Electronic Communications of the EASST
Volume 86 (1) Year 2026

deRSE26 - Selected Contributions of the 6th Conference for Research Software Engineering in Germany (Part 1)

Edited by Bernd Flemisch, René Caspart, Bernadette Fritzsch, Florian Goth, Jean-Noël Grad, Florian Thiery

Visualizing Python Software Structures via Dynamic and Static Analysis With the Kieker Framework

Daphné Larrivain, Shinhyung Yang, Wilhelm Hasselbring

DOI: 10.14279/eceasst.v86.1.2780

License: This article is licensed under a CC-BY 4.0 License.

Electronic Communications of the EASST (https://eceasst.org).
Published by **Berlin Universities Publishing**
(https://www.berlin-universities-publishing.de/)

Visualizing Python Software Structures via Dynamic and Static Analysis With the Kieker Framework

Daphné Larrivain [1], **Shinhyung Yang** [2], **Wilhelm Hasselbring** [3]

[1] daphne.larrivain@ecole.ensicaen.fr
ENSICAEN, France
[2] shinhyung.yang@email.uni-kiel.de
Kiel University, Germany
[3] hasselbring@email.uni-kiel.de
Kiel University, Germany

Abstract: The Kieker observability framework is a tool that provides users with the means to design a custom observability pipeline for their application. Originally tailored for Java and Fortran, supporting Python with Kieker is worthwhile: Python is a popular language to implement research software, thus making structural insights of Python applications highly valuable. Our Python analysis pipeline combines static and dynamic analysis of a given software system. We present the evaluation of the tool's execution performance and the evaluation of the layout algorithms for visualizing the software structure. Our emphasis of the present paper is on the importance of different visualization techniques, and the evaluation of the visualization engine in this context. We discovered that our visualization engine processed nested graphs of a NumPy example in 34 seconds where as prior approaches does not terminate and consumes more than one hour.

Keywords: research software engineering, reverse software engineering, software visualization, software analysis

1 Introduction

In the research software engineering community, it is generally accepted that better software leads to better research [Gob14]. In software engineering, it is known that clear architectural foundations have strong impact to better software [NSB+08, Has18]. Building on these principles, it is recommended to design robust and maintainable software architectures in research contexts, recognizing their critical role in enhancing the quality and impact of research outcomes [DEC+25].

Reverse software engineering increases the maintainability of legacy and existing software through reconstructing their architecture [CDC11]. This enables visual access to the target software's structure, which provides intuition without prior knowledge of the software code, ranging from identifying differences between intended and implemented architectures to understanding unfamiliar software.

Visualizing the structure of a given software system is the first step toward reconstructing its architecture, and later restructuring the architecture for improved maintainability. In the

present paper, we report on improving and extending the Kieker framework to efficiently visualize Python software structures.

The Kieker observability framework [HWH12, HH20, YRJ+25] was successfully used to visualize research software implemented in Fortran [HJS25a, JSGH23] and Perl [ZHC15]. The analyzed Fortran code is modeling and simulation software (Category 1.1 of [HDB+25]), while the analyzed Perl code is research data management software (Category 3.6 of [HDB+25]). Kieker itself is technology research software [HKN26]. Besides visualizing research software, Kieker was often used for analyzing commercial software, mainly implemented in Java [ZOLL11, QZL+14, QGZ+15b, QGZ+15a, JLQ+16, JLQ+18, Dab12, MDTS13, SH20], but also with COBOL [KHGH12].

Today's research software often relies on the Python language, which has been increasingly popular among research software engineers. However, maintaining and extending existing research software only from the code could be hindered by the lack of the previous knowledge of the original researcher who has left the project without adequate documentation [GAA18]. Thus, we aim to provide visual access to the target software structure with our combined static and dynamic software analysis.

Static analysis extracts information from the source code, whereas dynamic analysis uncovers runtime behavior. Combining the two allows for a more comprehensive view. The combination is known as combined analysis [AB05, HJS25b]. Our approach for Python software builds on the previous combined analysis for Fortran software [JSGH23]. Using the Kieker observability framework as a basis, the existing tools were extended to bring a custom analysis and visualization pipeline to Python.

In this paper, we present two main contributions: first, we adapted the existing combined analysis approach, originally developed for Fortran applications, to effectively analyze Python applications; second, we incorporated the Tulip framework [AAB+17] into our workflow, which significantly enhances the visualization performance.

We start with a look at related work in Section 2, before the implementation is presented in Section 3. Section 4 presents the evaluation of the tool's execution performance and the visual layout algorithms. Section 5 draws conclusions and discusses future work.

2 Related Work

Architecture recovery is an ongoing topic in software engineering. Ghaleb et al. [GAA18] reviews different methodologies and techniques to recover the architecture of a legacy system by reverse engineering through static- and dynamic analysis and presents the cases of program comprehension using different types of UML diagrams. The use of sequence diagrams prominently focused here, and from other research as well [GAA20, SMIJ21].

Since AST was first introduced [McC62], it has been essential in building compiler frontends [Sco09]. Researchers try to build compiler helper tools exploiting the different language ASTs [RP96, FLZW97, BC20, WYZH22]. PyKokkos [AZBG21] is an effort for inter-language interoperability by converting Python AST to C++ AST.

There are many types of static code analysis tools using abstract syntax tree (AST). Among them are AST matchers [GP19]. An AST is a tree-like structure that represents the syntactic

organization of source code, where each node corresponds to the associated code construct. According to the characteristics of simple and flat Python development, ASTs are used as the main means to mine context information of Python code fragments [MYX22]. Arcade [SMMG20] provides limited Python support for preventing architectural decay through a modular pipeline with five subsystems: recovery, decay detection, measurement, visualization, and prediction. Arcade does not support dynamic analysis. Unlike Arcade, we focus on the combined analysis of Python software, especially on the visualization. Kieker's existing combined analysis, originally designed for Fortran [JSGH23], offers the baseline analysis pipeline adaptable to Python. In this work, the Fortran workflow involves three steps: Fxtran[1] translates target source code into an AST. Then, Kieker provides two tools: fxca (FXTran Code Analysis)[2] and SAR (Static Architecture Recovery):[3] fxca converts the input AST into the SAR-compatible format, and SAR generates an architectural model from the input.

3 Implementation

To describe our implementation, we fist introduce the employed Kieker analysis pipeline in Section 3.1. Dynamic and static analysis for Python are introduced in Sections 3.2 and 3.3, respectively. Section 3.4 explains the combined analysis. The architecture visualization via automatic layout engines is described in Section 3.5.

3.1 The Kieker Analysis Pipeline

The Kieker observability framework offers a broad set of tools for software monitoring and analysis. The Kieker tools use the TeeTime high-throughput framework [WHO17], which implements the pipe-and-filter architectural pattern. With TeeTime, each *stage* represents a filter, a processing step that transforms incoming data into output data. Pipes serve as connections between each stage, enabling data to flow from one stage to another. For tools to be chained together in the pipe-and-filter pattern, the output format of one stage must match the input format of the next stage. When this requirement is met, users can combine TeeTime stages to build custom TeeTime 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 summarized in Table 1. The resulting processing pipeline is shown in Figure 1 in three stages, and we explain it in dedicated sections. The analysis begins from both *Dynamic Analysis* (Section 3.2) and *Static Analysis* (Section 3.3) separately, then (2) *Combined Analysis* (Section 3.4) produces the entire architectural description. Lastly (3) graphic files are generated in *Visualization* (Section 3.5).

3.2 Dynamic Analysis

Figure 1, lower left corner, illustrates the dynamic analysis in our pipeline. Kieker's Python support for dynamic analysis [SDJG23] is based on OpenTelemetry [GB23], a language- and

[1] https://github.com/pmarguinaud/fxtran
[2] https://github.com/kieker-monitoring/kieker/tree/2.0.3/tools/fxca
[3] https://github.com/kieker-monitoring/kieker/tree/2.0.3/tools/sar

Idx	Tool	Description, GitHub repository	Origin	Rev.	Lang.	Release
T1	Pyparse	analyzes Python code and outputs AST for SAR kieker-monitoring/pyparse	New	New	Python	0.0.1
T2	OtktInst	(Otkt Instrument tool) applies Otkt DSL to instrument Python code kieker-monitoring/OtktInst	New	New	Python	0.0.1
T3	GGVIS	(Grouped Graph Visualizer) renders graphs and exports to PDF/SVG/PNG kieker-monitoring/GGVIS	New	New	Python	0.0.1
T4	OtktDSL	(OpenTelemetry to Kieker Translation DSL) describes a mapping from OTel span to Kieker record kieker-monitoring/OtktDSL	Kieker	Improved	Python, Java, Xtext	1.0.0
T5	SAR	(Static Architecture Recovery) builds architecture models from received AST description kieker-monitoring/kieker/tree/main/tools/sar	Kieker	Fixed	Java	1.15.4
T6	DAR	(Dynamic Architecture Recovery) converts Kieker traces to architecture models kieker-monitoring/kieker/tree/main/tools/dar	Kieker	As-is	Java	1.15.4
T7	MOP	(Model OPeration) merges and compares architecture models kieker-monitoring/kieker/tree/main/tools/mop	Kieker	As-is	Java	1.15.4
T8	MVIS	(Model Visualization and Statistics Tool) exports architecture models to graphics kieker-monitoring/kieker/tree/main/tools/mvis	Kieker	As-is	Java	1.15.4

Table 1: We developed three new tools (T1, T2, T3), revised two Kieker tools (T4, T5), and reused three Kieker tools as-is (T6, T7, T8) for our combined analysis pipeline. The new tools are implemented in Python, while the existing tools are mainly implemented in Java and Xtext [EEK+12]. These tools are annotated to the pipeline stages in Figure 1.

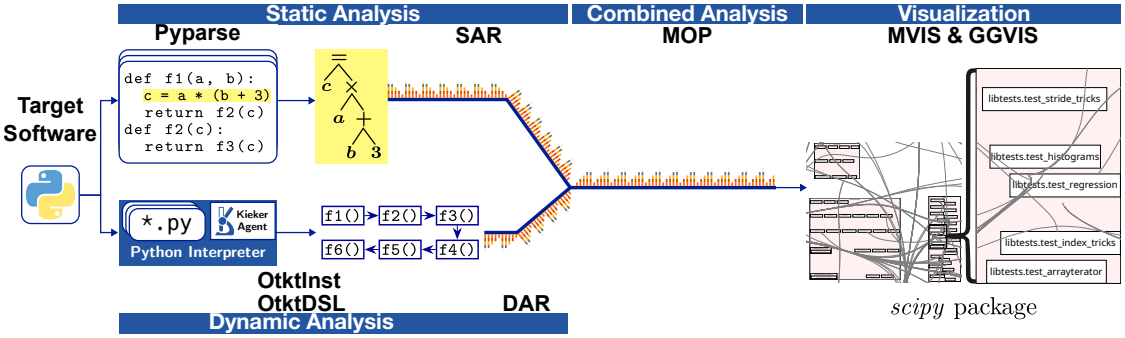


Figure 1: Our Python software visualization pipeline stages, combining and visualizing the analysis results both from the static (upper route) and the dynamic (lower route) software analysis.

platform-agnostic observability framework. Kieker's dynamic analysis allows Kieker uses the OpenTelemetry for Kieker Translator domain-specific language (OtktDSL) [Sim24] (T4 in Table 1) to translate OpenTelemetry spans into Kieker traces, which allow for Kieker's dynamic analysis of Python software.

Dynamic analysis requires method-level tracing over consecutive method calls. It allows tracing the application's control flow and locating bottlenecks between function calls and between modules. Thus it requires code-level instrumentation tailored to the desired insights. While this is application-specific to a degree, our Otkt Instrument (OtktInst) tool (T2 in Table 1) partially automates the instrumentation process by annotating each function with the `@instrument` decorator. Then, OtktDSL is responsible for two tasks: (1) creating a decorator that instruments functions in the target program, and (2) creating a OpenTelemetry Exporter that translates OpenTelemetry spans to Kieker traces and sends them to a Kieker collector. Because the existing combined analysis assumes Java-style fully qualified names, we modified the original OtktDSL to address it using function names combining their module and class names. This ensures our Dynamic Architecture Recovery (DAR)[4] tool (T6 in Table 1) to correctly generate a dynamic architectural model from the received Kieker traces.

3.3 Static Analysis

Figure 1, upper left corner, illustrates the static analysis in our pipeline. Kieker's existing approach, originally designed for Java [HWH12] and Fortran [JSGH23], offers principles adaptable to Python. The existing approach for Fortran involves three steps: Fxtran translates the code into AST, Kieker's fxca converts the AST to a specific format compatible with our Static Architecture Recovery (SAR) tool (T5 in Table 1), that generates a static architectural model from the received static information. This outlines a general process:

$$\text{source code} \rightarrow \text{AST} \rightarrow \text{SAR-compatible format} \rightarrow \text{SAR tool}$$

However, the existing approach for Fortan programs required some changes to read the AST from Python code, because ASTs are language-specific. Also, SAR requires extra information, which

[4] https://github.com/kieker-monitoring/kieker/tree/main/tools/dar

we had to address in our implementation. Therefore we implemented Pyparse (T1 in Table 1).[5], which replaces Fxtran. It generates ASTs and extracts two key types of relationships: function calls and data flow. These are then passed to the SAR tool to generate static architectural model. Compared to existing Python-based static analysis approaches, our approach is distinguishable because we extract function calls and data flow from the given AST, which is complementary to our dynamic architectural model for completing the combined architectural model, which allows for analyzing both static and dynamic perspective of the target model.

3.4 Combined Analysis

Figure 1, middle, shows where the dynamic and static architectural models are combined as combined architectural model via the Model Operation MOP tool (T7 in Table 1),[6] We reused the MOP tool without modifications. However, while inspecting the existing approach with the DAR and SAR tools, we had to adapt the naming conventions in SAR to match the SAR results with the DAR results.

3.5 Software Structure Visualization

A key objective of the visualization is to layout the software structure graph to reveal the 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 improve readability, nodes needed to be grouped according to their package affiliation.

Graphviz,[7] the default visualization toolkit of Kieker, presents notable limitations in handling nested graph layouts [SNA+25]. Among its suite of layout engines, the *dot* and the *fdp* tools support nested structures. The dot tool employs a hierarchical layout that yields cleaner diagrams but struggles to scale with increasing graph complexity. Conversely, fdp uses a force-directed approach that accommodates larger graphs, though it often suffers from excessive edge crossings. Moreover, both engines reach scalability limits when faced with highly complex architectures.

To overcome these limitations, we developed the Grouped Graph Visualizer (GGVIS) tool (T3 in Table 1).[8] Our new tool is built on the Tulip framework [AAB+17], using its *tulip-python* library.[9] The requirements for our Graphviz alternative includes (1) supporting nested structures either natively or through flexible configuration, while also (2) allowing scripting and compatibility with standard formats such as GraphML and dot. Kieker's MVIS tool exports raw graphics from an architecture model in GraphML or dot format. Then, GGVIS reads the output graphics and run two algorithms in Algorithms 1 and 2 sequentially, (1) going through all nodes and cluster them hierarchically, and (2) coordinates all nested nodes in the right position recursively until the root node.

One of the main challenges in selecting a new visualization tool was that most existing solutions are either Graphviz-based, GUI-centric, or web-based. Tulip, however, met all the neces-

[5] https://github.com/kieker-monitoring/pyparse
[6] https://github.com/kieker-monitoring/kieker/tree/main/tools/mop
[7] https://www.graphviz.org/
[8] https://github.com/kieker-monitoring/GGVIS
[9] https://pypi.org/project/tulip-python/

Algorithm 1 Cluster Nodes into Hierarchical Subgraphs

```
1: clusters ← empty dictionary
2: for each node in graph do
3:     segments ← split node name by "."
4:     parent ← main graph
5:     // Create missing parent subgraphs
6:     for each segment except last do
7:         clusterName ← join segments up to current
8:         if clusterName not in clusters then
9:             Create subgraph clusterName
10:            Add to parent
11:            clusters[clusterName] ← new subgraph
12:        end if
13:        parent ← clusters[clusterName]
14:    end for
15:    // Add node to current subgraph
16:    clusterKey ← join segments except last
17:    targetCluster ← clusters[clusterKey]
18:    Add node to targetCluster
19: end for
```

Algorithm 2 Bounding Box Layout Strategy

```
1: list ← List recursively from bottom up all subgraphs
2: for each graph in list do
3:     Apply layout algorithm and store result away
4:     for each direct node of graph do
5:         Update the old coordinates to the new ones
6:     end for
7:     for each bounding box node box do
8:         Update the old coordinates to the new ones
9:         diff ← new position − old position
10:        for each node inside box do
11:            Translate node position by diff
12:        end for
13:    end for
14:    Create bounding box node For graph
15:    Add it to its parent graph
16: end for
```

sary criteria: it supports nesting, offers scripting capabilities, and handles both dot and GraphML formats effectively.

4 Evaluation

The following sections present the evaluation of the tool's execution performance and the evaluation of the visual layout algorithms. We provide a virtual machine file as a replication package where all scripts used in the evaluation are pre-configured.[10] We chose a set of Python application as our examples. NumPy, SciPy, Matplotlib, and Pillow provides the foundation for many Python libraries used by researchers [CBST24, PZK+22]. Anytree is a tree library frequently used by researchers and inspired other research software [MML+25, Far24, PC20, ALT+21, YMK+23, LFSS24]. We chose UXsim as a recent research software example published by the Journal of Open Source Software.[11] We used Graphviz version 2.43.0 and mentioned all other software versions in Table 1 and Table 2.

4.1 Pipeline Performance Evaluation

The pipeline was tested on several Python projects of various sizes. 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.

	Anytree	**UXsim**	**Pillow**	**Matplotlib**	**NumPy**	**SciPy**
Version	2.12.1	1.8.2	12.0.0	3.10.3	2.4.0	1.17.0
# files	61	67	294	919	1124	1217
Pyparse	1s	2s	10s	3m 53s	2m 57s	7m 15s
SAR	3s	6s	10s	7m 26s	10m 44s	21m 58s
DAR	2s	4s	1s	2s	*N/A*	2s
MOP	3s	6s	5s	26m 13s	*N/A*	28s
MVIS	2s	3s	2s	1m 10s	19m 22s	3s
GGVIS	1s	1s	1s	4m 28s	34s	1s
Total	12s	22s	29s	43m 12s	33m 37s	29m 47s

Table 2: Durations in minutes and seconds for the pipeline stages of Figure 1. The first column lists the stages, whose execution times where measured for the six example Python projects. NumPy could not be instrumented, thus, running the dynamic analysis DAR and MOP was not possible.

Table 2 shows execution times for six example Python projects. In Table 2, a few observations deserve to be highlighted. First, as the number of files increases, processing time of static analysis – Pyparse and SAR – increases accordingly. Instrumentation succeeded for Anytree,[12]

[10] https://doi.org/10.5281/zenodo.16735614
[11] https://joss.theoj.org/
[12] https://github.com/c0fec0de/anytree

UXsim [Seo25], and Pillow[13] without issues. Matplotlib[14] and SciPy[15] show low DAR times due to reduced data input. By design, dynamic analysis is more focused than static analysis. However, NumPy could not be instrumented due to application-level limitations. Thus, the dynamic analysis was not possible. This means we will not be able to gather dynamic information such as call counts. Finally, regarding SciPy, the MOP tool failed to process the static model. Despite this, DAR and SAR models were structurally valid and produced syntactically correct graphs via MVIS and GVIS. This explains the low processing time.

4.2 Visualization Layout Evaluation

Tools	Graph data	**UXsim**	**Anytree**	**NumPy**
fdp	ungrouped	3s	2s	1m 50s
dot	ungrouped	1m 9s	18s	$>$ 1h
fdp	grouped	6s	2s	16m 36s
dot	grouped	$>$ 1h	$>$ 1h	$>$ 1h
GGVIS	grouped	1s	1s	34s
	# nodes	66	74	573
	# edges	287	231	7915

Table 3: Execution times for different layout engines. Our new GGVIS tool supports grouping, and it runs on the Tulip framework.

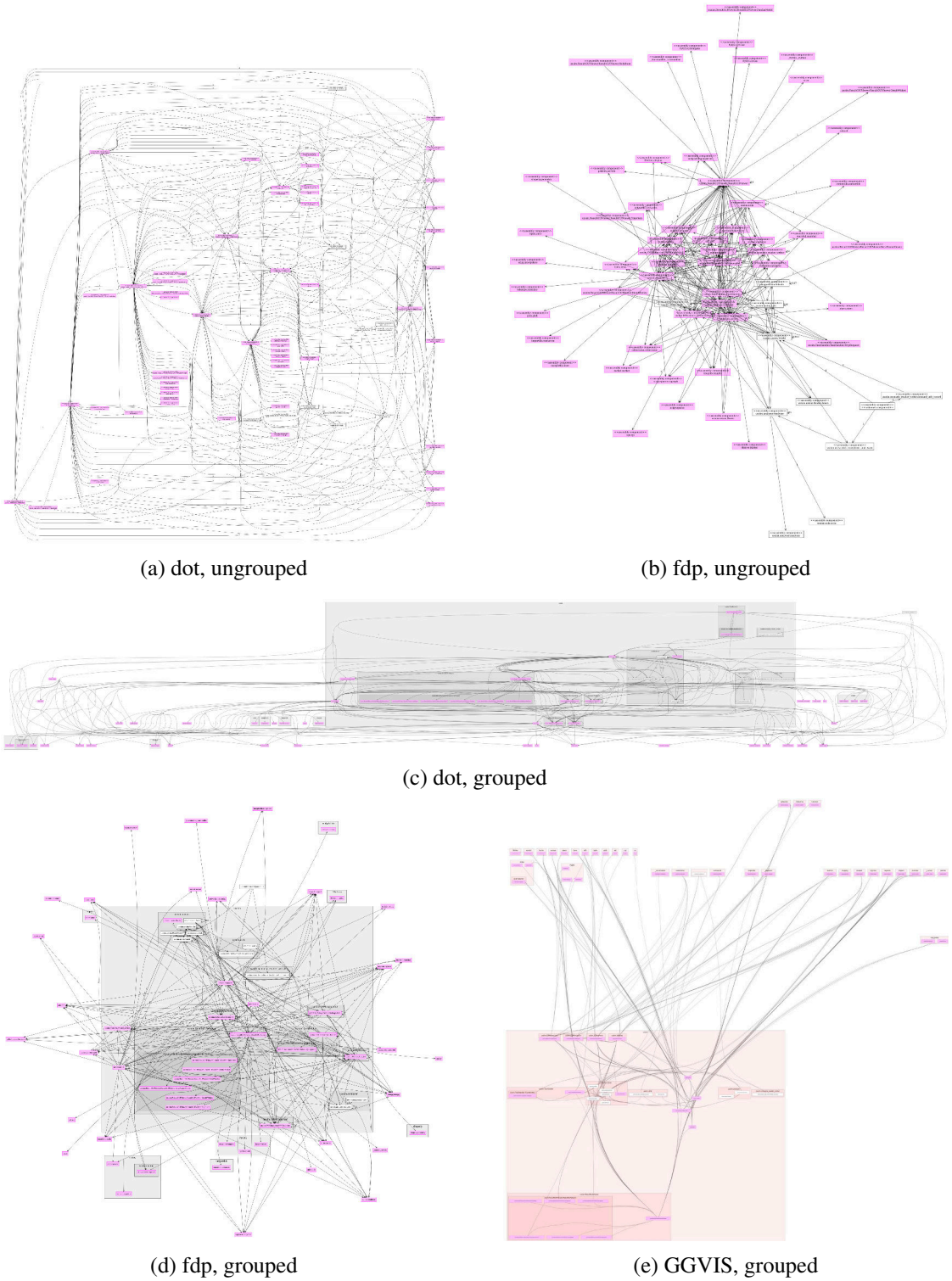
Table 3 shows the execution times for different layout engines. We compared Graphviz dot and fdp tools for visualizing ungrouped and grouped (i.e., nested) graphs, where dot and fdp are the only Graphviz tools supporting nesting, as well as our new GGVIS tool. As the graph size increases, dot shows performance issues, not being able to complete the given task within runtime limit of one hour. fdp holds up better, but still shows very high computing times when it is applied to NumPy. In detail, MVIS yields a 678 KiB dot file from analysis of the NumPy example. As reported in Table 3, GGVIS processes it in 34s, where as the Graphviz fdp tool requires more than 16m and the Graphviz dot tool overran the time out duration of 1h.

Figure 2 illustrates the differences in layout engines. It presents multiple visualizations of the same UXsim code base, generated using different layout engines. UXsim is a traffic flow simulator [Seo25]. As a first step, the relationships between components are extracted from the code base and translated into a description compatible with the layout engines.

We produce two structural variants:

- Ungrouped: all components (files) are placed at a single hierarchical level.
- Grouped: components are nested within their respective packages.

[13] https://github.com/python-pillow/Pillow
[14] https://github.com/matplotlib/matplotlib
[15] https://github.com/scipy/scipy



(a) dot, ungrouped

(b) fdp, ungrouped

(c) dot, grouped

(d) fdp, grouped

(e) GGVIS, grouped

Figure 2: Visualizations of the UXsim code base. Captions for each subfigure is composed of two information. (1) different layout engines: dot, fdp, or GGVIS. (2) groupings: the nodes in each diagram is either grouped (nested) or ungrouped. For a detailed inspection, all figures in the paper are available online [LYH26].

Both variants contain the same components, the same number for nodes and edges, but the hierarchical grouping differs. Conceptually, this corresponds to placing the same items into different box configurations, see Figures 2d and 2e.

Figures 2a to 2d use the Graphviz layout engine suite. Figure 2e uses GGVIS, our custom tool built on top of the Tulip layout engine suite. Tulip integrates and optimizes Graphviz algorithms while providing additional capabilities. Transitioning away from standard Graphviz was necessary, as its layout engines lack the computational performance required for our dataset.

- Figure 2a: dot layout engine on the ungrouped description (Sugiyama layout algorithm). Some of the relationships are short and noticeable but most of them are prolonged, making its proximity relationships difficult to interpret.
- Figure 2b: fdp layout engine on the ungrouped description (Fruchterman–Reingold layout algorithm). The proximity relationship is preserved, but the occurrences of edge-crossings over nodes, especially in the center, has to be improved.
- Figure 2c: dot layout engine on the grouped description (Sugiyama layout algorithm). Compared to Figure 2a, the grouped nodes provide more intuition, but it grows only horizontally rather than grow in both direction. The generation of diagram overran the time out duration of one hour.
- Figure 2d: fdp layout engine on the grouped description (Fruchterman–Reingold layout algorithm). The output is modest but has room to improve, e.g., decrease edge crossings and improve edge bundling.
- Figure 2e: GGVIS on the grouped description (FM^3 algorithm by Hachul and Jünger). The result is readable, proximity relationships are preserved. Figure 3 shows an extract of Figure 2e. It took one second to generate the diagram.

Figures 4 and 5 illustrate the NumPy architecture rendered using the fdp and the GGVIS tools, respectively. The rendering performance has improved significantly: fdp required 16m 38s, whereas GGVIS completes the task in approximately 30s to 60s, depending on the desired output format (e.g., PDF, SVG, PNG).

5 Conclusions and Future Work

This work extends the Kieker observability framework to improve its software-visualization support for Python applications. 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. A replication package for the presented combined analysis approach, integrating static and dynamic analysis, is available [Lar25].

Throughout the development process, some 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 (abstract syntax tree) matching was developed specifically for the pipeline. Furthermore, a more intuitive visualization method

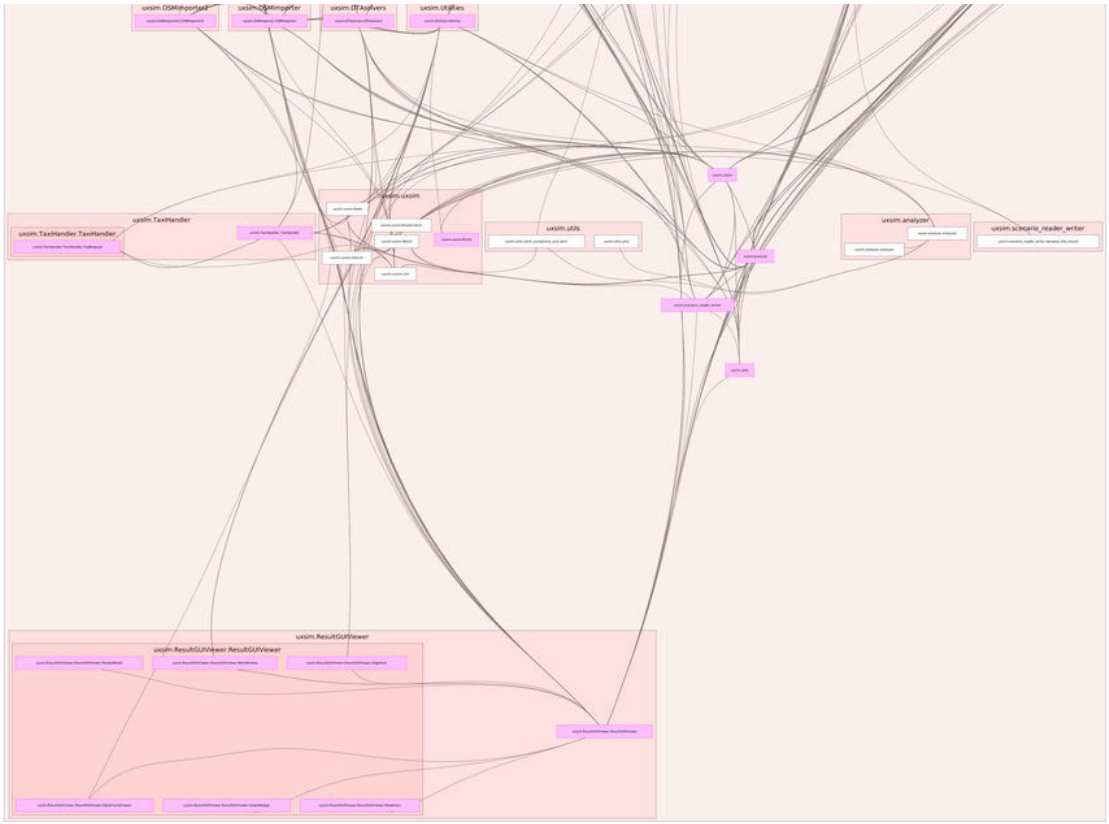


Figure 3: Extract of Figure 2e, which shows the UXsim architecture visulization, grouped with GGVIS.

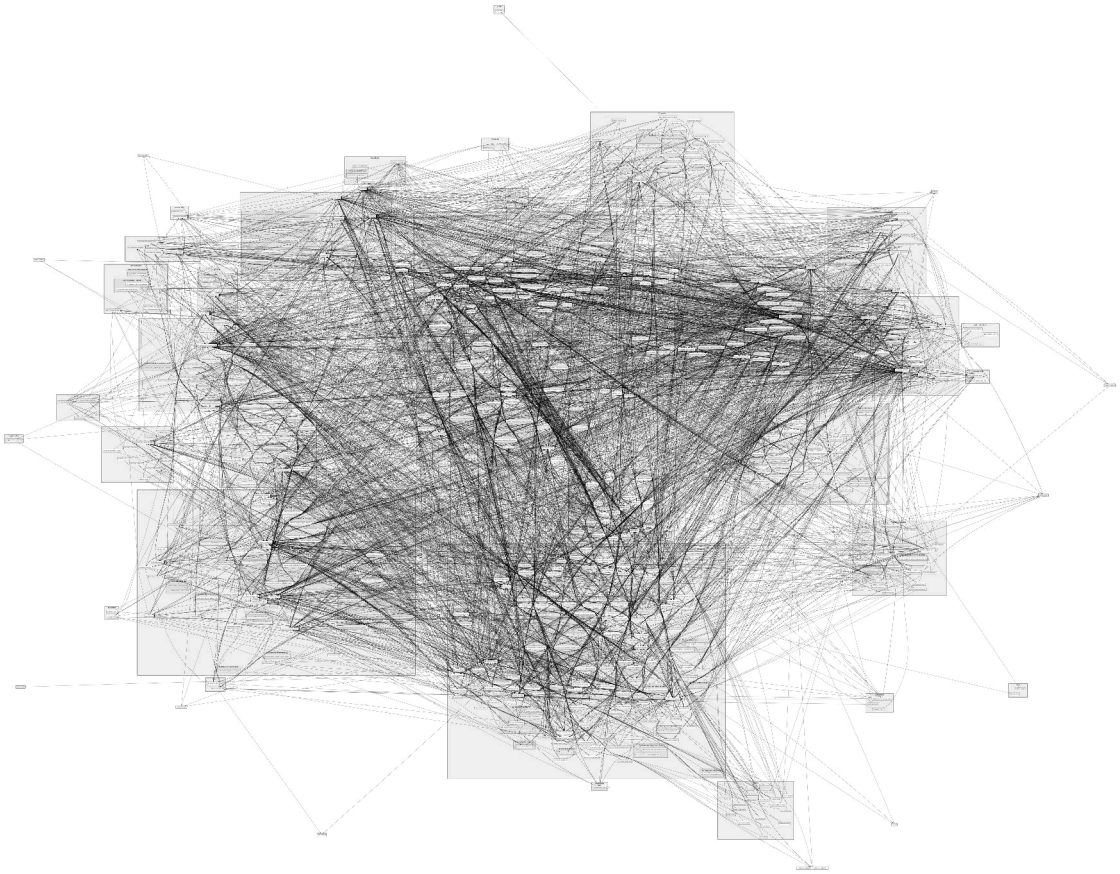


Figure 4: NumPy visualization with fdp

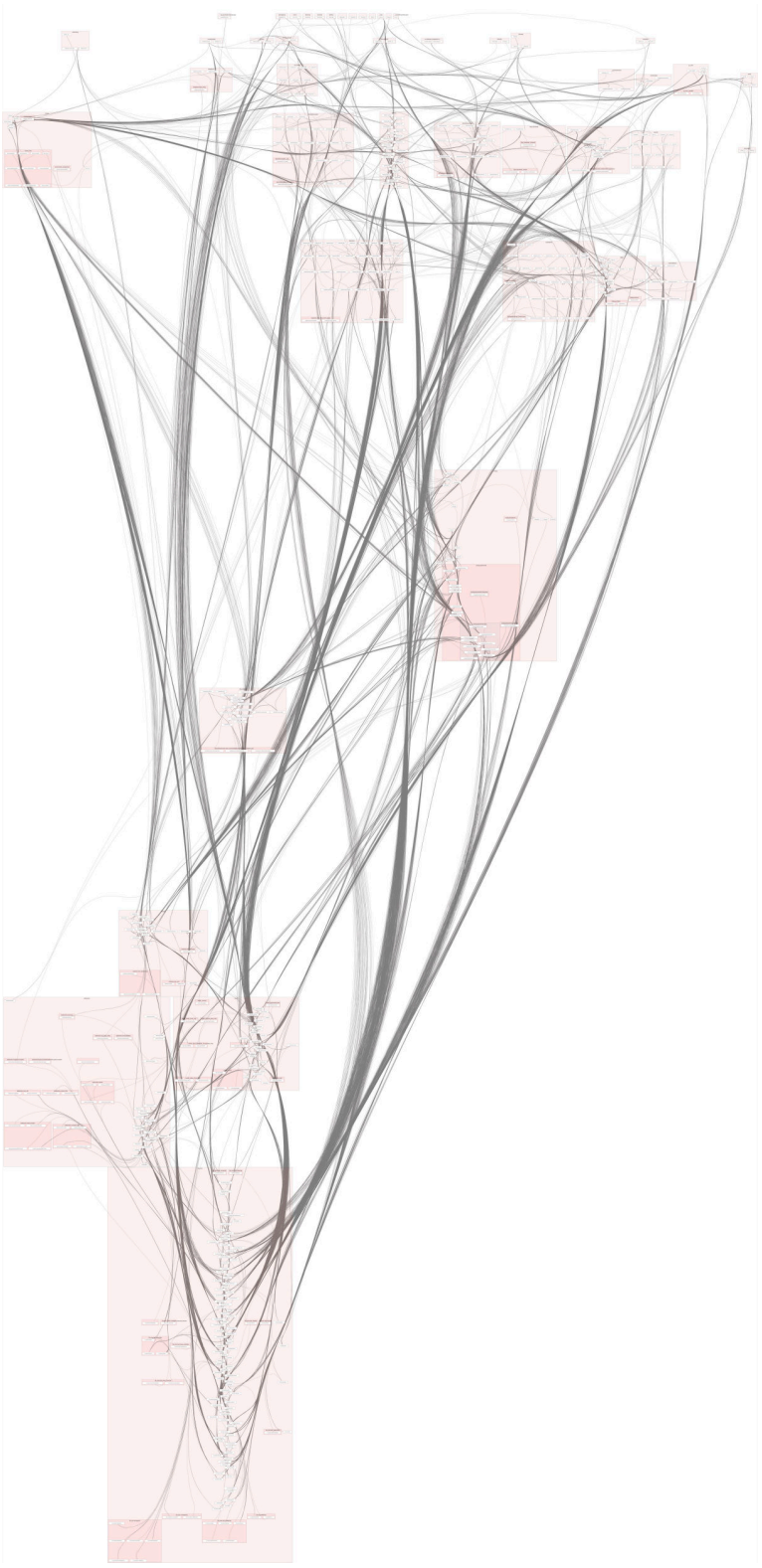


Figure 5: NumPy visualization with GGVIS

was introduced, and limitations in current layout tools were revealed, highlighting an area for future improvement.

A more detailed inspection of the Python software structures is a follow-up activity, toward reconstructing and later restructuring the software architecture for improved maintenance.

While Python has become the most popular language, there are demands for existing and new software developed with C/C++, inside and outside the RSE communities. Kieker has an early version of support for the C and C++ languages.[16] It was developed for the combined analysis of Fortran application, and the next steps include monitoring Python's C/C++ native libraries to further increase the observability of Python software.

We expect that our approach will likely become even more relevant with the increasing amount of AI-generated software in the future. For research software engineers it will become a challenge to comprehend and understand the AI-generated code, and software structure visualizations may help with this task. At the same time, it may also be beneficial to integrate AI support into our pipeline.

6 Data Availability

We published a replication package on Zenodo,[17] and vector graphics of generated analysis diagrams are also available.[18] All git repository addresses of our analysis tools on GitHub are listed in Table 1.

7 Use of AI

The authors are not aware of any AI usages in the research artifacts and manuscripts they worked on, however we don't examine the use of AI done by other contributors.

Acknowledgements: This research is funded by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. 528713834, and UK Research and Innovation (UKRI) through the UKRI Metascience Research Grants program (Reference S26368).

Bibliography

[AAB+17] D. Auber, D. Archambault, R. Bourqui, M. Delest, J. Dubois, A. Lambert, P. Mary, M. Mathiaut, G. Melançon, B. Pinaud, B. Renoust, J. Vallet. Tulip 5. In *Encyclopedia of Social Network Analysis and Mining*. Pp. 1–28. Springer, Aug. 2017. doi:10.1007/978-1-4614-7163-9_315-1

[AB05] C. Artho, A. Biere. Combined Static and Dynamic Analysis. *Electronic Notes in Theoretical Computer Science* 131:3–14, 2005. Proceedings of the First Interna-

[16] https://github.com/kieker-monitoring/kieker-lang-pack-c
[17] https://doi.org/10.5281/zenodo.16735614
[18] https://doi.org/10.6084/m9.figshare.33238413.v1

tional Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL 2005).
doi:10.1016/j.entcs.2005.01.018

[ALT+21] H. Atwi, B. Lin, N. Tsantalis, Y. Kashiwa, Y. Kamei, N. Ubayashi, G. Bavota, M. Lanza. PYREF: Refactoring Detection in Python Projects. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Pp. 136–141. 2021.
doi:10.1109/SCAM52516.2021.00025

[AZBG21] N. Al Awar, S. Zhu, G. Biros, M. Gligoric. A performance portability framework for Python. In *Proceedings of the 35th ACM International Conference on Supercomputing*. ICS '21, p. 467–478. Association for Computing Machinery, New York, NY, USA, 2021.
doi:10.1145/3447818.3460376
https://doi.org/10.1145/3447818.3460376

[BC20] J. Bispo, J. M. Cardoso. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX* 12:100565, 2020.
doi:https://doi.org/10.1016/j.softx.2020.100565
https://www.sciencedirect.com/science/article/pii/S2352711019302122

[CBST24] O. Castro, P. Bruneau, J. Sottet, D. Torregrossa. Landscape of High-Performance Python to Develop Data Science and Machine Learning Applications. *ACM Comput. Surv.* 56(3):65:1–65:30, 2024.
doi:10.1145/3617588
https://doi.org/10.1145/3617588

[CDC11] G. Canfora, M. Di Penta, L. Cerulo. Achievements and challenges in software reverse engineering. *Communications of the ACM* 54(4):142–151, Apr. 2011.
doi:10.1145/1924421.1924451

[Dab12] R. Dabrowski. On Architecture Warehouses and Software Intelligence. In *Proceedings of the 4th International Mega-Conference on Future Generation Information Technology (FGIT 2012)*. LNCS 7709, pp. 251–262. Springer, 2012.
doi:10.1007/978-3-642-35585-1_35

[DEC+25] S. Druskat, N. U. Eisty, R. Chisholm, N. Chue Hong, R. C. Cocking, M. B. Cohen, M. Felderer, L. Grunske, S. A. Harris, W. Hasselbring, T. Krause, J. Linxweiler, C. C. Venters. Better Architecture, Better Software, Better Research. *Computing in Science & Engineering* 27(2):45–57, 2025.
doi:10.1109/mcse.2025.3573887

[EEK+12] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, M. Hanus. Xbase: implementing domain-specific languages for Java. In *Proceedings of the 11th International Conference on Generative Programming and*

Component Engineering (GPCE 2012). Pp. 112–121. ACM, 2012.
doi:10.1145/2371401.2371419

[Far24] L. A. Farthofer. StreamGen: a Python framework for generating streams of labeled data. *Journal of Open Source Software* 9(104):7206, 2024.
doi:10.21105/joss.07206
https://doi.org/10.21105/joss.07206

[FLZW97] G. C. Fox, X. Li, Q. Zheng, Z. Wu. A Prototype of Fortran-to-Java Converter. *Concurr. Pract. Exp.* 9(11):1047–1061, 1997.
doi:10.1002/(SICI)1096-9128(199711)9:111047::AID-CPE348 3.0.CO;2-V
https://doi.org/10.1002/(SICI)1096-9128(199711)9:111047::AID-CPE348\protect\protect\leavevmode@ifvmode\kern+.2222em\relax3.0.CO;2-V

[GAA18] T. A. Ghaleb, M. A. Alturki, K. Aljasser. Program comprehension through reverse-engineered sequence diagrams: A systematic review. *Journal of Software: Evolution and Process* 30(11):e1965, 2018.
doi:https://doi.org/10.1002/smr.1965

[GAA20] T. A. Ghaleb, K. Aljasser, M. A. Alturki. Enhanced Visualization of Method Invocations by Extending Reverse-engineered Sequence Diagrams. In *2020 Working Conference on Software Visualization (VISSOFT)*. Pp. 49–60. 2020.
doi:10.1109/VISSOFT51673.2020.00010

[Gob14] C. Goble. Better Software, Better Research. *IEEE Internet Computing* 18(5):4–8, Sept. 2014.
doi:10.1109/mic.2014.88

[GB23] D. Gomez Blanco. *Practical OpenTelemetry: Adopting Open Observability Standards Across Your Organization*. Apress, 2023.
doi:10.1007/978-1-4842-9075-0

[GP19] H. Gulabovska, Z. Porkoláb. Towards More Sophisticated Static Analysis Methods of Python Programs. *15th IEEE International Scientific Conference on Informatics*, 2019.
doi:10.1109/Informatics47936.2019.9119307

[Has18] W. Hasselbring. Software Architecture: Past, Present, Future. In *The Essence of Software Engineering*. Pp. 169–184. Springer, Cham, 2018.
doi:10.1007/978-3-319-73897-0_10

[HDB+25] W. Hasselbring, S. Druskat, J. Bernoth, P. Betker, M. Felderer, S. Ferenz, B. Hermann, A.-L. Lamprecht, J. Linxweiler, A. Prat, B. Rumpe, K. Schoening-Stierand, S. Yang. Multi-Dimensional Research Software Categorization. *Computing in Science & Engineering* 27(2):59–68, 2025.
doi:10.1109/mcse.2025.3555023

[HH20] W. Hasselbring, A. van Hoorn. Kieker: A monitoring framework for software engineering research. In *Software Impacts*. 2020.
doi:10.1016/j.simpa.2020.100019

[HJS25a] W. Hasselbring, R. Jung, H. Schnoor. Software Architecture Evaluation of Earth System Models. *Journal of Software Engineering and Applications* 18(3):113–138, 2025.
doi:10.4236/jsea.2025.183008

[HJS25b] W. Hasselbring, R. Jung, H. Schnoor. Software Architecture Evaluation of Earth System Models. *Journal of Software Engineering and Applications*, 2025.
doi:10.4236/jsea.2025.183008

[HKN26] W. Hasselbring, D. S. Katz, R. van Nieuwpoort. Technology Research Software: An Often Overlooked Category of Research Software. *Computing in Science & Engineering* 28(2):59–68, 2026.
doi:10.1109/mcse.2026.3662117

[HWH12] A. van Hoorn, J. Waller, 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 '12, pp. 247–248. ACM, Apr. 2012.
doi:10.1145/2188286.2188326

[JLQ+16] W. Jin, T. Liu, Y. Qu, J. Chi, D. Cui, Q. Zheng. Dynamic Cohesion Measurement for Distributed System. In *Proceedings of the 1st International Workshop on Specification, Comprehension, Testing, and Debugging of Concurrent Programs (SCTDCP 2016)*. Pp. 20–26. ACM, 2016.
doi:10.1145/2975954.2975956

[JLQ+18] W. Jin, T. Liu, Y. Qu, Q. Zheng, D. Cui, J. Chi. Dynamic structure measurement for distributed software. *Software Quality Journal* 26(3):1119–1145, 2018.
doi:10.1007/s11219-017-9369-3

[JSGH23] R. Jung, H. Schnoor, S. Gundlach, W. Hasselbring. Architecture Recovery from Fortran Code with Kieker. *Softwaretechnik-Trends* 43(1):38–40, 2023. 13th Symposium on Software Performance, 7.-9. November 2022, Stuttgart.
https://dl.gi.de/handle/20.500.12116/43643

[KHGH12] H. Knoche, A. van Hoorn, W. Goerigk, W. Hasselbring. Automated Source-Level Instrumentation for Dynamic Dependency Analysis of COBOL systems. *Softwaretechnik-Trends* 32(2):45–46, May 2012.
doi:10.1007/BF03323473

[Lar25] D. Larrivain. Dynamic and Static Analysis of Python Software Replication Package VM. *Zenodo*, 2025.
doi:10.5281/zenodo.16735613

[LFSS24] K. L. Lucas, T. C. Ford, J. L. Stern, J. X. Situ. PySysML2: Building Knowledge from Models with SysML v2 and Python. In Verma et al. (eds.), *The Proceedings of the 2023 Conference on Systems Engineering Research*. Pp. 3–17. Springer Nature Switzerland, Cham, 2024.

[LYH26] D. Larrivain, S. Yang, W. Hasselbring. Vector images of Analysis Results in the Paper: Visualizing Python Software Structures via Dynamic and Static Analysis With the Kieker Framework. 8 2026.
doi:10.6084/m9.figshare.33238413.v1

[McC62] J. McCarthy. Towards a Mathematical Science of Computation. In *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*. Pp. 21–28. North-Holland, 1962.

[MDTS13] V. Markovets, R. Dabrowski, G. Timoszuk, K. Stencel. Know Thy Source Code. In *Proceedings of the 6th Balkan Conference in Informatics (BCI '13)*. Volume 1036, pp. 128–131. CEUR, 2013.
https://ceur-ws.org/Vol-1036/p128-Markovets.pdf

[MML+25] F. Madeira, N. Madhusoodanan, J. Lee, A. Eusebi, A. Niewielska, S. Butcher. Taxonomy Resolver: A Python package for building and filtering taxonomy trees. *Journal of Open Source Software* 10(106):7604, 2025.
doi:10.21105/joss.07604
https://doi.org/10.21105/joss.07604

[MYX22] L. Ma, H. Yang, J. Xu. Code Analysis with Static Application Security Testing for Python Program. *J Sign Process Syst 94, 1169–1182*, 2022.
doi:https://doi.org/10.1007/s11265-022-01740-z

[NSB+08] E. Y. Nakagawa, E. P. Machado de Sousa, K. de Brito Murata, G. de Faria Andery, L. B. Morelli, J. C. Maldonado. Software Architecture Relevance in Open Source Software Evolution: A Case Study. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*. Pp. 1234–1239. 2008.
doi:10.1109/compsac.2008.171

[PC20] M. S. Phan, A. Chessel. GeNePy3D: a quantitative geometry python toolbox for large scale bioimaging. *F1000Research* 9:1374, 2020.
doi:10.12688/F1000RESEARCH.27395.1
https://doi.org/10.12688/f1000research.27395.1

[PZK+22] F. Psallidas, Y. Zhu, B. Karlas, J. Henkel, M. Interlandi, S. Krishnan, B. Kroth, V. Emani, W. Wu, C. Zhang, M. Weimer, A. Floratou, C. Curino, K. Karanasos. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines. *SIGMOD Rec.* 51(2):30–37, July 2022.
doi:10.1145/3552490.3552496
https://doi.org/10.1145/3552490.3552496

[QGZ+15a] Y. Qu, X. Guan, Q. Zheng, T. Liu, L. Wang, Y. Hou, Z. Yang. Exploring community structure of software call graph and its applications in class cohesion measurement. *Journal of Systems and Software* 108:193–210, 2015. doi:10.1016/j.jss.2015.06.015

[QGZ+15b] Y. Qu, X. Guan, Q. Zheng, T. Liu, J. Zhou, J. Li. Calling Network: A New Method for Modeling Software Runtime Behaviors. *SIGSOFT Softw. Eng. Notes* 40(1):1–8, Jan. 2015. doi:10.1145/2693208.2693223

[QZL+14] Y. Qu, Q. Zheng, T. Liu, J. Li, X. Guan. In-depth measurement and analysis on densification power law of software execution. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*. Pp. 55–58. 2014. doi:10.1145/2593868.2593878

[RP96] L. D. Rose, D. A. Padua. A MATLAB to Fortran 90 Translator and Its Effectiveness. In Yew (ed.), *Proceedings of the 10th international conference on Supercomputing, ICS 1996, Philadelphia, PA, USA, May 25-28, 1996*. Pp. 309–316. ACM, 1996. doi:10.1145/237578.237627 https://doi.org/10.1145/237578.237627

[Sco09] M. L. Scott. *Programming Language Pragmatics (3. ed.)*. Academic Press, 2009.

[SDJG23] S. Simonov, T. Düllmann, R. Jung, S. Gundlach. Instrumenting Python with Kieker. *Softwaretechnik-Trends* 43(1), 2023. 13th Symposium on Software Performance, 7.-9. November 2022, Stuttgart. https://dl.gi.de/handle/20.500.12116/43639

[Seo25] T. Seo. UXsim: lightweight mesoscopic traffic flow simulator in pure Python. *Journal of Open Source Software* 10(106):7617, 2025. doi:10.21105/joss.07617

[SH20] H. Schnoor, W. Hasselbring. Comparing Static and Dynamic Weighted Software Coupling Metrics. *Computers* 9(2):1–21, Mar. 2020. doi:10.3390/computers9020024

[Sim24] S. Simonov. Domain Specific Language Support for Kieker and OpenTelemetry Interoperability. Master's thesis, Kiel University, 2024. https://oceanrep.geomar.de/61743/

[SMIJ21] B. Satwik, K. G. Manoj, S. R. Immadi, S. Jayaraman. Visualization of Distributed Program Execution. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. Pp. 1–6. 2021. doi:10.1109/ICCCNT51525.2021.9579575

[SMMG20] M. Schmitt Laser, N. Medvidovic, D. Minh Le, J. Garcia. ARCADE: An Extensible Workbench for Architecture Recovery, Change, and Decay Evaluation. In

ESEC/FSE 2020. 2020.
doi:10.1145/3368089.3417941

[SNA+25] B. Samodra, V. Nora, F. Arifiansyah, G. Soekidjo, M. Koyimatu. Development of Graph Generation Tools for Python Function Code Analysis. *JITK*, 2025.
doi:10.33480/jitk.v10i3.6177

[WHO17] C. Wulf, W. Hasselbring, J. Ohlemacher. Parallel and Generic Pipe-and-Filter Architectures with TeeTime. In *IEEE International Conference on Software Architecture Workshops (ICSAW)*. Pp. 290–293. 2017.
doi:10.1109/ICSAW.2017.20

[WYZH22] K. Wang, M. Yan, H. Zhang, H. Hu. Unified Abstract Syntax Tree Representation Learning for Cross-Language Program Classification. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. Pp. 390–400. 2022.
doi:10.1145/3524610.3527915

[YMK+23] G. Ybytayeva, O. Mamyrbayev, N. Khairova, N. Rizun, S. Berdali, K. Mukhsina. Creating a Thesaurus "Crime-Related Web Content" Based on a Multilingual Corpus. In Khairova et al. (eds.), *Proceedings of the 7th International Conference on Computational Linguistics and Intelligent Systems. Volume II: Computational Linguistics Workshop, Kharkiv, Ukraine, April 20-21, 2023*. CEUR Workshop Proceedings 3396, pp. 77–87. CEUR-WS.org, 2023.
https://ceur-ws.org/Vol-3396/paper6.pdf

[YRJ+25] S. Yang, D. G. Reichelt, R. Jung, M. Hansson, W. Hasselbring. The Kieker Observability Framework Version 2. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*. ICPE '25, pp. 11–15. Association for Computing Machinery, New York, NY, USA, 2025.
doi:10.1145/3680256.3721972

[ZHC15] C. Zirkelbach, W. Hasselbring, L. Carr. Combining Kieker with Gephi for Performance Analysis and Interactive Trace Visualization. *Softwaretechnik-Trends* 35(3):26–28, Nov. 2015. (Symposium on Software Performance 2015).
https://dl.gi.de/handle/20.500.12116/40767

[ZOLL11] Q. Zheng, Z. Ou, L. Liu, T. Liu. A Novel Method on Software Structure Evaluation. In *Proceedings of the 2nd IEEE International Conference on Software Engineering and Service (ICSESS '11)*. Pp. 251–254. 2011.
doi:10.1109/ICSESS.2011.5982301