Towards a DevOps Approach for Software Quality Engineering

Juan F. Pérez, Weikun Wang, Giuliano Casale
Department of Computing
Imperial College London
London, UK
{j.perez-bernal,weikun.wang11,g.casale}@imperial.ac.uk

ABSTRACT

DevOps is a novel trend in software engineering that aims at bridging the gap between development and operations, putting in particular the developer in greater control of deployment and application runtime. Here we consider the problem of designing a tool capable of providing feedback to the developer on the performance, reliability, and in general quality characteristics of the application at runtime. This raises a number of questions related to *what* measurement information should be carried back from runtime to design-time and what degrees of freedom should be provided to the developer in the evaluation of performance data. To answer these questions, we describe the design of a filling-the-gap (FG) tool, a software system capable of automatically analyzing performance data either directly or through statistical inference. A natural application of the FG tool is the continuous training of stochastic performance models, such as layered queueing networks, that can inform developers on how to refactor the software architecture.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.2.9 [**Software Engineering**]: Management—*Software quality assurance*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering*

Keywords

Software Performance Engineering; Quality of Service; Monitoring; Design-time Application Models

1. INTRODUCTION

Recent years have seen the rise of the DevOps approach for software development [13], which aims at closing the gap between development and operations, providing timely feedback to the application developer to speed-up the development cycle. In particular, a number of decisions made during development may have a large impact on the application performance, but the actual effect of such decisions is only measurable once the application is deployed and monitoring data is available. The problem is therefore how software performance methods can help bridging the gap between runtime performance data and the higher level of abstraction required by the developer to be able to reason on the quality of an application design and possibly identify refactoring actions.

In this paper we describe the design of a tool, referred to as the filling-the-gap (FG) tool, to enhance and automate the delivery of application performance information to the developer. The FG tool has two main objectives. The first objective is to provide the data to parameterize application design-time Quality-of-Service (QoS) models, improving their accuracy, by relying on the monitoring information collected at runtime. To this end, the FG tool implements a set of statistical routines to estimate the QoS models' parameters, which, given the flexibility of being executed offline, can be computationally intensive and make use of the extensive datasets collected at runtime, with the aim of producing more accurate results. The procedure of executing the estimation routines aiming at improving the accuracy of the QoS models is referred to as *FG Analysis*.

The second objective of the FG tool is to provide the developer with a report of the application behavior at runtime. Relevant information includes, among others, the application compliance with Service-Level Objectives (SLO)s, the effective QoS offered, and the deployment cost. Based on these results, the developer can make informed decisions regarding the application architecture and deployment, with the aim of improving the QoS or reducing the incurred cost.

Recent approaches in software performance engineering are related to the FG tool. The Palladio Component Model [2] offers the capability of integrating application component models with resource and usage models, which can be used for QoS analysis by means of transformations to performance models, such as Layered Queueing Networks (LQN) [5]. Also, Descartes [3] provides a modeling language that focuses on online performance prediction. On the other hand, monitoring frameworks, such as Kieker [15], enable the application-level performance monitoring, including filters that allow the selection of data for further analysis.

The FG tool presented in this paper relies on a monitoring framework capable of providing both application and system-level monitoring metrics. As in other tools, such as Kieker, this information can be used for further analysis of the application behavior at runtime. The FG tool, however,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WOSP-C'15, January 31, 2015, Austin, Texas, USA.
Copyright © 2015 ACM 978-1-4503-3340-5/15/01 ...$15.00.
http://dx.doi.org/10.1145/2693561.2693564.

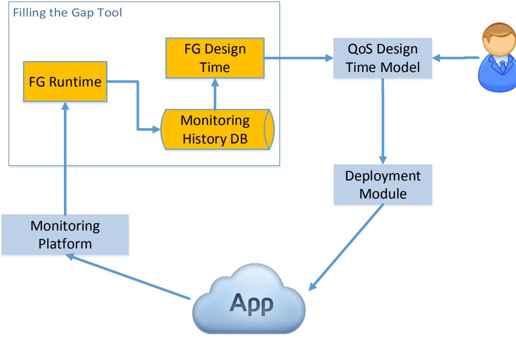

Figure 1: FG tool architecture

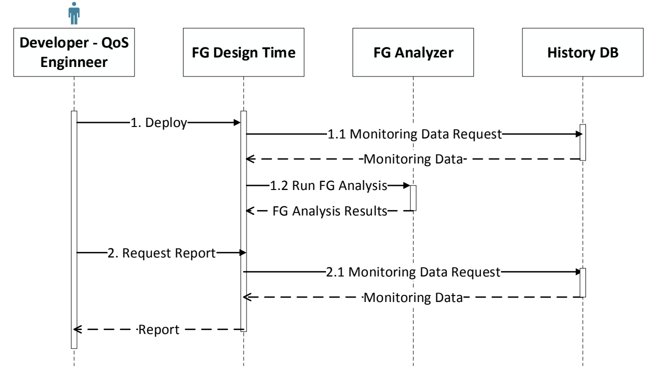

Figure 2: Filling the Gap workflow

differs from existing approaches in that it can reason on design-time models, namely extended LQN models, to deliver more accurate inferences of the model parameters from runtime monitoring data. Thus, rather than simply monitoring, the FG tool is envisioned as a machine-learning component that is aware of the application software architecture, and can use this to improve parameter learning. The objective is thus to automatically collect the appropriate monitoring information, and execute the estimation routines required to keep the model up-to-date, allowing the developer to have a more accurate view of the application performance, not only through the direct analysis of the monitoring information, but also through application of statistical inference to learn from monitoring data the parameters of the design-time application models. This enables the developer to test potential application refactorings using design-time models that are appropriately parametrized with up-to-date runtime data.

The remainder of this paper is organized as follows. In Section 2 we introduce the architecture of the FG Tool. This is followed in Section 3 by a review of methods that may be used for performance inference in the FG tool. In sections 4 and 5 we identify the control knobs that the FG Tool should expose for configuration and analysis. Finally, Section 6 discusses conclusions and future work.

2. ARCHITECTURE

We consider 3 main components for the FG tool: the FG Design-Time component, the FG Runtime component, and the Monitoring History Database. We now describe these components and their interactions.

2.1 Components

As illustrated in Figure 1, the FG Runtime component is executed with the application, and we assume a monitoring platform is in place to collect data relevant for the application. The FG Runtime component connects to the monitoring platform to collect the data necessary for FG analysis, thus this component needs to be configured with the queries necessary to obtain all the data relevant for FG analysis. How these data requirements are determined is illustrated in Section 4. The FG Runtime component saves the received data, maybe after some pre-processing into the Monitoring History DB. This DB is the second component of the FG tool, storing all the monitoring data relevant for FG analysis. Since this data can grow large very easily, both sampling and pre-processing can be implemented in the FG Runtime component to limit the amount of monitoring data stored in the Monitoring History DB.

The third component of the FG tool is the FG Design-Time component, which executes the estimation routines to update the parameters of the application QoS models. To this end, it accesses the Monitoring History DB to retrieve the relevant data, process it, and provide it as input to the implemented estimation methods, as those described in Section 3. The result of this analysis is passed to the QoS Design-Time models by means of common XML files, timestamped so that the developer can select the parameters obtained in different conditions. For instance, the developer can implement a modification to the application architecture, expecting a QoS gain. When the application is deployed, the FG tool will save and analyze the monitoring data, providing a new set of parameter values for the QoS models, which are specific for this deployment. The developer can use this information to test if the QoS expected with a given design was in fact achieved.

2.2 Workflow

Having defined the FG components and their roles, we now present the workflow for FG analysis. The operation of the FG component can be divided in three main stages, as follows:

1. Deployment and Configuration: this step is performed by the user, interacting with the FG Design-time component, to launch and configure the FG tool.
2. Analysis: this step is performed by the FG Design-Time component, to execute the FG analysis.
3. Reporting: in this step the user interacts with the FG Design-time component to obtain information about the behavior of the application at runtime, including results from the FG analysis.

These steps are depicted in Figure 2, which illustrates the FG tool workflow. As mentioned in the previous section, the user interacts with the FG Design-Time component to launch and configure the FG analysis (e.g., the frequency with which it must be executed). This information is used by the FG Design-Time component to perform the FG Analysis, for which it must first query the Monitoring History DB to obtain the necessary information. The user can also interact with the FG Design-Time component to obtain reports about the application effective QoS, as well as about the results of the FG analysis.

3. ESTIMATION TECHNIQUES FOR FG

One of the ultimate objectives of the FG tool is to provide accurate estimates for the parameters of the design-time application QoS models. These QoS models can be queueing networks, layered queueing networks, or other abstractions that support QoS analysis based on a description of the application main characteristics. In fact, these QoS models should be able to handle what-if analyses performed at design time, as well as to support optimization routines, e.g., for optimal resource provisioning. Typically, these models are initially parametrized using expert-knowledge or data collected in small deployments. The FG tool aims at obtaining estimates based on monitoring data collected at runtime, once the application has been deployed, improving the developer knowledge of the application offered QoS.

For the FG tool, we are particularly interested in layered queueing network models [5], which capture the contention experienced when multiple users attempt to access the available hardware and software resources, and the interaction between them. Further, we focus on closed models as these are well-suited for software systems, which can be seen as being composed of layers, the interactions of which are typically caused by admission control or finite threading limits [14]. However, the methodology can be easily extended to open models. To parameterize these models it is necessary to estimate the inter-request submission times, modeled as think times, as well as the resource consumption exerted by each request. Inter-request times are easy to obtain, since this information (or data from which this can be extracted) is typically tracked by application- or container-level logs.

Resource consumptions, also called demands, are however harder to obtain as these are not explicitly tracked by logs, and the deep monitoring instrumentation typically required pose unacceptably large overheads, especially at high resolutions. Since application requests can complete in a few milliseconds, individual monitoring becomes too expensive to perform in a production system. To cope with this issue, our approach is to take coarse-grained measurements and apply statistical inference to obtain *mean* resource demand estimates. Existing mean demand estimation approaches mainly rely on regression against utilization data [6, 8, 10, 17]. However, utilization measurements are not always available, for example, in Platform-as-a-Service (PaaS) deployments where the resource layer is hidden to the application and thus protected from external monitoring.

To overcome these issues we have recently proposed two methods for demand estimation that avoid the use of utilization measurements, and are therefore suitable for applications deployed on both IaaS and PaaS. The first one is the Gibbs sampler in [16], referred to as *GQL*, is based on measurements of the outstanding number of requests at each resource, which is equivalent to the queue length in queueing models. These measurements can be obtained from application and container logs, by looking only at requests' arrival and departure timestamps. The second method [12], referred to as *MINPS*, is based on response-time data, which can be obtained by active probing or by simple injection of timers in the application code.

In addition to these methods, the FG tool implements existing demand estimation methods, particularly to take advantage of utilization measurements available on IaaS deployments. The tool supports the following three existing demand estimation methods: the utilization-based optimization (UBO) method from [9], the utilization-based regression (UBR) method from [17], and the FCFS regression method from [7].

Table 1: Information required by the QoS model

Info set	Item	Struct.	Estim.
Resource	Resources (CPUs, disks)	X	
	Resource multiplicity (number of cores)	X	
	Resource scheduling policy	X	
Workload	Request classes (URIs)	X	
	User population		X
	Sequence of resources used by each request class	X	
	Resource consumption of each request class		X
	Users' think time		X
Environment	Environmental stages	X	
	Average duration of each stage		X
	Transition probabilities between stages		X
	Efficiency factor in each stage		X

4. FG CONFIGURATION

In this section we provide additional details regarding the configuration of the FG Analysis.

4.1 User input

For ease of use, the FG tool should require a limited input from the user. This information will be

- Frequency (F): this parameter is used to determine with which frequency the FG analysis must be run. It is expected to be run sporadically, since the FG analysis relies on computationally-intensive estimation routines, and these routines provide better results if more monitoring information is available. During the test stage, the FG analysis could be run daily to rapidly gain knowledge about the application behavior, but once in production the FG analysis is expected to be run on a monthly basis.
- Horizon of analysis (H): this parameter defines the length of the observation interval used for FG analysis. If this is set equal to x hours, it means that only the data collected during the last x hours is considered for FG analysis.
- Monitoring Intensity (MI): this parameter is used to determine how intensive the data collection must be, setting the value of a single parameter in the range $(0,\ 100]$. The value x of this parameter indicates that $x\%$ of the total available samples are collected randomly. A value of 100 would therefore imply the collection of all available samples, increasing the data-collection overhead, but limiting the time necessary to obtain a significant number of samples. Therefore, large values are suggested for test deployments, while small values are best suited for production.

Table 2: Monitoring Data required for the FG analysis

Parameter	Data Required	Level		Platform		Data Collector
		App.	VM	IaaS	PaaS	
Population	Total Number of requests	X		X	X	App. DC
Resource Consumption	Utilization		X	X		Sigar/Collect
	Throughput	X		X	X	App. DC
	Queue Length	X		X	X	App. DC
	Response Times	X		X	X	App. DC
	Queue Length (arrival)	X		X	X	App. DC
Think time	Throughput	X		X	X	App. DC
	Total Number of Requests	X		X	X	App. DC
	Mean Number Requests	X		X	X	App. DC
Stage duration, transition probs. and efficiency	Start-up duration		X	X		Start-up DC
	Availability (Up/Down)		X	X		Availability DC
	CPU Steal		X	X		Collectl

- Maximum Collection Window (MCW): some of the estimation methods for FG analysis require a complete trace, that is, a record with all the events (e.g. all the calls to an application method) in a time interval. As this requires the activation of a significant number of data collectors, which may incur in undesired overhead, the MCW parameter allows the user to fix the maximum length of the collection period where complete traces are being collected. This means that the data collectors required for complete data traces will be activated during a period of maximum length MCW, and then deactivated for a period long enough to comply with the MI parameter. For testing, the MCW parameter can be set to a large value, posing little or no constraints to the data collection. For production, the MCW can be set to a small value, in the order of minutes, which together with a small MI guarantee little overhead to collect the data required by the FG component.

4.2 FG setup

This is an automatic step that determines which routines need to be run in the FG Analysis, and what monitoring information must be collected for this purpose. Since the main objective of the FG Analysis is to parameterize the design-time QoS model, this model determines the specific requirements in terms of monitoring information. Here we focus on a QoS model based on Layered Queueing Networks [5, 14], which can be evaluated with tools such as LQNS or LINE [11]. We consider the extended LQN model underlying LINE, which adds to the standard LQN a random environment to describe changes in the application beyond the control of the application manager [4]. A random environment can model for instance temporary VM failures or high-contention in virtualized deployments. For this model we determine the three main sets of information required as listed in Table 1. We also indicate if each of the items corresponds to structural information, or if it can be estimated using monitoring data.

From the last column in Table 1 we identify six parameter sets that can be estimated, and must therefore be provided by the FG analysis using the monitoring data collected at runtime. We now consider the information sources needed for each of these parameters. The data monitors/collectors mentioned are assumed to be provided by a monitoring system, for example the MODAClouds monitoring platform [1]. Table 2 summarizes the following description, indicating whether a metric is available at IaaS or PaaS level, and which data collector is able to collect this information.

Population: to estimate the total user population we need to record the number of active requests executing each of the *main* application methods. We therefore setup a *monitor* for the execution of each of these methods.

Resource consumption: to estimate the resource (CPU) consumption there are a number of alternative methods, based on different sets of information, as discussed in Section 3. We consider three main options:

- Utilization and throughput: a number of methods use these two quantities to estimate the CPU consumption for each request class. The utilization information is only available in IaaS deployments, and will therefore be collected only in those cases. Thus, for IaaS deployments, we setup a VM-level utilization collector. For the throughput, we need to track the number of calls to each application method in the QoS model. We therefore setup an application-level data collector to register the calls to each of these methods.
- Queue lengths: one of the methods presented in Section 3 relies on the queue lengths at each resource, which is equivalent to the number of threads executing each application method in each resource. To collect this data we rely on an application monitor for each method, that register the calls to each method and its corresponding response time. From this information, it is possible to reconstruct the number of threads executing each method as a function of time, from which the time-average queue lengths can be readily obtained.
- Response times and queue lengths at arrival time: Section 3 also describes an estimation method based on the response times attained by a thread executing an application method, and on the number of threads executing the same method, as observed by the thread just before it starts execution. As in the previous case, this information can be collected by setting an application monitor for each application method that register the time of each call to the method and the response time experienced by the calling thread.

Think time: the mean think time is an unobserved quan-

tity that can be obtained by indirect measurements. Specifically, using Little's law, we can estimate the mean waiting time as the ratio between the mean number of inactive users and the overall request throughput. For the overall request throughput we setup a monitor for each of the main application methods, to register the successful execution of each of these methods. Summing up the number of executions of these methods over a time interval provides an estimate of the overall throughput. On the other hand, the mean number of inactive users can be obtained as the difference between the total population and the mean number of active users. These two quantities can be obtained using the monitors described above to obtain the total population. Notice that for *open models*, we can replace the estimation of populations and think times by that of the request arrival rate, which can be obtained similarly to the throughput.

Stage duration: the extended LQN model we consider includes four random-environment *stages*, namely start-up, failure, low and high contention. These stages describe the state of the *resources* on which the application is deployed. To estimate the mean duration of a sojourn in each of these stages we require different monitoring information.

Start-up: for the duration of this stage we need to setup a VM-level collector to register the start-up times of the VMs on which the application is deployed.

Failure: in this case we setup an availability monitor on the VM state, which registers an *up* or *down* state. From this information it is possible to re-construct the length of the up and down periods, and to estimate the mean duration of the failure stage.

Low/High contention: for these two phases we first need the length of the *up* period, collected in the previous item. Next, we need to identify periods where the underlying resource is being heavily used by other applications. To this end, we set up VM-level monitors on the CPU Steal Time, which is the percentage of time that the hypervisor assigns the CPU cycles to a process different from the application VM. This information can be later processed to differentiate periods of low and high contention, and to extract the mean durations of these periods. Notice that these measurements are available for IaaS platforms, but not on PaaS deployments. The case of PaaS will be considered as part of future work.

Stage transition probabilities: the estimation of the stage transition probabilities can be done with the same monitors deployed for the stage duration. From the information collected by those monitors we can determine, for instance, the fraction of times that the completion of a visit to the failure stage is followed by a visit to the high-contention stage, versus a visit to the low-contention stage. These fractions are the maximum-likelihood estimators of the stage transition probabilities, and are therefore enough to estimate them.

Stage efficiency factor: the efficiency factor is used to determine how the resource processing rate changes in each stage, compared to a baseline. In our case with four environmental stages, we take the *low-contention* stage as the baseline, and therefore its efficiency factor is 1. For the start-up and failure stages, the efficiency factor is 0, since the resources cannot process any request in these stages. The estimation is then limited to the high-contention stage, where its efficiency factor $HC < 1$ reflects that the application VM receives less CPU cycles compared to the low-contention stage. The estimation in this case is performed using the CPU Steal Time, for which a monitor has already been deployed to estimate the stage duration.

5. FG ANALYSIS

In this section we describe how the FG analysis is executed. The information required to configure the FG Design time component are the user-defined parameters (see Section 4): F, H, MI, and MCW.

- According to the MCW and MI parameters set by the user, the FG Design-Time component modifies the deployed data collectors, to activate them during certain time intervals to collect complete traces. After a period of collection, it deactivates the collectors if necessary to comply with the user requirements.
- According to the F and H parameters, the FG Design-Time component executes the routines for FG analysis, querying the Monitoring History DB to obtain the information required during the horizon H, pre-processing it, and executing the corresponding analysis routine.

The routines available for FG Analysis vary depending on the parameters to be estimated. Table 3 lists the routines for each parameter, depending on the data required, and the type of data pre-processing required, if any. The routines can be classified in three main groups:

RNT: this is a single routine in charge of estimating the request numbers and throughputs. The estimation of these parameters is performed by a single routine because it entails counting the number of active requests, executing the main application methods, along the observation horizon. From these counts, both the request numbers and their throughputs can be estimated. A time-window pre-processing is required to divide the observation horizon into smaller intervals, or time-windows, to obtain observations of the throughput.

ENV: this is a single routine that puts together all the information available regarding the environmental stages to determine the mean duration of the visits to each stage, the stage transition probabilities, and the stages' efficiency factor. No pre-processing is assumed in this case as the raw observations are required for estimation.

Resource consumption: this is a group of routines focused on estimating the request resource consumption. In this case, a number of routines are available, of which we consider three here. The GQL and the MINPS are the estimation methods described in Section 3. As stated before, the GQL method relies on samples of the queue lengths, while MINPS requires observations of response times and queue lengths at arrival times. We also consider UBR, the regression-based technique proposed in [17] that requires utilization and throughput samples; UBO, the optimization-based method proposed in [9] that, in addition, uses average response times; and the FCFS regression method from [7] that makes use of response times and queue length samples. To simplify the usage of the different estimation methods, we have defined a common data format that includes fields for CPU utilization, request arrival timestamps, response times, throughput, and mean response times. In particular, for the throughput and mean response times, a time-window

Table 3: Estimation routines and input data for FG analysis

Parameter	Data Required	Preprocessing		Routine
		Data Format	Time-windows	
Population	Total Number of requests		X	RNT
Resource Consumption	Utilization	X	X	UBR
	Throughput	X	X	UBO[1]
	Queue length	X	X	GQL
	Response Times	X	X	MINPS
	Queue length (arrival)	X	X	FCFS
Think time	Throughput		X	RNT
	Total Number of requests		X	RNT
	Mean Number Requests		X	RNT
Stage duration and transition probs. and efficiency	Start-up duration			ENV
	Availability (Up/Down)			ENV
	CPU Steal			ENV

processing is required as the observation horizon needs to be partitioned in smaller time-windows, for each of which a single observation is obtained.

The results of the FG Analysis, i.e., the values of the estimated parameters, are stored in timestamped XML files that can be used to parameterize the design-time models.

6. CONCLUSION

In this paper, we have introduced a tool to fill the gap between development and operations. Our focus has been on designing the tool, identifying architecture and user requirements. We believe that the main contribution of this paper is to highlight, given the large number of performance parameters that could be exposed in the feedback loop from the runtime to the developer, which would be the most relevant from a software performance engineering perspective. Currently, we are working towards releasing an initial version of the tool, which will be made available in Spring 2015 on www.modaclouds.eu.

7. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union Seventh Framework Programme FP7/2007-2013 under grant agreement no. 318484 (MODAClouds), and from an AWS in Education Research Grant.

8. REFERENCES

[1] M. Balduini, E. di Nitto, M. Miglierina, V. Munteanu, G. Casale, J. F. Pérez, and W. Wang. MODAClouds D6.3.1 - Monitoring platform - initial release, 2013.

[2] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In *Proc. of the 6th WOSP*, pages 54–65, 2007.

[3] F. Brosig, N. Huber, and S. Kounev. Architecture-level software performance abstractions for online performance prediction. *Science of Computer Programming*, 90:71–92, 2014.

[4] G. Casale, M. Tribastone, and P. G. Harrison. Blending randomness in closed queueing network models. *Performance Evaluation*, 82(0):15 – 38, 2014.

[5] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. Soft. Eng.*, 35:148–161, 2009.

[6] A. Kalbasi, D. Krishnamurthy, J. Rolia, and S. Dawson. Dec: Service demand estimation with confidence. *IEEE Trans. Soft. Eng.*, 38:561–578, 2012.

[7] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson. Estimating service resource consumption from response time measurements. In *Proc. of the 4th VALUETOOLS*, 2009.

[8] D. Kumar, L. Zhang, and A. Tantawi. Enhanced inferencing: estimation of a workload dependent performance model. In *Proc. of the 4th VALUETOOLS*, 2009.

[9] Z. Liu, L. Wynter, C. H. Xia, and F. Zhang. Parameter inference of queueing models for IT systems using end-to-end measurements. *Performance Evaluation*, 63(1):36–60, 2006.

[10] D. Menascé. Computing missing service demand parameters for performance models. In *CMG 2008*, pages 241–248, 2008.

[11] J. F. Pérez and G. Casale. Assessing sla compliance from palladio component models. In *Proc. of the 2nd Workshop on Management of resources and services in Cloud and Sky computing (MICAS)*, 2013.

[12] J. F. Pérez, S. Pacheco-Sanchez, and G. Casale. An offline demand estimation method for multi-threaded applications. In *MASCOTS*, pages 21–30, 2013.

[13] J. Roche. Adopting devops practices in quality assurance. *Commun. ACM*, 56(11):38–43, 2013.

[14] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. on Soft. Eng.*, 21(8):689–700, 1995.

[15] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proc. of the 3rd ICPE*, 2012.

[16] W. Wang and G. Casale. Bayesian service demand estimation using gibbs sampling. In *MASCOTS*, 2013.

[17] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proc. of the 4th ICAC*, 2007.

[1] This method also requires average response times.