass issues such as the decision of which underlying cloud computing platforms to use, and the need of tracking pricing policies of services provided by different clouds platforms. There are several challenges that must be addressed in the Cloud Computing context, especially regarding methods, tools, and techniques to support developers to design, build and deploy complex software systems that make full use of the cloud technology. The particular nature of Cloud Computing applications creates specific requirements that also demand changes in terms of the development of such applications, encompassing methodologies and techniques for requirements elicitation, architecture, implementation, deployment, testing, and evolution of software [6, 7]. Moreover, to realize the vision of Cloud Computing in its full potential, new systems models must be developed in order to allow developers to focus only on the functional aspects of the applications, and handle the distribution and parallelization of the cloud as an orthogonal concern [8].

In software development, there is an essential need to reduce costs, effort, and time to market of software products [9]. In this perspective, software product lines (SPLs) [10, 13] represent an increasingly popular technology to support the derivation of a wide range of applications by promoting the systematic reuse of components and other software artifacts. SPLs have become a mainstream technique to the development of software systems that share a common set of commonalities and contain variabilities that distinguish specific products, thus supporting the develop-
ment of a family (or product line) of related products. Among the benefits achieved with the SPL approach, it can be highlighted a greater reuse of software artifacts, reduction of the production time, greater flexibility, increased quality of the developed products, ability to perform customization, etc.

Recent research has approached the synergic development of SPLs to the Service-Oriented Architecture (SOA) [14] concept, which is intrinsically related to the Cloud Computing paradigm. Both have common goals, such as promoting reuse of services, which enable rapid and easy composition of loosely coupled distributed software applications, rather than repeatedly redeveloping them for new systems. Moreover, they enable capitalizing on reuse to achieve desired benefits such as productivity gains, decreased development costs, higher reliability and others [15, 24].

As Cloud Computing is still an emergent area and it does not have a common standardized technological model, cloud platforms are not implemented using common standards, each one having its own APIs, development tools, virtualization mechanisms and governance characteristics. Considering the fact that such heterogeneity of Cloud Computing environments hampers the development of applications using different cloud services, SPL is a promising approach for this context since these specificities regarding cloud platforms and other aspects can be catered as variabilities to core features. The use of the SPL paradigm enables the specification and implementation of cloud applications as the different services facilities provided by cloud platforms, leading to the features that represent the variabilities in SPL. In the domain engineering phase of the SPL development, the different cloud platform services are represented as alternative features of a given service. In the application engineering phase, SPL enables the configuration of cloud applications according to the requirements, since it is possible to choose the proper cloud platform service that fit the application needs. However, we need to add expressiveness to the feature model 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.

The goal of this paper is twofold: (i) to propose a seamless adaptation of the SPL-based development to support important specificities of cloud-based applications, and; (ii) to report the experience of developing a SPL for the Health Watcher (HW) System [16], a Web-based system that allows citizens to register complaints and consult information regarding the public health system of a city. The main goal of this system is to improve the quality of the services offered by health institutions by enabling the public to report complaints about the quality of these services and to obtain information about health care units.

The HW system was selected to serve as running example of this paper because it is a real, non-trivial application, having quality requirements found in several information systems, e.g. Web user interface, persistence, concurrence, distribution, and implementation technologies such as Java servlets [19], JDBC (Java Database connection) and RMI (Remote Method Invocation). Moreover, HW was developed in a well-defined and documented layered architecture based on the MVC (Model-View-Control) architectural pattern, widely known and used for Web systems [20].

HW was developed to cater the needs of two actors: user and employee. The user can be any citizen that accesses the system through the Internet and that can make his/her complaint or consult information related to the public health services. When a complaint is registered by a user, it is forwarded to a specific department, which will deal with the complaint and return a response upon its investigation. This response will be registered in the system and will be available to be consulted by users.

Users can register a complaint in regards to different elements, having three options to make the register: (i) complaint about food, representing cases of suspected ingestion of contaminated food; (ii) complaint regarding mistreatment of animals or diseases transmitted by contaminated animals, and; (iii) complaints covering other cases (e.g. hygiene problems in restaurants, sewage leaks, etc.). After choosing one of these items, the citizen fills out a form for registering the complaint in the system. In order to investigate the registered complaints, the employees of sanitary control and health system departments can access the HW system by providing their username (login) and password.

2. THE HEALTH WATCHER SYSTEM

Health Watcher (HW) [16] is a Web-based system that enables citizens to register complaints and consult information related to the public health system of a city. The main goal of this system is to improve the quality of the services offered by health institutions by enabling the public to report complaints about the quality of these services and to obtain information about health care units.

The HW system was selected to serve as running example of this paper because it is a real, non-trivial application, having quality requirements found in several information systems, e.g. Web user interface, persistence, concurrence, distribution, and implementation technologies such as Java servlets [19], JDBC (Java Database connection) and RMI (Remote Method Invocation). Moreover, HW was developed in a well-defined and documented layered architecture based on the MVC (Model-View-Control) architectural pattern, widely known and used for Web systems [20].

HW was developed to cater the needs of two actors: user and employee. The user can be any citizen that accesses the system through the Internet and that can make his/her complaint or consult information related to the public health services. When a complaint is registered by a user, it is forwarded to a specific department, which will deal with the complaint and return a response upon its investigation. This response will be registered in the system and will be available to be consulted by users.

Users can register a complaint in regards to different elements, having three options to make the register: (i) complaint about food, representing cases of suspected ingestion of contaminated food; (ii) complaint regarding mistreatment of animals or diseases transmitted by contaminated animals, and; (iii) complaints covering other cases (e.g. hygiene problems in restaurants, sewage leaks, etc.). After choosing one of these items, the citizen fills out a form for registering the complaint in the system. In order to investigate the registered complaints, the employees of sanitary control and health system departments can access the HW system by providing their username (login) and password.

3. CLOUD PLATFORMS AND SERVICES

Aiming to illustrate our approach, we have implemented several functionalities of HW system using two Cloud Computing platforms, Amazon Web Services (AWS) [17] and Google App Engine (GAE) [18]. These platforms were selected because they are the most known solutions on the market. In addition, they offer a wide range of services, a good support for the development of cloud applications through a well-defined API, and the services provided by these platforms 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 several 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 enable creating relational database instances, such as MySQL and Oracle.
GAE platform is focused on supporting Web applications hosting. Virtualization and elasticity clearly observed in the AWS platform are practically imperceptible in GAE, since virtualization management and elasticity are automatically done 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 from 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 specific file.

One of the most remarkable characteristics of Cloud Computing is its pricing model. Cloud providers charge their services per-usage, hence there is no need to up-front commitment by users, thus enabling them to request and use only the necessary amount. In general, the pricing model supported by cloud providers is able to charge for computing at a very fine-grained level. Examples are measuring storage, bandwidth, and computing resources consumed.

In order to present a clear idea of what drives costs in cloud providers, we show the pricing model of AWS and GAE platforms. As these cloud providers offer pay-as-you-go services, there is a separate cost for the use of each service. Table 1 depicts some of these costs. The rows of the table represent features implemented for the HW system (first four rows) and infrastructure elements (last two rows), while the columns represent the price for using each feature/service for each provider. It is noteworthy that it is not easy to model the costs to operate HW at scale because cloud providers use many pricing dimensions to charge their services. For instance, when an Amazon EC2 instance is created to host the HW Web application, AWS will charge for the amount of time that the server is running and for the data that it is fetched. Additionally, the pricing for many of the services reflects volume discounts based on usage, that is, the more the service is used, the less it costs per event. Thus, pricing for services tends to decline over time.

Table 1. Costs regarding the use of cloud resources from AWS and GAE platforms for the HW features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>AWS</th>
<th>GAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistence</td>
<td>$0.11 per hour for SmallDB, with instance deployed in a single availability zone</td>
<td>$0.24 (GB per month)</td>
</tr>
<tr>
<td>Login System</td>
<td>No additional charge</td>
<td>No additional charge</td>
</tr>
<tr>
<td>Log System</td>
<td>$0.14 per hour for each Amazon SimpleDB machine</td>
<td>$0.24 (GB per month)</td>
</tr>
<tr>
<td>File Storage</td>
<td>$0.125 per GB up to 1TB/month</td>
<td>$0.13 (GB per month)</td>
</tr>
<tr>
<td>Virtual machines instances</td>
<td>$0.34 per hour for a large Amazon EC2 instance running Linux</td>
<td>No additional charge</td>
</tr>
<tr>
<td>Data transfer (out)</td>
<td>$0.12 per GB up to 10TB/month</td>
<td>$0.12 (per GB)</td>
</tr>
</tbody>
</table>

In regards to the SPL development, the task of deploying the HW on the cloud is even more complicated since it is necessary to examine the core assets, the product-specific assets, and the interaction between them in order to figure out the deployment cost. Therefore, one of the main challenges to use the SPL approach for cloud applications is how to find the true development costs, involving all of the cost associated with maintaining the services and data placed on the cloud. In this context, Section 4 describes an approach to estimate the costs to deploy an application on the cloud by using the information provided by the extended feature model proposed in our approach.

4. DEVELOPING A SPL ON THE CLOUD

4.1 Proposed approach

SPL approaches usually identify commonalities (similarities) between all members of the family as well as characteristics that vary among them, the variabilities. Thus, the members of a family have a basic set of common functionalities and associated variabilities that individualize each of these members. In this perspective, commonality and variability analysis is a powerful concept that enables specifying members of a SPL by taking into account services available by cloud service providers.

We have proposed an adaptation to the domain engineering activity that encompasses the elaboration of the application’s feature model. Feature models [11, 12] represent commonalities, variabilities, and variation-related constraints in terms of features and their relationships, being also used in the domain analysis activity and also in product derivation of a SPL. In general, feature models are structured as a tree, in which features are represented by nodes of this tree and the variations between features are represented by edges and feature groups, so that the hierarchical organization of the diagram describes the key concept starting from more general concepts to more specific concepts as they descend the tree. Features can be [11]: (i) mandatory, i.e. the feature must necessarily be included in a product; (ii) optional, i.e. the feature may or may not be included if the feature from which it is derived is selected; (iii) or-inclusive, among the set of related features at least one of them is selected, and; (iv) alternative, among the set of related features exactly one of them is selected.

There are multiple notations for feature modelling. In our approach, we have chosen to use an extended feature model inspired in the proposed one by Czarnecki et al. [21]. The feature model proposed by these authors enables introducing attributes to features, where an attribute is any characteristic of a feature that can be measured. The reason of introducing attributes or any kind of information is to enable a more concise representation of feature models. Of course, each attribute could be represented as a subfeature, but this quickly leads to very large and less understandable feature models. In this perspective, we ground on this idea of introducing attributes to features in the feature model with the notion of properties, where these properties (in the form of <name, type, value> triples) regarding a feature represent any information about it, in special, in our context, cloud-related information such as pricing, availability, elasticity support, Quality of Service (QoS) parameters, etc. Figure 1 shows a generic extended feature model that can be used to model cloud SPLs. For instance, the X1 feature contains information about the cloud service that is used by this feature, such as pricing model, availability, response time, etc. In our approach, we use the feature model for distinguishing properties of the system as well as expressing development alternatives.
It is noteworthy that this idea of including properties is very flexible and goes beyond simple attributes but actually augments the feature model encompassing any sort of information that may be useful in such model. Under this assumption, we can use these properties regarding the features to make inferences about the products generated from the SPL. In addition, our proposal enables to monitor dynamic aspects of the application that may affect the choice of a product. For instance, let us consider the pricing aspect of the products generated from a cloud-oriented SPL. Following the algorithm depicted in Figure 2, after defining the combinations of the features that represent variabilities in the SPL to generate the respective products (line 1), it is calculated the price of each generated product \( p \) based on these combinations.

Since the price of a product is the sum of the prices of all features that compose the product, for each feature \( f \) of a product \( p \), the price of \( f \) is added to the current price of \( p \) (line 4). To calculate the price of a feature (procedure \( \text{price}(f) \) in line 4), we use the \( \text{Pricing} \) property of the extended feature model regarding the cloud services used by the feature \( f \), by consulting the respective service provider about the prices. This can be a complex task because different cloud providers can adopt different pricing models, a typical situation in Cloud Computing environments.

Thus, with the prices of all products, the user can select one of them in the application engineering process based on their costs. It is very important to highlight that although we are concerned to the pricing aspect of cloud services in this paper, we could use any feature property to select the proper product.

4.2 The HW-CSPL software product line

The HW-CSPL (Health Watcher Cloud Software Product Line) was developed from the original Health Watcher (HW) system [15]. HW was originally designed to run in Java containers (e.g., Apache Tomcat) so that for an appropriated execution over different cloud platforms, it was necessary to know their specific requirements. The commonalities observed in HW-CSPL were proposed from the requirements and features in the original HW system. In turn, the different service facilities provided by cloud platforms led to the features that represent the variabilities in the SPL.

Figure 3 illustrates the HW-CSPL extended feature model. This model contains mandatory features representing commonalities, such as: (i) Deployment, which defines what cloud platform is used to deploy the application; (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. For each one of these features, there are alternative feature groups. For instance,
the Persistence feature offers two options for application’s data persistence, relational or non-relational, respectively represented by the Relational Persistence and Non-Relational Persistence features. In turn, the Non-Relational Persistence feature offers two additional options for non-relational persistence. In addition, the Relational Persistence feature contains a property called Pricing, which refers to the pricing model adopted by the cloud service (in this case it refers to the address of the AWS pricing model definition).

In order to enable variation in the proposed SPL, we have adopted the well-known conditional compiling technique. In conditional compiling, preprocessor directives indicate pieces of code that should be compiled or not based on the value of preprocessor variables, and such decision may be at level of a line (or a set of lines) of code or to a whole source code file. For instance, Figure 4 describes an excerpt of code of the HealthWatcherFacade class, which enables the access to the business classes of the system. In Figure 4, the conditional compiling variables relational or non-relational (lines 10 and 15) are used to determine when the enclosed code regarding two features (Relational Persistence and Non-Relational Persistence) must be compiled. As HW-CSPL was implemented using the Java programming language, and since it does not have native support for preprocessor directives, we have used a third-party tool called Antenna Preprocessor [22], released as an Eclipse IDE plug-in, to annotate the source code.

```
1. public class HealthWatcherFacade
2. {
3.  private PersistenceMechanism returnValue = null;
4.  if (ConstantHM.isRelational())
5.  {
6.      try
7.      {
8.         // Relational Persistence feature
9.         if (relational)
10.            returnValue = PersistenceMechanism.getInstance();
11.        #endif
12.     
13.     // Non-Relational Persistence feature
14.     if (nonRelational)
15.            returnValue = PersistenceMechanism.JDO.getInstance();
16.          #endif
17.     
18.     // Persistence mechanism connection
19.     returnValue.connect();
20.     }
21.     catch (PersistenceMechanismException x) {
22.        printlnStackTrace();
23.     }
24.     // ...
25. }
26. }
```

Figure 4. Example of variability with conditional compiling in the HealthWatcherFacade class.

Figure 5 depicts the main four HW-CSPL aforementioned features using the services provided by Amazon Web Services and Google App Engine cloud platforms. Due to space restrictions and as we have used the same technique for all features, Section 4.2.1 illustrates the use of the conditional compiling technique just for the Persistence feature since the same applies for the other features.

### 4.2.1 An example: the Persistence feature

In its original version, HW stores data in memory, using arrays, or in a local relational database, using the JDBC (Java Database Connectivity) API, which defines methods for accessing a database. For the Persistence feature, we have implemented other persistence mechanisms that make use of the facilities provided by AWS and GAE cloud platforms. The local relational database was replaced by the Amazon RDS relational database service, and we have developed another non-relational persistence mechanism that uses the GAE’s DataStore service, a text-based database.

According to the original HW architecture, the access to the data layer is done through the Factory design pattern [20]. A class called AbstractRepositoryFactory works as an access point to the classes that manipulate data related to the application model elements. The UML Class Diagram in Figure 6 shows the classes related to the Persistence feature: (i) JDORepository class, which implements the GAE’s non-relational persistence mechanism; (ii) RDBRepositoryFactory class, which implements the relational persistence mechanism using the relational MySQL database service provided by Amazon RDS service, and; (iii) ArrayRepositoryFactory, which implements the original persistence format, which is storing data in memory using arrays.

![Figure 5. Health Watcher functionalities using cloud services from AWS and GAE platforms.](image)

![Figure 6. UML Class Diagram illustrating the Factory classes related to the Persistence feature.](image)

In Figure 6, the AbstractRepositoryFactory class is responsible for providing methods to data persistence, so that an object of this class is used by the business layer to define which implementation of the persistence mechanism will be used. To do so, the conditional compiling is used based on the selected feature. Figure 7 shows the method getRepositoryFactory, which returns an object related to the persistence implementation that will be used by the business layer. In line 8 of Figure 7, if the conditional compiling variable persistence has value “relational”, then the line 9 is considered to the compilation process. In this case, the persistence mechanism will be done using a relational database, represented by a new object of the RDBRepositoryFactory class.

### 5. EVALUATION

In Section 3, we have described how to establish commonalities and variations of cloud-oriented SPL. We have also described how our proposed extended feature model and application algorithm can be used to derive cloud product line applications and to compute the costs of the different products. This Section presents an evaluation of such proposed approach. In Section 5.1, we analyze the design stability of the HW-CSPL software product line, and in Section 5.2 we discuss about cost estimation regarding the features that compose different products in the SPL. Finally, in
Section 5.3, we point out some lessons learned with the development of the proposed SPL.

5.1 Change impact analysis

We have quantitatively analyzed to what extent the introduction of each feature entails undesirable change propagations in the original HW implementation. This analysis relies on a suite of typical change impact measures [23], such as number of added or modified lines of code (LOC), among others. The purpose of using these metrics is to quantitatively assess the propagation effects, when introducing or changing a specific feature, in terms of different levels of abstraction, e.g. classes, operations, and LOC.

As observed in Table 2, the SPL development process applied in this paper have reduced the amount of LOC needed to be added, changed or removed, as well as classes and methods, mainly due to the use of the conditional compiling technique, probably if compared to developing different modules for each cloud platform, for example. It is important to highlight that we have simply added new features for the existing code but we have not changed the original HW architecture, design patterns or dataflow.

Table 2. Change impact measures in HW-CSPL.

<table>
<thead>
<tr>
<th>Action</th>
<th>Measures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changed classes</td>
<td>83</td>
</tr>
<tr>
<td>Changed methods</td>
<td>178</td>
</tr>
<tr>
<td>Added classes</td>
<td>15</td>
</tr>
<tr>
<td>Added methods</td>
<td>171</td>
</tr>
<tr>
<td>Changed inheritance relations</td>
<td>0</td>
</tr>
<tr>
<td>Added LOC</td>
<td>1514</td>
</tr>
<tr>
<td>Changed LOC</td>
<td>247</td>
</tr>
<tr>
<td>Removed LOC</td>
<td>4</td>
</tr>
</tbody>
</table>

5.2 Cost estimation

We have quantitatively analyzed to what extent the introduction of each feature entails undesirable change propagations in the original HW implementation. This analysis relies on a suite of typical change impact measures [23], such as number of added or modified lines of code (LOC), among others. The purpose of using these metrics is to quantitatively assess the propagation effects, when introducing or changing a specific feature, in terms of different levels of abstraction, e.g. classes, operations, and LOC.

As observed in Table 2, the SPL development process applied in this paper have reduced the amount of LOC needed to be added, changed or removed, as well as classes and methods, mainly due to the use of the conditional compiling technique, probably if compared to developing different modules for each cloud platform, for example. It is important to highlight that we have simply added new features for the existing code but we have not changed the original HW architecture, design patterns or dataflow.

Table 2. Change impact measures in HW-CSPL.

<table>
<thead>
<tr>
<th>Action</th>
<th>Measures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changed classes</td>
<td>83</td>
</tr>
<tr>
<td>Changed methods</td>
<td>178</td>
</tr>
<tr>
<td>Added classes</td>
<td>15</td>
</tr>
<tr>
<td>Added methods</td>
<td>171</td>
</tr>
<tr>
<td>Changed inheritance relations</td>
<td>0</td>
</tr>
<tr>
<td>Added LOC</td>
<td>1514</td>
</tr>
<tr>
<td>Changed LOC</td>
<td>247</td>
</tr>
<tr>
<td>Removed LOC</td>
<td>4</td>
</tr>
</tbody>
</table>

5.3 Lessons learned

By proposing a process to develop a SPL from the original HW system to generate products to be run on the cloud, we aimed at obtaining a flexible, dynamically changeable product. Our main objective was to analyze the HW features deployed in different cloud platforms, intending also to observe what features would present the best prices. For example, if AWS platform has a better price for storage and GAE has a better price for hosting the application, then the application can be deployed in GAE and be configured to store data using the Amazon S3 service provided by AWS.

We have verified that producing different modules for each cloud platform would be very tricky and error prone, with repeated code and a big size project. So, we have decided to use the conditional Google App Engine (GAE) cloud platforms. For this analysis, we have generated some products resulting from different combinations of the aforementioned features and daily execution of the operations, simulating a normal use of the application.

Table 3 shows the estimated deployment costs for these HW-CSPL products in the considered cloud platforms. The first column of Table 3 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 platform) for Deployment (D), Persistence (P), and File Storage (FS) features. Finally, the last three columns show the cost (in dollars) for executing the cloud product with the selected features for a time period of 7, 30 and 90 days. It is important to highlight that as Login System feature has no costs when the Google’s authentication service is being used, it was not included in Table 3, as well as local options for running the application. As the values for Log System feature are very small for AWS and GAE platforms, never reaching $0.01 for this analysis, this information was omitted.

Table 3. Estimated deployment costs for HW-CSPL products in AWS and GAE cloud platforms.

<table>
<thead>
<tr>
<th>HW-CSPL Products</th>
<th>Features</th>
<th>Cost (dollars per days)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>$/7</td>
</tr>
<tr>
<td>HW1</td>
<td>AWS</td>
<td>AWS</td>
</tr>
<tr>
<td>HW2</td>
<td>AWS</td>
<td>GAE</td>
</tr>
<tr>
<td>HW3</td>
<td>GAE</td>
<td>AWS</td>
</tr>
<tr>
<td>HW4</td>
<td>GAE</td>
<td>GAE</td>
</tr>
<tr>
<td>HW5</td>
<td>GAE</td>
<td>GAE</td>
</tr>
<tr>
<td>HW6</td>
<td>GAE</td>
<td>GAE</td>
</tr>
<tr>
<td>HW7</td>
<td>AWS</td>
<td>GAE</td>
</tr>
<tr>
<td>HW8</td>
<td>AWS</td>
<td>AWS</td>
</tr>
</tbody>
</table>

By analyzing Table 3, it is possible to observe that the GAE cloud platform offers better options in terms of price, since its free quota is bigger if compared to AWS. This means that, using the GAE platform, the user will start to pay later than AWS, for example. In this analysis, HW5 product (highlighted row in Table 3) is the cheapest option to be deployed using only GAE’s services. However, as described in Section 5.1, our approach envisions monitoring the resource usage, so that if costs roughly changes or free quotas provided by the platforms are exceeded, then this cost analysis can be redone and another product can be selected.
compiling technique that enables the selection of parts of the code according to preprocessor variables, which can be created, changed, or excluded anytime. For example, considering the File Storage feature, if it is verified that AWS is offering a cheaper storage service when compared to GAE, then the $s3$ conditional compiling variable is set to the true value, thus enabling the code responsible for storing the HW images to use Amazon S3 service. In addition, we can disable the code responsible for storing images at GAE Blobstore by setting the conditional variable googleImg to the false value. This process is similar for the other features. After choosing the desired parts of code that are enabled, the developer can easily generate the deployment file and host the application to start using the selected services.

As observed in Table 2, the SPL development process applied in this paper have reduced the amount of LOC needed to be changed, as well as classes and methods to be created. We have simply added new options for the existing code, with caution to not change HW’s original architecture, design patterns or dataflow.

Although out of the scope of this paper, an important issue to be highlighted is the detailed monitoring of costs at runtime, which will provide information about better costs related to the services that are being used by the features. The optimal ease is to dynamically deploy the features according to the best prices in the different cloud platforms. Currently, our cost algorithm computes the cost of all products and the user makes use of this pricing information to statically choose a product and the respective features to be deployed. However, as the pricing policy adopted by service providers can change at runtime and over time, it is necessary a dynamic monitoring cost algorithm.

6. RELATED WORK

To the best of our knowledge, there are no proposals in the literature that apply techniques to develop software product lines in the Cloud Computing context. As SOA and Cloud Computing paradigms share some concepts related to service-orientation (such as the notions of service itself, service provider and consumer and the contractual relationships between them, abstraction, loose coupling, and encapsulation), a natural way of thinking would be to extend or take advantage of some SPL-based approaches for SOA. As we already mentioned, SOA and SPL approaches to software development have common goals, such as encouraging an organization to reuse existing assets and capabilities rather than repeatedly redeveloping them for new systems, thus enabling rapid and easy composition of loosely coupled distributed software applications and other benefits [15, 24]. However, as we present in this Section, the literature does not present many proposals for developing service-oriented applications as software product lines. Here we highlight some interesting ideas provided by these works and that could be applied to our approach in future works.

Lee et al. [25] and Lee and Kotonya [26] propose a method that consists of an adaptation from feature-oriented product line engineering applied to service-orientation. The key idea is to split the overall method in two main activities, namely feature analysis and service analysis. Somewhat similar to the domain engineering in traditional SPL approaches, the feature analysis is concerned to identify externally visible characteristics (commonalities and variabilities) of products in a product line and organizing them in an exploitable feature model. Then, this feature model is analyzed through a feature binding analysis, which consists of the identification of services features that represent a major functionality of a system and may be added or removed as a service unit. In turn, the service analysis is intended to refine and restructure the feature model previously generated by introducing a separation of two distinctive service categories: (i) behavioral (or orchestrating) services, whose main role is the composition of other services and to deal with preconditions, effects, and invariants associated with these services in order to maintain system integrity, and; (ii) computational (or molecular) services, which are the basic building blocks (that can be composed of other atomic services) and can be reused as-is by behavioral services. This separation of concerns made in these works seems interesting because it is intended to capture observable service-orientation characteristics (e.g. those regarding system integrity) usually present in service-based applications and to try to correlate them with the feature model used as the main artifact of the SPL approach.

In turn, Medeiros et al. [27] propose an approach in which SOA applications are developed as software product lines. This approach starts with an identification activity separated in two sub-activities: (i) component identification, which receives as input the feature model regarding the product line and produces a list of possible components, and; (ii) service identification, which receives as input business process models and produces a list of service candidates for the product line architecture. Next, the variability analysis activity receives as input the lists of components and services identified previously and tries to reduce the number of candidates defining how variability will be implemented within services and components based on architectural decisions. Finally, the architecture specification activity receives the considered architectural decisions and is able to create different architectural views of the components and services, such as structural view, dependency view, etc. As highlighted in this work, an important aspect that can be considered in future works is to take into account business process models and architectural decisions since these elements can provide important information to the product line engineering process.

7. FINAL REMARKS

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 Computing environments that hampers the development of applications using different cloud services.

To address this problem, this paper reports our experience in applying the software product lines (SPL) approach in the developing of complex cloud-based applications. The contributions of the paper include: (i) a seamless adaptation of the SPL development process both at the domain and the application engineering phases in order to represent important information to cloud applications as attributes of the feature model, and to provide an algorithm that calculates the cost of the different products to enable a product static selection based on the total pricing; (ii) the details of the development of a complex real-world application following our SPL process and their evaluation considering the design stability and the cost estimation regarding the features that compose different products in the SPL, and; (iii) the lessons learned with the development of the application using the proposed approach.

It is worthwhile to mention that our SPL development process is generic and can be applied to other complex application domains. The use of extended feature model is not new but we are not
aware of any other proposal in the cloud domain. According to the
evaluation presented in Section 5, we can conclude that our SPL
approach is well suited for the use in the development of complex
cloud-based applications.

Future works include the extension of this proposal to continuous-
ly monitoring [28] the application cost at runtime in order to
trigger dynamic adaptation for huge cost saving or even for
achieving a trade-off between availability, elasticity, and cost.
Once consumer plays an important role in the service-orientation
context, we also intend to improve our approach in order to con-
sider consumer-side aspects regarding the use of cloud services.

8. ACKNOWLEDGEMENTS
This work was partially supported by the Brazilian Academic and
Research Network (RNP) through the AltoStratus CTIC Project,
and by the Brazilian National Council for Scientific and Techno-
logical Development (CNPq). Thais Batista and Flavia C. Delica-
to are partially supported by CNPq through the grants
and Thiago Sena are partially supported by the Rio Grande do
Norte Research Foundation (FAPERN), through the Project PPP-
III-013/2009. Nélio Cacho is also partially supported by the Bra-
zilian National Institute of Software Engineering (INES), founded
by CNPq through the grant 573964/2008-4.

9. REFERENCES
Distributed Systems Laboratory, University of California at
Berkeley, USA.
research challenges. Journal of Internet Services and Applica-
tions 1, 1 (Apr. 2010), 7-18.
Computing. NIST Special Publication. National Institute of
Standards and Technology, USA.
Implementation, management and security. CRC Press, USA.
cloud application perspective. International Journal of
Information Technology and Knowledge Management 2, 2
(Dec. 2010), 669-673.
cloud technologies. Computing Research Repository, Cornell
University, USA.
cloud from software modules. Proc. of the 2009 ICSE Work-
shop on Software Engineering Challenges of Cloud Comput-
ing. CLOUD'09. IEEE Computer Society, USA, 32-37.
[9] van der Linden, F. et al. 2007. Software product lines in ac-
tion: The best industrial practice in product line engineer-
ing. Springer-Verlag New York, USA.
and patterns. Addison-Wesley, USA.
(FODA) Feasibility Study. Technical Report. Software Engi-
neering Institute, Carnegie Mellon University, USA.
[12] Czarnecki, K., Helsen, S. Feature-based survey of model
transformation approaches, IBM Systems Journal 45, 3 (Jul.
2006), 621-645.
Engineering: Foundations, principles, and techniques.
Springer-Verlag New York, USA.
USA.
Applications – PL for SOA. Proc. of the 7th Int. Workshop
on Automated Specification and Verification of Web Sys-
tems. WWW 2011. Electronics Proceedings in Theoretical
Persistence as aspects, Software – Practice & Experience 36,
7 (Jun. 2006), 711-759.
[20] Buschmann, F. et al. 1996. Pattern-Oriented Software Archi-
tecture: A system of patterns. John Wiley & Sons, Ltd.
of the 1st ACM SIGPLAN/SIGSOFT Conference on Genera-
tive Programming and Component Engineering. GPCE’02.
Lecture Notes in Computer Science 2487. Springer-Verlag
London, 156-172.
[22] Antenna Preprocessor:
http://antenna.sourceforge.net/wtkpreprocess.php
software maintenance. IEEE Transactions on Software Engi-
neering 11, 9 (Sep. 1985), 849-856.
Proc. of the 11th Int. Conf. on Software Engineering Re-
oriented product lines. Proc. of the 12th Int. Software Prod-
uct Line Conference. SPLC 2008. IEEE Computer Society,
USA, 275-284.
with product line engineering. IEEE Software 27, 3 (May
2010), 35-41.
[27] Medeiros, F. et al. 2009. Towards an approach for service-
oriented product line architectures. Proc. of the 3rd Work-
shop on Service-Oriented Architectures and Software Prod-
uct Lines: SOAPL 2009. Software Engineering Institute,
Carnegie Mellon University, USA, 151-156.
360-degree compared. Proc. of the 4th Grid Computing Envi-
ronments Workshop. GCE’08. IEEE Computer Society, USA,
1-10.