Model-driven Application Refactoring to Minimize Deployment Costs in Preemptible Cloud Resources

Daniel J. Dubois[1], Catia Trubiani[2], Giuliano Casale[1]
[1]*Imperial College London, Department of Computing, London, United Kingdom, {daniel.dubois,g.casale}@imperial.ac.uk*
[2]*Gran Sasso Science Institute (GSSI), L'Aquila, Italy, catia.trubiani@gssi.infn.it*

***Abstract*—Performance assessment of cloud-based applications requires new methodologies to deal with the complexity of software systems and the variability of cloud resources. In this paper, we address the problem of reducing the total costs for running cloud-based applications while fulfilling service-level objectives (SLOs). To this end, we define an approach to refactor a cloud application in such a way that, when it is deployed, it requires less computational capacity and therefore less resources. We experimented our approach on top of a modified optimal provisioning heuristic designed for preemptible cloud resources and the results show that it reduces deployment costs, up to 60% when compared to the same approach, but without model-driven application refactoring.**

***Keywords*-software performance engineering; cloud computing; application refactoring; resource provisioning.**

I. Introduction

Cloud computing has become popular in organizations as a mean to outsource part of their IT infrastructures and lower their costs. As the number of cloud providers and offerings are increasing, it is becoming more challenging for organizations to make decisions on which cloud resources to rent for deploying their applications. Cloud resources are not only characterized by different performance in terms of processors, memory, etc., but also by different pricing models that result in different levels of reliability. For example, Amazon EC2 spot instances and Google CE's preemptible VMs are rented at a cheaper price than on-demand resources, but they carry the continuous risk of being claimed back by the cloud provider. While these types of resources operate at lower costs than others, they introduce new issues since they can become unavailable at any time.

Model-driven approaches stem from the idea of creating and quantitatively analyzing performance models from high-level specifications. Example of languages are UML diagrams [6] and Palladio [2, 19], which can be translated into performance models that are easier to analyze such as queueing networks (QN) [1]. Based on the generated performance models, model-driven approaches optimally refactor [18] the application architecture with the purpose of meeting SLOs [5, 26].

The focus of this paper is on the performance evaluation of cloud-based applications. We aim to exploit predictive performance results and provide feedback while considering two main sources of information: *(i)* software knowledge, *i.e.*, by experimenting design alternatives that optimize the application model, and *(ii)* resource knowledge, *i.e.*, by selecting cloud resources that minimize the deployment costs. To this end, the adoption of software refactorings is aimed at improving the system performance at the application level, when reasoning on predictive quantitative results may avoid an expensive rework, possibly involving the overall software system. Once the application model has been refactored, we apply a heuristic method to find optimally cheap cloud resources for the deployment.

In our previous work [8, 9] we have introduced an approach to minimize the costs of cloud resources while optimizing the allocation of software components. Instead, in this work we additionally consider model-driven application refactorings and we evaluate their impact in the process of minimizing cloud resources costs. By application refactoring we mean the set of software modifications that are usually correctness-preserving transformations aimed at improving the quality of the software. For example, the interaction between two software components might be refactored to improve performance by sending fewer messages with more data per message. This transformation does not alter the semantics of the application, but it can significantly improve the overall performance.

The goal of this paper is two-fold. First, at design time, we support software designers providing them a quantitative evaluation about how to efficiently deploy the software components into the available cloud resources. This will guide designers to initial architectural decisions for the application and to plan some alternatives (*e.g.*, provide alternative and compatible components to implement similar functionalities) with a quantitative evaluation. Second, architectural alternatives and the quantitative evaluation methodology can be repeated after the system has been deployed to suggest system modifications (*e.g.*, a software/hardware change) at run-time. Such suggestions can be used to guide a periodical manual or automatic reconfiguration on the running system.

The added value of our approach is that we jointly consider software and hardware knowledge to minimize the costs of renting cloud resources instead of focusing on just one aspect. We estimate the effectiveness of our approach by comparing the cost reduction obtained by using our model-driven application refactoring with respect to the state-of-the-art deployment approaches for preemptible cloud resources that do not use them. To the best of our knowledge, our approach is currently the only one that considers the costs of preemptible resources and their probabilities to be lost and recovered in its performance model.

The paper is organized as follows. Section II describes a motivating example. Section III states the research problem. Section IV presents our solution. Section V shows the experimental results. Section VI surveys related work. Section VII concludes the paper and outlines future research directions.

II. Running Example

Let us consider the SPECjAppServer J2EE benchmark application [21], which represents an enterprise-level business-to-business e-commerce workload of a realistic complexity. We use its performance model and evaluation conducted in [16], where actual observations have been validated vs model predictions. Figure 1 reports the application model expressed as a QN derived from the software model, which includes the following main components of the application: (*i*) users sending requests to the system; (*ii*) production line stations processing the manufacturing; (*iii*) application servers managing user requests; (*iv*) database servers including the processing and the I/O components.

The research problem we address in this paper is to minimize the cost of running this large-scale application on a modern cloud system characterized by different types of resources and different pricing models while guaranteeing the fulfillment of performance requirements. To run this optimization process we assume to have the following information: (*i*) a performance model of the software application and the available cloud resources; (*ii*) a pricing evaluation for using the modeled cloud resources along with their estimated availability; (*iii*) quality requirements (SLOs), *e.g.*, the required service response time; (*iv*) a system reconfiguration model where design alternatives are modeled as set of possible modifications for the original application model.

For example, the system reconfiguration model of our running example can specify the following refactorings. First, *software component replacement* means that some software components can be replaced with more efficient ones, *e.g.*, application servers of a vendor can be replaced by others of another vendor. Second, *software component merge* means that two or more software components can be replaced by one that offers both functionalities, *e.g.*, the processing and the I/O components of the Database can be replaced by a single Database component offering both functionalities. Third, *software component reassignment* means that multiple instances of the same software components may redirect and specialize the load they can process, *e.g.*, the first replica of the application server serves in login/logout requests, whereas the second one manages user profile update requests.

Note that the actual application of software refactorings can be limited by *design constraints*, for example a legacy component may not be allowed to be replaced or merged because no compatible alternatives exist. Moreover, a component may not be replicated due, for example, to licensing issues and this would prevent the reassignment of its load.

III. System Model and Problem Statement

A. System Model

The system model we consider is composed of two parts:

Application Model: we model the application as a closed QN and a set of constraints that define the SLOs. A QN is a formalism composed as a graph in which nodes are interconnected software components and requests spend part of their execution time being processed in these components. We have chosen a closed QN model because it can be generated from application models (*e.g.*, UML-based models [6]) and be efficiently evaluated using fluid models [17, 19].

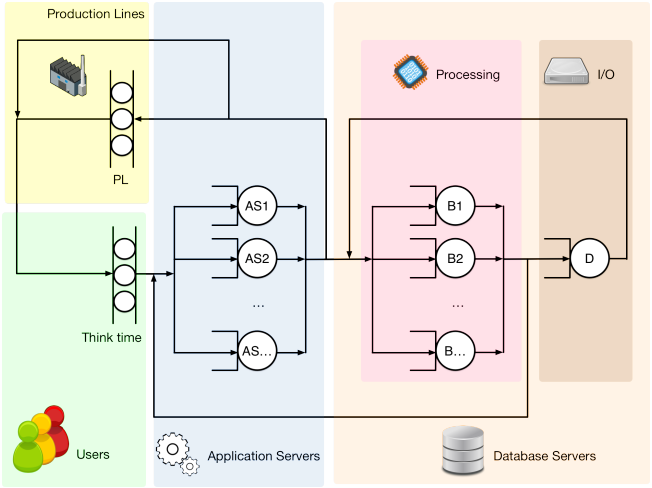


Figure 1. Queueing Network (QN) model for SPECjAppServer [16].

Table I
Application and Resource Parameters

M	Software components, $0 \leq m \leq M$, where $m = 0$ represents the delay node that models the think time.
K	Classes of requests, $1 \leq k \leq K$. This parameter is used to differentiate different types of requests (*e.g.*, the creation of an entry in a database and the update of the same entry can be two different classes of requests).
N_k	Total number of requests of class k in the system. This parameter specifies the system workload.
$\mu_{m,k}$	Class service rate. Average number of class-k requests completed at component m in a time unit.
$v_{m,k}$	Request visits. Number of times a class-k request visits component m before being completed. This parameter summarizes information about how requests are routed among the software components.
SLO_k	SLO on the response time for class-k requests.
R	Types of resources that can be rented, $0 \leq r \leq R$.
λ_r	Nominal service rate of resources of type r. It expresses the number of requests that the resource can process in a time unit when hosting a software component that has class service rate equal to 1. This is a measure of the total computational capacity (*e.g.*, Amazon EC2's ECU).
a_r	Availability of a resource of type r. It represents the percentage of time in which the resource is expected to be available.
$c(r,a)$	Expected price for renting a single resource of type r with availability a for a fixed amount of time.

Resource Model: we model the set of possible resources that can host software components with information about their rate (a metric of speed), cost, and availability. The cost and availability parameters are determined by the type of preemptible cloud and the price prediction strategy.

Table I schematically reports the main parameters for the application and the resources.

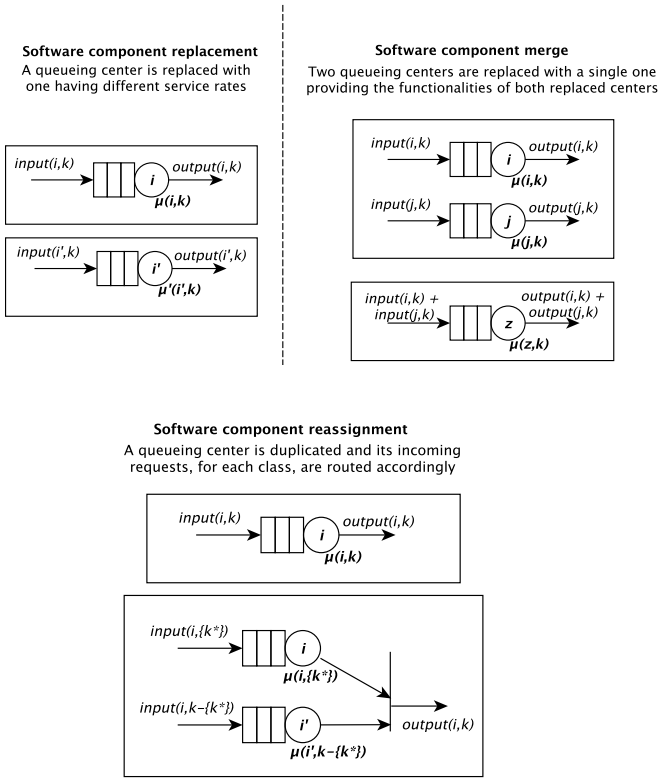


Figure 2. Software refactorings.

B. Reconfiguration Decisions

Our system can be reconfigured according to a set of *software refactoring decisions* (*e.g.*, replications and substitutions of software components) aimed at modifying the application itself while guaranteeing consistency, and a set of *deployment decisions*, which is used to specify the allocation of software components to real cloud resources. Table II reports the parameters involved in refactoring decisions.

Software refactoring decisions. These are defined as a set of modifications in the parameters of the application model. The software refactorings that we consider are shown in Figure 2 and described hereafter:

Software component replacement: a software component (*e.g.*, an Application Server) is replaced with a different software component that provides the functionalities of the replaced component (*e.g.*, another Application Server from a different vendor). Applying this refactoring on the component i using a replacement component i' characterized by service rate $\mu_{i',k}$ means that the service rate of the original component $\mu_{i,k}$ is replaced with the new one. The set of component replacements is defined as V_m in Table II.

Software component merge: two distinct software components (*e.g.*, a Web Server and an Application Server) are replaced with a software component that provides the functionalities of the two replaced components. The QN is transformed by replacing the service rate of the first software component with the new software component in the same way as in the previous case. The second component is simply removed from the QN, and its input and output links are redirected to the replaced component. The set of merge components is defined as $W_{m1,m2}$ in Table II.

Software component reassignment: it is the functional separation of a software component instance into two ones responsible for different classes of requests (*e.g.*, an instance of *AS* is divided into two instances: one to register new users and one to authenticate new users). The result of this refactoring applied on component i and request class k is that a new component replica i' is added to the QN, with $\mu_{i',k} = \mu_{i,k}$. The QN topology is then changed in such a way that all the requests of class k are rerouted from component i to component i'.

Deployment decisions. These are allocation decisions for each software component to a set of cloud resources. Given a set of Y rented resources, they can be expressed through an *allocation matrix* [8]:

$$D = [d_{m,y}],\ 1 \leq m \leq M,\ 1 \leq y \leq Y$$

The allocation matrix specifies the amount of service rate λ_r associated to the rented resource y that is used to deploy the software component m. D is able to express both simple and complex deployment scenarios in which a component m is associated to multiple resources (*i.e.*, replication and load balancing) and when a rented resource y is hosting multiple components (*i.e.*, resource partitioning).

Design constraints. These constraints are used to limit the application of refactoring and deployment decisions when a software component cannot be replaced or replicated. These constraints are defined as the non-replaceable set *CRep*, and the non-replicable set *CLim* in Table II.

Table II
REFACTORING PARAMETERS AND DESIGN CONSTRAINTS

V_m	Set of alternative software components that can be used to replace component m, $1 \leq m \leq M$, to apply a software replacement refactoring on it.
$W_{m1,m2}$	Set of alternative software components that can be used to replace components $m1$ and $m2$, $1 \leq m1, m2 \leq M$, to merge them.
CRep	Set of software components that cannot be replaced, limiting the application of software replacement and merge refactorings.
CLim	Set of software components that cannot be replicated, limiting the application of reassignment refactorings and replication deployment decisions.

C. Problem Statement

Given as input the parameters reported in Tables I and II, we want to decide:

- A refactored queueing network *QN'* derived from *QN* through a set of application refactorings that do not violate the design constraints *CRep* and *CLim*.
- The list of Y resources $y_1, y_2, \ldots \in R$ that need to be rented to deploy the software components.
- The allocation matrix $D = [d_{m',y}]$, $1 \leq m' \leq M'$, $1 \leq y \leq Y$, that specifies the amount of component m' that is allocated to the rented resource y, while taking into account the design constraints *CLim*.

The following optimization problem is considered:

$$\min \quad \sum_{y=1,\ldots,Y} \hat{c}_y$$

$$\text{s.t.} \quad \sum_{m'=1,\ldots,M'} d_{m',y} \leq \hat{\lambda}_y, \forall y \qquad (1)$$

$$SVC_k(D) \leq SLO_k, \forall k \qquad (2)$$

The goal is to minimize the total cost of renting cloud resources, while fulfilling the performance goals. M' and m' refer to the parameters of the refactored queueing network *QN'*; $\hat{c}_y = c_r$ and $\hat{\lambda}_y = \lambda_r$ are respectively the cost and nominal rate of a rented resource y of type r. The first constraint avoids deploying to a resource more software components that it can actually fit. In the second constraint, $SVC_k(D)$ represents the actual response times derived from the evaluation of the refactored queueing network *QN'* when deployed in the resources Y according to the allocation matrix D, which must always be smaller than the required SLO_k, for each request class k.

IV. Approach

A. Overview

Our approach focuses on the decision-making steps of the iterative process illustrated in Figure 3. The starting point is constituted by the system model, which depends on the application and the environment (*e.g.*, expected number of users, expected price fluctuations, etc.), while the output is the information on how to deploy the application.

We identified three main steps of our decision-making process: (*i*) apply to the initial model the *merge and replace* refactorings to reduce the overall computational needs of the system; (*ii*) *reassign requests* and create component replicas to reduce the complexity of the QN; (*iii*) calculate an optimal deployment by deciding which resources to rent and how to map components to them.

After the third step, we obtain a decision on the desirable configuration of the system that minimizes the overall costs, which may be used to perform a reconfiguration of the deployed system. Since the application requirements and the environment may change overtime, we expect our approach to be implemented as the decision-making part of a loop in which the optimization and adaptation processes are repeated when there are significant changes in the system model.

B. Algorithm

Our algorithm has the goal of finding an optimal solution to the problem stated in Section III-C. In particular, the inputs are the application model of the system – expressed or translated into a QN, the resource model, the refactoring parameters, and the design constraints. The outputs are: the refactored QN, a list of the resources to be rented, and the allocation matrix that associates each software component to the cloud resources. The actual algorithm has been implemented in MATLAB and available for download in [20].

Step 1 – Replace and merge software components: we modify the application model to take advantage of the alternative components that can replace one or more existing components. A detail of the algorithm for replacement is shown in Figure 4, with some high-level functions explained in Table III. The idea is to create a new QN for the application by replacing one or more components that are not in the *CRep* set in such a way that the component substitution reduces the utilization of the component. To do this, we calculate the utilization (U) of replaceable components by evaluating the QN before and after the refactoring (lines 4 and 12). If the utilization becomes lower, then we perform the substitution. In the case of merge, the algorithm and the idea is very similar, with the difference that two components are replaced by a single one.

Step 2 – Duplicate bottleneck and reassign requests: in this step we modify the QN in such a way that software components that are bottleneck for a particular class of requests are duplicated to reduce their load, and all the requests of the bottleneck class are forwarded to the new replica. Figure 5 reports a detail of the algorithm we developed, with some high-level functions explained in Table III. The idea is that we first identify the components that can be duplicated (*i.e.*, do not violate our design constraints, identified by the input

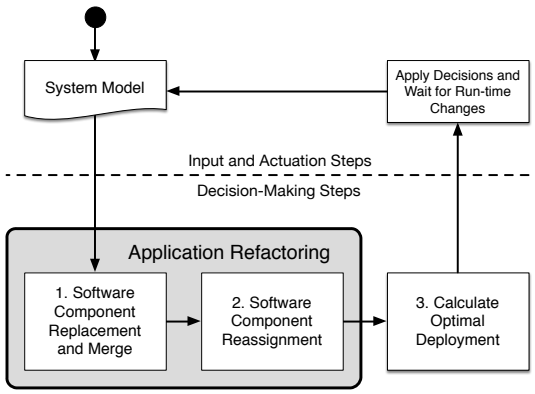


Figure 3. State diagram of our iterative approach.

```
1:  function swRefactorReplacement(QN, SLO, V, CRep)
2:      λ̂ ← findOptimalRates(QN, SLO)
3:      newQN ← QN
4:      U ← evalUtilization(newQN, λ̂)
5:      for i ∈ (QN.getComponents() \ CRep) do
6:          for alt ∈ V_i do
7:              altQN ← newQN
8:              replaceComponent(altQN, i, alt)
9:              altU ← evalUtilization(altQN, μ̂)
10:             if altU(i) < U(i) then
11:                 newQN ← altQN
12:                 U ← altU
13:             end if
14:         end for
15:     end for
16:     return newQN
17: end function
```

Figure 4. Algorithm for refactoring the queueing network *QN* and the given *SLO*, with the alternative components in $V = \{V_1, \ldots, V_m\}$ and replaceable components constraints *CRep*. The result is a new refactored queueing network *newQN* in which replaced components have a lower utilization than the ones in the original QN.

```
1: function swRefactorReassignment(QN, SLO, CLim)
2:     λ̂ ← findOptimalRates(QN, SLO)
3:     newQN ← QN
4:     nodes ← QN.getNonDelayNodes()
5:     for n ∈ nodes do
6:         if notDuplicable(n, QN, CLim) then
7:             nodes ← nodes − n;
8:         end if
9:     end for
10:    repeat
11:        stop ← true
12:        nodes ← nodes − QN.getSingleClassNodes()
13:        μ̂ ← QN.μ × QN.v × λ̂
14:        for k ∈ QN.getClasses() do
15:            minμ̂(k) ← min(μ̂(nodes, k))
16:        end for
17:        for n ∈ nodes, k ∈ QN.getClasses() do
18:            if μ̂(n, k) = minμ̂(k) then
19:                u ← duplicateNode(n)
20:                newQN ← newQN.addNode(u)
21:                newQN ← newQN.redirect(n, u, k)
22:                if notDuplicable(n, newQN, CLim) then
23:                    nodes ← nodes − n
24:                end if
25:                stop ← false
26:            end if
27:        end for
28:    until stop
29:    return newQN
30: end function
```

Figure 5. Algorithm for refactoring the queueing network *QN* with the given *SLO*, in such a way that bottleneck components that are not constrained by the *CLim* parameter can be replicated and all the requests of the bottleneck classes are redirected to the replicas. The result is the new refactored queueing network *newQN*.

parameter *CLim*) and that are responsible for processing more than a single class of requests. If there is a component m that has the lowest aggregate service rate:

$$\hat{\mu}_{m,k} := \mu_{m,k} \times v_{m,k} \times \hat{\lambda}_m$$

for a given class k, then the component is duplicated and all the requests of class k are forwarded to the new component. Here, $\mu_{m,k}$ and $v_{m,k}$ are input parameters characterizing the QN (see Table I), while $\hat{\lambda}_m$ is the nominal resource rate of the resource hosting the component.

Step 3 – Calculate optimal deployment: in this step we calculate the optimal deployment using a modified version of the state-of-the-art approach developed in [8], which is currently the only one that considers the effect of a limited resource availability that is typical of preemptible cloud resources. In this paper we introduce support for design constraints. In particular, we first assign the constrained software components (*CLim*) to the cheapest available resources that are big enough to contain them without replication, and then we execute the original optimal deployment heuristic on the remaining resources. This step finally produces the set of resources to be rented and the allocation matrix corresponding to the refactored QN.

C. Benefits and Limitations of Application Refactoring

The *Software component replacement and merge* refactorings guarantee that one or more software components are replaced with faster components according to the modeled system load. However, this comes with an additional cost of evaluating the QN every time it is refactored.

Table III
DESCRIPTION OF THE FUNCTIONS USED IN THE PSEUDOCODE

findOptimalRates	Calculate the minimal rates $\hat{\lambda}$ for each node that satisfied the given SLO using a bisection-based minimization method.
evalUtilization	Calculate the utilization of each node under the assumption that such node has its rate scaled by $\hat{\lambda}$.
notDuplicable	Test whether a node can be duplicated or not, according to the design constraints.
QN.addNode	Add a node to the QN.
QN.getComponents	Get the list of nodes corresponding to application components.
QN.getClasses	Get the list of all the classes of request k.
QN.getSingleClassNodes	Get the list of nodes that represent application components that process a single class of requests.
QN.getNonDelayNodes	Get the list of nodes that represent delay nodes.
QN.redirect	Redirect all the requests of a given class k that are sent to a node, to another node.
replaceComponent	Replace the rate $\mu_{i,k}$ of the given component i with the rate of an alternative component $\mu_{alt,k}$, for each possible class k.
duplicateNode	Create a replica of the given node.

The *Software component reassignment* refactorings, when applicable, replicate existing components to separate the classes of requests among the replica. These refactorings do not improve the performance of the original QN, but generate a configuration that leads the deployment optimization algorithm to give better results. The reason is that the new refactored QN has a higher chance of having software components that specialize in a single class of requests, therefore different classes can be scaled independently from the remaining ones in the deployment step, leading to a significant reduction of required resources and costs.

V. Evaluation

We evaluate our system under different scenarios inspired by a real system. The next paragraphs describe the setting for our experiments, and the results we have obtained. For each experiment we show the standard deviation over a set of 20 repeated runs. The MATLAB source code and the Amazon EC2 price traces we used to perform these experiments are available in [20].

A. Experiment Setting

Application setting. As an example application we use a system model derived from the SPECjAppServer benchmark, as illustrated in Section II. The system is composed of an Application Server (AS), and a Database subsystem. The Database subsystem is modeled as two components, a processor component DB-CPU and the storage component DB-I/O. The closed QN is depicted in Figure 6, which is an approximate version of the QN previously shown in Figure 1. The system has 5 classes of requests: (*k1*) NewOrder, (*k2*) ChangeOrder, (*k3*) OrderStatus, (*k4*) CustStatus, and (*k5*) WorkOrder.

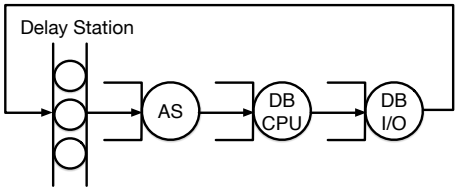


Figure 6. Approximate QN for the SPECjAppServer benchmark [16].

Table IV
SPECJAPPSERVER PARAMETERS FOR 1 ECU (HEAVY PROFILE)

Parameter	k1	k2	k3	k4	k5
Number of requests (req)	100	50	150	50	200
Think time (s)	3	3	3	3	5
$\hat{\mu}_{AS}$ (req/s)	77.04	73.31	378.8	393.7	41.29
$\hat{\mu}_{DB\text{-}CPU}$ (req/s)	93.98	96.53	403.2	480.8	29.29
$\hat{\mu}_{DB\text{-}I/O}$ (req/s)	892.9	787.4	1724	3333	595.2

We assume that, if every component of the system runs on a cloud resource of unitary computation power (*e.g.*, 1 Amazon ECU: Elastic Computing Unit), we obtain the same behavior reported in [16] in terms of service rates for each component. The actual parameters of the model are shown in Table IV, where the first two rows (*i.e.*, number of users N_k, representing the requests and the think time) report the system workload, whereas the last three rows represent service rates whose values have been calculated as the inverse of the service demands in [16].

Resources setting. As resource setting we re-use the resource model in [8]. This model is based on the Spot VMs offered by Amazon, which are cheaper instances that may be lost at any time depending on the bid price that is offered. To decide which price to offer we choose a resource availability $a_r = 95\%$, which was shown to be optimal in [8]. The actual price is estimated from real historical Amazon EC2 traces which cover a 14-month period up to January 2011 that can be downloaded from [20]. It is important to notice that the price prediction strategy used here is based on the assumption that behavior from past traces is correlated with future traces; however, depending on the situation this assumption may be different. The purpose of our approach is not to propose or replace existing methods for forecasting resource prices, but to complement them.

B. Parameters

The input parameters for our experiments are:

Number of requests (N): total number of requests in the system, as the sum of the requests for each class k. The default profile of requests is proportioned to its total number, *i.e.*, the *Heavy Load* scenario in [16].

Profile (N_k): the profile specifies how the total requests in the system are divided into classes. As default values we consider the number of requests for each class reported in Table IV. This parameter is equivalent to the number of users of the QN (see Table I).

Replaceable set (V): we consider only the AS as a replaceable component with replaceable set (V_{AS}). We use five alternative replacements $V_{AS}(x)$ ($x = 1,\ldots,5$) where, if class $k = x$, we have an alternative component with the service rate for that class $\mu_{AS,x}$ halved, and the service rates for the other classes doubled. This parameter allows us to verify if our algorithm actually takes advantage of those alternatives when performing software replacement refactorings.

Design constraints on node replicability (CLim): this parameter contains the set of nodes that cannot be replicated.

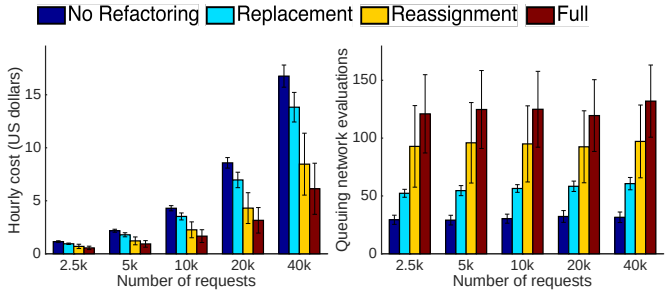
Performance constraints (SLOs): maximum mean response time for the 5 classes: 100ms, 100ms, 20ms, 20ms, 1000ms; maximum response time for the 80% of the requests is double than the maximum mean response time. These constraints are kept constant for all the experiments and are derived from the measurements reported in [16] for the high load scenario.

The output parameters we report are: (*i*) *hourly cost*, which represents the expected hourly cost we have to pay to deploy the system in the Amazon EC2 spot instances; (*ii*) *QN evaluations*, which is the number of times QN models are evaluated by our algorithm. The evaluation of QN models is the operation that dominates the execution time, therefore we use it as a cost metric that is related to processing time. In our experimental setting we evaluate the QN using LINE [17] v0.7.1 on MATLAB r2005b with the following hardware: 2.5 GHz Intel Core i7 quad core, 16GB RAM. Each QN evaluation takes approximately 1.5–2 seconds for our case study under a workload of 10,000 requests.

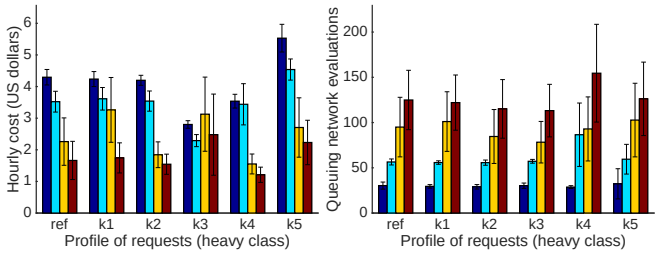
C. Results

Varying the number of requests. Figure 7a reports the results we got while modifying the number of users between 2,500 and 40,000 to analyze the scalability of our approach. From the figure we can see that the hourly cost is reduced in all situations and that the number of QN evaluations is independent from the number of users. This means that our approach is suitable for large-scale situations and that the time to produce a solution is fast enough for the approach to be used at run-time, since a QN evaluation took at most 2s in the reference case.

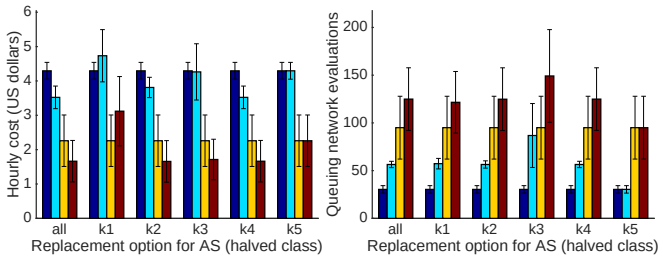
Varying the profile of requests. Figure 7b shows the effects we obtain by halving a particular class of requests (from $k1$ to $k5$) and doubling the remaining ones. In all the experiments the approach with all model-driven application refactorings costs significantly less than the one with no refactorings. It is worth to notice that since the approach we propose is a best-effort heuristic, we can increase the likelihood of reducing the cost, but we cannot guarantee its optimality because the deployment algorithm heuristic is non-deterministic and dependent on the initial situation. For this reason we may encounter situations like the $k3$ in Figure 7b where the reassignment may be slightly more expensive. Also in this experiment, different profiles do not significantly affect the convergence speed in terms of QN evaluations.



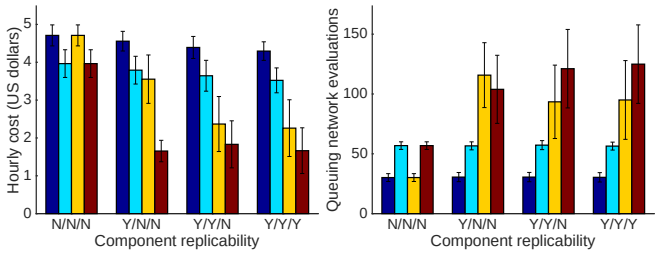
(a) Varying the number of the requests.



(b) Varying the profile of the requests.



(c) Varying the set of Application Server (AS) replacement.



(d) Varying the design constraints in terms of component replicability ("Y" yes, "N" no). Components are ordered in this way: Application Server, Database Server, Database I/O.

Figure 7. Experiment results.

Varying the replaceable set. Figure 7c reports the results we get while modifying the replacement option for the Application Server (*AS*). In addition to the reference case that considers all five possible substitutions, we consider alternative cases in which only one of the original substitutions is available. For each case $k = k1, \ldots, k5$, we consider only one substitution in which the service rate $\mu_{AS,k}$ is halved and the service rates of the other classes are doubled. In the cases of k2, k3, and k4, the algorithm replaces the AS component and reduces the cost. In the case of k5, the algorithm does not replace the AS component since it does not reduce the utilization of the component, and therefore the cost with or without the replacement refactoring is the same. Finally, in the case for $k1$ the algorithm chooses a reconfiguration that reduces the utilization, but the cost becomes higher.

Varying the design constraints. Figure 7d reports the results we get while modifying the design constraints on component replicability. As expected, we can notice that increasing design constraints makes our optimization algorithms less powerful, in fact the improvement gap in costs is reduced when new constraints are added. Figure 7d also reports the QN evaluations and, as expected, the addition of constraints decreases the number of evaluations.

D. Discussion

The experiments discussed in the previous section show that our approach is able to reduce the costs in all scenarios up to 60% when compared to an approach that does not use model-driven application refactorings. In most of our experiments the reassignment refactoring is typically responsible for 50% of such improvement, while the replacement refactoring (which depends on the alternative components provided as input) is responsible for the remaining 10%. The cost for this improvement is paid in terms of additional QN evaluations, which has shown an increase of up to 4 times. However, since our QNs are evaluated in a matter of seconds in our system, and the increment in convergence speed is constant, we expect our approach to be fast enough to be used do drive periodical reconfigurations of the system.

Another important point is that our approach is a best-effort heuristic applied to a NP-hard problem, therefore we cannot guarantee to always improve the results in every situation. Most of our experiments have shown a significant decrease in the costs for the majority of refactorings, whereas in few cases we found small cost increase. The reason behind this small cost increase is that the algorithm reaches borderline situations, with local optima that are difficult to optimize, which is typical in best effort heuristics. We do not think this is a significant problem since the increase, according to our experience, has always been small. However, there is still the option to spend additional QN evaluations (and therefore computation time) to use the approach with and without refactorings and choose the best one. We will study this possibility as follow-up work.

VI. Related Work

Automated resource provisioning has been previously studies in works such as [4, 15, 24, 27]. However, cloud computing introduces the important dimension of variable resource availability since the adoption of preemptible instances aims at minimizing costs without guaranteeing continuity in the computation.

In the literature several approaches have been defined to forecast price dynamics to make cost-efficient provisioning decisions [12, 13, 25]. Some other works take into account the possibility to lose resources and provide recovery techniques based on checkpointing or replication strategies [7, 14]. Others provide model-driven tools to analyze the costs and the performance of cloud applications at early stages of their development [11]. Finally, some

research jointly considers pricing models and the variable resource availability [3, 10, 28], and self-organizing auto-scaling techniques [10] to optimize the load allocation. In our previous work [8, 9] we applied a heuristic-based approach to find the cheapest cloud resources for the application deployment while taking into account the variable availability that is typical of spot and preemptible instances. However, to the best of our knowledge, none of the existing approaches proposes the application refactoring as support for the automated resource provisioning.

An extensive overview of existing research in the field of software refactoring (not only related to performance problems) is provided in [18], where a need for processes and tools that address refactoring in a more consistent, generic, scalable and flexible way is identified. We are interested to refactorings that can be applied to software performance models and few approaches have been introduced up today. Our previous work in this context makes use of performance antipatterns: (*i*) in [22] we introduced a methodology to rank performance antipatterns and optimize the refactoring process; (*ii*) in [23] we explored the synergies in combining antipatterns with bottleneck analysis.

VII. Conclusions

In this paper we have presented a model-based approach for optimizing the costs of running cloud-based applications by means of model-driven application refactorings, *i.e.*, experimenting software alternatives that optimize the application model, with the purpose of minimizing the cost of deploying them in the cloud. The peculiarity of our approach is that it can be used in situations characterized by a variable resource availability, such as Amazon EC2 spot instances and Google preemptible instances. We have shown through experiments that our approach is able to reduce the costs up to 60% by using model-driven application refactorings as a means for improving the optimal deployment of the application in terms of cost minimization. A drawback of our approach is that it requires a higher computational cost than the state-of-the-art, offering to the user different trade-offs between time requirements and quality of the optimal solution.

As future work we plan to: (*i*) assess the generalization of our approach to other industrial case studies, possibly from different domains; (*ii*) experiment with different price prediction strategies. Besides this, computational resources, like memories, are of key relevance and we want to investigate how much they affect application performance and cost.

Acknowledgment

Daniel J. Dubois has been partially funded by the People Programme (Marie Curie Actions) SPANDO FP7 project (FP7-629982). Catia Trubiani has been supported by the Microsoft Azure Research Award for the DESPACE project. Giuliano Casale has been supported by the European Unions Horizon 2020 research and innovation programme under grant agreement No. 644869 (DICE).

The data generated in this paper are available at: http://dx.doi.org/10.5281/zenodo.50673.

VIII. References

[1] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, Closed, and Mixed Networks of Queues with Different Classes of Customers. *J. ACM*, 22(2):248–260, Apr. 1975.

[2] S. Becker, H. Koziolek, and R. H. Reussner. The Palladio component model for model-driven performance prediction. *JSS*, 82(1), 2009.

[3] U. Bellur, A. Malani, and N. C. Narendra. Cost Optimization in Multi-site Multi-cloud Environments with Multiple Pricing Schemes. In *IEEE CLOUD '14*, pages 689–696, 2014.

[4] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *HotCloud*, 2009.

[5] P. Clements and et Al. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, MA, 2003.

[6] V. Cortellessa and R. Mirandola. PRIMA-UML: a performance validation incremental methodology on early UML diagrams. *Sci. Comput. Program.*, 44(1):101–129, 2002.

[7] S. Di, Y. Robert, F. Vivien, D. Kondo, C.-L. Wang, and F. Cappello. Optimization of cloud task processing with checkpoint-restart mechanism. In *IEEE SC '13*, pages 1–12, 2013.

[8] D. J. Dubois and G. Casale. Autonomic Provisioning and Application Mapping on Spot Cloud Resources. In *ICCAC '15*, pages 57–68, 2015.

[9] D. J. Dubois and G. Casale. OptiSpot: minimizing application deployment cost using spot cloud resources. *Cluster Comp.*, 2016.

[10] D. J. Dubois, G. Valetto, D. Lucia, and E. Di Nitto. Mycocloud: Elasticity through Self-organized Service Placement in Decentralized Clouds. In *IEEE CLOUD '15*, pages 629–636, 2015.

[11] D. Franceschelli, D. Ardagna, M. Ciavotta, and E. Di Nitto. SPACE4CLOUD: A Tool for System Performance and Cost Evaluation of Cloud Systems. In *MultiCloud '13*, pages 27–34, 2013.

[12] B. Javadi, R. K. Thulasiram, and R. Buyya. Characterizing spot price dynamics in public cloud environments. *Future Generation Computer Systems*, 29(4):988–999, 2013.

[13] S. Karunakaran, V. Krishnaswamy, and R. Sundarraj. Decisions, Models and Opportunities in Cloud Computing Economics. In *Service Research and Innovation*, pages 85–99. 2013.

[14] S. Khatua and N. Mukherjee. Application-centric resource provisioning for Amazon EC2 spot instances. In *Euro-Par '13*, pages 267–278. 2013.

[15] J. Kirschnick, J. Alcaraz Calero, L. Wilcock, and N. Edwards. Toward an architecture for the automated provisioning of cloud services. *Communications Magazine, IEEE*, 48(12):124–131, 2010.

[16] S. Kounev and A. P. Buchmann. Performance Modeling and Evaluation of Large-Scale J2EE Applications. In *CMG '03*, pages 273–283, 2003.

[17] Line solver http://line-solver.sourceforge.net.

[18] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.

[19] J. F. Perez and G. Casale. Assessing SLA Compliance from Palladio Component Models. In *MICAS '13*, pages 409–416, 2013.

[20] SPANDO EU Project. SPANDO-Refactor implementation and traces: http://www.spando.org/software.

[21] SPEC. SPECjAppServer2002 Design Document https://www.spec.org/jAppServer2002/docs/DesignDocument.html.

[22] C. Trubiani, A. Koziolek, V. Cortellessa, and R. H. Reussner. Guilt-based handling of software performance antipatterns in palladio architectural models. *J. of Syst. and Soft.*, 95:141–165, 2014.

[23] C. Trubiani, A. D. Marco, V. Cortellessa, N. Mani, and D. C. Petriu. Exploring synergies between bottleneck analysis and performance antipatterns. In *ICPE '14*, pages 75–86, 2014.

[24] B. Urgaonkar and A. Chandra. Dynamic Provisioning of Multi-tier Internet Applications. In *ICAC '05*, pages 217–228, 2005.

[25] P. Wang, Y. Qi, D. Hui, L. Rao, and X. Liu. Present or future: Optimal pricing for spot instances. In *IEEE ICDCS '13*, pages 410–419, 2013.

[26] C. M. Woodside, G. Franks, and D. C. Petriu. The Future of Software Performance Engineering. In *FOSE '07*, pages 171–187, 2007.

[27] Q. Zhang, L. Cherkasova, and E. Smirni. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *ICAC '07*, pages 27–27, 2007.

[28] Q. Zhang, Q. Zhu, and R. Boutaba. Dynamic resource allocation for spot markets in cloud computing environments. In *IEEE UCC '11*, pages 178–185, 2011.