ITM Web of Conferences **85**, 01013 (2026) https://doi.org/10.1051/itmconf/20268501013
ISSF 2026

Code Visualizer

Sai Manoj Veera Reddy Challa[1], *Suhas Varma* Rudraraju[1], *Srinivas Reddy* Bellapure[1], *Raj Kumar* Dharavath[1] and *Padma Vijetha Dev* Bakkaiahgari[1]

[1]Dept. of Computer Science and Engineering, Gokaraju Rangaraju Institute of Engineering and Technology, Hyderabad, India

Abstract. Grasping the runtime behaviour of software, especially how data evolves and control flows, is essential but becomes progressively difficult as software complexity increases. Conventional debugging approaches, such as using print statements or text-based debuggers, frequently fail to deliver adequate insight into the dynamic states of intricate data structures like nested collections, custom objects, and their interconnections. This constraint can hinder effective debugging for developers and present considerable challenges for students acquiring programming concepts. To address these issues, this paper introduces the Code Visualizer, an interactive, web-based tool aimed at offering a clear, step-by-step visualization of Python program execution. The visualizer enables users to carefully track changes in variables, scopes, and the call stack in real time. It visually displays fundamental Python data structures such as lists, dictionaries, sets, deques, and user-defined objects, along with their memory references, thereby elucidating concepts like object identity and aliasing. The architecture of the system is client-server based, featuring a backend built with Python Flask that utilizes Python's bdb (Basic Debugger) module for detailed execution tracing, alongside a responsive frontend created using HTML (HyperText Markup Language), CSS (Cascading Style Sheets), JavaScript, and SVG (Scalable Vector Graphics) for vibrant visualization.

1 Introduction

Understanding the runtime behavior of software systems, i.e., the evolution of data and the flow of control, is one of the most important aspects of programming. This understanding, however, becomes increasingly harder as the software system under consideration gets more and more complex. Existing debugging strategies, such as the use of print statements and textual-based debuggers in Integrated Development Environments (IDEs), fail to provide the desired level of insight into the runtime behavior of complex data structures such as nested data structures and object aliasing. This lack of insight poses a major problem for students who are trying to grasp programming concepts and for software developers[1] who try to debug their software systems. Though recent advances in Computer Science Education (CSE) tools have provided a platform for the visualization of software systems, these tools have several shortcomings. For example, tools such as CodeLens [2] provide excellent static visualization of software systems but fail to show the evolution of software systems during the execution of the system. Recent advances in the visualization of software debugging, such as the one proposed by Kräuter et al. (2024) [3], though very promising, are limited to specific languages and platforms. Recent advances in Large Language Models (LLMs) have provided a new dimension to the debugging and visualization of software systems by using AI not only to write the software system but also to help in the visualization of the software system by translating the software system written in different languages into a unified backend.

To overcome these challenges, this paper presents the Code Visualizer, an interactive web-based tool aiming to provide a clear, step-by-step view of program execution. Unlike previous tools, our tool takes advantage of a hybrid method using the bdb module in Python to trace program execution deterministically, with the incorporation of an LLM to translate inputs (Pseudocode, Java, C++) into executable code to visualize execution. The tool also specifically visualizes basic data structures, as well as their memory reference links, to clarify abstract program concepts such as object identity. The contributions of this paper are:

1. A client-server visualization tool that can dynamically visualize complex data structures and their object references.
2. A novel method to incorporate an LLM to translate inputs (Java, C++, Pseudocode) into a visualizable Python code to significantly extend the tool's potential in education.
3. A two-mode visualization tool for the program's call stack with timeline cards and historical trees.
4. A web interface that can be configured to reduce cognitive load, unlike traditional IDE debuggers.

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

2 Literature Review

Several strategies have been proposed for code visualization and debugging. The following section discusses existing tools to situate our contribution on Code Visualizer.

2.1 IDE-Integrated and language-specific visual debuggers

Early prototypes like the functional program visualizer by Whitington and Ridge [1] laid the groundwork for visual debugging. A number of tools have been proposed that extend conventional IDEs or focus on particular programming languages. Uke et al. [4] proposed a tool for error resolution in C++, which is a visual debugger. Kobayashi et al. [5] proposed an extension to VS Code for Java programming, called TEDVIT. The tool allows instructors to define their own rules for visualization. However, it is restricted to language-specific interfaces such as JDI. Lee et al. [6] proposed dpvis, which is a Python package for dynamic programming visualization. The tool is restricted to Python programming and requires knowledge of Python programming. Faust et al. [7] proposed Anteater, which is restricted to Python programming using an AST-based approach.

The proposed Code Visualizer is language-independent and web-based, which is facilitated by LLM translation.

2.2 Web-based and Educational Visualizers

Significant numbers of visualizers are web-based applications. Arora et al. developed the "Graph Algorithm Visualizer" in [8], which focuses on the implementation of pathfinding algorithms. Kulkarni et al. developed the "Algorithm Visualizer" for sorting and graph algorithms in [9]. Ugile et al. conducted an analysis on the design requirements in [10] for such applications. Although useful, they mostly focus on the implementation of algorithms rather than the execution.
Libraries such as EasyTracker developed by Barollet et al. in [11] provide assistance with the control of execution for Python and C code. Other applications such as CrossCode developed by Hayatpur et al. in [12] enable the user to traverse through different levels of execution abstraction. CodeVisions developed by Ahmed and Mohammed in [13] assist developers of educational software using static analysis. Other recent contributions include P-Inti developed by Halaharvi in [14], Sorting Visualizer developed by Shahaba et al. in [15], and algorithm visualizations in general by Simoňák in [16]. The novelty of our tool is the focus on the execution of the code in general rather than the implementation of algorithms, and the use of LLMs to bridge the language gaps.

3 System architecture

The architecture of Code Visualizer is a client-server based, and it is shown in Fig. 1. This system has a Python, Flask web framework-based backend and a dynamic HTML (HyperText Markup Language), CSS (Cascading Style Sheets), JavaScript frontend executed on the user's web browser.

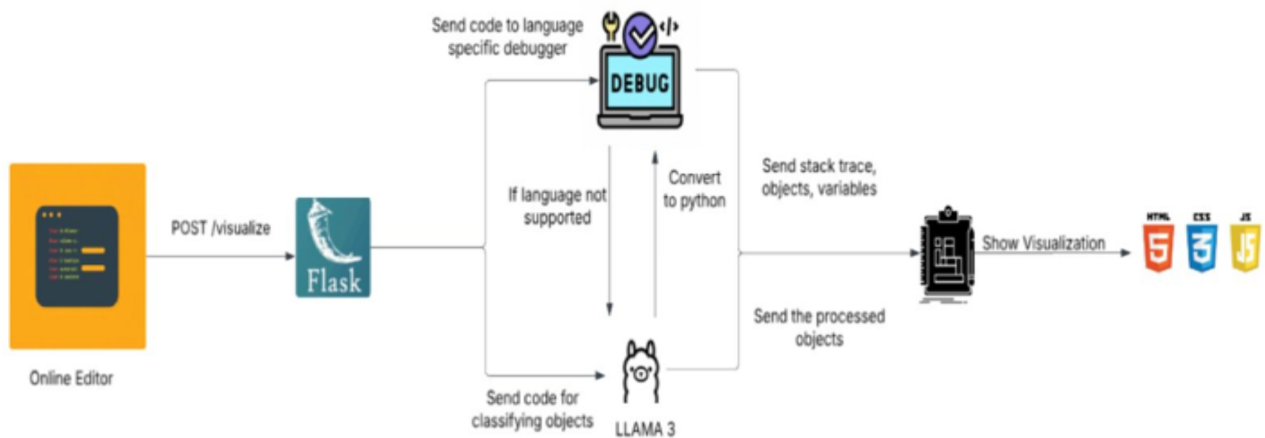

Fig. 1. System Architecture

3.1 Backend design and implementation

The code in the backend is implemented using the Flask framework with Python. The code in the

backend can essentially be regarded as the engine used in the execution of code and sophisticated processing of data in a manner that can allow for visualization. The code in the backend has a custom debugger class named WebDebugger, which is a subclass of the basic debugger, namely bdb. By this inheritance mechanism, the system can tap into the execution process of code, which is performed by the interpreter. The system overrides three very important methods of the basic debugger. The user_line() method of the basic debugger is overridden in the code in the backend. This method is called before each statement in the code is executed. The current location in the code (line NO.), the current state of local variables, global variables, and the results obtained from the execution of code in the frame of execution, which is very important for visualization, are recorded. The user_call() method of the basic debugger is overridden in the code in the backend. The user_call() method is used in the recording of functions. The user_return() method of the basic debugger is overridden in the code in the backend. The user_return() method is used in the recording of functions. The methods for checking the variables to be displayed in a better manner for the purpose of visualization of program data have also been incorporated in the code in the backend. The _is_visualizable_candidate(value) function browses through the variables. The candidates with the appropriate Python data types, namely lists, dictionaries, sets, deques, and classes, are identified. The HTML markers in the string returned by the _safe_repr function, which indicates that the object points to another object that is visualizable, can also be interpreted by the frontend to draw arrows between the different data structures.

Simultaneously, the backend also carefully maintains the tracking of function call events to gather the necessary information to construct the call stack visualization.

3.2 Frontend Design and Implementation

The frontend part, represented as index.html, provides a facility to create an interactive environment to input the code and execute it along with the visualization display facility. The user interface consists of major and important resizable panels. An ACE editor represents a section for writing the code. The "Visualization Panel" represents the major part where visualization of the data structures and their associations occurs. Secondly, other supporting panels include "Output Panel," "Variables Panel," and "Call Stack Panel," which help to display the results and the current values of local and global variables, respectively.

One of the major operations that the frontend performs includes the visualization implementation. To illustrate object linking concepts like object referencing, represented by metadata provided by backend communication, the drawArrow(.) function creates paths to draw arrows that link all these HTML tags. The arrows can change dynamically based on the visualization location changes made by user intervention.

4 Results and Discussions

This section includes the use cases of the Code Visualizer tool. It also includes a comparison with the existing state-of-the-art solutions.

4.1 Illustrative Use Cases

4.1.1 Execution of Basic Python Programs and Data Structures

To explain the basic execution visualization of the tool, let's take a look at a basic Python program that initializes a list as well as a dictionary. As depicted in Fig. 2, the user interface of the Code Visualizer tool during the execution phase of the initialization code snippet clearly illustrates the execution state. The Variables Panel accurately shows the variables my_list and my_dict, thereby confirming that the interpreter has successfully executed the code snippet.

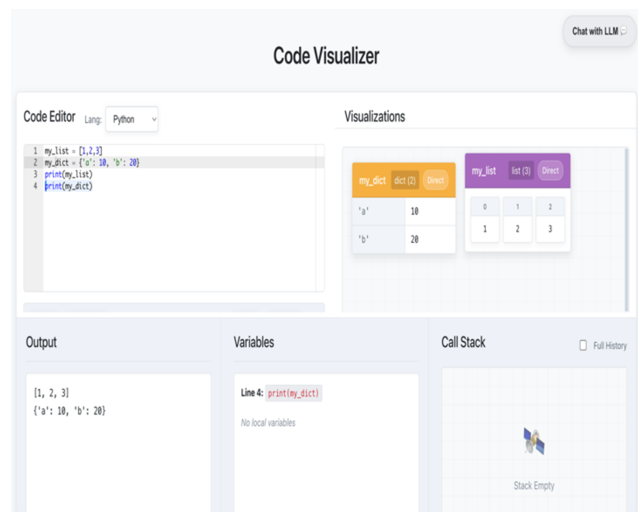

Fig. 2. Lists and Dicts

4.1.2 Execution of Complex Object References

To explain the visualization of complex object references, let's take a look at a Python program that includes a basic Node class to implement a linked list. Fig. 3. illustrates the visualization of complex object references using a basic linked list implementation. Once the structure is established as head→n1→n2→n3, the visualization tool accurately shows n1, n2, and n3 as separate objects in the visualization panel. The visualization tool clearly illustrates that these objects exist independently. The directed arrows from the next attribute of n1 to n2 accurately show that these objects are not copied; rather, they are referenced.

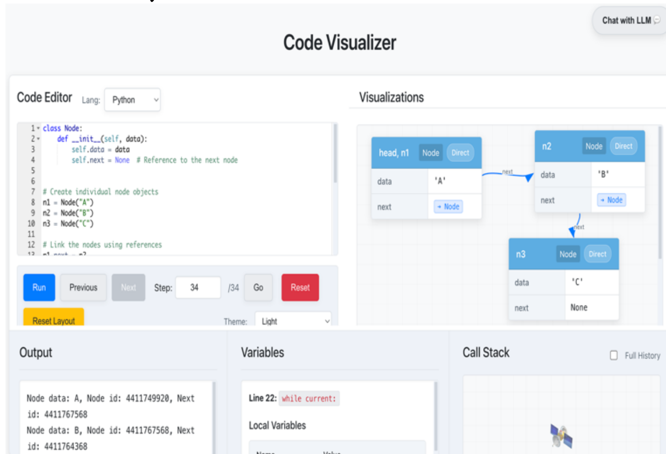

Fig. 3. Linked List

4.1.3 Call stack visualization

Fig. 4. accurately illustrates the "Full History" view of the execution of the entire code snippet to compute fibonacci(3). The SVG code accurately illustrates a tree diagram to show all function calls.

4.2 Comparative Analysis with Existing Tools

In order to affirm the added value of Code Visualizer, a comparison have been made with other important existing tools that have been discussed in the literature.

4.2.1 Dynamic vs. Static Visualization

Indeed, most existing tools like CodeLens [2] focus on static code analysis and/or static execution diagrams generation. Though these tools can be very efficient for overviewing the code's dependencies, they do not offer the level of precision provided by Code Visualizer in terms of state evolution during execution time. Code Visualizer provides a dynamic visualization of the state after each line's execution, like a "movie" representing the evolution of the program's memory, similar to the one used in EasyTracker [11], but with a modernized SVG-based rendering engine.

4.2.2 Language Flexibility via LLMs

One of the most important drawbacks of recent tools like TEDVIT [5] or the OCaml Visualizer [1] is that they are restricted to a single language. As shown in Section III, our system offers a unique feature: it uses an LLM to translate the input language (e.g., C++ or Pseudocode) to Python before visualization. This allows our tool to be a logic visualization tool for

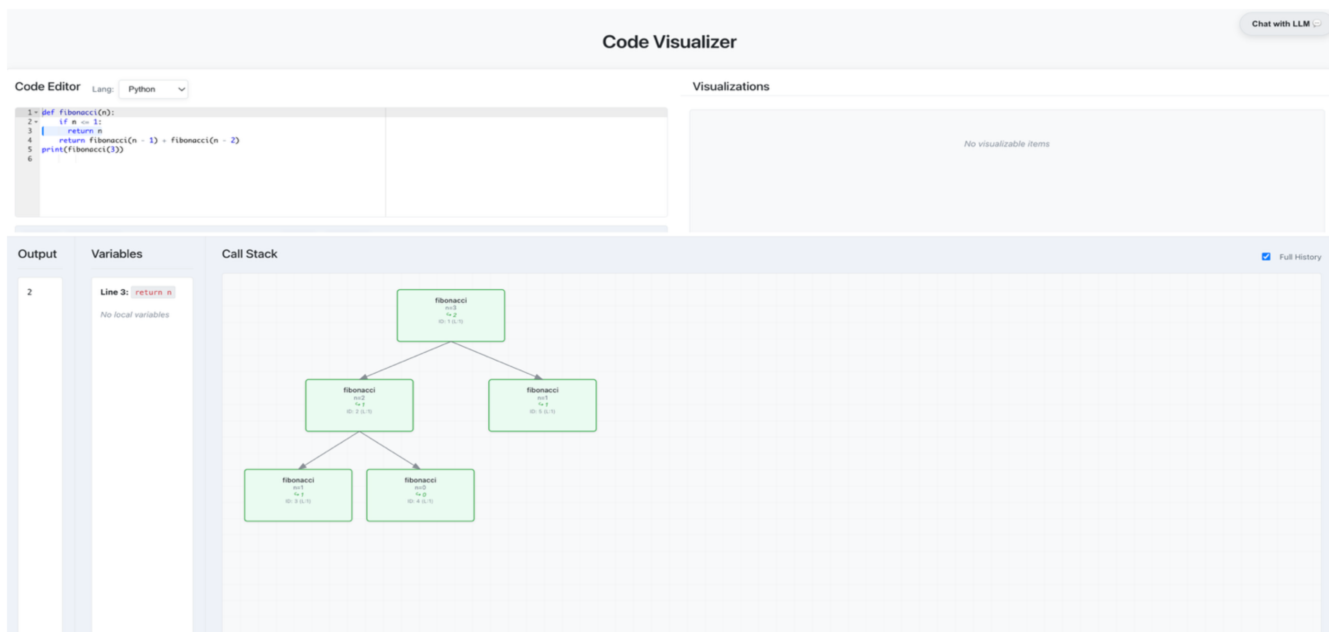

Fig. 4. Call Stack

multilingual classrooms, which is not the case for common execution tracing tools.

4.2.3 Cognitive Load

Standard IDE debuggers offer a very high level of precision but present a high cognitive load due to their complex interfaces. As shown in research on CrossCode [12], it appears that multi-level visualization can be very efficient in reducing this load. Our user interface follows this principle by separating the Visualization Panel and the Call Stack (historical tree view). The SVG-based historical call stack offers a visual representation of the recursion history, which can be more interesting for educational purposes than the stack view found in common debuggers.

5 Conclusion

In this paper, Code Visualizer, an interactive web-based tool, was introduced with the aim of filling the gap between static code analysis and dynamic execution. It was achieved through the integration of a debugger built on top of the Flask web framework and an SVG-based frontend. The tool has managed to successfully visualize complicated relationships, such as object aliasing and call stacks, which are normally difficult to understand through text-based debuggers. From the comparative analysis, it was established that although the conventional debuggers are good at executing code, Code Visualizer has managed to differentiate itself through the support of multi-language logic visualization through the integration of LLM, which enables the tool to visualize Java or Pseudocode logic through proxy. This has made the tool suitable for university-level algorithms classes, where logic is more important than syntax. Future plans aim to improve the rendering engine for large-scale data structures and the LLM prompts for edge cases.

References

Journal articles

1. J. Whitington, T. Ridge, Visualizing the Evaluation of Functional Programs for Debugging, arXiv preprint arXiv:2411.00618 (2024)
2. Y. Guo, S. Bettaieb, Q. Hu, Y. Le Traon, Q. Tang, CodeLens: An Interactive Tool for Visualizing Code Representations, arXiv preprint arXiv:2307.14902 (2023)
3. T. Kräuter, P. Stünkel, A. Rutle, Y. Lamo, The Visual Debugger: Past, Present, and Future, in *DE '24* (Lisbon, Portugal, 2024)
4. S. Uke, A. Patankar, S. Patwardhan, S. Phand, A. Patil, A Comprehensive Visual Debugger for Error Resolution in C++, in *2024 15th ICCCNT* (2024)
5. K. Kobayashi et al., Program Learning Support System with Visualization Reflecting Teacher's Intent, in *Proc. 32nd Int. Conf. Comput. Educ. (ICCE)* (2024)
6. D. H. Lee et al., dpvis: A Visual and Interactive Learning Tool for Dynamic Programming, in *Proc. 56th ACM Tech. Symp. Comput. Sci. Educ.* **1**, 1–14 (2025)
7. R. Faust, K. Isaacs, W. Z. Bernstein, M. Sharp, C. Scheidegger, Anteater: Interactive Visualization of Program Execution Values in Context, arXiv preprint arXiv:1907.02872 (2024)
8. S. Arora, K. C. Tripathi, M. L. Sharma, Graph Algorithm Visualizer, *Iconic Research and Engineering Journals* **6**, 150–153 (2022)
9. A. Kulkarni, S. Padave, S. Shrivastava, V. Kawtikwar, Algorithm Visualizer, *Int. J. for Research in Applied Science and Engineering Technology* **11**, 1818–1823 (2023)
10. R. L. Ugile, R. B. Barure, G. S. Sawant, L. Malphedwar, Review article on the visualization of algorithms, *IRJMETS* **4**, 1986–1988 (2022)
11. T. Barollet et al., EasyTracker: A Python Library for Controlling and Inspecting Program Execution, in *Proc. CGO* (Edinburgh, UK, 2024), 1–14
12. D. Hayatpur, D. Wigdor, H. Xia, CrossCode: Multi-level Visualization of Program Execution, in *Proc. 2023 CHI Conf. Human Factors Comput. Syst.*, 1–13 (2023)
13. Z. Ahmed, M. Mohammed, CodeVisions: Static Code Analysis for Creating Education-Oriented Applications, in *Proc. ACM Tech. Symp. Comput. Sci. Educ.* (2023)
14. S. Halaharvi, P-Inti: Interactive Visual Representation of the Programming Concepts for Learning and Instruction, M.S. thesis, Univ. of Victoria (2023)
15. S. Shahaba et al., Sorting Algorithm Visualizer, *Int. J. Creative Res. Thoughts* **12**, e854–e860 (2024)
16. S. Simoňák, Algorithm visualizations as a way of increasing the quality in computer science education, in *2016 IEEE 14th SAMI* (2016), 153–157