Performance Issues? Hey DevOps, Mind the Uncertainty!

Catia Trubiani, Gran Sasso Science Institute

Pooyan Jamshidi, University of South Carolina

Jurgen Cito, Massachusetts Institute of Technology

Weiyi Shang, Concordia University

Zhen Ming Jiang, York University

Markus Borg, RISE Research Institutes of Sweden AB

// DevOps is a novel trend that aims to bridge the gap between software development and operation teams. This article presents an experience report that better identifies performance uncertainties through a case study and provides a step-by-step guide to practitioners for controlling system uncertainties. //

DEVOPS IS A recent trend that integrates development (dev) and operational (ops) teams.[5] One of the reasons for this type of integration is the need to continuously adapt software system designs based on operational uncertainties, such as workload fluctuations (WFs) and resource availability. Such uncertainties inevitably affect the dependability characteristics of systems (e.g., performance and reliability) that may degrade as well as produce negative consequences. For instance, a software system can perform well initially with high throughput and low response time, but its performance may suddenly worsen because of reasons like WFs or software upgrades. To make informed decisions, DevOps teams must be aware of uncertainties in the entire DevOps life cycle so that they are able to interpret data, models, and results accordingly.

Support for this process can be provided by 1) identifying sources of uncertainty in a performance-aware DevOps scenario, 2) elaborating on how these uncertainties are manifested in input data, design models, and operational results, and 3) performing a sensitivity analysis that quantifies the impact of these sources of uncertainty for results interpretation. It is therefore necessary to put in place a set of methodologies that model and control these uncertainties so that violations of performance requirements can be detected, thereby linking the operational performance issues to the decision process of software designers.

In literature, uncertainty is embedded in the concept of variability (i.e., the natural variation of some parameters), and it is very relevant in the analysis process.[1,2] Therefore, it is important to incorporate some form of uncertainty representation[12] into the engineering process and identify software characteristics that are not completely known. There exist performance prediction approaches that provide a sensitivity analysis of many parameters by monitoring a system or considering reasonable guesses by the domain experts. However, such model-based approaches are typically used for predicting system performance as a result of system configuration[6] or external uncertainties. These estimations are only approximations and

may result in low accuracies that could be misleading during the software development process.[14]

This article has two main contributions: 1) we make the developers aware of the system uncertainties and 2) we run a sensitivity analysis to highlight the main system criticisms that lead to performance issues. In doing so, we demonstrate that it is essential to bring the sources of uncertainty up front in the DevOps process to apprise the developers of these uncertainties and guarantee the stakeholders' performance expectations.

Related Work

This section briefly reviews related works that have been defined to model, analyze, and minimize the software system's uncertainties. Note that variability can be considered a specific case of uncertainty because it includes the specification of parameters subject to varying values, whereas uncertainty also considers the lack of knowledge.[12]

Modeling and Analyzing Uncertainty

The uncertainty concept is discussed in many scientific fields. Kennedy and O'Hagan distinguish between six sources of uncertainty for models implemented in source code:[8]

- *Parameter uncertainty*: originates from calibrating the model, i.e., finding the actual parameter input values
- *Model uncertainty*: the difference between the real-world process and the code output given to the system model
- *Residual uncertainty*: originates from an inherently unpredictable real-world process; even under stable conditions, such a process might produce different output when repeated

- *Parametric uncertainty*: introduced when some of the input conditions are not specified by the parameter input, either intentionally or because of an uncontrollable process
- *Observation error*: occurs when actual observations are used to calibrate the system model
- *Code uncertainty*: variations in the output that are produced from executing a system model on a given platform.

Ramirez and coauthors introduced an uncertainty taxonomy that establishes a common vocabulary for the self-adaptive software community.[10] This taxonomy is composed of three phases of the development life cycle, i.e., requirements, design, and runtime. In particular, the authors identified 26 sources of uncertainty, ranging from missing requirements to sensor noise. This taxonomy highlights three aspects of uncertainty (i.e., location, level, and nature[9]) and facilitates the understanding of 1) where uncertainty manifests in the model, 2) what the level of uncertainty is (from deterministic knowledge to not even being certain about being uncertain), and 3) whether the uncertainty is caused by lack of measurement data or by inherent randomness in the model.

Minimizing Uncertainty

Minimizing uncertainty results is an open research challenge. Considerable research effort has been invested to minimize the uncertainty in software engineering experiments. Three main techniques have been adopted in this context. First, a widely adopted resolution of minimizing performance-related uncertainty is achieved using repeated measurements.[7] The instability of performance measurements leads to uncertainties of performance

evaluation results and may be misleading and incorrect without providing measurements of variation.[7] Georges et al.[3] recommended computing a confidence interval for repeated performance measurements when implementing a performance evaluation. With the knowledge of variation from repeated measurement, rigorous statistical techniques can be used to minimize uncertainty. Second, another way of minimizing uncertainty is to gain more knowledge about the nature of the system, based on extensive modeling and simulation. With models, it is possible to specify the uncertainty of parameter values through probability distribution functions.[13] In this way, Monte-Carlo-based simulations enable the extraction of parameter value samples, which minimize uncertainty. Goldsby and Cheng[4] developed a model-based approach that generates a system model to simulate system behavior in complex environments. Developers use this type of model to interactively understand the uncertainty in such environment. Finally, another method that minimizes uncertainty is to provide more information about the subject system, thereby reducing uncertainties. Yuan et al.[15] enriched the monitoring of systems by attaching more runtime information.

Decisions in a Performance-Aware DevOps Life Cycle Leading to Uncertainty

In this section, we describe the envisioned performance-aware DevOps life cycle whose high-level illustration is reported in Figure 1. A central element of this life cycle is represented by the models used for decision making and may be formal models that predict the system runtime (e.g., queuing models) and system models that test design changes. For these
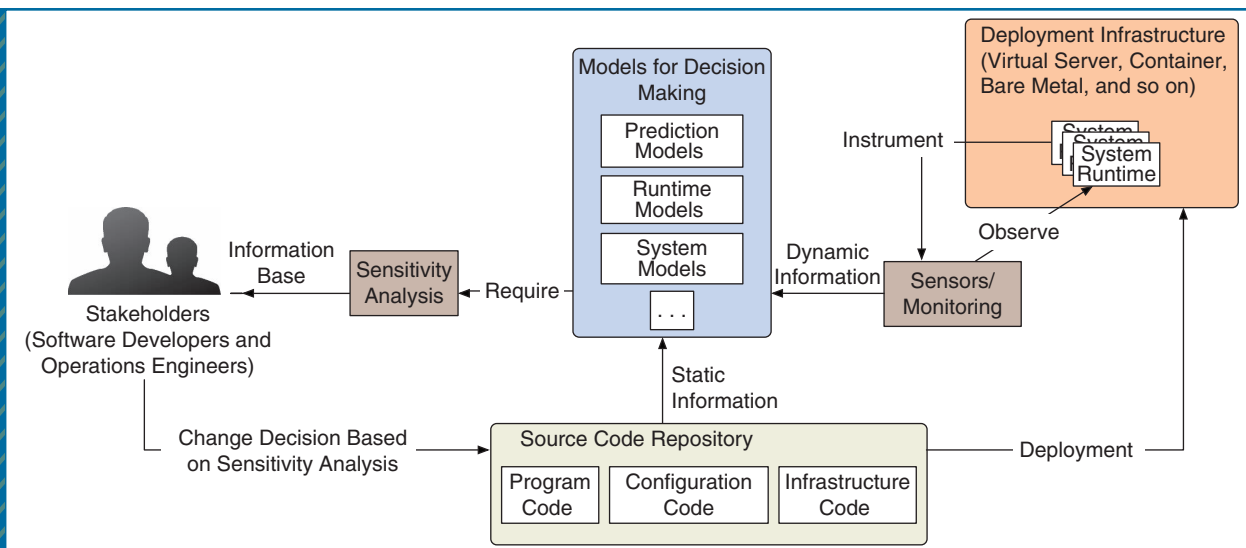
FIGURE 1. A performance-aware DevOps life cycle under uncertainty.

models, there are two types of inputs: static information in source code repositories and dynamic information instrumented or observed during the system runtime. A sensitivity analysis, i.e., the study of how the uncertainty in the output can be apportioned to different sources of uncertainty in the input,[11] closes the cycle. Stakeholders use this information base to form an understanding of the system at various levels, e.g., influencing their decisions in changing the running system by altering the source code, configurations, infrastructure, or even redeploying the system.

Design decisions are influenced by the uncertainty of the system under development. [9] To this end, we elaborate on the high-level decisions taken by the stakeholders that may affect the performance analysis results.

1) *Deployment infrastructure* (*DI*): The physical or virtual infrastructure used to deploy a system can have a tremendous

effect on the uncertainty of various performance characteristics, especially in public cloud infrastructures.

2) *Software versions and code changes* (*SCs*): The uncertainty of the performance characteristics can also be directly introduced by a code change through a software developer or indirectly by an operator's decision to upgrade to a different version of the software.

3) *Configuration parameters* (*CPs*): The small adjustments of the software CPs can have a tremendous effect on the uncertainty of the runtime characteristics of a system, e.g., the SQLite benchmarking has often been found to be incorrect.

4) *WF*: The performance of a system is a function of its workload, and a linearly increasing load may have nonlinear effects on a system. Wrongfully interpreting future states of a system when higher workloads occur

can lead to uncertain decisions that affect CPs.

5) *Monitoring and sensor accuracy* (*MS*): Operators rely on active monitoring, instrumentation, and sensors to observe and retrieve information about the (internal) state of a system, and adjusting the accuracy of sensors (e.g., through sampling) determines an inherent tradeoff between the visibility of the state and the introduced overhead.

Case Study

We conducted a controlled experiment as a case study to illustrate the effects on uncertainty caused by the DevOps decisions depicted in the previous section. The goal of this case study was to demonstrate the performance impact of various DevOps decisions. We assessed and quantified both the Dev (e.g., code or configuration changes) and the Ops (e.g., hardware- and workload) side changes, which may impact system performance. Specifically, we

measured different performance characteristics of Apache Cassandra while changing the DI, source code repository, and workload. For the sake of illustration, we focused on the throughput (i.e., sensors/monitoring, as shown in Figure 1) of the benchmark queries against the Cassandra engine. However, a broader definition of performance can be extended to any quantitative nonfunctional property (e.g., reliability and security) of the system. We modeled potential actions that could affect uncertainty as a discrete decision space $D = \{DI, SC, CP, \text{ and } WF\}$. DI represents the space of underlying hardware (i.e., the DI); $SC = \{v_1, \ldots, v_n\}$ depicts the space of software releases (i.e., SC); $CP = \{c_1, \ldots, c_n\}$ is a set of configuration options that can be set for a particular version of software; $WF = \{w_1, \ldots, w_n\}$ models the workload change on the system as a set of relative percentages of read and write operations; and finally, |D| represents the number of combinations for all of these decision parameters.

Experimental Setting

We measured the performance characteristics of Apache Cassandra under different environmental changes (Table 1), which represent the dimensions of concrete instances of the decision space D. For our Cassandra case study, we conducted systematic measurements (for measuring the performance indicators of one configuration for a specific system version in specific environments and for specific workloads). We ran the benchmark for 10 min with the same operation repeated multiple times. Before the next measurement, the Cassandra database was cleaned and restarted to ensure each measurement began with the same initial state. The Cassandra

database was left idle for 10 s, i.e., the warmup period before the start of each measurement round. We used the Yahoo! Cloud Serving Benchmark [(YCSB) https://github.com/brianfrankcooper/YCSB] for generating different workloads and collecting the performance indicators of the system. More specifically, we used the YCSB workload generator to first define the data set and load it into the database, and secondly to execute operations against the data set while measuring performance. YCSB is a standard benchmarking tool that has been used extensively for performance measurements of key-value and cloud-based data engines.

We observed output parameters in many different combinations, e.g., hardware change, workload change, version change, workload-version change, and hardware-workload-version change. In particular, we measured system performance considering six configuration options (leading to a total of 1,024 system configurations) in different environments, i.e., two hardware environments, three software versions, and six workloads.

Table 2 provides a summary of the underlying hardware options in this case study. To understand the performance with respect to varying incoming request rate behavior, in light of different choices made in the decision space of the Apache Cassandra benchmark system, we ran six core YCSB workloads:

1) *Workload A* (*update heavy*): This workload has a 50/50 mix of reads and writes.
2) *Workload B* (*read mostly*): This workload has a 95/5 reads/writes mix.
3) *Workload C* (*read only*): This workload is 100% read.
4) *Workload D* (*read latest*): New records are inserted, and the most recently inserted records are the most popular.
5) *Workload E* (*short ranges*): Short ranges of records are queried, rather than individual records.
6) *Workload F* (*read-modify-write*): The client will read a record,

Table 1. An overview of the case study subject system.

System	Domain	|D|	|DI|	|SC|	|CP|	|WF|
Cassandra	Database	1,024	2	3	6	6

|D|: the number of all possible decisions; |DI|: the number of hardware environments; |SC|: the number of analyzed software versions; |CP|: the number of configuration options; |WF|: the number of different analyzed workloads.

Table 2. A summary of the hardware platforms on which configurable software systems were measured.

ID	Type	NC	IS	CPU	CCR	RAM (GB)	Disk
h1	NUC	4	x86_64	i5-4250U	1.30	15	SSD
h2	NUC	2	x86_64	Celeron	2.13	7	SCSI

NC: the number of CPUs; IS: instruction set; CCR and CPU: clock rate (GHz); RAM: memory size; SSD: solid-state drive.

Decision	ID	Source	Target	Top	Bottom	|Top/bottom|	Correlation	Correlation (10%)
DI	ec_1	h2-A-V3	h1-A-V3	0.0980	0.1569	0.0589	0.0364	−0.0078
SC	ec_2	h1-A-V3	h1-A-V2	0.0490	0.0588	0.0098	−0.1266	−0.0527
	ec_3	h1-A-V3	h1-A-V1	0.1176	0.0376	0.08	0.1424	0.0696
WF	ec_4	h2-A-V3	h2-B-V3	0.0392	0.0686	0.0294	−0.1732	0.0139
	ec_5	h2-A-V3	h2-C-V3	0.1373	0.1275	0.0098	0.0318	0.0381
	ec_6	h2-A-V3	h2-D-V3	0.1471	0.1176	0.0295	0.0088	0.0172
	ec_7	h2-A-V3	h2-E-V3	0.0490	0.0686	0.0196	−0.0704	0.0127
	ec_8	h2-A-V3	h2-F-V3	0.0686	0.1373	0.0687	0.0217	0.0078
SC-WF	ec_9	h1-A-V3	h1-B-V1	0.1078	0.1765	0.0687	0.1001	−0.0302
DI-SC-WF	ec_{10}	h2-A-V3	h1-B-V1	0.1078	0.1176	0.0098	−0.0327	0.0192

Table 3. The results. Top/bottom: the percentage of top/bottom common configurations between source and target.

modify it, and write back the changes.

We considered three different versions of Apache Cassandra for our measurements: SC = v_1 = 1.2.19, v_2 = 2.2.8, and v_3 = 3.10. We chose the most recent release, the next-closest release (2.2.8), and a very distant version of the system (1.2.19). We also artificially injected white noise of 10% into the sources to investigate the performance impact as a result of measurement noise. More details regarding the configuration options, considered environments as well as all of measurement data are publicly available at https://github.com/pooyanjamshidi/uncertainty.

Results

We analyzed the percentage of the top/bottom configurations that are common across the environmental changes. Table 3 shows the results, and uncertainty indicates a dramatic influence when we consider variations along hardware, workload, version changes, or combinations of them. More specifically, we considered throughput as a metric and derived 10% of top (highest throughput) and 10% of bottom (lowest throughput).

The results show that the percentage of common configurations is very low. In particular, we found that in ec_6 (where ec stands for environmental change in which the source environment is different from the target), practitioners have a greater likelihood of achieving top throughput; at the same time, they have a higher change in ec_9 to spot performance issues if they choose the same configuration (however, the likelihood is still low). Also, the major gap (0.08) between top and bottom values is showed by ec_3, which demonstrates that in this environment change, it is more likely that they find a high-performing configuration and not observe a performance issue. This means that, despite uncertainties in the parameters, ec_6 is the system configuration that shows the highest system throughput and ec_9 is the one that is most likely to cause performance issues; whereas ec_3 is the one whose input uncertainties largely influence performance analysis results.

This sensitivity analysis supports system administrators in their task of setting the best configuration of the system using specific hardware, workload, and version of the software or their combinations. In the last two columns of Table 3, we report the Spearman rank correlation values that were calculated using the original sources and the artificially injected data (i.e., white noise of 10%), respectively. We observe that the rank correlation, despite being weak, still shows a decreasing trend. This observation highlights the importance of handling uncertainty in the DevOps process as well as determining the sources of uncertainty that are attributable to environmental changes and

measurement noise. For instance, if a developer selected a configuration based on previous measurements in a certain environment, he or she should be careful when choosing a configuration for the system in another environment because some environmental changes are more prone to uncertainty than others.

Our numerical results provide evidence which suggests that if uncertainty is not handled properly, performance issues may arise. For example, if a system configuration is selected based on a model trained by measurement data that are collected in an environment with a different workload, it may lead to a suboptimal configuration. As a result, systems may encounter higher deployment costs or more failures because of larger memory allocations or threads that spin up. This can be more problematic in critical domains, such as robotics. A detailed experience has been presented in the Model-Based Adaptation for Robotics Software project (https://github .com/cmu-mars), in which power models are used under budget constraints to adapt to perturbations, such as environmental or internal resources changes (e.g., battery level). Pareto optimal configurations are swapped at runtime based on the environmental conditions of the robot (e.g., its remaining battery level). We determined that when the model is inaccurate, Pareto optimal configurations are chosen incorrectly, and it results in a mission failure, as shown in the demo at https://youtu .be/ec6BhQp2T0Q.

Given these consequences, if practitioners are aware of the uncertainty, they can opt to

- conduct additional experiments that further reduce the

uncertainty (e.g., repetitive measurements combined with statistical methods are widely used in prior research to reduce uncertainty)
- identify and handle the root cause of the uncertainty (e.g., if the DI introduces the uncertainty, one should consider control or leverage a more stable infrastructure)
- if the uncertainty cannot be easily reduced or handled, uncertainty quantification approaches (e.g., forward-uncertainty propagation or inverse-uncertainty quantification) can be utilized to determine how likely certain outcomes are if some aspects of the system (e.g., optimal configurations) are not deterministically known.[12]

Similar results have been observed in other systems (including a compiler, a satisfiability solver, a database engine, and a video encoder) across different environmental changes (for more details, see https://github.com/pooyanjamshidi /ase17). Note that the considered uncertainties are demonstrated to be relevant for some software systems, but they are far from being exhaustive. Further applications may show other characteristics that have not been evidenced so far because it indeed difficult to link performance issues to a finite list of system configuration settings.

In conclusion, we discuss the lessons learned and their implications as a result of our study in the following two dimensions: 1) identifying sources of uncertainties and 2) modeling and controlling the uncertainties.

Identifying Sources of Uncertainty
The identification of the sources of uncertainty is challenging and two different strategies can be used for this scope: 1) the bottom-up approach that is based on the knowledge of the possible sources of uncertainty in the given model and 2) the top-down approach, which is derived from the complete lack of knowledge about possible uncertainties. We followed the bottom-up approach,[9] and we began by enumerating various decision points in the performance-aware DevOps life cycle, as shown in Figure 1. Then, we further limited the decision points to only those that lead to performance uncertainty and we narrowed them down using five sources of uncertainty.

Modeling and Controlling Uncertainties
Once we have identified the sources of uncertainties, we want to minimize their impact by modeling and analyzing the performance variation caused by such uncertainties and then devise approaches that control their impact. In our case study, we observed the following sources of uncertainty.

DI
Our results show that within the same release and workload, changing the DI can have an impact on system performance ranging from small to large. For example, for version 3.10, the throughput can be more than doubled it is switched between two different hardware platforms. Therefore, to limit such uncertainties, it is important to conduct user-acceptance testing or closely monitor the performance of the canary deployment[5] prior to implementing full-fledged infrastructure changes.

SC
Our results show that the optimal configurations for one version of Cassandra

ABOUT THE AUTHORS

CATIA TRUBIANI is an assistant professor at the Gran Sasso Science Institute. Trubiani received a Ph.D. from the University of L'Aquila. Her research interests include model-based performance analysis and feedback on software architectures under uncertainties and optimization of large-scale software systems. She received the Best Research Paper Award at the European Conference on Software Architectures, the Microsoft Azure Research Award, and the Best Research Paper Award at the International Conference on Performance Engineering. Contact her at catia.trubiani@gssi.it.

WEIYI SHANG is an assistant professor and research chair on ultra-large-scale systems at Concordia University, Canada. His research interests include big data software engineering, software engineering for ultra-large-scale systems, and software log mining. Shang received a Ph.D. from Queen's University, Canada. He received the SIGSOFT Distinguished Paper Award at ICSE 2013 and the Best Paper Award at WCRE 2011. Contact him at shang@encs.concordia.ca.

POOYAN JAMSHIDI is an assistant professor at the University of South Carolina. Jamshidi received a Ph.D. from Dublin City University. His research interests are at the intersection of software engineering, systems, and machine learning, with a focus on the areas of distributed machine-learning systems. Contact him at pjamshid@cse.sc.edu.

ZHEN MING JIANG is an associate professor in York University's Department of Electrical Engineering and Computer Science. His research interests include software engineering and computer systems, with special interests in software performance engineering, software analytics, and source-code analysis. Jiang received a Ph.D. from the School of Computing at Queen's University. He received several Best Paper Awards at ICST 2016, ICSE 2015 (the Software Engineering in Practice track), ICSE 2013, and WCRE 2011, respectively. Contact him at zmjiang@cse.yorku.ca.

JURGEN CITO is a postdoctoral researcher at the Massachusetts Institute of Technology. His research interests include developing approaches that produce reliable infrastructure through synthesis and repair techniques. Cito received a Ph.D. from the University of Zurich, where he worked on techniques that help software developers reason about performance problems in their development workflows. Contact him at jcito@mit.edu.

MARKUS BORG is a senior researcher at the RISE Research Institutes of Sweden AB and an adjunct senior lecturer at Lund University, where he also received a Ph.D. His research interests include requirements engineering and software testing, particularly for systems powered by machine learning. He is a board member of Swedsoft and a Member of the IEEE. Contact him at markus.borg@ri.se.

would probably not yield the best performance after a system upgrade. As shown in Table 3, when switching between versions (ec_2 and ec_3), there is less than 12% of top configurations shared between the two versions (i.e., before and after system update). Therefore, it is vital to examine the performance impact of various CPs for each version. However, because of the large combination of the configuration space and the rapid changes in DevOps, performance-testing-reduction techniques (e.g., experimental design or redundancy detection) should be used to efficiently explore the system configuration space.

CPs

Among different hardware platforms, software versions, and workloads, the percentage of common configurations is very low. Within the same kind of setting (i.e., same hardware, workload, and release version), the throughput can vary by up to nine-times difference among different Cassandra configurations. This clearly shows how important CPs are in terms of system performance. However, to minimize and control the uncertainty caused by CPs, it is necessary to isolate and study the most relevant ones.

WF

Similar to the aforementioned three aspects, for Cassandra, the optimal configurations do not translate when different workloads are exercised. As the system keeps evolving, the user behavior coevolves. It is therefore important to periodically verify and update the performance-testing workload.

MS

The measurement noise can impact the validity of the performance results, although its overall effect can be small. Hence, it is helpful to cross-reference the measurement data to ensure their validity.

References

1. M. Autili, V. Cortellessa, D. Di Ruscio, P. Inverardi, P. Pelliccione, and M. Tivoli, "Eagle: Engineering software in the ubiquitous globe by leveraging uncertainty," in *Proc. ACM Symp. Foundations of Software Engineering (FSE)*, 2011, pp. 488–491.

2. D. Garlan, "Software engineering in an uncertain world," in *Proc. Int. Workshop Future Software Engineering Research (FoSER)*, 2010, pp. 125–128.

3. A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proc. ACM SIGPLAN Conf. Object-Oriented Programming Systems and Applications (OOPSLA)*, 2007, pp. 57–76.

4. H. J. Goldsby and B. H. Cheng, "Automatically generating behavioral models of adaptive systems to address uncertainty," in *Proc. Int. Conf. Model Driven Engineering Languages and Systems (MoDELS)*, 2008, pp. 568–583.

5. M. Httermann, *DevOps for Developers*. Apress, 2012.

6. P. Jamshidi, N. Siegmund, M. Velez, C. Kastner, A. Patel, and Y. Agarwal, "Transfer learning for performance modeling of configurable systems: An exploratory analysis," in *Proc. Int. Conf. Automated Software Engineering (ASE)*, 2017, pp. 497–508.

7. T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proc. Int. Symp. Memory Management (ISMM)*, 2013, pp. 63–74.

8. M. C. Kennedy and A. O'Hagan, "Bayesian calibration of computer models," *J. Roy. Statistical Soc.: Statistical Methodology Series B*, vol. 63, no. 3, pp. 425–464, 2001.

9. D. Perez-Palacin and R. Mirandola, "Dealing with uncertainties in the performance modelling of software systems," in *Proc. Int. Conf. Quality of Software Architectures (QoSA)*, 2014, pp. 33–42.

10. A. J. Ramirez, A. C. Jensen, and B. H. Cheng, "A taxonomy of uncertainty for dynamically adaptive systems," in *Proc. Int. Symp. Software Engineering Adaptive and Self-Managing Systems (SEAMS)*, 2012, pp. 99–108.

11. A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto, *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. Hoboken, NJ: Wiley, 2004.

12. R. C. Smith, *Uncertainty Quantification: Theory, Implementation, and Applications*. Philadelphia: Siam, 2013.

13. A. Aleti, C. Trubiani, A. van Hoorn, and P. Jamshidi, "An efficient method for uncertainty propagation in robust software performance estimation," *J. Syst. Softw.*, vol. 138, pp. 222–235, Apr. 2018.

14. C. M. Woodside, "Regression techniques for performance parameter estimation," in *Proc. WOSP/SIPEW Int. Conf. Performance Engineering*, 2010, pp. 261–262.

15. D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2012, pp. 293–306.