Towards Holistic Continuous Software Performance Assessment

Vincenzo Ferme, Cesare Pautasso
Faculty of Informatics, USI Lugano, Switzerland
firstname.lastname@usi.ch

ABSTRACT

In agile, fast and continuous development lifecycles, software performance analysis is fundamental to confidently release continuously improved software versions. Researchers and industry practitioners have identified the importance of integrating performance testing in agile development processes in a timely and efficient way. However, existing techniques are fragmented and not integrated taking into account the heterogeneous skills of the users developing polyglot distributed software, and their need to automate performance practices as they are integrated in the whole lifecycle without breaking its intrinsic velocity. In this paper we present our vision for holistic continuous software performance assessment, which is being implemented in the BenchFlow tool. BenchFlow enables performance testing and analysis practices to be pervasively integrated in continuous development lifecycle activities. Users can specify performance activities (e.g., standard performance tests) by relying on an expressive Domain Specific Language for objective-driven performance analysis. Collected performance knowledge can be thus reused to speed up performance activities throughout the entire process.

1. INTRODUCTION

We live in the "Continuous" era of software [15]. Companies are adopting DevOps [24], and more recently also NoOps [19], in their software development processes, as they introduce more and more automation and velocity in their software development and release lifecycles characterised by continuous improvement based on feedback [23, 29]. Continuous improvement is defined as "the ongoing improvement of products, services or processes through incremental and breakthrough improvements" [3]. When this applies to software, continuous software improvement (CSI) relates to the iteration over the activities of continuous integration (CI), testing, delivery, deployment (CD), use, and run-time monitoring applied towards releasing software of increasingly good quality, complying with the velocity requirements of agile software development [8, 15].

Although the opportunity to provide more computational resources to cope with developers' non optimised code is a solution that initially may work, it is likely not to be sustainable over time, when the capacity to process an increasingly large transaction volume is demanded. Performance analysis and assessment has always been fundamental for software systems' success [32, 8], and nowadays, with development practices striving to achieve multiple releases to production up to every hour, day, week, month or financial quarter [29, 15] (depending on the context and the company), performance assessment, to complement the testing of functional requirements, is mandatory to guarantee an efficiently functioning deployed software [29, 8]. Industry and academia identified the relevance of integrating performance analysis in agile software development lifecycles [9, 47], and analysed and begun to tackle some of the challenges.

The main challenges [9] in achieving a successful integration, are due to the characteristics of the context, e.g., timeliness to reach the deployment of a new incremental release and the need for heavy automation to achieve it. Other challenges are due to the professional figures that are involved, e.g., the need to simplify the way developers can obtain performance insights [47]. Any performance testing practice introduced in this context should enable and assist the different figures taking part in continuous improvement [15] to implement performance-related activities independently from their performance expertise. In this paper we focus on the research challenges which mostly occur in the intersection between the needs of agile software development practices, the contexts in which they are applied, the types of developed software, and the characteristics of the professional figures working on those software. These concern the integration of performance engineering activities within the full CSI lifecycle, how to describe the corresponding objective-driven performance testing activities and how to execute them under the constraint of providing useful feedback without breaking the velocity of the CSI lifecycle.

1.1 Motivation and Vision

Our goal is to enable developers, quality assurance engineers and operations engineers, with different levels of performance expertise and different objectives, to fully integrate performance practices in the CSI lifecycle. The aim is to empower the targeted users to gain actionable performance knowledge about the developed software complying with the velocity requirements of CI. They can then use the gained knowledge to drive the decisions about improvements leading to better software quality. Given the diversity in the roles of the targeted users, our solution acknowledges the importance of considering the different points of view on the software performance they might have (e.g., a developer is interested in a single service's performance, while an operations engineer is interested on the end-to-end performance of a multi-services application).

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.

ICPE '17 Companion, April 22–26, 2017, L'Aquila, Italy.
© 2017 ACM. ISBN 978-1-4503-4404-3/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3053600.3053636

Current approaches propose solutions that focus on users with specific performance skills or target systems seen as white boxes and implemented with specific programming languages, and are often not supported by tools automating the entire performance assessment process. More research is needed to speed up performance testing activities executed by diversified user profiles so that they can be integrated in the CSI lifecycle without affecting its velocity.

We approach the challenges of integrating performance engineering practices in the continuous software improvement process with an holistic approach where the different user profiles, their heterogeneous performance skills, and the velocity and automation requirements are all key to the success of the integration. More in detail we propose: 1) to declaratively specify performance tests (e.g., a stress test [32]) and **objective-driven tests** (e.g., capacity planning, performance optimisation) together with the specification of the configuration and deployment of the system under test (SUT), and the specification of data collection and analyses requirements, by relying on a **declarative** and **SUT-aware** Domain Specific Language (**DSL**) [30]; 2) to enable the seamless integration of the mentioned performance analysis in the continuous integration, and testing [15] activities of the CSI lifecycle; 3) to enable fast performance feedback, mainly by **reusing accumulated performance knowledge** in all the mentioned activities of the lifecycle to **seamlessly integrate performance engineering activities** matching the velocity requirements.

1.2 Targeted Software

The **types of software systems** we consider are: 1) realised by means of one or more interacting services; 2) exposing APIs so that clients can access the system's functionality to be tested; 3) packaged in containers (e.g., Docker); 4) publishing monitoring data, needed to characterise the system's performance.

Recent studies about DevOps practices show that tools are widely used in the process, and developers and software operations engineers heavily rely on tools automating and assisting their work [11]. Our holistic approach is also supported by a tool: BenchFlow [13], providing an end-to-end framework for black/gray-box performance testing and analysis. BenchFlow integrates state of the art technologies, such as Docker, Faban, and Apache Spark, to reliably execute performance tests and automatically collect and validate performance data. More in detail, BenchFlow works well with any Docker packaged software whose deployment is described using a Docker Compose deployment descriptor, realized by one or more services orchestrated by Docker Swarm and exposing APIs to other services or clients. Although we decided to rely on Docker technologies for the implementation of the proposed solutions, those are not dependent of the specific technology. BenchFlow was initially developed by us for reliable performance benchmarking of Workflow Management Systems middleware [13], and we are extending it to support what we propose in this paper. BenchFlow will provide a DSL for automating the execution of standard performance tests and objective-driven performance tests, and for collecting performance data. The BenchFlow tool and the DSL are designed to provide fast performance feedback by collecting and reusing performance knowledge about the SUT over the different CSI lifecycle activities. The BenchFlow tool will offer complete control over the performance analysis for expert users, while coping with knowledge gaps of non-expert users.

2. CHALLENGES AND RELATED WORK

2.1 CSI Lifecycle Integration

*Define suitable ways to express during which activities of the Continuous software improvement lifecycle to **integrate** which performance tests/queries, limiting the impact on the "continuity" of the lifecycle.*

The integration of performance testing, analysis and engineering activities within the CSI lifecycle is a widely recognized challenge in the literature [9, 47]. CI lifecycle automation [23] best practices organise tests in workflows, pipelines, and steps. These can grow to include additional performance-related activities described using DSLs for performance testing [5, 48, 49, 41], which explicitly support workload definition and testbed configuration. The design of Domain-Specific Languages is also studied in the literature [16, 44], with a large number of possible choices for language syntax and semantics. Regarding performance testing related DSLs, the large number of possible choices led to different decision made by DSL designer, that designed language using YAML [6], UML [4], code [48], or custom formalisms (e.g., in [41]). All the mentioned performance testing focused DSLs support the definition of Load Functions, Workloads, Simulated Users, and Test Data, other DSLs also support TestBed management, analysis of client-side performance data for checking defined conditions, and definition of configuration tests.

In our research we build on the already proposed DSLs, and as we are going to show in Sect. 3.1, we provide a DSL supporting all the features mentioned above, and also SUT-awareness, collection and analysis of server side performance data, integration with CI tools, and last but not least, advanced performance tests definition required to deal with Objective-Driven Performance Testing, and for providing Fast Performance Feedback.

2.2 Objective-Driven Performance Testing

*Enable users to declaratively express **objective-driven performance questions** [47], and get pre-analysed and interpreted answers.*

Westermann et al. [49] present the concept of goal-driven performance testing, mainly related to exploration of the performance space for different configuration of software systems (i.e., the space described by all the possible combinations of the configuration variables). In attacking the complexity of the performance space exploration, they rely on experiment based modeling techniques (selected by the user) and propose techniques for efficiently selecting the points in the space to explore in order to reduce the time needed for the exploration while matching a user-provided precision level. They also introduce the possibility for the users to control the time allocated for the exploration, and specify stop conditions related to the precision of the built performance model. The queries the user can specify in the work by Westermann et al. are limited to configuration exploration. Other work proposed different exploration techniques to answer different kinds of questions: capacity planning [42], performance optimisation in the presence of constraints [46, 10, 50], white and black box regression detection [25], performance bugs identification and prediction [43], and charac-

terise the performance of server infrastructures [20] and of software components [27]. These techniques rely on different strategies for space definition and exploration, mainly based on search-based software engineering techniques [1], or machine learning [31] models [50]. It is difficult to pick the best technique apriori since there is evidence that the most effective techniques are influenced by different factors, as for example the input data [17].

In our research we plan to follow hybrid techniques, combining experiment-based and model-based performance engineering approaches, because they build performance models on which they can base decisions about how to optimise the performance space exploration [39, 36, 2]. The main reasons are that we propose techniques that work for heterogeneous developed software, as discussed in Sect. 1.2, and our users have an heterogeneous performance expertise, that might not be suitable to deal with the complexity of pure performance models based approaches. The main limitations of the state of the art relevant to our proposal are mainly related to the advanced skills required by the users to use the proposed techniques and the limited expressiveness (e.g., [49]), the implementation of a reduced set of statistical modeling techniques by the different proposed solutions, and the limited support for automatically analysing the SUT behaviour to determine whether this is stable enough to apply the proposed techniques.

2.3 Fast Performance Feedback

*Provide **fast performance feedback** to the users, so that they can continuously gain performance knowledge about the SUT and use this knowledge to improve the same.*

This challenge requires automatic codification and collection of the generated performance knowledge in the CSI lifecycle, as well as definition of the conditions in which the collected knowledge can be reused for speeding up the performance testing and analyses activities. This also requires determining the conditions for identifying suitable unit, functional, integration, end-to-end, and performance tests for detecting performance regressions over time as well as for cross-branch/version performance evaluation [52], or to be reused to respond to different users' objectives.

In the software engineering literature, test results reuse techniques is often applied to functional tests [51]. Functional test results are suitable to be reused because are deterministic (if well designed). This helps to reduce number of executed performance tests across different builds of the same project [51]. For what concerns reuse of performance test results, the main limitation is the non deterministic behaviour of the same, and the fact their results are influenced by the hardware on which they are executed. For enabling reuse, performance tests results need to be reproducible [38, 18], consistent and validated. These results can be obtained within dedicated testing environments or using a shared testbed. The former allows to compare new results with previously obtained ones. For this purpose, current best practices rely on clusters of similar machines to parallelise and speed up performance activities [34]. To guarantee performance stability such machines are rarely virtualized. Shared testbeds may introduce noise due to known possible interferences of co-located workloads on the same machine [26]. Results are thus more challenging to reuse but are still valuable as these environments often closely resemble production environments, where virtualization is commonly employed to increase hardware utilization.

Reuse techniques for performance test results rely on performance models [49] and some of them work also cross-platform. They are applied in contexts where the developed software is stable in functionality over time, e.g. in Software Product Lines [35], or by reusing unit/functional tests results augmented with performance instrumentation [22] to speed up the performance testing activities.

3. HOLISTIC CONTINUOUS SOFTWARE PERFORMANCE ASSESSMENT

In this section we present our vision of an holistic approach for continuous software performance assessment. Fig. 1 provides an overview on the different solutions we propose in this section and how they integrate with the activities of the CSI lifecycle relevant for the discussed solutions.

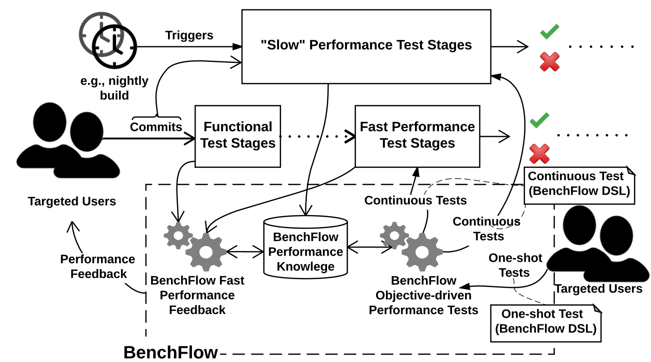

Figure 1: Holistic Continuous Software Performance Assessment with BenchFlow

3.1 A DSL for Declarative Continuous Performance Assessment

BenchFlow provides a declarative DSL to define performance tests and performance analyses activities, and integrate them in the CSI lifecycle, mainly in the continuous integration and continuous testing activities of the same. The BenchFlow DSL adopts the *YAML* syntax because it provides a concise notation, intended to be readable both by humans and by machines. BenchFlow DSL shares many concepts with the performance testing related DSLs, as discussed in Sect. 2.

Additionally, BenchFlow DSL allows users to declaratively express objective-driven performance tests. Users define **objective-driven performance tests** by relying on templates provided by BenchFlow for expressing tests' requirements such as the test objectives and test types, metrics of interest, stop conditions (e.g., maximum test execution time) and which parameters to vary during the execution of the test. Listing 1 shows an example on how the BenchFlow DSL enables the definition of a particular kind of objective, related to the exploration of the performance of the SUT when varying different parameters (i.e., Configuration testing [32]). The example shows how a user can define a configuration test simulating from 100 to 1000 interacting users with increasing steps of 100 simulated users during the exploration, varying the number of CPUs from 1 to 4,

the *SIZE_OF_THREADPOOL* SUT configuration variable from 5 to 100 with steps of 5, and with a fixed amount of 5GB of RAM. After defining the parameters, the user can indicate which parameters he/she wants to explore and optionally on which services realizing the SUT, as well as defining the stop conditions.

```
[...]

objective:
  type: exploration

  parameters:
  - users:
      range:
          100...1000
      step: +100
  - memory: 5GB
  - cpus:
      range: 1...4
  - SIZE_OF_THREADPOOL
      :
      range: 5...100
      step: +5
  - ...

  explore:
  - users
  - cpus on
      someService
  - SIZE_OF_THREADPOOL
      on
      someOtherService

  - ...

  stop:
  - max_exec_time = 1h
  - ...

[...]
```

Listing 1: Configuration testing with the BenchFlow DSL[1]

BenchFlow integration with Docker for managing the SUT configuration and lifecycle makes the DSL SUT-aware, which further simplifies the way users can specify performance tests. This feature is exposed to the users by relying on the Docker Compose standard, by which users can define the deployment descriptor of the SUT, and extended by BenchFlow to improve reusability of the deployment descriptors, and simplify the configuration management and SUT deployment (e.g., by providing the possibility to specify dependencies together with the deployment configuration declaratively for different test executions).

Templates written in the BenchFlow DSL provide reusable artifacts for defining performance tests for specific types of software. For example, when testing WfMSs middleware, different steps are required for deploying the necessary artifacts (e.g., workflow models [21]) before being able to deploy performance tests [14]. Also, specific performance data is collected by querying a database after the workload has been applied. The BenchFlow DSL can be used to describe how to automate the entire process, from the automated deployment of workflow models to the specification of performance data to collect and analyse.

To drive the integration of performance practices in the CSI lifecycle, the DSL natively supports the concepts of pipelines and automation steps. The BenchFlow DSL works together with existing CI/CD tools for generating the corresponding automated performance engineering activities that can be easily embedded within existing build pipelines.

3.2 Objective-Driven Hybrid Performance Testing

Users dealing with system performance investigation, have different objectives [33] that are influenced, e.g., by their role in the CSI lifecycle, and the type of the SUT. Objective-driven performance testing allows the users to express the objective of their performance investigations, and rely on BenchFlow for obtaining an actionable answer.

[1]The listing only report the section of the DSL definition related to the objective, omitting other configurations related for example to the load function.

Objectives' Taxonomy and Characteristics - We defined a taxonomy to categorise different kinds of objectives, based on three different levels of abstractions: base objectives, objectives and meta objectives. Base objectives correspond to well known performance tests [32], e.g., stress test or configuration test. Objectives refer to specific types of performance engineering activities, e.g., capacity planning and performance optimisations. These are different from the base objectives, because they require a more in depth performance analysis to be carried out in the presence of constraints. Objectives are based on base objectives, e.g., to investigate performance optimisation when specific loads (defined by base objectives) are applied to the system. Meta objectives instead rely on performance knowledge collected through objectives and base objectives, and enable the users to express more open-ended questions to BenchFlow, as for example comparing the performance of different systems using a benchmark.

Objective-driven performance queries can be executed once or continuously evaluated. One-shot objectives are executed following explicit user requests, while continuous objectives are tightly integrated with the automated CSI lifecycle. An example of one-shot objective is configuration testing, where a user requests to explore different configuration options to determine the most suitable one related to the given performance objectives. This is an activity usually done once in pre-release. Regression detection is an example of continuous objective, because it is an objective tightly integrated with the evolution of the SUT and should be automatically performed at every change.

Answering Objective-driven performance testing queries requires an expert system implementing the strategies to answer to them based on one or more performance models used to explore the space described by the objective settings provided by the user. To ensure the validity of the obtained empirical results, BenchFlow evaluates if the system complies with the assumptions implied by the corresponding objective. For example, stability: requests are accepted and no errors are produced, and the system can reach the steady state [7]. We plan to implement different exploration strategies in BenchFlow, and study how to select the most suitable given the objective and the SUT characteristics. While BenchFlow attempts to automatically select the most suitable strategy, expert users, who need complete control on the objective-driven performance test, may override or constrain the strategy selection.

Different performance engineering activities might have different velocity requirements. Performance tests applied at every build of the SUT, that usually happens after every code related commit, need to provide feedback to the users about the performance of the system, that is useful to define the next steps and committed code. Other performance tests, e.g., executed on nightly builds, usually can last an entire night, because the feedback they provide is not used in the immediate next coding steps by the developers, but further on for analysing the performance of the SUT. The selection of strategies must be driven by those velocity requirements. For example some exploration strategies might work better/faster with a small number of services composing the software [37], or depending on the combination of parameters to monitor, some solutions are more efficient/-faster than the others [45].

3.3 Model-based Performance Knowledge Reuse and Prediction

To ensure that the proposed techniques add value without slowing down the lifecycle's velocity too much, we propose to accumulate and reuse knowledge across the entire CSI lifecycle. This way, empirical evidence obtained in previous iterations can possibly save time in future iterations as it may enable BenchFlow to avoid repeating experiments with already known results.

Agile practices propose to develop projects relying on one or more project repositories, and different branches to parallelize the work of different teams of developers, that is then integrated before the release [52]. In this context there are many possibilities for accumulating and reusing performance knowledge, for speeding up the execution of the single performance test, but also for deciding to skip some more sophisticated (and long) performance tests, if simple ones fail, e.g., because they do not reach predefined criteria [40]. To do so, it is important to evaluate and select the testbed resources where to execute the various performance tests to ensure the reliability and repeatability of accumulated results.

It is imperative that the proposed techniques add value to the users without significantly slowing down the lifecycle's speed. Thus, we propose different ways to enable fast performance feedback to the users. BenchFlow automatically detects whether or not the SUT is ready to be subjected to performance testing, given the objective specified by the user (e.g., because of its instability in reaching the steady state). Thus time is not invested in producing invalid performance results. Additionally, BenchFlow can determine that certain data it collects and metrics computations it performs are not needed by the stated objectives and can be avoided.

For what concerns the reuse of performance knowledge when integrated in the CSI lifecycle, we enable the possibility to reuse unit/functional/integration tests, usually executed on every commit with code changes, to provide an initial characterisation of the SUT performance behaviour, that might be sufficient to determine the system it is not in a state for which it makes sense to proceed to more advanced performance tests. We rely on techniques introduced in the literature to enable the possibility to use these tests to characterise the performance behaviour, in particular for determining the set of suitable tests to be used, for guaranteeing the reliability of collected knowledge [22], and for computing platform independent metrics (e.g., number of calls from a service to another one, number of database calls, number of exception, static source code metrics, resource utilization efficiency, absolute RAM usage). We further propose to allow users to state whether a change in performance is expected.

Objective-driven testing enables us to understand the test requirements, and to determine the suitability of reusing collected performance knowledge across consecutive executions of the same tests, across executions of different tests, and tests executed in different moments in the lifecycle. Moreover the reuse can happen when the tests are executed on the same code repository branches, or across different branches, and if the system is composed of many in-house developed or third-party services that are deployed as part of the SUT, the collected knowledge can also be reused for determining the performance of the overall software by relying on the performance knowledge of the single services [28, 12]. When executing the same test multiple times, we can verify whether performance behaviour has changed by updating the performance model corresponding to the test. In doing so, we plan to optimise the way we select experiments to reduce the time needed to detect if the performance behaviour has changed, and a complete test is required, or we can reuse accumulated knowledge. Performance models are also used to determine the reuse possibility across different tests, belonging to different objectives. As an example, if we explore the performance of a SUT in specific conditions (e.g., a given configuration), we can reuse the results in other tests exploring again the same conditions.

The main techniques we plan to use are experiment based statistical modeling and prediction techniques (e.g., Kriging, MARS [45]). Due to the velocity constraint for applying performance practices in the context, we plan to evaluate and rely on techniques that enable us to obtain stable and reproducible results [38, 18], in limited amount of time, selected according to the test objective. We combine and apply these techniques before the execution of a test (to determine whether the test should indeed be performed), during its execution (to monitor the SUT behavior and possibly abort the test), and after the execution (to validate the results and determine whether it is possible to reuse the accumulated knowledge).

4. CONCLUSION AND FUTURE WORK

In this position paper we propose an holistic approach to the integration of performance engineering practices in continuous software improvement lifecycles. The approach is holistic because it embraces the different needs, viewpoints and skills of users participating in the lifecycle, proposing a tool-enabled and automated solution where the users can specify performance tests relying on a declarative, SUT-aware DSL.

BenchFlow's DSL allows the definition of standard performance tests, as well as objective-driven performance tests, so that the users can express their performance knowledge objectives relying on templates and automatically obtain answers to those objectives. The integration of BenchFlow in the CSI lifecycle helps to collect performance knowledge on the SUT in different activities of the same. For example, experiments can be performed during continuous integration, nightly builds or continuous deployment. We propose to reuse the collected knowledge to speed up performance testing activities of future iterations of the lifecycle. To save time, BenchFlow can warn the users about the SUT not being ready for particular performance tests, or obtain answers to performance queries without re-executing time-consuming tests.

In the near future we plan to release the support for base objectives definition and automation in the BenchFlow tool. We then plan to start experimenting and evaluating techniques for fast performance feedback, while enabling more and more objectives and meta objectives over time. We also plan to experiment with the integration of BenchFlow with Travis, a widely used CI tool.

5. ACKNOWLEDGEMENTS

This work is funded by the "BenchFlow" project (DACH Grant Nr. 200021E-145062/1) project.

6. REFERENCES

[1] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *IST*, 51(6):957–976, 2009.

[2] T. Ahmad and D. Truscan. Automatic performance space exploration of web applications using genetic algorithms. In *SAC '16*, pages 795–800. ACM, 2016.

[3] ASQ. Continuous Improvement. http://asq.org/learn-about-quality/continuous-improvement/overview/overview.html.

[4] M. Bernardino, E. M. Rodrigues, and A. F. Zorzo. Performance testing modeling. In *SAC '16*, pages 1660–1665. ACM Press, 2016.

[5] M. Bernardino, A. F. Zorzo, and E. M. Rodrigues. Canopus: A Domain-Specific Language for Modeling Performance Testing. In *ICSEA*, pages 157–167, 2014.

[6] Blazemeter. Taurus: Automation-friendly framework for Continuous Testing. http://gettaurus.org.

[7] A. B. Bondi. *Foundations of Software and System Performance Engineering*. Process, Performance Modeling, Requirements, Testing, Scalability, and Practice. Addison-Wesley Professional, 2014.

[8] J. Bosch. *Continuous Software Engineering*. Springer, 2014.

[9] A. Brunnert, A. van Hoorn, F. Willnecker, et al. Performance-oriented DevOps: A Research Agenda. 2015.

[10] S. Di Alesio, S. Nejati, L. C. Briand, et al. Stress testing of task deadlines - A constraint programming approach. *ISSRE*, 2013.

[11] DZONE. State of DevOps Report. Technical report, 2016.

[12] A. Faisal, D. Petriu, and M. Woodside. A Systematic Approach for Composing General Middleware Completions to Performance Models. In *Fundamental Approaches to Software Engineering*, pages 30–44. Springer, 2014.

[13] V. Ferme, A. Ivanchikj, and C. Pautasso. A Framework for Benchmarking BPMN 2.0 Workflow Management Systems. *BPM*, 9253(Chapter 18):251–259, 2015.

[14] V. Ferme, A. Ivanchikj, C. Pautasso, et al. A Container-centric Methodology for Benchmarking Workflow Management Systems. *CLOSER*, 2:74–84, 2016.

[15] B. Fitzgerald and K.-J. Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 2017.

[16] M. Fowler. *Domain-Specific Languages*. Pearson, 2010.

[17] R. Gao, Z. M. Jiang, C. Barna, et al. A Framework to Evaluate the Effectiveness of Different Load Testing Analysis Techniques. In *ICST*, pages 22–32. IEEE, 2016.

[18] I. P. Gent. The Recomputation Manifesto. 2013.

[19] M. Gualtieri and G. ODonnell. Augment DevOps With NoOps, 2016.

[20] M. Hauck, M. Kuperberg, N. Huber, et al. Deriving performance-relevant infrastructure properties through model-based experiments with Ginpex. *Softw Syst Model*, 13(4):1345–1365, 2013.

[21] D. Hollingsworth. The Workflow Reference Model. Technical report, 1995.

[22] V. Horký, F. Haas, J. Kotrč, et al. Performance Regression Unit Testing: A Case Study. In *Relationship of DevOps to Agile, Lean and Continuous Deployment*, pages 149–163. Springer, 2013.

[23] J. Humble and D. Farley. *Continuous Delivery*. Reliable Software Releases through Build, Test, and Deployment Automation. Pearson, 2010.

[24] R. Jabbari, N. bin Ali, K. Petersen, et al. What is DevOps?: A Systematic Mapping Study on Definitions and Practices. In *XP '16 Workshops*, pages 12–21. ACM, 2016.

[25] T. Kalibera and P. Tůma. Precise Regression Benchmarking with Random Effects: Improving Mono Benchmark Results. In *Formal Methods and Stochastic Models for Performance Evaluation*, pages 63–77. Springer, 2006.

[26] Y. Koh, R. Knauerhase, P. Brett, et al. An Analysis of Performance Interference Effects in Virtual Environments. In *ISPASS*, pages 200–209. IEEE, 2007.

[27] R. Kolb, D. Ganesan, D. Muthig, et al. Goal-Oriented Performance Analysis of Reusable Software Components. *ICSR*, 4039(27):368–381, 2006.

[28] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, 2010.

[29] E. I. Laukkanen, J. Itkonen, and C. Lassenius. Problems, causes and solutions when adopting continuous delivery - A systematic literature review. *IST*, 82:55–79, 2017.

[30] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[31] T. M. Mitchell. Machine learning. *McGraw Hill series in computer science*, 1997.

[32] I. Molyneaux. *The Art of Application Performance Testing*. From Strategy to Tools. O'Reilly Media, Inc., 2014.

[33] PractiTest. State of Testing Report. Technical report, 2016.

[34] K.-T. Rehmann, C. Seo, D. Hwang, et al. Performance Monitoring in SAP HANA's Continuous Integration Process. *SIGMETRICS Perf. Eval. Rev.*, 43(4):43–52, 2016.

[35] S. Reis, A. Metzger, and K. Pohl. A reuse technique for performance testing of software product lines. 2006.

[36] D. Rudolph and G. Stitt. An interpolation-based approach to multi-parameter performance modeling for heterogeneous systems. *ASAP*, 2015-September:174–180, 2015.

[37] B. M. Rutherford and L. P. Swiler. Response Surface (Meta-model) Methods and Applications. In *ICPE'13*, 2006.

[38] G. K. Sandve, A. Nekrutenko, J. Taylor, et al. Ten Simple Rules for Reproducible Computational Research. *PLOS Computational Biology*, 9(10):e1003285, 2013.

[39] A. Sarkar, J. Guo, N. Siegmund, et al. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T). In *ASE*, pages 342–352. IEEE, 2015.

[40] G. Schermann, J. Cito, P. Leitner, et al. Towards quality gates in continuous delivery and deployment. In *ICPC*, pages 1–4. IEEE, 2016.

[41] K. Spafford and J. S. Vetter. Aspen - a domain specific language for performance modeling. *SC*, page 84, 2012.

[42] S. Spinner, G. Casale, F. Brosig, et al. Evaluating approaches to resource demand estimation. *Perf. Eval.*, 92:51–71, 2015.

[43] S. Tsakiltsidis, A. Miranskyy, and E. Mazzawi. On Automatic Detection of Performance Bugs. In *ISSREW*, pages 132–139. IEEE, 2016.

[44] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages. *SIGPLAN Not.*, 35(6):26–36, 2000.

[45] L. Van Gelder, P. Das, H. Janssen, and S. Roels. Comparative study of metamodelling techniques in building energy simulation: Guidelines for practitioners. *Simulation Modelling Practice and Theory*, 49:245–257, 2014.

[46] R. Vuduc, J. W. Demmel, and J. A. Bilmes. Statistical Models for Empirical Search-Based Performance Tuning. *IJHPCA*, 18(1):65–94, 2004.

[47] J. Walter, A. van Hoorn, H. Koziolek, et al. Asking "What"?, Automating the "How"? - The Vision of Declarative Performance Engineering. *ICPE*, pages 91–94, 2016.

[48] Y. Wang. *Automating experimentation with distributed systems using generative techniques*. PhD thesis, University of Colorado, 2006.

[49] D. Westermann. *Deriving Goal-oriented Performance Models by Systematic Experimentation*. PhD thesis, KIT Scientific Publishing, 2014.

[50] F. Wu, W. Weimer, M. Harman, et al. Deep Parameter Optimisation. In *GECCO*, pages 1375–1382. ACM, 2015.

[51] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, 2012.

[52] L. Zhu, L. Bass, and G. Champlin-Scharff. DevOps and Its Practices. *IEEE Softw.*, 33(3):32–34, 2016.