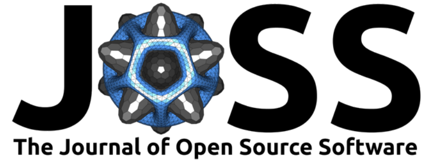


MooBench: A micro-benchmark for performance overhead measurement of observability tools

David Georg Reichelt [1,2¶], **Shinhyung Yang** [3], **Marcel Hansson** [4], and **Wilhelm Hasselbring** [3]

1 Lancaster University Leipzig **2** Universitätsrechenzentrum (URZ) Leipzig **3** Kiel University **4** University of Hamburg ¶ Corresponding author

DOI: 10.21105/joss.10400

Software

- Review
- Repository
- Archive

Editor: Jack Atkinson

Reviewers:

- @abhishektiwari
- @Ahanaf-Aziz
- @shrishar

Submitted: 02 April 2026
Published: 07 July 2026

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

In partnership with

JOSS Special Issue for RSECon26 Research Software Papers

Summary

Understanding the runtime behavior of software is inherently difficult due to the unpredictability of the software's behavior itself and the non-determinism of underlying layers, such as Just-In-Time (JIT) compilation in virtual machines, operating system scheduling, and CPU frequency scaling. Observability tools aim to answer questions regarding runtime behavior of software, such as "How much time did this request take?" or "How often did method A call method B?" (Majors et al., 2022). These questions are answered using telemetry data, i.e., measurement data that is obtained from the code execution. To collect telemetry data, additional code needs to be executed, which introduces overhead. This overhead affects both system performance and the accuracy of the measurements themselves. The MooBench microbenchmark measures this overhead and contains factorial experiments that facilitate breaking down this overhead into its root causes.

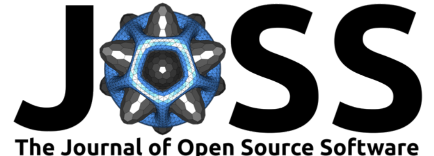
The MooBench benchmark was originally developed to examine the performance overhead of the Kieker observability framwork (Hasselbring & van Hoorn, 2020; Hoorn et al., 2012; Yang, Reichelt, Jung, et al., 2025) in Java (Waller & Hasselbring, 2013) and was extended as a general overhead measurement microbenchmark for various observability tools, currently within the Java and Python ecosystem. In this paper, we describe why it is needed, how it is structured, and how it is used in research.

Statement of need

Observability tools are crucial for managing software system health and optimizing operational costs; consequently, a wide variety of tools compete in the market (Siegfried et al., 2025). The relevance of distributed tracing tools increased with the rise of microservice architectures, where system behavior across microservices needs to be understood (Sigelman et al., 2010). While observability generally consists of the three pillars: traces, metrics, and logs, with distributed tracing especially gaining wide adoption (Janes et al., 2023). Concerning the categorization of research software (Hasselbring et al., 2025), MooBench is technology research software (Hasselbring et al., 2026): It is developed and used in research, but it is also intended for use in non-academic contexts for evaluating and selecting observability tools.

State of the field

In the context of microservices, various studies have examined the overhead of tracing. For example, Nõu et al. (2025) investigated the overhead of OpenTelemetry and Elastic APM using open-source applications, and examined the root causes of the overhead using profiling. Ahmed et al. (2016) compared the overhead of commercial observability tools and the open-source tool



Pinpoint for regression detection. Eder et al. (2023) compared the overhead using serverless applications.

Besides the microservice context, tracing overhead has been examined for operating systems and High-Performance Computing (HPC). For operating systems, the overhead of tools for tracing the kernel itself (Desnoyers & Dagenais, 2006; Gebai & Dagenais, 2018) and eBPF, that allows tracing within both the kernel and the user space (Domaschka et al., 2023; Volpert et al., 2025), have been examined in case studies. In the context of HPC, research has addressed the impact of overhead (Hunold et al., 2022) and the combination of instrumentation and sampling to reduce it (Ilsche et al., 2015).

The aforementioned studies utilize macrobenchmarks to examine overhead in realistic use cases. While macrobenchmarks provide an indication of overhead in similar scenarios, they often fail to isolate the baseline overhead or identify its specific sources. When looking at specific distributed cases, the overhead sources are influenced by the processes running in parallel, their scheduling, and the network behavior. MooBench addresses this by providing a microbenchmark that measures the fundamental overhead of observability frameworks in a controlled environment. Using different configurations, it enables factorial experiments that isolate root causes of overhead, specifically distinguishing between instrumentation, data collection, and data serialization.

Furthermore, macrobenchmarks are also time-consuming to execute. The MooBench microbenchmark was included early in the Kiekers CI setup (Waller et al., 2015), enabling the detection of performance regressions and the check of performance improvements in daily development.

Software design

MooBench consists of two parts: the minimal and configurable system under test, and the automation of benchmarking observability frameworks. Figure 1 gives an overview of the architecture. The following subsections describe the system under test, the automated setup of observability frameworks and the continuous integration process.

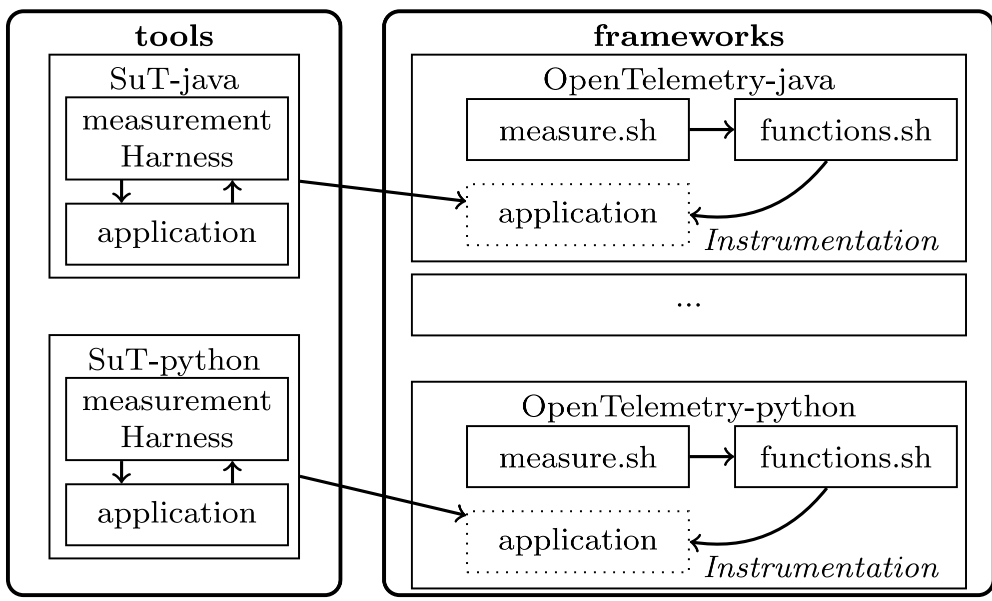
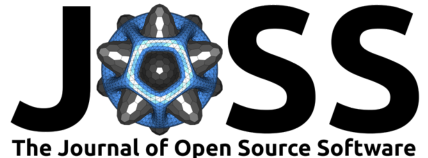


Figure 1: Architecture of MooBench.



System under test

The system under test consists of two parts: The measurement harness, and the application itself.

The measurement harness manages start and end of executions, and the storage of result data.

The structure of the application itself is built to represent the overhead of observability tools in the most simple way. The overhead of observability tools is caused by data collection. The amount of collected data – and therefore the overhead – scales with the number of observed method executions. However, the resource consumption of both, the application methods and the associated telemetry code, is non-deterministic and influenced by various factors, including class loading, Just-in-Time compilation (JIT), and garbage collection. To reduce the variance, the core of MooBench's SuT is a single method (`monitoredMethod`) that calls itself recursively `$RECURSION_DEPTH` times. Because the JIT compiler could potentially optimize away these recursive calls, the last method call contains a busy wait for `$METHOD_TIME` nanoseconds. Figure 2 visualizes the call tree.

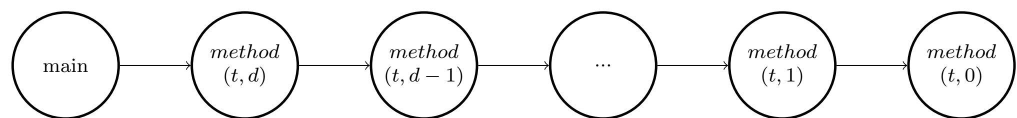


Figure 2: Call tree of the MooBench microbenchmark showing the recursive method calls to control the stack depth (Reichelt et al., 2023).

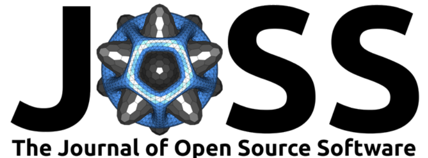
This system under test is currently implemented in Java and Python. It is planned to extend it to JavaScript and Go.

Automated setup of observability frameworks

Each observability framework requires a unique setup and has different capabilities. For example, OpenTelemetry can be attached to the JVM using `-javaagent`, which requires the property `otel.instrumentation.methods.include` to be set to a specific list of classes. OpenTelemetry can export data to Zipkin, Jaeger, or Prometheus. Furthermore, OpenTelemetry can be started with an empty `otel.instrumentation.methods.include` or without export, which enables measuring the overhead of instrumentation without data collection or data collection without serialization. Due to these different configurations, it is necessary to implement configuration scripts for each framework individually.

MooBench stores these configuration scripts in `frameworks/$TECHNOLOGY-$LANGUAGE` (e.g., `frameworks/OpenTelemetry-java`). The scripts typically contain a `measure.sh` for starting the experiment, `functions.sh` with specific download or execution functions, `labels.sh` containing the names of the configurations and `config.rc` containing optional definitions of additional environment variables.

To measure performance in managed runtimes like the JVM, the managed runtime needs to be started multiple times. Within each instance, the workload must be repeated until a given count of warmup iterations is finished, and finally, the measurement iterations within the managed runtime need to be executed (Georges et al., 2007). MooBench implements this process by executing `$NUM_OF_LOOPS` loops, where each loop runs all configurations of one framework. Inside of each run, `$TOTAL_NUM_OF_CALLS` defines the number of iterations, i.e., repetitions of all calls to `monitoredMethod`. The execution data are stored into CSV files, and after the execution is finished, the warmup iterations are truncated.



Continuous Integration

Continuous testing is done via GitHub Actions: For every framework with name `$FRAMEWORK`, a workflow named `execute$FRAMEWORK.yaml` exist, that checks whether executing the default configuration for the framework yields the expected count of measurements. Furthermore, continuous benchmarking is executed via GitHub Actions. To avoid polluting the repositoy with data, the results are stored in the repository moobench-data.

Research impact statement

There have been numerous case studies that examined overhead of observability frameworks, and its root causes. Furthermore, there have been studies working on MooBench's continuous execution for regression benchmarking.

Overhead analysis

Eichelberger & Schmid (2014) describe and develop SPASS-meter, an observability tool for Java and Android applications. To evaluate its overhead, they integrated SPASS-meter into MooBench (Eichelberger et al., 2016). Based on this work, Knoche & Eichelberger (2018) compared the overhead measurement of SPASS-meter and Kieker on a Raspberry Pi. They found that the results were reproducible across different Raspberry Pi units, indicating that such hardware provides a viable means for researchers to achieve reproducible performance benchmarks.

Reichelt et al. (2021) integrated OpenTelemetry into MooBench and compared the overhead of Kieker, OpenTelemetry and inspectIT using MooBench. They found that the tracing overhead for Kieker was 4.6 μs, OpenTelemetry was 6.8 μs, and inspectIT was 10.9 μs. This indicates that Kieker's overhead was significantly lower than the overhead of OpenTelemetry at the time of the study.

Waller & Hasselbring (2012) examined how multi-core environments influence the overhead of observability with Kieker. They concluded that across all tested systems, asynchronous writing on multi-core architectures leads to a significant reduction of overhead.

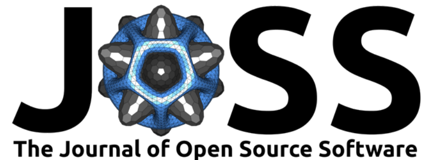
Yang et al. (2024) compared the overhead of Cloudprofiler (Yang et al., 2023), a monitoring application for stream processing workloads, to Kieker and OpenTelemetry. They found that Cloudprofiler's overhead (2.28 μs) is on average lower than the overhead introduced by Kieker (7.127 μs) and OpenTelemetry (higher, but only visible in a graph).

In our most recent study (Reichelt et al., 2026a), we integrated Pinpoint, Scouter, Skywalking, and Elastic APM into MooBench, measured the overhead of all frameworks, and examined the root causes of overhead. We found the rank order of overhead to be

$$\text{Kieker} < \text{OpenTelemetry} < \text{Elastic APM} < \text{Skywalking} < \text{inspectIT},$$

with all observability frameworks scaling linearly. For Scouter and Pinpoint, we observed that their overhead increases very slow since they lose records, i.e., they contain functional bugs. By profiling the overhead using async-profiler, we attributed the overhead to the categories time function calls, metadata management, call tree collection, memory management, and queue management. Notably, we found that a significant portion of the overhead is caused by extensive metadata management. The software version for this study was awarded the Available, Functional, and Reusable badges for the ICPE 2026 Artifact Evaluation Track (Reichelt et al., 2026b).[1]

[1] https://icpe2026.spec.org/tracks-and-submissions/artifact-evaluation-track/



Observability framework improvement and optimization

Strubel & Wulf (2016) examined how rewriting Kieker's monitoring component could reduce tracing overhead. Using MooBench, they demonstrated that their suggested rewrite reduced the overhead to 17% of the original measurement.

Reichelt et al. (2023) examined different options for overhead reduction for tracing: Using source code instrumentation instead of AspectJ (*The AspectJ Project*, 2026), storing only limited metadata, using a different queue and aggregating performance data before writing it to the monitoring queue. Using MooBench, they found that on their examined hardware, they could reduce the overhead from 4.77 ns to 0.4 ns per method call.

Reichelt, Bulej, et al. (2024) compared the overhead of the instrumentation frameworks AspectJ, ByteBuddy (Winterhalter, 2026), DiSL (Marek et al., 2012), Javassist (Chiba & Nishizawa, 2003), and direct source code instrumentation. To do so, they extended the Kieker-java scripts of MooBench. Through these extensions of Kieker and MooBench, they found that while source code instrumentation has the lowest overhead, ByteBuddy, DiSL and Javassist also have comparably low overhead. AspectJ causes significantly higher overhead than the others. Based on these findings, the Kieker framework transitioned to using the Kieker ByteBuddy agent as its default.

Continuous execution for regression benchmarking

Several authors worked on continuously executing MooBench benchmarks for spotting performance regressions in observability frameworks. Waller et al. (2014) started to include MooBench's benchmarking into CI. To reduce perturbations from concurrently running software, they configured Jenkins to execute measurements on a separate server. This setup has been running continuously to date, with regular hardware and system software updates. Additionally, a performance measurement setup for GitHub Actions was developed (Reichelt, Jung, et al., 2024). This additional setup utilizes GitHub's default runners, that, despite sharing the execution infrastructure with others, show low standard deviation and lower execution times than the original Jenkins setup. The measurement data from GitHub have been uploaded to Nyrkiö, a performance change detection tool for GitHub Actions. Using this tool, past regressions could be reproduced (Yang, Reichelt, Ingo, et al., 2025). In the future, it is planned to automatically detect performance regressions.

AI usage disclosure

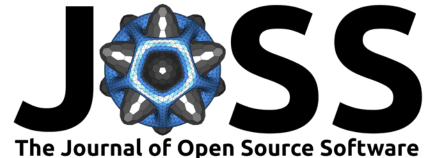
There is no known AI usage, however, AI usage in PRs is not monitored.

Acknowledgements

We acknowledge contributions over the years from André van Hoorn, Jan Waller, Reiner Jung, Christian Wulf, Florian Fittkau, Ferit Altin, Serafim Simonov, and Nils Christian Ehmke.

References

Ahmed, T. M., Bezemer, C.-P., Chen, T.-H., Hassan, A. E., & Shang, W. (2016). Studying the effectiveness of application performance management (APM) tools for detecting performance regressions for web applications: An experience report. *Proceedings of the 13th International Workshop on Mining Software Repositories*, 1–12. https://doi.org/10.1145/2901739.2901774



Chiba, S., & Nishizawa, M. (2003). An easy-to-use toolkit for efficient Java bytecode translators. *Proc. Int. Conf. On Generative Programming and Component Engineering*, 364–376. https://doi.org/10.1007/978-3-540-39815-8_22

Desnoyers, M., & Dagenais, M. R. (2006). The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. *OLS (Ottawa Linux Symposium)*, *2006*, 209–224. https://www.kernel.org/doc/ols/2006/ols2006v1-pages-209-224.pdf

Domaschka, J., Volpert, S., Maier, K., Eisenhart, G., & Seybold, D. (2023). Using eBPF for database workload tracing: An explorative study. *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, 311–317. https://doi.org/10.1145/3578245.3584313

Eder, C., Winzinger, S., & Lichtenthäler, R. (2023). A comparison of distributed tracing tools in serverless applications. *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 98–105. https://doi.org/10.1109/SOSE58276.2023.00018

Eichelberger, H., Sass, A., & Schmid, K. (2016). From reproducibility problems to improvements: A journey. *Softwaretechnik-Trends (Proceedings of the Symposium on Software Performance (SSP 2016))*, *36*(4). https://dl.gi.de/handle/20.500.12116/40621

Eichelberger, H., & Schmid, K. (2014). Flexible resource monitoring of Java programs. *Journal of Systems and Software*, *93*, 163–186. https://doi.org/10.1016/j.jss.2014.02.022

Gebai, M., & Dagenais, M. R. (2018). Survey and analysis of kernel and userspace tracers on Linux: Design, implementation, and overhead. *ACM Comput. Surv.*, *51*(2). https://doi.org/10.1145/3158644

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices*, *42*(10), 57–76. https://doi.org/10.1145/1297027.1297033

Hasselbring, W., Druskat, S., Bernoth, J., Betker, P., Felderer, M., Ferenz, S., Hermann, B., Lamprecht, A.-L., Linxweiler, J., Prat, A., Rumpe, B., Schoening-Stierand, K., & Yang, S. (2025). Multidimensional research software categorization. *Computing in Science & Engineering*, *27*(2), 59–68. https://doi.org/10.1109/mcse.2025.3555023

Hasselbring, W., Katz, D. S., & Nieuwpoort, R. van. (2026). Technology research software: An often overlooked category of research software. *Computing in Science & Engineering*, *28*(in press). https://doi.org/10.1109/MCSE.2026.3662117

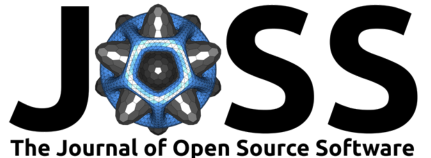
Hasselbring, W., & van Hoorn, A. (2020). Kieker: A monitoring framework for software engineering research. *Software Impacts*, *5*, 100019. https://doi.org/10.1016/j.simpa.2020.100019

Hoorn, A. van, Waller, J., & Hasselbring, W. (2012). Kieker: A framework for application performance monitoring and dynamic software analysis. *Proceedings of the 3rd Joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, 247–248. https://doi.org/10.1145/2188286.2188326

Hunold, S., Ajanohoun, J. I., Vardas, I., & Träff, J. L. (2022). An overhead analysis of MPI profiling and tracing tools. *Proceedings of the 2nd Workshop on Performance Engineering, Modelling, Analysis, and Visualization Strategy*, 5–13. https://doi.org/10.1145/3526063.3535353

Ilsche, T., Schuchart, J., Schöne, R., & Hackenberg, D. (2015). Combining instrumentation and sampling for trace-based application performance analysis. *Tools for High Performance Computing 2014: Proceedings of the 8th International Workshop on Parallel Tools for High Performance Computing, October 2014, HLRS, Stuttgart, Germany*, 123–136. https://doi.org/10.1007/978-3-319-16012-2_6

Janes, A., Li, X., & Lenarduzzi, V. (2023). Open tracing tools: Overview and critical comparison.



Journal of Systems and Software, *204*, 111793. https://doi.org/10.1016/j.jss.2023.111793

Knoche, H., & Eichelberger, H. (2018). Using the Raspberry Pi and Docker for replicable performance experiments: Experience paper. *Proc. 8th ACM/SPEC Int. Conf. On Performance Engineering*, 305–316. https://doi.org/10.1145/3184407.3184431

Majors, C., Fong-Jones, L., & Miranda, G. (2022). *Observability engineering*. O'Reilly Media, Inc.

Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., & Qi, Z. (2012). DiSL: a domain-specific language for bytecode instrumentation. *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*, 239–250. https://doi.org/10.1145/2162049.2162077

Nõu, A., Talluri, S., Iosup, A., & Bonetta, D. (2025). Investigating performance overhead of distributed tracing in microservices and serverless systems. *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*, 162–166. https://doi.org/10.1145/3680256.3721316

Reichelt, D. G., Bulej, L., Jung, R., & Van Hoorn, A. (2024). Overhead comparison of instrumentation frameworks. *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, 249–256. https://doi.org/10.1145/3629527.3652269

Reichelt, D. G., Jung, R., & Hoorn, A. van. (2024). Overhead measurement noise in different runtime environments. *Softwaretechnik-Trends (Proceedings of the 15th Symposium on Software Performance 2024)*. https://dl.gi.de/handle/20.500.12116/45533

Reichelt, D. G., Kühne, S., & Hasselbring, W. (2023). Towards solving the challenge of minimal overhead monitoring. *Comp. 14th ACM/SPEC International Conference on Performance Engineering*, 381–388. https://doi.org/10.1145/3578245.3584851

Reichelt, D. G., Kühne, S., & Hasselbring, W. (2021). Overhead comparison of OpenTelemetry, inspectIT and Kieker. *Short Paper Proceedings of Symposium on Software Performance 2021*. https://ceur-ws.org/Vol-3043/short3.pdf

Reichelt, D. G., Yang, S., Hansson, M., & Hasselbring, W. (2026a). Benchmarking the overhead of distributed tracing agents. *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering*, (in press). https://doi.org/10.1145/3777884.3797004

Reichelt, D. G., Yang, S., Hansson, M., & Hasselbring, W. (2026b). Benchmarking the overhead of distributed tracing agents – dataset. *Zenodo*. https://doi.org/10.5281/zenodo.17639772

Siegfried, G., Crossley, M., Byrne, P., Bridges, A., & Caren, M. (2025). *Magic quadrant for observability platforms* [Research Report]. Gartner, Inc. https://www.gartner.com/doc/reprints?id=1-2LF3Y49A&ct=250709&st=sb

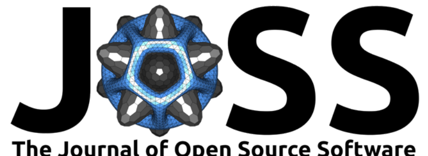
Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., & Shanbhag, C. (2010). *Dapper, a large-scale distributed systems tracing infrastructure*. http://research.google.com/pubs/archive/36356.pdf

Strubel, H., & Wulf, C. (2016). Refactoring Kieker's monitoring component to further reduce the runtime overhead. *Softwaretechnik-Trends (Proceedings of the Symposium on Software Performance (SSP 2016))*, *36*(4). https://dl.gi.de/handle/20.500.12116/40624

The AspectJ Project. (2026, January 16). The Eclipse Foundation. https://eclipse.dev/aspectj/

Volpert, S., Winkelhofer, S., Domaschka, J., & Wesner, S. (2025). Towards eBPF overhead quantification: An exemplary comparison of eBPF and SystemTap. *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*, 122–129. https://doi.org/10.1145/3680256.3721311

Waller, J., Ehmke, N. C., & Hasselbring, W. (2015). Including performance benchmarks into



continuous integration to enable DevOps. *ACM SIGSOFT Software Engineering Notes*, *40*(2), 1–4. https://doi.org/10.1145/2735399.2735416

Waller, J., Fittkau, F., & Hasselbring, W. (2014). Application performance monitoring: Trade-off between overhead reduction and maintainability. *Proceedings of the Symposium on Software Performance*. https://oceanrep.geomar.de/id/eprint/25653

Waller, J., & Hasselbring, W. (2013). A benchmark engineering methodology to measure the overhead of application-level monitoring. *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013*, *1083*, 59–68. https://ceur-ws.org/Vol-1083/paper7.pdf

Waller, J., & Hasselbring, W. (2012). A comparison of the influence of different multi-core processors on the runtime overhead for application-level monitoring. *International Conference on Multicore Software Engineering, Performance, and Tools*, *7303*, 42–53. https://doi.org/10.1007/978-3-642-31202-1_5

Winterhalter, R. (2026). Byte Buddy: runtime code generation for the Java virtual machine. *Spotify*. https://bytebuddy.net/

Yang, S., Jeong, J., Scholz, B., & Burgstaller, B. (2023). *Cloudprofiler: TSC-based inter-node profiling and high-throughput data ingestion for cloud streaming workloads*. https://arxiv.org/abs/2205.09325

Yang, S., Reichelt, D. G., & Hasselbring, W. (2024). Evaluating the overhead of the performance profiler Cloudprofiler with MooBench. *Softwaretechnik-Trends (Proceedings Des 15th Symposium on Software Performance 2024)*, *44*(4). https://dl.gi.de/handle/20.500.12116/45557

Yang, S., Reichelt, D. G., Ingo, H., & Hasselbring, W. (2025). Detection of performance changes in MooBench results using Nyrkiö on GitHub Actions. *Softwaretechnik-Trends (Proceedings Des 16th Symposium on Software Performance 2025)*, *45*(4). https://dl.gi.de/handle/20.500.12116/47944

Yang, S., Reichelt, D. G., Jung, R., Hansson, M., & Hasselbring, W. (2025). The Kieker observability framework version 2. *Companion of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE 2025)*, 11–15. https://doi.org/10.1145/3680256.3721972