

Overhead Comparison of OpenTelemetry and Kieker in Python

David Georg Reichelt 
Lancaster University Leipzig /
URZ Leipzig

Shinhyung Yang 
Software Engineering
Kiel University

Ferit Altin
Lancaster University Leipzig

Wilhelm Hasselbring 
Software Engineering
Kiel University

Abstract

The overhead of distributed tracing is a core property of a tracing framework. It is especially relevant in fast-evolving languages like Python, where new versions of the language might change the overhead significantly. In this context, being able to measure the overhead and understanding its causes is crucial.

To address this challenge, we present the extension of the MooBench benchmark for measuring OpenTelemetry’s tracing overhead in Python. Based on this extension, we compare the overhead between OpenTelemetry and Kieker. We find that both are higher by order of magnitude than in their Java counterparts. Furthermore, we find that Kieker for Python has significantly lower overhead than OpenTelemetry for Python due to its extensive metadata gathering.

1 Introduction

Tracing overhead is one main selection criterion for an observability framework: If the tracing overhead is too high, the observability framework cannot be used, or at least cannot be used for all events that should be traced. Also, high overhead makes it impossible to detect performance changes [5]. Tracing overhead has multiple causes: The instrumentation itself, the collection of data, passing the data into the queue, and processing them further, e.g., by serialization. These causes work different in different programming languages: Instrumentation technologies differ, the metadata that can be obtained differ, and synchronization mechanisms for the queue differ. This is especially relevant in currently supported Python releases (prior to 3.13), where the Global Interpreter Lock (GIL) prevents the parallel execution of Python code.

Tracing overhead has been examined in Java for different tracing frameworks [3, 8]. In this work, we compare the tracing overhead of OpenTelemetry and Kieker in Python. Prior to this work, no overhead benchmark for OpenTelemetry in Python existed. Therefore, we (1) describe how we extended

MooBench, so it supports the overhead measurement for OpenTelemetry, and (2) analyze the measurement results and compare them to Kieker’s tracing overhead in Python.

2 Measurement Approach

In this section, we first describe the basic overhead measurement approach and then how we extended it for OpenTelemetry.

2.1 Basic Overhead Measurement

MooBench is a tool for overhead measurement of observability tools [9]. It consists of a system under test and measurement scripts for different observability tools. The system under test’s (SuT) core is a method that recursively calls itself for a given tree depth `$RECURSION_DEPTH`. In the measurement scripts, this SuT is instrumented with different observability tools. Usually, the measurement scripts contain configurations of different factorial experiments that make it possible to measure the overhead for instrumentation without serialization, or serializations to different data sinks (e.g., TCP and binary on the hard disk).

Alongside the Java tools, MooBench supports overhead measurement on Kieker [7] in Python [6] with Kieker’s language pack for Python.¹ Kieker implements two instrumentation approaches: (1) `PostImportLoader` (post), and (2) `InstrumentOnImportFinder` (pre). For a target function `func()` in a target module `m`, `PostImportLoader` replaces it with `instrument(func)`, `post-import`. On the other hand, `InstrumentOnImportFinder` reads the abstract syntax tree (AST) of `m` while loading the module. It decorates `func()` with a `@instrument` annotation `pre-import`, which then triggers `instrument(func)`.

¹<https://github.com/kieker-monitoring/kieker-lang-pack-python>

2.2 Extension for OpenTelemetry

To extend MooBench for OpenTelemetry-Python, the scripts for automating the instrumentation with Python were added. OpenTelemetry has two instrumentation variants for Python: Auto instrumentation² and manual instrumentation. Since auto instrumentation is built for tracing HTTP calls, like calls processed by Django, and not method calls itself, we chose to use manual instrumentation.

Manual instrumentation requires the adaptation of the SuT’s source code itself. In the monitored application, this requires having a tracer available, starting the span and ending it, ideally using a `with` block. Since the tracer is only available with activated OpenTelemetry tracing, this would require conditionally executing the span starting and ending, both in regular and exceptional cases. These numerous conditional statements can potentially change the performance of the measurement of other tracing tools. Therefore, we decided to implement a separate method that is called when OpenTelemetry is activated. In this method, the content of the original method is put into a `with` block, that starts and ends the span automatically. Listing 1 depicts this methods structure.

Listing 1: OpenTelemetry Method

```
def monitored_method_otel(method_time,
    rec_depth, tracer):
    with tracer.start_as_current_span("
        monitored.method") as span:
        ... // Original monitored method code
```

3 Measurement Results

To compare the overhead of Kieker and OpenTelemetry using Python, we executed measurements on PCs running Rocky Linux 9.8 running Python 3.12.13 on AMD Ryzen 7 5700G.

3.1 Overview

The measurement results for MooBench’s default configuration (2000000 iterations, call tree depth 10, 10 loops) are depicted in Table 1. The lowest measured value for full monitoring is 99.72 μ s for full tracing using **Binary TCP - pre**, which is substantially higher compared to 1 to 5 μ s, which we see in Java for different tracing frameworks [8]. Similarly to Simonov et al. [6], we see that post instrumentation weaving incurs higher overhead than pre-compilation weaving.

In the configuration with full data gathering (Binary TCP - pre / OpenTelemetry Zipkin), we see that OpenTelemetry has significantly higher overhead. Since OpenTelemetry’s probes are manually inserted, this indicates that this overhead is purely created by OpenTelemetry’s own processing.

Notably, the *No Instrumentation* configuration takes significantly more time for OpenTelemetry than

Configuration	Mean /ns	Std. Dev. /ns
Kieker-python		
<i>No Instrumentation</i>	533.13	6.90
Deactivated probe - post	1325.10	13.53
Deactivated probe - pre	1344.42	22.69
No logging - post	46177.70	442.39
No logging - pre	13395.90	174.15
Binary TCP - post	167600.00	8271.11
Binary TCP - pre	99717.60	2683.21
OpenTelemetry-python		
<i>No Instrumentation</i>	559.24	23.61
OpenTelemetry No Export	130227.00	1608.22
OpenTelemetry Zipkin	266775.00	2255.22

Table 1: Execution Times Using Different Timers

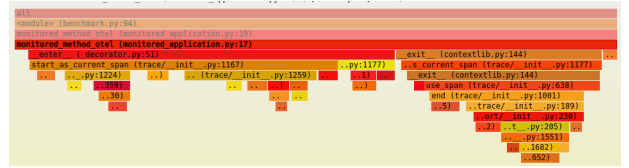


Figure 1: Flamegraph of OpenTelemetry

for Kieker. This effect is reproducible. It is caused by the OpenTelemetry modules being installed using `pip`; if they are not installed, both have the same overhead.

All measurements were first executed with Python 3.9.25. This yielded significantly higher values (≈ 750 ns baseline, binary TCP A Kieker ≈ 261 ns, OpenTelemetry Zipkin 424000 ns). Therefore, while the values are stable for one version, we see that Python’s performance improvements across versions yield significant changes. Hence, also future updates and optimizations based on the measurements should be used with caution, and might change with new Python versions.

Our full data, including the measurements for Python 3.9.25, are available in our replication dataset.³

3.2 Flamegraph Analysis

To further analyze the results, we used `py-spy` in order to obtain flamegraphs for Kieker’s and OpenTelemetry’s full instrumentation. Flamegraphs usually consist of the `monitored_method` and the instrumentation overhead. Exemplary flamegraphs zoomed into the instrumentation overhead is visualized in Figure 1 and Figure 2.

They show the reason for the overhead difference: While Kieker only obtains basic necessary metadata, OpenTelemetry does more complex processing. In addition to storing the span, it decides on whether the span is sampled, stores metadata such as arbitrary attributes that are added as key-value pairs,

²<https://opentelemetry.io/docs/zero-code/python/>

³<https://doi.org/10.5281/zenodo.21775347>

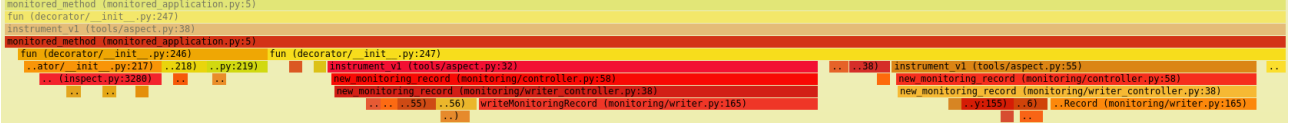


Figure 2: Flamegraph of Kieker, Zoomed into the Last Method Call

stores references to different contextual information, and maintains references to the parent span. Listing 2 depicts all the information that are stored in one OpenTelemetry span, and that therefore need to be obtained beforehand.

Listing 2: Creation of OpenTelemetry span

```
span = _Span(
    name=name,
    context=span_context,
    parent=parent_span_context,
    sampler=self.sampler,
    resource=self.resource,
    attributes=sampling_result.attributes,
    span_processor=self.span_processor,
    kind=kind,
    links=links,
    instrumentation_info=self
        instrumentation_info,
    record_exception=record_exception,
    set_status_on_exception=
        set_status_on_exception,
    limits=self._span_limits,
    instrumentation_scope=self
        instrumentation_scope,
    record_end.metrics=record_end.metrics,
)
```

4 Related Work

Gocht et al. [1] extended the tracing framework ScoreP for Python, and measured its overhead. They found that for full tracing using Python’s `settrace` interface, including line-level information, an overhead of $17.9\mu s$ is incurred per method call. This is significantly higher than our previously measured values, which is expected due to the extensive metadata obtained by the fine-grained tracing of `settrace`.

Based on this existing tracing implementation, Gocht-Zech et al. [2] reduced the overhead by converting the tracing code into C and implementing selective instrumentation. They found that using the C-API of Python, the overhead gets reduced by up to 74%. Hence, using the C-API is also a promising candidate to further reduce the already low overhead of Kieker for Python.

Eghbali and Pradel [4] implemented DynaPyt, an approach to weave dynamic analysis code into Python’s instrumentation hooks. In comparison to tracing, dynamic analysis allows more fine-grained analysis, e.g., instrumentation at statement level. They find that tracing all hooks increases the execution time by a factor of 1.2 to 16, depending on the underlying project.

5 Summary

In this paper, we examined the overhead of tracing in Python using different tracing agents. We found that the overhead is significantly higher than for the same frameworks using Java. This is caused by the nature of Python as a language with GIL and without compilation. Furthermore, we found that Kieker for Python is significantly faster than OpenTelemetry for Python; this is mostly caused by extensive metadata handling in OpenTelemetry.

In future work, we see three main directions: (1) Python 3.13 brings the option to deactivate GIL and to enable Just In Time compilation. Both are expected to bring substantial performance improvement. It would be valuable to evaluate their effects on the tracing overhead. (2) Besides OpenTelemetry and Kieker, other observability frameworks like Pinpoint and Elastic APM offer Python instrumentations. An inclusion of them into MooBench and a in-depth examination of their overhead sources would be valuable. (3) Besides Python, other languages are widely used in production code, such as TypeScript, Rust, and Go. Including them into MooBench would enable a comparison of the same agent across languages.

References

- [1] A. Gocht, R. Schöne, and J. Frenzel. “Advanced Python performance monitoring with Score-P”. In: *arXiv* (2020). DOI: 10.48550/arXiv.2010.15444.
- [2] A. Gocht-Zech, A. Grund, and R. Schöne. “Controlling the runtime overhead of Python monitoring with selective instrumentation”. In: *ProTools*. IEEE. 2021, pp. 17–25. DOI: 10.1109/ProTools54808.2021.00008.
- [3] D. G. Reichelt, S. Kühne, and W. Hasselbring. “Overhead Comparison of OpenTelemetry, inspectIT and Kieker”. In: *SSP 2021*. 2021. URL: <https://ceur-ws.org/Vol-3043/>.
- [4] A. Eghbali and M. Pradel. “DynaPyt: a dynamic analysis framework for Python”. In: *ESEC/FSE*. 2022, pp. 760–771. DOI: 10.1145/3540250.3549126.
- [5] D. G. Reichelt, S. Kühne, and W. Hasselbring. “Automated identification of performance changes at code level”. In: *QRS*. 2022. DOI: 10.1109/QRS57517.2022.00096.
- [6] S. Simonov et al. “Instrumenting Python with Kieker”. In: *SSP*. 2022. URL: <https://dl.gi.de/handle/20.500.12116/43639>.
- [7] S. Yang et al. “The Kieker Observability Framework Version 2”. In: *Companion of the 16th ACM/SPEC ICPE*. 2025, pp. 11–15. DOI: 10.1145/3680256.3721972.
- [8] D. G. Reichelt et al. “Benchmarking the Overhead of Distributed Tracing Agents”. In: *ICPE*. 2026, pp. 147–161. DOI: 10.1145/3777884.3797004.
- [9] D. G. Reichelt et al. “MooBench: A micro-benchmark for performance overhead measurement of observability tools”. In: *JOSS* 11 (2026). DOI: 10.21105/joss.10400.