Performance-Aware Refactoring of Cloud-based Big Data Applications

Chen Li, Giuliano Casale
Department of Computing
Imperial College London
London, U.K
Email: {chen.li1, g.casale}@imperial.ac.uk

***Abstract*—The problem of optimizing performance of cloud-based Big Data applications is critical in the cloud computing domain. In this paper, we propose a performance-aware approach that not only considers the cost of the cloud resources but also focuses on the dependency constraints among the deployed components to help dynamically refactoring the application. Furthermore, our approach closes the gap between design time models and runtime models. The feedback can be provided to application designers to iteratively enhance the application design and improve the application deployment. We have experimented our approach on the Wikistats application and the results show that the proposed approach effectively refactors the deployment based on different QoS metrics (*e.g.*, utilization, cost, availability) of the cloud resources and dependency constraints.**

***Keywords*-software performance engineering, cloud computing, application refactoring, resource management**

I. Introduction

In recent years, many approaches have been proposed to tackle the problem of how to close the gap between development and operations. Since the abstraction levels between system runtime and design time are different, it is essential for the developer to (i) obtain the runtime information for identifying the potential problems, especially performance issues; (ii) reflect them into design time models to reason on the quality of an application design and infer refactoring decisions for the application deployment.

To provide efficient deployment decisions, a developer needs to consider the target deployment environment. Different from transitional centralized and distributed systems, the cloud resources is more dynamic and transparent. The computing resources are not static and application deployment on cloud are virtualization-based. Thus, it is difficult to maintain the dependency relationship among the components after deploying them on the cloud. Furthermore, monitoring the particular aspects of runtime behaviour to guide the design time models refactoring on the cloud platform is also a challenge [1]. Other QoS Metrics, *e.g.*, cost, availability, also need to be considered to calculate the cost of renting cloud resources since all the cloud resources, *e.g.*, platform, storage, are provided as services. Customer needs to pay the resources they are using. However, mature solutions are not available to solve all of the above issues at the same time.

This work is partially supported by the European Commission grant no. 644869, DICE.

In this paper, we present a novel research proposal, a performance-aware deployment approach. Our approach (i) considers the dependency plan among the logical components which share the physical nodes, (ii) creates a refactoring plan of application deployment to address the performance issues (*i.e.,* Excessive Calculation), and (iii) provides the feedback to the design time models. Furthermore, we also consider other characteristics, *e.g.*, available hardware resources and budget, to customize the cloud resources according to actual demands.

In this work, we use Unified Modeling Language (UML) to specify the architecture model at the design stage to identify the behaviours and structure of applications. The UML performance attributes, *e.g.*, *hostdemand* (the CPU requirements of an operation), will be provided by MARTE [2] profile and DICE profile (a recently proposed UML profile for supporting Big data technology) [3]. The performance model chose here is Layered Queueing Networks (LQNs) model [4]. The motivation of choosing LQNs is an application may be seen as a network of buffers and processing elements that exchange messages, and it is thus quite natural to map them into a queueing network model. Furthermore, the core elements of LQN models are semantically similar to the corresponding elements of UML activity and deployment diagrams. Moreover, the LQN solvers such as LINE[1] or LQNS[2] can be used to provide analytical methods to solve the LQN model and generate analysis results, *e.g.*, CPU utilization, throughput. Our approach assumes that user has the following information: (i) the performance bounds, *e.g.*, thresholds on the CPU utilization of a single server; (ii) the availability of the resources, used and unused resources; (iii) average recovery time of a single server, the hourly cost of the single server and budget; (iv) a dependency matrix that defines the dependency relationship among the components.

This paper is organized as follows. We reviewed related work in Section 2. In Section 3, we describe the system model and problem statement through a running example. Section 4 presents our approach. Based on the above example, a case study is provided in Section 5. Finally, we conclude the paper and outlines future research directions in Section 6.

[1]http://line-solver.sourceforge.net/
[2]https://github.com/layeredqueuing/V5/tree/master/lqns

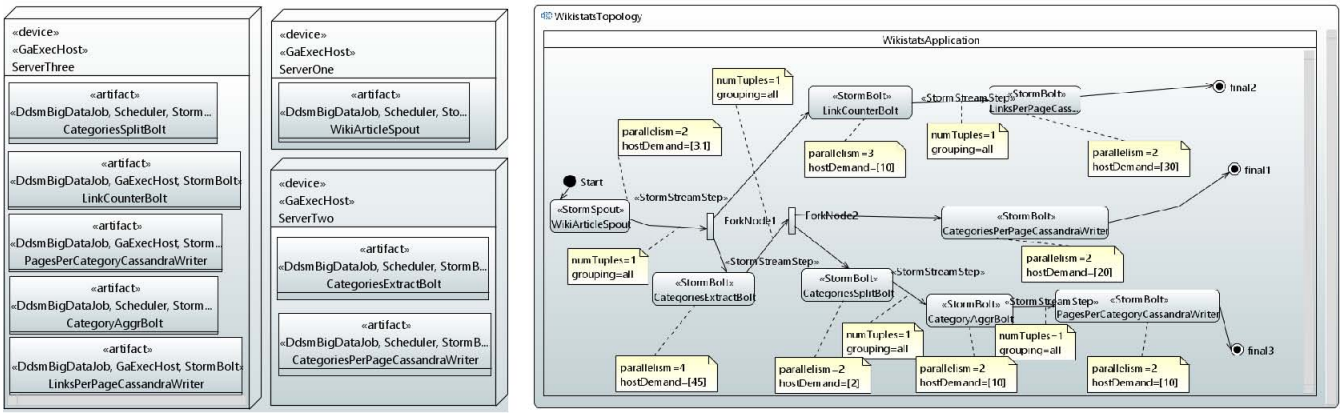

Fig. 1: The deployment diagram and activity diagram of Wikistats example.

II. Related Work

Software performance issues (*e.g.*, Unbalanced Processing) are largely studied in the industry. They are related to the incorrect software decisions at different hierarchical levels (architecture, deployment, or project management). Several research works have been published in the area. Bodik et al. apply rich statistical models of the application's performance, simulation-based methods for finding an optimal control policy, and change-point methods to find abrupt changes in performance [5]. J.Kirschnick et al. propose an approach to automating deployment and management of the virtual infrastructure and software of services deployed in the cloud [6]. Urgaonkar et al. proposed a flexible queuing model to determine how much resources to allocate and a combination of predictive and reactive methods to determine when to provision these resources [7]. Dubois et al. present a model-based approach for optimizing the costs of running cloud-based applications by means of model-driven application refactorings [8]. Another research direction is optimizing performance for streaming applications. Cardellini et al. present and evaluated a general formulation of the optimal DSP replication and placement (O-DRP) as an integer linear programming problem, which takes into account the heterogeneity of application requirements and infrastructures resources [9]. Liu et al. propose a deployment framework that enables streaming applications to run on IaaS clouds with satisfactory performance and minimal resource consumption [10]. Most of the above works focus on the optimizing the application performance with the concerning of the resource consumption. However, few of them (i) consider the dependency constraints among the software components, (ii) and relocates the components to balance the utilization of the servers without violating the dependency constraints.

To provide the feedback to the design time models, several approaches integrate performance analysis into software design process to generate performance models such as queueing networks (QN) [11], stochastic Petri nets [12] and layered queueing networks [13] from design time models. [14] and [15] present techniques for deriving performance models from UML model. Woodside et al. propose a tool PUMA which provides a unified interface between different software models and generates different performance model, *e.g.*, stochastic Petri nets, QN and LQN [16] [17]. Though these studies remain relevant in many areas, there is a shortage of methods of automatically deriving performance models for the cloud based Big data applications, *e.g.*, Apache Storm, Hadoop. To address this issue, we develop Tulsa [18]. It takes the UML model as the architecture model, annotated with MARTE and DICE profiles to capture runtime performance information of Big data applications, and LQN model as performance model. Our approach closes the gap between design time and runtime and provides feedback to the customer to help refactoring application deployment.

III. System Model and Problem Statement

In this section, we present the system model and define the performance issues. For better understanding, first, we introduce a running example. For the sake of clarity, we summarize the notation used in this paper (see Table 1).

A. Running Example

We use WikiStats[3] as a running example of the application that analyzes web pages from the popular Wikimedia web site and stores the result of the analysis into a database. The WikiStats application is composed of 8 components and deployed on 3 devices. Fig. 1 shows the application model expressed as a UML model including deployment and activity diagrams.

In our example, if $TCom$ = 2, *ServerThree* might cause an issue since it holds 5 components in deployment diagram. If its utilization, using the prediction given by LINE, $SerUti_{ServerThree} \geq TUti$, then an excessive calculation issue will be detected. This leads to the deployment refactoring, that is, three components need to be moved to the new server(s). If the relocation does not violate the $ComDep$, the number of needed server nodes is lower than $SerAva$ and the new hourly cost is lower than $BudH$, the component relocation will be accepted. Note the actual application deployment

[3]https://github.com/dice-project/DICE-WikiStats

TABLE I. Main notation adopted in the paper

Symbol	Description
N_r	The total number of server nodes
N_r^{idle}	the number of idle server nodes
N_r^{used}	the number of used server nodes
$TCom$	the maximum number of components on a server
$TUti$	the upper bound of utilization of a server node
$ComNum_i$	the number of components on $server_i$, i $\in(0,N_r^{used})$
$SerUti_i$	the CPU utilization of $server_i$, i $\in(0,N_r^{used})$
$BudH$	the hourly budget cost
$CosSerH$	the average hourly cost of one server
$SerAva$	the average availability of one server
$norWorkT$	the normal working time of a component without any failure
$RecT$	the average time to recover a server
$ComDep$	the dependency matrix

refactorings also needs to consider the system or customer constraints, for example, there may not have available servers for the refactorings due to the budget.

B. System Model

The system model we consider is composed of three parts: architecture model, performance model and resource model.

Architecture model is represented as an UML model annotated with MARTE and DICE profile which provide performance annotations. The dependency relationship among the logical components is defined by the designer. It can be represented as a matrix $ComDep$. We use Tulsa to transform the architecture specified by UML model to a performance model (i.e., LQN model). More details of the model transformation can be found in our previous work [13] [18]. Resource model captures the computing resources, where N_r represents the total number of available server nodes we can use on the Cloud. N_r^{idle} and N_r^{used} stand for the number of idle servers and used servers. Each server node is characterized by the measures shown in Table 1, which cover costs, availability, time, and dependency matrix.

C. Problem Statement

Our approach focuses on performance bottlenecks detected by a high utilization of a device, that is, Excessive Calculation (EC). This may be caused by an unbalanced utilization of the devices originated in a bad deployment of processes among the hardware resources. Thus, the main goal of our approach is component allocation which aims at deploying the application components on the hardware resources while reaching a balanced CPU utilization without violating the components dependency and the design constraints. Furthermore, how to propagate the redeployment within the architecture model by means of the performance model to help designer reason the design flaws also need to be addressed.

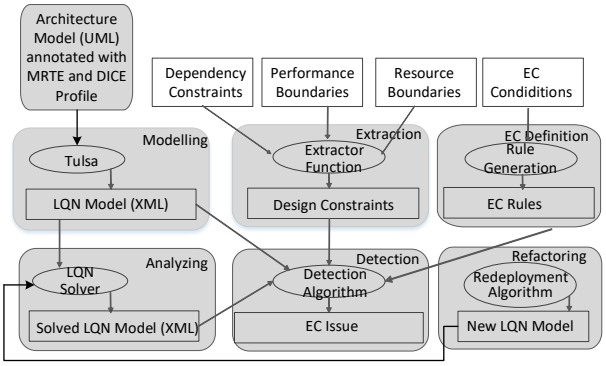

Fig. 2: Workflow of our approach

IV. Approach

A. Main idea

Our approach can be summarized by the following six steps. Fig. 2 shows the corresponding workflow of our approach.

- Step 1: Modelling. This step is focused towards transforming the UML model to LQN model by using Tulsa. Tulsa will generate an XML format LQN model which follows the XML schema of LQN.
- Step 2: Analyzing. The LQN solver (*i.e.*, LINE) is used to solve the XML format LQN model and to generate the analysis results, an XML format file as well.
- Step 3: Extraction. Extracting the constraints to be used in the deployment refactoring, *e.g.*, performance bounds and dependency matrix.
- Step 4: EC definition. Excessive Calculation is defined as EC rules (*i.e.*, trigger conditions) for performance issues detection.
- Step 5: Detection. LQN model, solved model, design constraints and EC rules will be used for detection algorithms to check if there is an excessive calculation issue in the current model.
- Step 6: Refactoring. Relocating the components which cause the EC issue without violating the design constraints.

B. Process

One of the key features of our approach is we leverage our previous work, Tulsa and LINE, to help to generate and solve the LQN model automatically. Thus, the application developer can identify the design flaws of the architecture model (*i.e.*, UML model) according to the analysis results of the corresponding performance model.

The following paragraphs explain the details of the step 3 to step 6. For the sake of simplicity, we assume that all the server nodes are homogeneous. This is often the case in actual cloud deployment since homogeneous resources considerably simplify load balancing and auto-scaling.

Extraction. In order to set the design constraints, our approach lets the user define the performance bounds, resource bounds and dependency matrix.

1) Performance bounds. Our approach considers two constraints, distribution of the components and CPU utilization of a server node. To avoid overload, our approach provides two performance bounds, $TCom$ and $TUti$, to set the upper bound of the number of components that can be deployed on a single server and CPU utilization.
2) Resource bounds. It includes two parts, server nodes and price. The sum of the number of the idle and used server nodes should equal the total number of the server nodes:

$$N_r = N_r^{idle} + N_r^{used} \tag{1}$$

If the new deployment strategy needs i, $i = 1, ..., n$, components and $i > N_r^{idle}$, then the refactoring cannot be done due to the limited available server nodes. The other bound is the budget. Our approach calculates the cost of the rent server nodes to check if it is beyond the budget. To obtain the cost, we first need to calculate the normal working time per hour without failure (*norWorkT*).

$$norWorkT = 1 - (RecT/3600) \tag{2}$$

The total cost should be lower than the budget.

$$(norWorkT * SerAva * CosSerH) * N_r^{used} < BudH \tag{3}$$

3) Dependency matrix. A matrix is created to track the dependency among the components. The row and column vectors represent the components, that is

$$Com : Com_i \times Com_j \rightarrow M \tag{4}$$

where *i, j* $\in (1, N_r^{used})$ and *M* is the matrix, $M_{n,m} \in \{0, 1\}$, 0 indicate that there is no dependency between Com_i and Com_j and 1 the opposite.

EC definition. EC problem occurs when a processor performs all of the work of an application or holds all of the application data. It can degrade performance. To better understand EC issue, we interpret it crossing different modelling level, architecture model and performance model.

1) UML Model: EC may be detected on UML deployment diagram. A server node is represented as a *Device* and a software component is represented as an *Artifact* in the deployment diagram. It might cause the EC problem if a *Device* holds the number of *Artifact* is greater than $TCom$.
2) LQN Model: In LQN model, EC is related to the *Entry* of the *processor*. A server node is regarded as a *processor* in LQN model. The main services of each *Artifact* is represented as the activities of *Entry* of the *processor* in the LQN model. Thus, it might cause the EC problem, if the number of the *Entry* of the *processor* is greater than $TCom$.
3) Solved LQN Model: The other trigger condition of the EC is the CPU utilization. The solved the LQN model provides the analysis results of the LQN model including the CPU utilization of each *processor* (*i.e.*, server node). If a *processor* utilization is greater than $TUti$, it might cause the EC problem.

Given the performance bounds, maximum number of components ($TCom$) and maximum utilization ($TUti$). The EC is defined by the following two rules:

- Rule 1: The number of the *Entry* on any *processor* needs to be lower than the $TCom$;
- Rule 2: The utilization of the *processor* needs to be lower than the $TUti$;

Detection and Refactoring. Based on the above *Extraction* and *EC definition*, the detection and refactoring strategy generation process can be described as follows:

1) Step 1: Calculate the number of components ($ComNum_i$) of $server_i$ ($i \in (0, N_r^{used})$) from the obtained LQN model to see if there is any server node that has the number of components is greater than the $TCom$. If the server node is found, goes to step 2 otherwise exits.
2) Step 2: Check if the utilization of $server_i$, which is found in step 1, from the obtained the solved LQN model is greater than the $TUti$. If the server node is found, goes to Step 3 otherwise exits.
3) Step 3: Check the dependency among the components of the $server_i$ based on the dependency matrix $ComDep$, selecting the candidate components which need to be migrated to the new server(s) and calculating the number of server nodes ($NSer$) for the component(s) reallocation. If $NSer$ is lower than the N_r^{idle}, goes to step 4 otherwise exits.
4) Step 4: Use the formula (2) and (3) to calculate the cost ($CosReDep$) of the new deployment. If the $CosReDep$ is lower than $BudH$, the deployment suggestion is approved otherwise exits.

C. Benefits and Limitations

Our approach considers the software life circle and provides a refactoring strategy to close the gap between design time and runtime, that is, it not only focuses on refactoring the architecture to address performance issues detected at runtime, but also helps to provid feedback to the design time model. The designer can easily obtain the performance model from the architecture model by using our Tulsa tool which ensures the precision and correctness of performance model. Our approach guarantees that there is no EC issue in then deployment while considering both the cost of the cloud resource and dependency relationship among the components, and deliver a balanced deployment configuration to the end user. It also helps designer to trace the performance issues on different stages and reflects the refactoring decisions back to the architecture model.

However, our approach assumes that designer uses UML to represent the architecture model (*i.e.*, activity diagram and deployment diagram) and use LQN model as the performance model. Without UML model, user has to manually define the LQN model according to their architecture model. This may lead to the extra efforts.

TABLE II. CPU utilization of the initial LQN model

Server Name	CPU Utilization
$ServerOne$	0.19
$ServerTwo$	0.39
$ServerThree$	0.99

TABLE III. CPU utilization of the new LQN model

Server Name	CPU Utilization
$ServerOne$	0.33
$ServerTwo$	0.66
$ServerThree$	0.66
$NewProcessor1$	0.99

V. Case Study

In this section, we use the above Wikistats example to demonstrate our approach. To implement the experiment, we need Eclipse 4.6, JDK 1.7, Papyrus Framework[4], Tulsa, LINE and MATLAB Compiler Runtime (MCR) R2015a. The following paragraphs describe the setting for the implementation and the results.

A. Modelling

The first step our experiment is building the system model.

Architecture model. Fig. 1 shows the the architecture model (*i.e.*, UML Model) of Wikistats application which is designed by using Papyrus editor in the Eclipse. It is annotated with core tags (*e.g., hostdemand*) of stereotypes of MARTE and DICE profiles.

Performance model. Our approach uses Tulsa to generate the Wikistats LQN model. The solved LQN model can be obtained with the help of LINE solver. Table 2 shows the CPU utilization of each server node from the solved LQN model of Wikistats.

B. Parameters

To run the experiment, we need to set the parameters for design constraints which include the performance bounds, resource bounds and dependency matrix. For the sake of simplicity, our algorithm accepts a *configuration* file, where the user can define those constraints, as an input parameter.

Performance bounds. In our example, we assume that the threshold on the number of components ($TCom$), which can be deployed on a single server, is 3. The upper bound of the CPU utilization of a single server ($TUti$) is 0.80.

Resource bounds. The total server nodes, N_r, for running the example is 20. Three server nodes are used, *ServerOne*, *ServerTwo* and *SeverThree*, for deploying Wikistats application. Thus, the N_r^{idle} and N_r^{used} are 17 and 3 respectively. We assume the hourly budget, $BudH$, for running the Wikistats applications is 80.00$. Since we consider all the server nodes are homogenous, hourly cost of each server, $CosSerH$, is 5.00$. The server nodes availability, $SerAva$, is set to 95% and average recovery time for a single server, $RecT$, is 810 seconds.

Dependency matrix. The dependency relationship is predefined by the designer. In our running case, it includes 8 components which are deployed on 3 server nodes. Their dependency relationship is (i) the component *WikiArticleSpout* is independent of others components, (ii) component *CategoriesPerPageCassandraWriter* dependents on the processing results of the component *CategoriesExtractBolt*. (iii) the component *LinksPerPageCassandraWriter* dependents on the processing results of the component *LinkCounterBolt*, (iv) the component *CategoriesSplitBolt*, *CategoryAggrBolt* and *PagesPerCategoryCassandraWriter* dependent on each other. The corresponding dependency array is defined as follows:

$$
\begin{aligned}
DepArray = \{&[(ServerOne), (< WikiArticleSpout >)],\\
&[(ServerTwo), (< CategoriesExtractBolt,\\
&CategoriesPerPageCassandraWriterThe >)],\\
&[(ServerThree), (< LinkCounterBolt,\\
&LinksPerPageCassandraWriter >,\\
&< CategoriesSplitBolt, CategoryAggrBolt,\\
&PagesPerCategoryCassandraWriter >)]\}
\end{aligned}
$$

C. Results

LINE uses fluid analysis to solve the LQN model and provides the performance analysis results, *e.g.*, CPU utilization, Response Time. The total execution time of LINE for our case is 0.146540 second.

Performance Bounds. Since the threshold on the number of component, $TCom$, is 3, the *ServerThree*, which holds 5 components, violates the bounds. Further more, the utilization of *ServerThree* is 0.99 (see Table 2) which is greater than the threshold on CPU utilization of single server ($TUti$ = 0.80).

Dependency Matrix. There are two dependency groups on the *ServerThree*, (*LinkCounterBolt*, *LinksPerPageCassandraWriter*) and (*CategoriesSplitBolt*, *CategoryAggrBolt*, *PagesPerCategoryCassandraWriter*). To meet the dependency requirement, if the component *i* needs to be relocated to a new server, all of it dependents components need to be moved at the same time. Since the $TCom$ is 3, either of the two groups can be migrated to a new server node.

Resource Bounds. To relocate the components of *ServerThree*, a new server node needs to be introduced to the system. The current available number of server, $SerAva$, is 17. The cost of adding a server node is 44.17$ which is lower than the $BudH$. Thus, the new deployment is under the predefined budget.

Refactoring. As results of the EC detection and refactoring, three entries initially deployed on *ServerThree*, AC5Entry, AC6Entry and AC7Entry (*i.e.*, component *CategoriesSplitBolt*, *CategoryAggrBolt* and *PagesPerCategoryCas-*

[4]https://eclipse.org/papyrus/

sandraWriter) are relocated to a new server node (*NewProcessor1*). After solving the new LQN model, the CPU utilization of *ServerThree* changes to 0.66 which is lower than $TUti$ (see Table 3). Thus, the new deployment decisions can be provided to designer to refactor the architecture model, especially the deployment diagram. Note that the number of components in the *NewProcessor1* is greater than $TCom$ and the CPU utilization is also greater than $TUti$. However, we cannot refactor them due to the dependency constraints. The user can either redesign the application or modify the bounds to solve this issue.

D. Discussion

The experiments discussed in the above section show that our approach not only considers the costs of the cloud resources but also helps to avoid the overload issue by monitoring the distribution of the components and the CPU utilization of server nodes. Furthermore, the proposed approach takes into account the dependency relationship among the deployed components and provides a balanced component relocation strategy without violating the dependency constraints.

Another important point is that our approach introduces a new methodology and a prototype to close the gap between runtime measurements and UML model. According to our knowledge, no mature methodology appears available in the research literature in the context of cloud-based Big data applications to address the difficult problem of going from measurements back to the application models, transforming application model to performance model to help to reason about system performance, and reflecting the analysis results back to the application model to guide the application refactoring.

VI. Conclusions

In this paper, we have presented a performance-aware approach for refactoring the deployment of cloud based Big data applications. The purpose of our approach is balancing the CPU utilization without violating software component dependency constraints while customizing the cloud resources according to the actual demand. The peculiarity of our approach is that it closes the gap between the design time models and runtime models by generating the performance model from the architecture model and providing feedback to the architecture model by means of the performance model. A limitation of our approach is that it requires UML model as the design time model since Tulsa takes UML model as a input model to generate the LQN model.

As future work we plan to (i) assess the generalization of our approach to other industrial case studies; (ii) work with performance anti-patterns experts and extend our approach to help to address some popular performance anti-patterns issues, *e.g.*, Circuitous Treasure Hunt, Empty Semi Trucks.

References

[1] Z. Zhou, L. Wang, Z. Cui, X. Chen, and J. Zhao, "Jasmine: A tool for model-driven runtime verification with uml behavioral models," in *Proceeding of the 11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*. Nanjing, China: IEEE, 2008, pp. 487–490.

[2] O. M. Group, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1*, June 2011. [Online]. Available: http://www.omg.org/spec/MARTE/

[3] DICE, *D2.1 Design and quality abstractions*, Jan 2016. [Online]. Available: http://www.dice-h2020.eu/deliverables/

[4] C. Woodside, *Tutorial Introduction to Layered Modeling of Software Performance*, http://www.sce.carleton.ca/rads/lqns/lqn-documentation/tutorialh.pdf, ACM, Feb 2013.

[5] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters," in *Proceeding of the 2009 conference on Hot topics in cloud computing (HotCloud'09)*. San Diego, USA: ACM, 2009.

[6] J. Kirschnick, J. A. Calero, L. Wilcock, and N. Edwards, "Toward an architecture for the automated provisioning of cloud services," *Communications Magazine*, vol. 48, no. 12, pp. 124–131, 2010.

[7] B. Urgaonkar and A. Chandra, "Dynamic provisioning of multi-tier internet applications," in *Proceeding of the 2nd International Conference on Autonomic Computing (ICAC'05)*. Seattle, USA: IEEE, 2005, pp. 217–228.

[8] D. Dubois, C. Trubiani, and G. Casale, "Model-driven application refactoring to minimize deployment costs in preemptible cloud resources," in *Proceeding of the 9th International Conference on Cloud Computing (CLOOUD'16)*. San Francisco, USA: IEEE, 2016, pp. 335–342.

[9] V. Cardellini, V. Grassi, F. Presti, and M. Nardelli, "Optimal operator replication and placement for distributed stream processing systems," *SIGMETRICS Performance Evaluation Review*, vol. 44, no. 4, pp. 11–22, 2017.

[10] X. Liu and R. Buyya, "Performance-oriented deployment of streaming applications on cloud," *IEEE Transactions on Big Data*, 2017.

[11] V. Cortelessa and R. Mirandola, "Deriving a queueing network based performance model from uml diagrams," in *Proceeding of 2nd ACM Workshop on Software and Performance (WOSP'00)*. Ottawa, Canada: ACM, 2000, pp. 58–70.

[12] J. Lopez-Grao, J. Merseguer, and J. Campos, "From uml activity diagrams to stochastic petri nets: Application to software performance engineering," in *Proceeding of the 4th International Workshop on Software and Performance (WOSP'04)*. Redwood City, CA: ACM, 2004, pp. 22–36.

[13] T. Altamimi, M. Zargari, and D. Petriu, "Performance analysis roundtrip: automatic generation of performance models and results feedback using cross-model trace links," in *Proceedings of CASCON'16*. ACM, November 2016.

[14] V. Cortellessa, A. D. Marco, and P. Inverardi, *Model-based Software Performance Analysis*. Springer, 2011.

[15] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: a survey," *Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.

[16] C. Woodside, D. Petriu, J. Merseguer, D. Petriu, and M. Alhaj, "Transformation challenges: from software models to performance models," *Software and Systems Modeling*, vol. 13, no. 4, pp. 1529–1552, 2014.

[17] C. Woodside, D. Petriu, D. Petriu, H. Shen, T. Israr, and J. Merseguer, "Performance by unified model analysis (puma)," in *Proceeding of the 5th ACM Workshop on Software and Performance*. Palma, Spain: ACM, 2005, pp. 1–12.

[18] C. Li, T. Altamimi, M. Zargari, G. Casale, and D. Petriu, "Tulsa: A tool for transforming uml to layered queueing networks for performance analysis of data intensive applications," in *Proceeding of the 14th International Conference on Quantitative Evaluation of SysTems (QEST'17)*. Berlin, Germany: Springe, 2017.