Towards Automating Representative Load Testing in Continuous Software Engineering

Henning Schulz
NovaTec Consulting GmbH
Karlsruhe, Germany

Tobias Angerstein
NovaTec Consulting GmbH
Leinfelden-Echterdingen, Germany

André van Hoorn
University of Stuttgart
Stuttgart, Germany

ABSTRACT

As an application's performance can significantly impact the user satisfaction and, consequently, the business success, companies need to test performance before delivery. Though load testing allows for testing the performance under representative load by simulating user behavior, it typically entails high maintenance and execution overhead, hindering application in practice. With regard to the trend of continuous software engineering with its parallel and frequently executed delivery pipelines, load testing is even harder to be applied.

In this paper, we present our vision of automated, context-specific and low-overhead load testing in continuous software engineering. First, we strive for reducing the maintenance overhead by evolving manual adjustments to generated workload models over a changing environment. Early evaluation results show a seamless evolution over changing user behavior. Building on this, we intend to significantly reduce the execution time and required resources by introducing online-generated load tests that precisely address the relevant context and services under test. Finally, we investigate minimizing the amount of components to be deployed by utilizing load-test-capable performance stubs.

ACM Reference Format:
Henning Schulz, Tobias Angerstein, and André van Hoorn. 2018. Towards Automating Representative Load Testing in Continuous Software Engineering. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3185768.3186288

1 INTRODUCTION

Software performance as one dimension of quality of service (QoS) is a crucial attribute of today's enterprise applications. As an example, Amazon found they lose 1 % of sales per 100 ms delay [7]. To prevent such a business loss, companies aim at testing their applications' performance before delivery. As Jin et al. [5] show, considering the correct workload when measuring performance is indispensable. Surveying 109 real-world performance bugs, they are able to ascribe 42 of the bugs to wrong workload assumptions. This finding motivates the need for performance tests simulating representative user behavior as it would be in a production environment, commonly known as load tests.

Disregarding its importance, practice showed that load testing often is not applied. Chen et al. [2] detect this circumstance to essentially originate from high complexity and overhead during the whole load testing lifecycle. For instance, finding representative workloads is challenging as well as maintaining them over evolving applications and users' behavior. Existing approaches address these challenges by generating workload models from production request logs [2, 6, 14] but still require manual changes to the finally generated load tests. Hence, maintaining the load tests still entails manual overhead. Furthermore, the authors identify long execution times of load tests to hinder its application.

Within the last years, industry strengthened the named challenges even more by introducing the new paradigm of continuous software engineering, including DevOps, microservice architectures, continuous integration, and continuous delivery [1, 9]. Services are now small and self-contained, and are developed and delivered in individual, automated pipelines. In addition, the release cycles are shortened, down to less than a day. Consequently, the time and resources available for load testing tend to be even less while now several load tests for several pipelines are needed and automation is added as a new requirement. With this context of continuous software engineering, representative load testing can only cope by being short-running, resource-saving and automatable, and respect the service self-containment.

With our approach, we want to enable the newly required features of load testing. We build on existing approaches to automated workload model generation and test execution and propose new contributions in the following fields. (1) As a basis, we admit a need for manual adjustments of generated load tests and propose an approach to minimizing them by evolving them over changes of the users' behavior and the application itself. (2) Corresponding to self-contained microservices, we target modularized load tests that can be executed against one or few services. (3) In addition to that, we plan to enrich the load tests with contextual information to restrict the load test scenario to the respectively relevant context. In combination with (2), we assume this approach to significantly reduce the test execution time. (4) In order to finally minimize the needed resources as well, we want to introduce performance stubs that can be used as replacements of dependent services during a load test.

Summing up, this paper presents our vision of load testing in continuous software engineering and provides insights into first evaluation results of our load test adjustment evolution approach.

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5629-9/18/04…$15.00
https://doi.org/10.1145/3185768.3186288

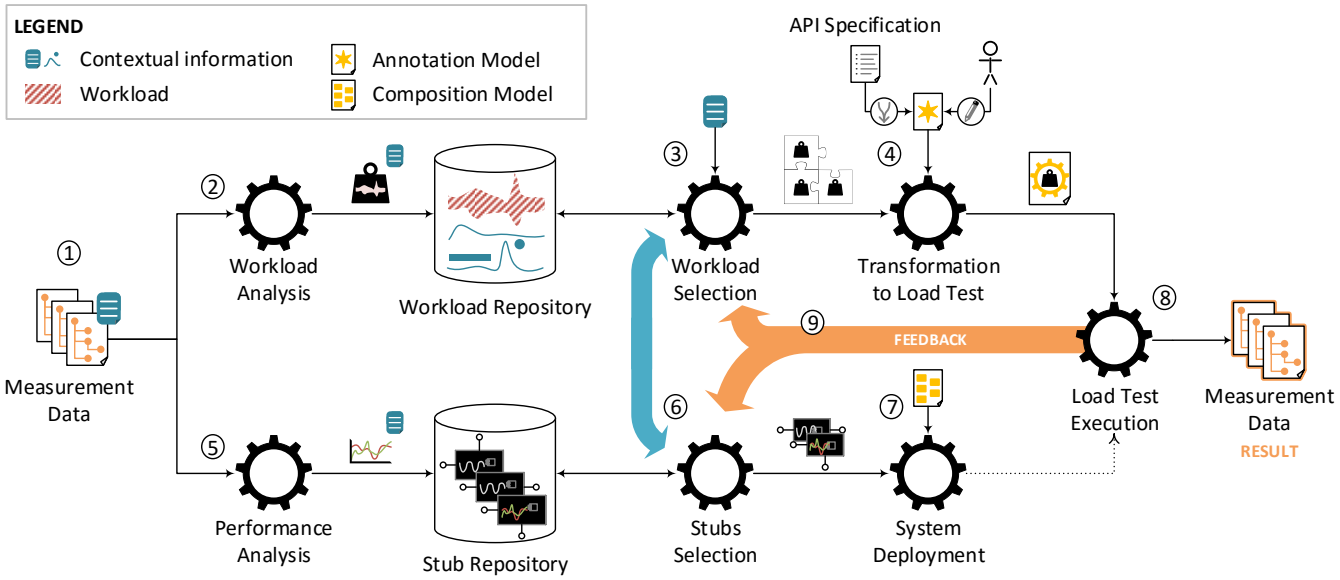

Figure 1: Visionary load testing process from measurement data recorded in production to load test results.

2 PROPOSED APPROACH

With our approach, we want to reform load testing with respect to continuous software engineering. Conceptually, we are going to replace fixed, human-defined load test scripts by time series of generated workload models from which context-relevant portions can be selected on demand. By furthermore restricting the tests to the services that are actually to be tested, we expect to minimize the test execution time and the amount of deployed services.

Figure 1 illustrates our new load testing process. The process starts with production monitoring data ① consisting of request logs, traces, and response times of the service interfaces. In addition, for later test selection, the data are enriched by various contextual information, e.g., marketing campaigns, public holidays, or sports events. In a workload analysis ②, the request logs are transformed into workload models by utilizing an existing approach like WESSBAS [14] and stored in a workload repository along with the contextual information. Continuously streaming measurement data to the workload analysis yields a time series of workloads and contexts.

Once a load test is to be executed, a context description serves as input and trigger ③. This description can hold contexts as recorded with the monitoring data, the services to be tested and, additionally, information about the requirements to the test, e.g., the available time for testing, the required confidence in the test results or the acceptable costs when testing in a cloud. Based on this description, we can automatically select the relevant sections from the workload time series in the workload repository and merge them into one or several workload models. In addition, we plan to restrict the workload models to the specified services by considering the initially collected traces. That is, if a backend service is to be tested, we compute the workload on the backend service resulting from the selected workload models and run the load directly to the backend without deploying any frontend services.

In the next step, the selected workload models are transformed into an executable load test ④. Using the existing approaches, this transformation step typically needs manual intervention for making the load tests actually executable. For instance, operations like ID correlation and input data specification are to be applied [14]. As we generate the load tests on demand, a user would have to do this intervention before every test execution. To overcome this drawback, we admit some adjustments to be necessarily done manually but evolve them over environment changes. For this purpose, we store the adjustments as an annotation in a separate, tool-independent model that only depends on the application, respectively the application programming interface (API). Hence, the annotation model can be applied to all different workload models for the same API. For adapting to changing APIs, we can utilize commonly used API specifications [9] to detect the changes and, if possible, adapt the annotation automatically or notify an application expert.

In parallel to the load test generation process, we plan to introduce a process for generating performance stubs. In doing so, we want to minimize the amount of services deployed for the load test and hence, save resources and costs. Resource saving is especially crucial in continuous software engineering, since several pipelines may run load tests in parallel. Simultaneously to the workload analysis, a performance analysis ⑤ utilizes the collected response times to calculate the performance behavior of the individual services. Here, we plan to base on Wert et al. [15] who derive such performance behavior from tests. Building on existing functional stubs, we want to merge the performance behavior and store the resulting performance stubs in another repository.

Corresponding to the workload selection, the performance stubs that replace the services which the tested services are dependent from are to be selected ⑥ and deployed together with the tested services ⑦. The knowledge about the services' deployment comes from a composition model, e.g., a docker-compose file. Since the

deployment step as well as the subsequent execution of the generated load tests ⑧ is already covered by existing work [4], we do not plan to investigate these steps.

A final step in our process will be a feedback of the test results to the workload and stub selection ⑨. For instance, if a performance regression could be detected, the impact on the user experience might be unclear, if only backend services were tested. In such a case, we can use the collected traces for determining the services calling the tested ones and, hence, the possibly impacted user interfaces. In a new load test, we then can test the services providing the user interfaces as well.

For integration into continuous integration and delivery pipelines, all manual steps in our process can be done offline. Specifying a template of the context description holding the services to be tested as well as the requirements, contextual information can be dynamically added from marketing databases, calendars etc. The annotation model can completely be defined in advance. In case of API changes, they can be detected before committing them to the code repository and thus, the programmer can be asked to instantly adapt the annotation.

3 EVALUATION

As illustrated at ④ in Figure 1, evolving manual adjustments to generated load tests is a fundament of our vision. For this reason, we provide early evaluation results of this approach. We show that we can specify the adjustments in an annotation once and apply them to changing load tests representing different usage behaviors without manual intervention. Due to space constraints, we only describe the most relevant results, but provide details online[1].

3.1 Experiment Setup

Our experiment is set up as follows. For a sample web shop, we define the required load test annotation as well as four different reference load tests representing the changing usage behavior. The annotation specifies the input values to the request parameters. Some values are specified directly while others have to be extracted from former requests by applying regular expressions.

In several iterations, we repeat the following steps. First, one of the reference load tests is randomly selected and executed. The resulting measurement data are used as input to WESSBAS [14] which generates a workload model. Next, the workload model is transformed to a JMeter[2] test plan, taking the annotation into account. After executing the test plan, we compare the measurement results to detect differences. Since the annotation is to be used to make load tests executable, the number of errors is the main metric of interest. In addition, we compare the CPU utilization as an indicator for the similarity of the load tests.

3.2 Experiment Results

In Figure 2, the errors per minute as well as the CPU utilization for each executed load test are shown. The reference load tests 1, 2, and 3 did not introduce any errors. Similarly, the corresponding generated load tests introduced zero errors except for the beginning of each test. Since the generated load tests implement a Markov

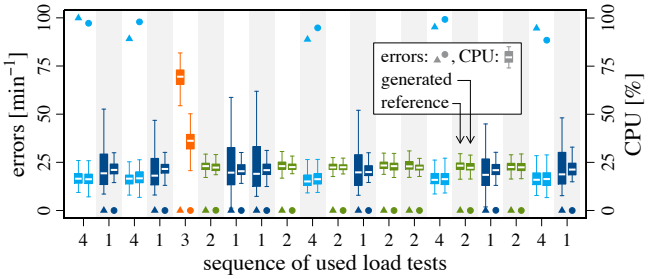

Figure 2: Errors per minute and CPU utilization for each used reference and generated load test.

chain, the order of the requests is random to some degree. For this reason, requests whose input data have to be retrieved from regular expression extractions fail if the response where the data are extracted from did not arrive yet. From testing, we know that load tests generated without the annotation continuously cause errors due to the missing regular expression extraction. Hence, we conclude that the load tests were correctly annotated.

One of the requests of load test 4 was intentionally configured with a wrong path and failed. In our experiment, the error rates of the reference test were between 88.8 and 99.9 per minute. The generated test also introduced the same error with rates between 88.4 and 99.2 per minute. Furthermore, the error rates correspond to the execution rates of the broken request in both cases. Consequently, for this load test, the annotation was correctly applied as well.

Considering the CPU utilization, the workloads executed by the reference load tests 2 and 4, and the corresponding generated tests appear not to differ significantly. The request counts and response times measured during the tests substantiate this hypothesis. However, there are apparent differences for the reference tests 1 and especially for test 3. During these tests, we observed overload effects. For instance, in some cases, the response times increased to up to 52 seconds while the minimum response time of the same request in the test was 66 milliseconds. For this reason, we assume that the workload generation did not properly work due to too high load, e.g., because the monitored requests differed from the ones JMeter executed. Regardless, our annotation approach does neither affect the intensity nor the order of requests. Hence, it is unlikely that the difference in the workloads is caused by our approach.

4 RELATED WORK

In regard to automated load testing, several approaches have been proposed in literature. Chen et al. [2] not only highlight common challenges to load testing but also provide a set of solutions to the majority of them, including automated generation of load tests from request logs. However, the authors do not explicitly respect continuous software engineering. For instance, they claim to execute the generation after several months to check if they have to adjust the original test. This manually attended procedure is not ready for dynamically generated load tests.

[1] https://continuity-project.atlassian.net/wiki/x/AQC2B
[2] http://jmeter.apache.org/

In addition to the work by Chen et al., further workload model generation approaches have been proposed. Some of them are based on Markov chains [8, 10] while others try to overcome the drawbacks of Markov chains by using sequences of requests [6], extended finite state machines [11], or adding guards and actions [14]. In the end, all these approaches enable automated load test generation but require manual adjustments like input data specification or ID correlation. With our approach, we want to minimize these adjustments in cases new load tests are generated. Furthermore, we want to add functionality to restrict the load tests to individual services.

Concerning the problem of manual adjustments, there has only been work that derives the difference of test and production workloads [13]. This approach can be used to determine if a new load test generation is worthwhile, but does not facilitate the new generation as such. Thus, it is not advantageous for online generation of load tests.

To the best of our knowledge, selecting load tests based on the context is a new approach. However, such approaches have been investigated for functional tests. As an example, Srivastava et al. [12] prioritize functional tests based on the introduced changes. We assume to need different optimization objectives for load tests but plan to build on the existing work if possible.

Finally, there is related work in the field of performance stubs. The field is already investigated for applications in an early development phase [3]. We want to use stubs also in late-phase load testing, when the performance behavior can already be determined from production monitoring. Determination of the performance behavior could be realized by building on the work by Wert et al. [15] who use systematic testing for deriving the behavior. By adapting this approach to production monitoring and merging the resulting performance behavior with existing functional stubs, we want to achieve the required load-test-capable stubs.

5 CONCLUSIONS

While always being challenging, the classic approach of load testing turns out not to be applicable in continuous software engineering. For instance, the maintenance and execution overhead is high and contradicts automated, parallel delivery pipelines generating highly frequent releases. For this reason, the load testing process has to be refined.

In this paper, we presented our vision of load testing in continuous software engineering as well as first evaluation results. As a basis, we presented an approach to minimizing the manual overhead for defining manual adjustments that are required to make generated load tests executable. By storing the adjustments in an annotation model only dependent from the application's API, we strive for evolving the adjustments over a changing environment without manual intervention. In a first evaluation, we showed that our annotation approach is suitable of evolving once created annotations over newly generated load tests if only the user behavior changes. Even if we could detect significant differences between reference and generated load tests, we assume they do not emanate from our approach. However, in future evaluations, we have to substantiate this hypothesis.

Building on the evolution approach, we are going to investigate the other ideas described in this paper. First, we address modularization of load tests in service granularity and allow for executing the tests directly on the service to be tested. Then, we investigate approaches for selecting a load test for a specific context, e.g., the target services, environmental circumstances like marketing campaigns or public events and requirements to the test. For integration into a continuous delivery pipeline, such a requirement could be a limit for the time available for testing as well as the costs when testing in a cloud. Finally, we address performance stubs that are capable for load testing. With this approach, we want to reduce the set of deployed services to the ones to be tested, while stubbing remaining services.

Altogether, we aim to reform the process of load testing, making it more flexible, automatable and less time- and resource-consuming and thus, applicable in continuous software engineering.

ACKNOWLEDGMENTS

This work is being supported by the German Federal Ministry of Education and Research (grant no. 01IS17010, ContinuITy).

The authors have benefited from discussions with our colleagues from the ContinuITy project, including Christoph Heger, Stefan Siegl, Alexander Wert, Dusan Okanović, Vincenzo Ferme, and Alberto Avritzer.

REFERENCES

[1] L. J. Bass, I. M. Weber, and L. Zhu. *DevOps - A Software Architect's Perspective*. SEI series in software engineering. Addison-Wesley, 2015.
[2] T.-H. Chen, M. D. Syer, W. Shang, Z. M. Jiang, A. E. Hassan, M. N. Nasser, and P. Flora. Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. *ICSE-SEIP 2017*.
[3] G. Denaro, A. Polini, and W. Emmerich. Early performance testing of distributed software applications. *ACM SIGSOFT Software Engineering Notes*, 29(1):94–103, 2004.
[4] V. Ferme and C. Pautasso. Towards holistic continuous software performance assessment. *ICPE 2017, Companion Proceedings*.
[5] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *PLDI 2012*.
[6] D. Krishnamurthy, J. A. Rolia, and S. Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Transactions on Software Engineering*, 32(11):868–882, 2006.
[7] G. Linden. Make data useful. https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt, 2006.
[8] D. A. Menascé. Load testing of web sites. *IEEE Internet Computing*, 6(4):70–74, 2002.
[9] S. Newman. *Building Microservices – Designing Fine-Grained Systems*. O'Reilly, 1st edition, 2015.
[10] G. Ruffo, R. Schifanella, M. Sereno, and R. Politi. Walty: A user behavior tailored tool for evaluating web application performance. *NCA 2004*.
[11] M. Shams, D. Krishnamurthy, and B. H. Far. A model-based approach for testing the performance of web applications. *SOQUA 2006*.
[12] A. Srivastava and J. Thiagarajan. Effectively priori- tizing tests in development environment. *ISSTA 2002*.
[13] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. N. Nasser, and P. Flora. Continuous validation of load test suites. *ICPE 2014*.
[14] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, and H. Krcmar. Wessbas: Extraction of probabilistic workload specifications for load testing and performance prediction – a model-driven approach for session-based application systems. *Software & Systems Modeling*, pages 1–35, 2016.
[15] A. Wert, J. Happe, and D. Westermann. Integrating software performance curves with the palladio component model. *ICPE 2012*.