Dynamic and Static Analysis of Python Software with Kieker

Daphné Larrivain
daphne.larrivain@ecole.ensicaen.fr
ENSICAEN, Caen, France

Shinhyung Yang
shinhyung.yang@email.uni-kiel.de
Kiel University, Kiel, Germany

Wilhelm Hasselbring
hasselbring@email.uni-kiel.de
Kiel University, Kiel, Germany

Abstract

The Kieker observability framework provides users with the means to design observability pipelines for their applications. Python's popularity has exploded over the years, thus making structural insights of Python applications highly valuable. Originally tailored for Java, adding Python support to Kieker is worthwhile. Our Python analysis pipeline combines static and dynamic analysis in order to build a complete picture of a given system.

1 Introduction

Visual access to a system's structure enables valuable insights, from identifying differences between intended and implemented architectures to understanding unfamiliar software [4].

Static analysis extracts key observations from source code, whereas dynamic analysis uncovers runtime behaviour. Combining the two allows for a more comprehensive view. The latter is known as combined analysis. This approach builds on earlier research [4].

This paper presents an extension of the Kieker observability framework [3, 9] for Python, enabling analysis and visualization of standard modules. Three tools were developed, and maintenance challenges identified. Section 2 reviews existing foundations, Section 3 outlines the pipeline, Section 4 details the analysis types, and Section 5 covers the evaluation.

2 Related Work

Existing static analysis tools include AST matchers [10]. An abstract syntax tree (AST) is a tree structure that represents the syntactic organization of source code, where each node corresponds to a given code construct.

Kieker's existing approach, originally designed for Fortran [4], offers principles adaptable to Python. The Fortran workflow involved three steps: Fxtran [1] translates target source code into an AST, Fxca converts the AST to a CSV format compatible with the Static Architecture Recovery (SAR) tool, and the SAR tool then generates an architectural model from the CSV.

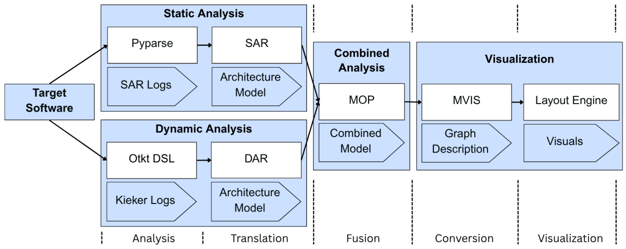

Figure 1: Combined analysis pipeline stages. Filter tools are shown in white, and the intermediate data formats are contained in arrow-like shapes.

In this paper, we present two main contributions. First, we brought the existing combined analysis approach, originally developed for Fortran applications, to Python. Second, we incorporated the Tulip visualization framework into our workflow, which significantly enhances the processing time and layout of nested graphs.

3 General Pipeline

The Kieker observability framework offers a broad set of tools. They follow the TeeTime pipes and filters architectural pattern [2]. In this model, filters are processing steps and pipes connect them, requiring compatible input and output formats. When this condition is met, users can combine filters to build custom processing pipelines and generate the desired output.

Since the Kieker framework was not originally designed to handle Python applications, it was necessary to adapt existing tools and create new ones as described in Table 1. The resulting processing pipeline is shown in Figure 1. The extended version provides a more in-depth exploration, including complete example architectures to illustrate the approach [7]. A replication package is available[2] as well as a virtual machine with the environment already set up.[3]

[1] https://github.com/pmarguinaud/fxtran

[2] https://github.com/kieker-monitoring/PCARP

[3] https://zenodo.org/records/16735614

Tool	Description	Origin	Revision	Language	Repository
Pyparse	analyzes Python code and outputs AST for SAR	New	New	Python	`https://github.com/kieker-monitoring/pyparse`
OtktInst	(Otkt Instrument Tool) applies Otkt DSL to instrument Python code	New	New	Python	`https://github.com/kieker-monitoring/OtktInst`
GGVIS	(Grouped Graph Visualizer) renders graphs and exports to PDF/SVG/PNG	New	New	Python	`https://github.com/kieker-monitoring/GGVIS`
Otkt DSL	(OpenTelemetry to Kieker Translation DSL) describes a mapping from OTel span to Kieker record	Kieker	Improved	Python, Java, Xtext	`https://github.com/kieker-monitoring/OtktDSL`
DAR	(Dynamic Architecture Recovery) converts Kieker logs to architecture models	Kieker	As-is	Java	`https://github.com/kieker-monitoring/kieker`
SAR	(Static Architecture Recovery) builds architecture models from received AST description	Kieker	Fixes	Java	`https://github.com/kieker-monitoring/kieker`
MOP	(Model OPeration) merges and compares architecture models	Kieker	As-is	Java	`https://github.com/kieker-monitoring/kieker`
MVIS	(Model Visualization and Statistics Tool) exports architecture models to graphics	Kieker	As-is	Java	`https://github.com/kieker-monitoring/kieker`

Table 1: We developed three new tools and revised two Kieker tools for our combined analysis pipeline.

4 Combined Analysis

Combined analysis is the process of fusing the insights from static and dynamic analysis to get a better picture of a given system. The general pipeline gathers data from both processes before fusing them with the Model Operation (MOP) tool.

4.1 Dynamic Analysis

Kieker's Python support [5] builds on OpenTelemetry, a cross-language observability framework. Using OtktDSL [6], OpenTelemetry spans are translated into Kieker records for dynamic analysis.

This bridge is technical, enabling data transfer but leaving interpretation and selection to the user. To ensure compatibility with Kieker tools, a Java-like architecture was adopted, reconstructing not just files but also classes for an object-oriented system view.

Application-specific analysis requires tailored instrumentation. Semi-automation is still possible via naming heuristics. Otkt-Instrument was developed to support this process. Since this is a hands-on process, some heuristics may have been missed. To this effect, new constraints can be easily added within the tool. However, in some outlier cases, application-specific systems may not be suitable for instrumentation, as shown in the evaluation.

4.2 Static Analysis

The previous Fortran research outlines a general process: source code → AST → CSV → SAR tool. However, this sequence couldn't be reused directly for Python. ASTs are language-specific, and Python's built-in module doesn't produce output compatible with existing Kieker tools. To solve this, we developed Pyparse. It performs AST generation and CSV conversion in one step, extracting two key types of relationships: function calls and data flow. These are then passed to the SAR tool for architecture reconstruction.

4.3 Visualization

A key objective of the visualization process was to lay out the graph to reveal the analyzed system's structure at a glance. The displayed components were flattened in the default output, without any meaningful grouping, which made the result difficult to interpret.

To address this, nodes were grouped based on their package affiliation. This led to the development of GGVIS, a tool built on the Tulip framework [1]. Tulip was chosen over the GraphViz dot utility due to the latter's limitations in handling nested graph layouts.

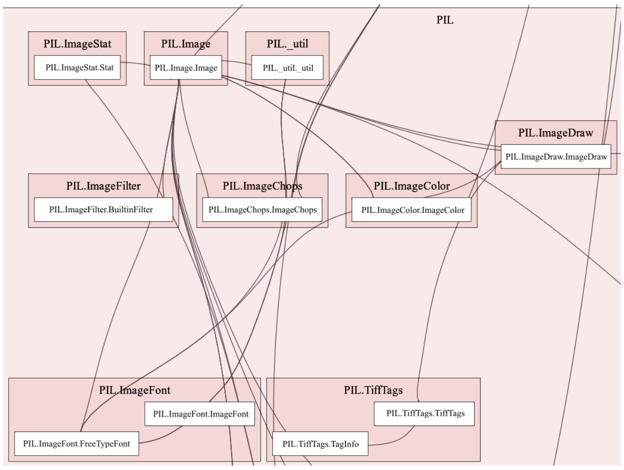

Figure 2: A close-up of the graph obtained by analyzing Pillow, one of the evaluated applications. The nodes were laid out in a nested manner with Tulip. The text was enlarged for readability.

5 Evaluation

We measured the duration of the entire pipeline, except for the data gathering phase of the dynamic analysis. This step was excluded because it includes the runtime overhead of the application under test, not the performance of the pipeline itself. The results were compiled in Table 2. A few key observations deserve to be highlighted. First, as the number of files increases, processing time increases accordingly.

	A.	U.	P.	Matplotlib	Numpy	Scipy
Pyparse	1s	2s	10s	3m53s	2m57s	7m15s
DAR	2s	4s	1s	2s		2s
SAR	3s	6s	10s	7m26s	10m44s	21m58s
MOP	3s	6s	5s	26m13s		28s
MVIS	2s	3s	2s	1m10s	19m22s	3s
GGVIS	1s	1s	1s	4m28s	34s	1s
Total	12s	22s	29s	43m12s	33m37s	29m47s

Table 2: Pipeline durations in minutes and seconds when analyzing standard Python modules. Note: A: Anytree[7], U: UXsim [8], and P: Pillow[8].

Matplotlib[4] and Scipy[5] show low DAR times due to limited input from dynamic analysis, which targets only executable parts of the application. Since some files are never run, they remain unaccounted for.

Numpy could not be instrumented due to application-specific limitations, which prevented dynamic analysis. Our instrumentation approach relies on wrapping Python functions. However, an internal key system of Numpy already employs this mechanism extensively. As a result, any attempt to modify it breaks the module.

Finally, regarding Scipy, the MOP tool failed to process the static model, relying solely on the dynamic model during the merge and disregarding the static one. This explains the low processing time. Nevertheless, the DAR and SAR models remained structurally valid and successfully generated correct graphs via MVIS and GVIS. The most plausible explanation for MOP's failure is that the static model was simply too large for the tool to handle.

For this evaluation, the objective was to push the tools to their limits. When applied to larger Python modules, the resulting graphs can become extremely large. In the case of Numpy, we are reaching around 600 nodes and 8000 edges. This raises questions about the practical utility of such an endeavor.

6 Conclusion

This work extends the Kieker observability framework to improve its support for Python applications. A combined analysis approach, integrating static and dynamic analysis, was replicated.[6] To implement this, a custom analysis pipeline was designed using built-in Kieker tools, alongside adaptations of existing components and the development of new ones.

Throughout the process, inconsistencies in tool outputs and maintenance challenges were identified. To resolve these issues, solutions from prior research were customized and integrated to better align with the project's requirements. A Python static analyzer based on AST matching was developed specifically for the pipeline. Furthermore, a more intuitive visualization method was introduced, and limitations in current layout tools were revealed, highlighting an area for future improvement.

The next steps involve repairing the existing tool suite (DAR, SAR, MOP, MVIS), whose build process is currently broken in the current Kieker distribution, and moving away from Graphviz, which performs poorly with large-scale graphs.

Acknowledgment This research is funded by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. 528713834.

References

[1] D. Auber et al. "Tulip 5". In: *Encyclopedia of Social Network Analysis and Mining*. Springer, Aug. 2017, pp. 1–28. DOI: 10.1007/978-1-4614-7163-9_315-1.

[2] C. Wulf, W. Hasselbring, and J. Ohlemacher. "Parallel and Generic Pipe-and-Filter Architectures with TeeTime". In: *ICSAW*. 2017, pp. 290–293. DOI: 10.1109/ICSAW.2017.20.

[3] W. Hasselbring and A. van Hoorn. "Kieker: A monitoring framework for software engineering research". In: *Software Impacts*. 2020. DOI: 10.1016/j.simpa.2020.100019.

[4] R. Jung et al. "Architecture Recovery from Fortran Code with Kieker". In: *Softwaretechnik-Trends*. SSP '22 43.1 (2023), pp. 38–40.

[5] S. Simonov et al. "Instrumenting Python with Kieker". In: *Softwaretechnik-Trends*. SSP '22 43.1 (2023), pp. 26–28.

[6] S. Simonov. "Domain Specific Language Support for Kieker and OpenTelemetry Interoperability". Master's thesis. Kiel University, 2024.

[7] D. Larrivain, S. Yang, and W. Hasselbring. *Dynamic and Static Analysis of Python Software with Kieker Including Reconstructed Architectures*. 2025. DOI: arXiv:2507.23425.

[8] T. Seo. "UXsim: lightweight mesoscopic traffic flow simulator in pure Python". In: *JOSS* 10.106 (2025), p. 7617. DOI: 10.21105/joss.07617.

[9] S. Yang et al. "The Kieker Observability Framework Version 2". In: ICPE '25. Toronto ON, Canada: ACM, 2025, pp. 11–15. DOI: 10.1145/3680256.3721972.

[10] H. Gulabovska and Z. Porkoláb. "Towards More Sophisticated Static Analysis Methods of Python Programs". In: Informatics'2019, pp. 225–230. DOI: 10.1109/Informatics47936.2019.9119307.

[7] https://github.com/c0fec0de/anytree
[8] https://github.com/python-pillow/Pillow
[4] https://github.com/matplotlib/matplotlib
[5] https://github.com/scipy/scipy
[6] https://github.com/kieker-monitoring/PythonCombinedAnalysis_replication-package