Including Performance Benchmarks into Continuous Integration to Enable DevOps

Jan Waller, Nils C. Ehmke, and Wilhelm Hasselbring
Software Engineering Group, Kiel University, 24098 Kiel, Germany
{jwa,nie,wha}@informatik.uni-kiel.de

ABSTRACT

The DevOps movement intends to improve communication, collaboration, and integration between software developers (Dev) and IT operations professionals (Ops). Automation of software quality assurance is key to DevOps success. We present how automated performance benchmarks may be included into continuous integration. As an example, we report on regression benchmarks for application monitoring frameworks and illustrate the inclusion of automated benchmarks into continuous integration setups.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – Testing tools. D.2.8 [**Software Engineering**]: Metrics – Performance measures. D.2.9 [**Software Engineering**]: Management – Software quality assurance (SQA).

General Terms

Measurement, Performance.

Keywords

Jenkins, Kieker, MooBench.

1. INTRODUCTION

Based upon our experience with regression benchmarks for application monitoring frameworks, we suggest to include automated benchmarks into continuous integration setups. Applying these automated benchmarks in an early stage of the development process enables an early detection and repair of performance issues before such issues are propagated into a release.

This approach contributes to the current efforts of the DevOps movement by presenting a case study of executing and analyzing regression benchmarks in continuous integration setups. Furthermore, we provide a vision for improved analyses and visualizations including automated notifications to the developers in charge.

The rest of this article is structured as follows: First, we introduce the DevOps movement (Section 2) and our case study setup (Section 3). In Section 4, a short overview on our present inclusion of benchmarks into continuous integration is given while our vision is sketched in Section 5. Finally, we provide a short summary and advice in Section 6.

2. DEVOPS: DEVELOPMENT + OPERATIONS

In addition to studying the construction and evolution of software, the software engineering discipline needs to address the *operation* of continuously running software services. Often, software development and IT operations are detached organizational units, with a high potential for misunderstanding and conflict. The *DevOps movement* intends to improve communication, collaboration, and integration between software developers (Dev) and IT operations professionals (Ops).

Automation is key to DevOps success: automated building of systems out of version management repositories; automated execution of unit tests, integration tests, and system tests; automated deployment in test and production environments. Besides functional acceptance tests, automated tests of non-functional quality attributes, such as performance, are required to ensure seamless operation of the software. In this article, we present how performance benchmarks may be included into continuous integration setups.

Continuous integration [3] and continuous delivery are enabling techniques for DevOps. A further important requirement for the robust operation of software services are means for *continuous monitoring* of software runtime behavior. In contrast to profiling for construction activities, monitoring of operational services should only impose a small performance overhead. The Kieker monitoring framework is an example of providing these means with a small performance overhead [5]. We report on how we include micro-benchmarks that measure the performance overhead of Kieker into the continuous integration process for this monitoring framework.

3. KIEKER WITH MOOBENCH

The *Kieker framework* [5] is an extensible framework for application-level performance monitoring of operational services and subsequent dynamic software analysis.[1] It includes measurement probes for the instrumentation of software systems and writers to facilitate the storage or further transport of collected observation data. Analysis plug-ins operate on the collected data to extract and visualize architectural models that can be augmented by quantitative observations.

In 2011, the Kieker framework was reviewed, accepted, and published as a recommended tool for quantitative system evaluation and analysis by the SPEC Research Group. Since then, the tool is also distributed as part of SPEC Research Group's tool repository.[2] Although is has originally been developed as a research tool, Kieker is used in several industrial systems.

A monitored software system has to share some of its resources (e. g., CPU-time or memory) with the monitoring framework. The amount of additional resources that are consumed by the monitoring framework is called *monitoring overhead*. In this article, we are concerned with monitoring overhead measured by the increasing response times of the monitored system.

[1] `http://kieker-monitoring.net/`
[2] `http://research.spec.org/projects/tools.html`

Benchmarks are used to compare different platforms, tools, or techniques in experiments. They define standardized measurements to provide repeatable, objective, and comparable results. In computer science, benchmarks are used to compare, for instance, the performance of CPUs, database management systems, or information retrieval algorithms [4].

The *MooBench micro-benchmark* has been developed to measure and compare the monitoring overhead of different monitoring frameworks [6]. We have identified three causes of application-level monitoring overhead that are common to most monitoring tools: (1) the instrumentation of the monitored system, (2) collecting data within the system, e. g., response times or method signatures, and (3) either writing the data into a monitoring log or transferring the data to an analysis system. With the help of our micro-benchmark, we can quantify these three causes of monitoring overhead and, for instance, detect performance regressions or steer performance tunings of the monitoring framework.

Thanks to irregularly performed manual benchmarks of the monitoring overhead of Kieker, we have detected several performance regressions after the releases of new versions over the last years. After their detection, these performance regressions have been transformed into tickets in our issue tracking system. This has enabled us to further investigate the regressions and to provide bug fixes for future releases.

The main challenge when patching performance regressions is identifying the code changes that have triggered the regression. With irregular manual benchmarks, lots of commits can contain the possible culprit. Ideally, our benchmark would have been executed automatically with each nightly build to provide immediate hints on performance problems with each change. Thus, the continuous integration of benchmarks provides the benefit of an immediate and automatic feedback to developers.

4. INCLUDING MOOBENCH INTO JENKINS

The *continuous integration setup* that is employed by Kieker is based upon Jenkins.[3]

As our initial approach, we developed a Jenkins plugin to execute the benchmark. This plugin directly executes MooBench from within Jenkins. However, this kind of execution violates the common benchmark requirement for an idle environment: Jenkins and its periodical tasks, as well as the running plugin itself, influence the benchmark results. This can be seen in Figure 1. Note that the changes in the response times of the purple graph are not caused by actual changes in the source code but rather by background tasks within Jenkins. Even remote execution with the help of a Jenkins master/slave setup, i. e., executing the benchmark within an otherwise idle Jenkins instance on a separate server, has only provided fluctuating results.

Finally, we have chosen a simpler approach: Instead of using complex plugins, we simply call a shell script at the end of each nightly build on Jenkins. This script copies the benchmark and the created Kieker nightly jar-file to an idle, pre-configured remote server (e. g., onto a cloud instance). There, the benchmark gets executed while Jenkins waits for the results. In addition to the usual analyses performed by MooBench, e. g., calculating mean and median with their confidence intervals and quartiles, we also create a comma-separated values (CSV) file with the mean measurement results. This file can be read and interpreted by a plot plugin within Jenkins. An example of such a generated plot based upon the Kieker nightly builds is presented in Figure 2.

As is evident by the display of the data in Figure 2, the plot plugin is rather limited. For instance, it is only capable of displaying the measured mean response times that still contain some variations. The display of additional statistical method, such as confidence intervals, would be beneficial to their interpretation.

In addition, we currently only display the gathered results rather than automatically notifying the developers when a performance anomaly occurs. The actual detection of the anomalies has to be performed manually. However, previous work on anomaly detection within Kieker results can be adapted for this scenario [1].

Finally, as is common with dynamic analysis approaches, the detection and visualization of performance regressions is only possible within benchmarked areas of Kieker. As a consequence, any performance regression caused by, for instance, unused probes or writers cannot be found. However, a more thorough benchmark requires a higher time cost (currently about 60 minutes per nightly build). Thus, a balance has to be found between benchmark coverage and time spent benchmarking.

Despite these remaining challenges in the current implementation, the inclusion of MooBench into the continuous integration setup of Kieker already provides great benefits. Any performance regressions are now detected immediately. Furthermore, the regressions can directly be linked to small sets of changes. Thus, diagnosis of performance problems is aided. The current and future implementations of our integration of benchmarks into Jenkins are available as open source software with MooBench.[4] Furthermore, the current state of our implementation is available with our continuous integration setup.[5]

5. EVALUATION AND VISION

In this section, we present our vision for improved analyses and visualizations of regression benchmarks within continuous integration setups. Of special interest are automated detections of performance regressions and corresponding notifications to the developers in charge. We demonstrate the capabilities of our envisioned approach with the help of our case study system by studying a previous performance regression.

Since our inclusion of MooBench into the continuous integration setup of Kieker, no additional major performance regressions have occurred. Instead of artificially creating an anomaly to demonstrate the capabilities of our setup, we have recreated earlier nightly builds and executed the benchmark as it would have been included. This post-mortem benchmarking also allows for an outlook on a more advanced visualization and anomaly detection than is currently realized within our Jenkins implementation.

Specifically, we have selected a performance regression that happened in March 2013 and that was detected in Kieker release version 1.7: An unintended increase of the first part of monitoring overhead (instrumentation) that was related to a bug in our implementation of adaptive monitoring. To further narrow down the cause of this regressions, we haven taken a look at the nightly builds between Kieker releases 1.6 and 1.7. For each build, we have run the MooBench benchmark in a configuration identical to the one used in our continuous integration setup. The resulting visualization of a few of the relevant benchmark results of the nightly builds is presented in Figure 3.

[3] `http://jenkins-ci.org/`

[4] `http://kieker-monitoring.net/MooBench/`

[5] `http://build.kieker-monitoring.net/job/kieker-nightly-release/plot/`

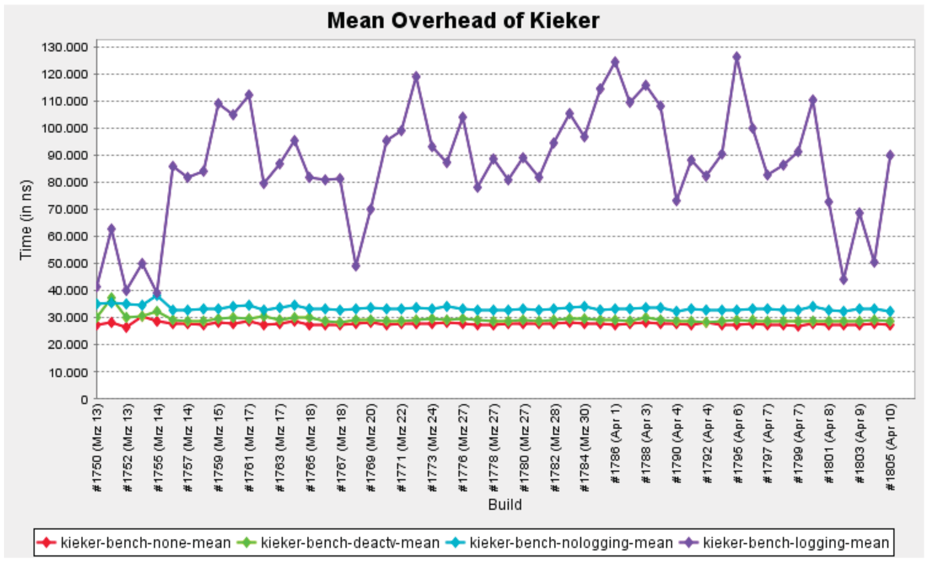


Figure 1: Initial inclusion of MooBench into Jenkins

In Figure 3, the mean benchmark results are depicted as stacked bars. Each bar is annotated to the right with its respective 95% confidence interval. The lowest bar is barely visible and represents the base time of the benchmark without any monitoring overhead. The other three bars correspond to the three causes of overhead (instrumentation, data collection, and writing). Our focus in this analysis is on the orange bar, representing the instrumentation. The actual anomaly is highlighted with a red ellipse.

The first four presented nightly builds are part of our analysis: two builds before and after the performance regression occurred. The final three builds demonstrate our bug fixing two and a half month after the performance regression. With the help of our presented vision of including benchmarks into continuous integration and performing automated anomaly detections on the results, e. g., similar to [1], the time to fix performance regressions can be reduced.

The necessity to automate the continuous execution of benchmarks has also been recognized in other contexts. For instance, the academical BEEN project is concerned with automated regression benchmarking [2]. An industrial approach for continuous benchmarking of web-based software systems is presented in [7].

6. SUMMARY AND ADVICE

Based on our experience with regression benchmarks, we suggest to include benchmarks into continuous integration early in the development process. In current practice, benchmark and monitoring instrumentation is often only integrated into the software when problems occur after release in the production environment of IT operations. Measurement instrumentation, i. e., benchmarks as well as monitoring, should be integrated into software development right from the start. Automated benchmarks are then able to detect performance issues that may have been introduced between versions, before these issues are propagated via the deployment pipeline. Monitoring then can aid in the early detection of further performance issues that only manifest in the actual deployment of the software.

References

[1] J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In *Proceedings of the 8th IEEE/ACM International Conference on Autonomic Computing (ICAC 2011)*, pages 197–200. ACM, June 2011.

[2] T. Kalibera, J. Lehotsky, D. Majda, B. Repcek, M. Tomcanyi, A. Tomecek, P. Tůma, and J. Urban. Automated benchmarking and analysis tool. In *Proceedings of the 1st International Conference on Performance Evaluation Methodologies and Tools (Valuetools 2006)*, pages 5–14. ACM, Oct. 2006.

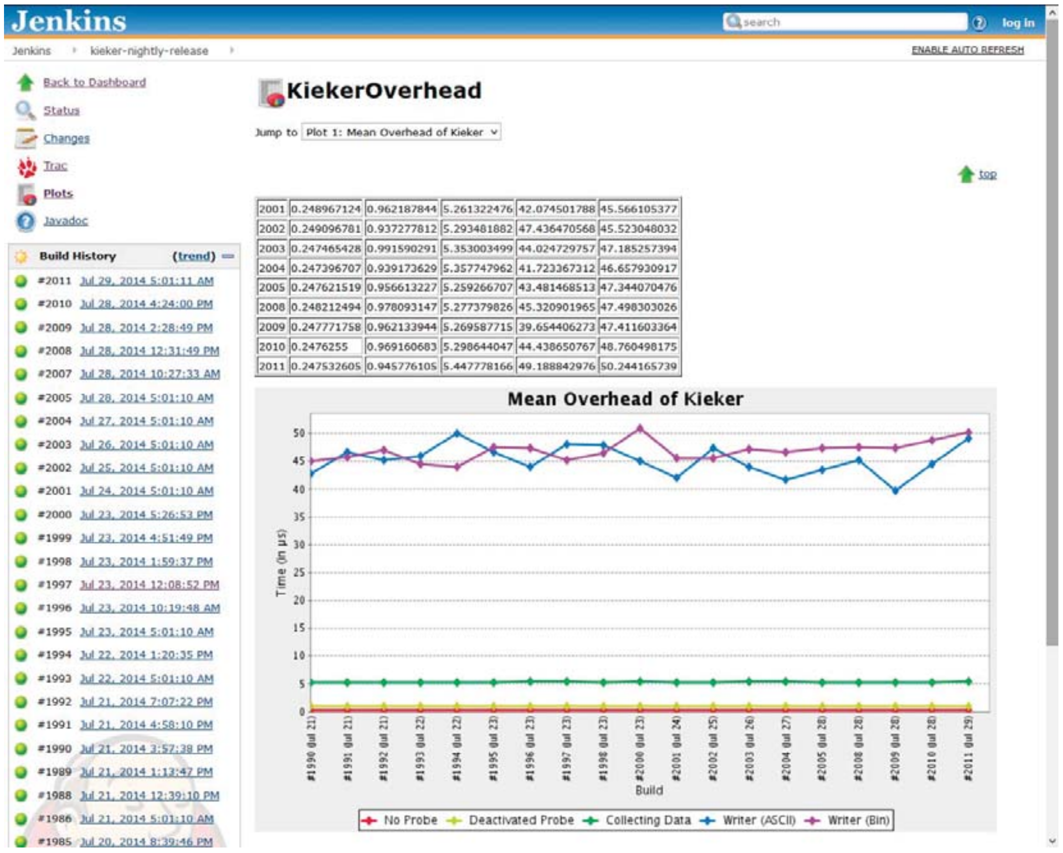
[3] M. Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, May 2014.

[4] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 74–83. IEEE Computer Society, May 2003.

[5] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, Apr. 2012.

[6] J. Waller and W. Hasselbring. A benchmark engineering methodology to measure the overhead of application-level monitoring. In *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*, pages 59–68. CEUR Workshop Proceedings, Nov. 2013.

[7] C. Weiss, D. Westermann, C. Heger, and M. Moser. Systematic performance evaluation based on tailored benchmark applications. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013)*, pages 411–420. ACM, Apr. 2013.



Build History (trend)

- #2011 Jul 29, 2014 5:01:11 AM
- #2010 Jul 28, 2014 4:24:00 PM
- #2009 Jul 28, 2014 2:28:49 PM
- #2008 Jul 28, 2014 12:31:49 PM
- #2007 Jul 28, 2014 10:27:33 AM
- #2005 Jul 28, 2014 5:01:10 AM
- #2004 Jul 27, 2014 5:01:10 AM
- #2003 Jul 26, 2014 5:01:10 AM
- #2002 Jul 25, 2014 5:01:10 AM
- #2001 Jul 24, 2014 5:01:10 AM
- #2000 Jul 23, 2014 5:26:53 PM
- #1999 Jul 23, 2014 4:51:49 PM
- #1998 Jul 23, 2014 1:59:37 PM
- #1997 Jul 23, 2014 12:08:52 PM
- #1996 Jul 23, 2014 10:19:48 AM
- #1995 Jul 23, 2014 5:01:10 AM
- #1994 Jul 22, 2014 1:20:35 PM
- #1993 Jul 22, 2014 5:01:10 AM
- #1992 Jul 21, 2014 7:07:22 PM
- #1991 Jul 21, 2014 4:58:10 PM
- #1990 Jul 21, 2014 3:57:38 PM
- #1989 Jul 21, 2014 1:13:47 PM
- #1988 Jul 21, 2014 12:39:10 PM
- #1986 Jul 21, 2014 5:01:10 AM
- #1985 Jul 20, 2014 8:39:46 PM

2001	0.248967124	0.962187844	5.261322476	42.074501788	45.566105377
2002	0.249096781	0.937277812	5.293481882	47.436470568	45.523048032
2003	0.247465428	0.991590291	5.353003499	44.024729757	47.185257394
2004	0.247396707	0.939173629	5.357747962	41.723367312	46.657930917
2005	0.247621519	0.956613227	5.259266707	43.481468513	47.344070476
2008	0.248212494	0.978093147	5.277379826	45.320901965	47.498303026
2009	0.247771758	0.962133944	5.269587715	39.654406273	47.411603364
2010	0.2476255	0.969160683	5.298644047	44.438650767	48.760498175
2011	0.247532605	0.945776105	5.447778166	49.188842976	50.244165739

Figure 2: Performance measurements within Jenkins

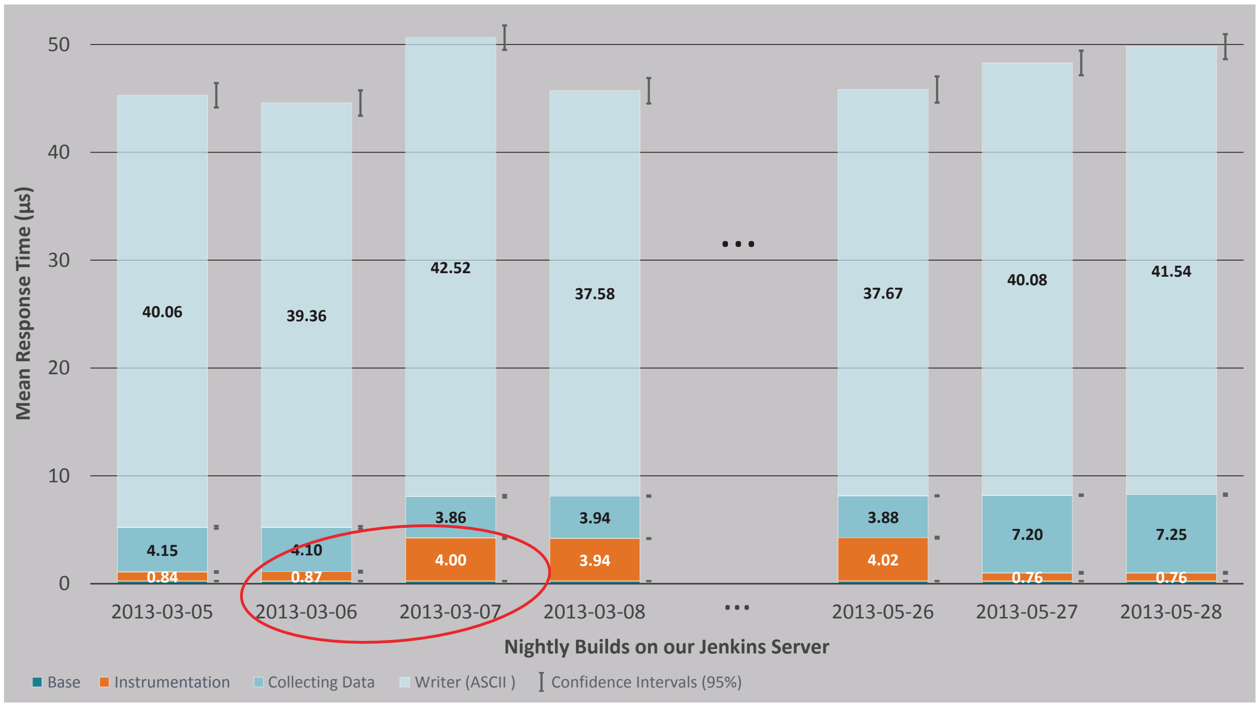


Figure 3: Scenario for detecting performance anomalies between releases via benchmarks in continuous integration