World Journal of Advanced Research and Reviews
eISSN: 2581-9615 CODEN (USA): WJARAI
Cross Ref DOI: 10.30574/wjarr
Journal homepage: https://wjarr.com/

(REVIEW ARTICLE)

Check for updates

Universal Java code execution tracer

R. Rajesh, Jetti Charitha *, Gunaganti Chandana and Pulimamidi Balu Shashank chary

Department of Computer Science and Engineering (Artificial Intelligence and Machine Learning), ACE Engineering College, Hyderabad, Telangana — 501 301, India.

World Journal of Advanced Research and Reviews, 2026, 30(01), 1132-1141

Publication history: Received on 24 February 2026; revised on 06 April 2026; accepted on 08 April 2026

Article DOI: https://doi.org/10.30574/wjarr.2026.30.1.0850

Abstract

Understanding program execution and analyzing algorithm efficiency are essential yet challenging tasks for students and developers. The system provides a multi-panel interface consisting of a code editor, a node-based graphical visualization of execution flow, and a step-by-step textual explanation of program behavior. The proposed system processes the input code only after it is fully written and syntactically correct, ensuring accurate parsing and reliable output. Once validated, the system will identify control structures such as loops, conditional statements, and recursion, and generate a node-based execution graph that illustrates the program flow dynamically. Each node represents a computational step, enabling users to trace execution in a structured and visual manner.

In addition, the system incorporates an automated complexity analyzer that determines time complexity (e.g., $O(n^2)$) and space complexity (e.g., $O(1)$) based on detected patterns and nesting levels. The visualizer supports both general Java programs and Data Structures and Algorithms (DSA), making it suitable for educational, debugging, and analytical purposes. Unlike existing tools, the proposed system integrates node-based visualization, real-time execution tracing, and automated complexity analysis in a single platform supporting both general Java programs and DSA problems.

Keyword: Java Code Visualization; Node-Based Visualization; Time Complexity; Space Complexity; Data Structures and Algorithms (DSA).

1. Introduction

Understanding how a program executes internally is a fundamental aspect of learning programming and algorithm design. However, many students and beginners face difficulty in visualizing the flow of execution, especially when dealing with complex constructs such as loops, nested structures, and recursion. Traditional learning approaches primarily rely on static code examples and theoretical explanations, which often fail to provide an intuitive understanding of how programs behave during runtime.

To address this challenge, code visualization tools have been introduced to bridge the gap between theory and practice. These tools help users observe program execution step-by-step, improving comprehension and debugging skills. However, most existing solutions either support limited programming constructs, lack interactive visualization, or do not provide integrated analysis of time and space complexity.

In this paper, we propose a Java Code Visualizer, an interactive web-based system designed to enhance program understanding through a multi-panel interface. The system allows users to write Java code in a dedicated editor and generates a node-based graphical representation of execution flow. Each node corresponds to a specific computational step, enabling users to track how the program progresses. In addition, a textual execution panel provides a detailed

* Corresponding author: Jetti Charitha

step-by-step explanation, while an integrated analyzer computes time and space complexity based on detected control structures.

A key feature of the proposed system is that visualization is generated only after the input code is completely written and syntactically correct, ensuring accurate parsing and reliable results. The system supports both general Java programs and Data Structures and Algorithms (DSA), making it suitable for educational and analytical purposes.

By combining graphical visualization, textual explanation, and complexity analysis, the proposed system provides a comprehensive platform for understanding program execution and algorithm efficiency, thereby improving learning outcomes and problem-solving skills.

1.1. Motivation

Understanding the execution of programs is a critical skill in computer science education, particularly for students learning Java and Data Structures and Algorithms (DSA). Many learners struggle to grasp how code executes step-by-step, especially when dealing with complex constructs such as nested loops, conditional statements, and recursion. Traditional teaching methods often rely on static examples, which do not effectively demonstrate dynamic program behavior. This gap in understanding can lead to difficulties in debugging, optimizing code, and analyzing algorithm efficiency. Therefore, there is a strong need for an interactive system that visually represents program execution and helps users intuitively understand how their code works.

1.2. Problem Definition

Despite the availability of various programming tools and compilers, there is a lack of integrated systems that provide a comprehensive view of program execution along with complexity analysis. Existing tools either focus only on code execution or provide limited visualization capabilities without supporting both general Java programs and DSA concepts. Additionally, many tools do not offer node-based visualization or step-by-step textual explanations, making it difficult for users to trace execution flow clearly. Another challenge is ensuring that the visualization is accurate and reliable, which requires proper validation of input code before processing. Hence, there is a need for a system that combines code visualization, execution tracing, and complexity analysis in a unified platform.

1.3. Proposed Solution Overview

To address the identified challenges, this paper proposes a Java Code Visualizer that provides an interactive and user-friendly environment for analyzing program execution. The system features a multi-panel interface consisting of a code editor, a node-based graphical visualization of execution flow, and a textual panel that displays step-by-step execution details. The input code is first validated to ensure syntactic correctness, after which it is parsed into an Abstract Syntax Tree (AST). Based on this structure, the system generates execution nodes representing individual operations and connects them to form a visual graph. Additionally, the system includes a complexity analyzer that estimates time and space complexity by examining control structures and nesting levels. This integrated approach enables users to better understand program behavior, making the system useful for both educational and debugging purposes.

2. Literature Review

2.1. Online Python Tutor – Interactive Program Visualization

- Authors: Philip J. Guo
- Journal/Publisher: ACM SIGCSE
- Year: 2013

2.1.1. Methodology

- Executes code step-by-step in browser
- Tracks variable states and memory
- Displays stack and heap visualization
- Uses execution tracing and AST-based interpretation

2.1.2. Advantages

- Highly interactive and beginner-friendly
- Clear visualization of variables and memory

- Supports multiple languages (Python, Java, C++)

2.1.3. Disadvantages

- Limited support for complex programs
- No node-based graph visualization
- Does not provide time/space complexity analysis

2.1.4. Results / Performance

- Widely used educational tool
- Improves student understanding significantly

2.1.5. Relevance to Project

- Provides inspiration for step-by-step execution
- Your system improves it with node-based visualization + complexity analysis

2.2. Jeliot 3 – Program Visualization System for Java

- **Authors:** Mikko Kuittinen et al.
- **Journal/Publisher:** ACM / University of Joensuu
- **Year:** 2006

2.2.1. Methodology

- Visualizes Java program execution
- Shows object creation and method calls
- Uses animation-based visualization

2.2.2. Advantages

- Specifically designed for Java
- Good for understanding object-oriented concepts
- Visual representation of execution

2.2.3. Disadvantages

- Outdated UI and limited interactivity
- No node-based execution graphs
- No automatic complexity analysis

2.2.4. Results / Performance

- Effective for teaching Java basics
- Used in academic environments

2.2.5. Relevance to Project

- Closest to your project domain (Java)
- Your system improves it with:
- Modern UI
- Node-based visualization

2.3. Algorithm Visualization Tool – VisuAlgo

- **Authors:** Steven Halim
- **Journal/Publisher:** National University of Singapore
- **Year:** 2012

2.3.1. Methodology

- Pre-built animations for DSA algorithms

- Step-by-step visualization of operations
- Focus on algorithm understanding

2.3.2. Advantages

- Excellent for learning DSA
- Clean and interactive UI
- Supports many algorithms

2.3.3. Disadvantages

- Does not accept custom code input
- Limited to predefined algorithms
- No code execution tracing

2.3.4. Results / Performance

- Widely used by students worldwide
- Improves algorithm learning

2.3.5. Relevance to Project

- Strong DSA visualization reference

2.4. Code Visualization using Abstract Syntax Trees (AST)

- **Authors:** Aho et al.
- **Journal/Publisher:** Compiler Design (Book/Research)
- **Year:** 2007

2.4.1. Methodology

- Converts code into AST representation
- Traverse's nodes to analyze structure
- Used in compilers and static analysis

2.4.2. Advantages

- Accurate representation of code structure
- Enables deeper analysis (loops, recursion)
- Foundation for code visualization

2.4.3. Disadvantages

- Complex to implement
- Not directly user-friendly

2.4.4. Results / Performance

- Industry standard for compilers
- High accuracy in parsing

2.4.5. Relevance to Project

- Core technique used in your system

2.5. Program Visualization: A Systematic Literature Review

- **Authors:** Marcelo M. T. da Silva et al.
- **Journal/Publisher:** IEEE / ACM Digital Library
- **Year:** 2019

2.5.1. Methodology

- Comprehensive review of program visualization tools
- Analysis of different visualization techniques (graphs, animations, memory views)
- Evaluation based on usability, learning impact, and performance
- Categorization of tools into educational and debugging systems

2.5.2. Advantages

- Provides detailed comparison of existing visualization techniques
- Identifies strengths and weaknesses of different tools
- Helps in selecting suitable approaches for system design

2.5.3. Disadvantages

- Does not provide a specific implementation
- Lacks practical system development
- No direct support for real-time execution visualization

2.5.4. Results / Performance

- Concludes that visualization significantly improves learning outcomes
- Highlights need for interactive and integrated systems
- Suggests combining multiple visualization techniques

2.5.5. Relevance to Project

- Supports the need for advanced visualization tools

3. System Architecture

The proposed Java Code Visualizer is designed using a modular system architecture to provide efficient code analysis, execution tracing, and visualization. The architecture consists of three main layers: the User Interface Layer, the Processing and Analysis Layer, and the Visualization and Output Layer. These layers work together to convert user-written Java code into an understandable visual and textual execution flow along with time and space complexity analysis.

3.1. User Interface Layer

The User Interface Layer is responsible for interaction between the user and the system. It provides a code editor on the left side, where users can enter Java programs. The interface is designed to be simple and interactive so that users can write, edit, and submit code easily. This layer also displays the final output, including the node-based visualization, step-by-step textual explanation, and complexity analysis. The main purpose of this layer is to provide a smooth environment for code entry and result observation.

3.2. Processing and Analysis Layer

The Processing and Analysis Layer acts as the core engine of the system. Once the user submits the Java code, this layer performs syntax validation to ensure correctness. After validation, the code is parsed into an Abstract Syntax Tree (AST), representing the structural hierarchy of the program. Using the AST, the system identifies key constructs such as loops, conditional statements, function calls, and recursion. Based on this analysis, the execution sequence is generated step-by-step, and the system evaluates algorithmic behavior to estimate time and space complexity.

3.3. Visualization and Output Layer

The Visualization and Output Layer present the analyzed execution flow in an understandable format. It converts each execution step into nodes and connects them with directed edges to represent control flow, creating a node-based graphical visualization. Along with the graphical output, it also provides a textual explanation of each execution step. The system displays time and space complexity at the top, helping users understand code efficiency. This layer ensures that both beginners and advanced learners can easily interpret program behavior.

3.4. Working Flow of the System

The working of the proposed system follows a sequence of steps. First, the user writes Java code in the editor. Next, the system validates the code to check for syntax correctness. Once validated, the code is parsed into an AST. Then, the execution steps are extracted and converted into node representations. After that, the complexity analyzer estimates time and space complexity. Finally, the system displays the node-based graph, the textual execution explanation, and the complexity results on the interface.

3.5. Architectural Benefits

The modular architecture of the proposed system offers several advantages. It separates code input, processing, and visualization into independent layers, making the system easier to design, manage, and extend. It also improves accuracy because only syntactically correct code is processed for visualization. In addition, the architecture supports both normal Java programs and Data Structures and Algorithms (DSA) code, making the system useful for education, debugging, and performance analysis.

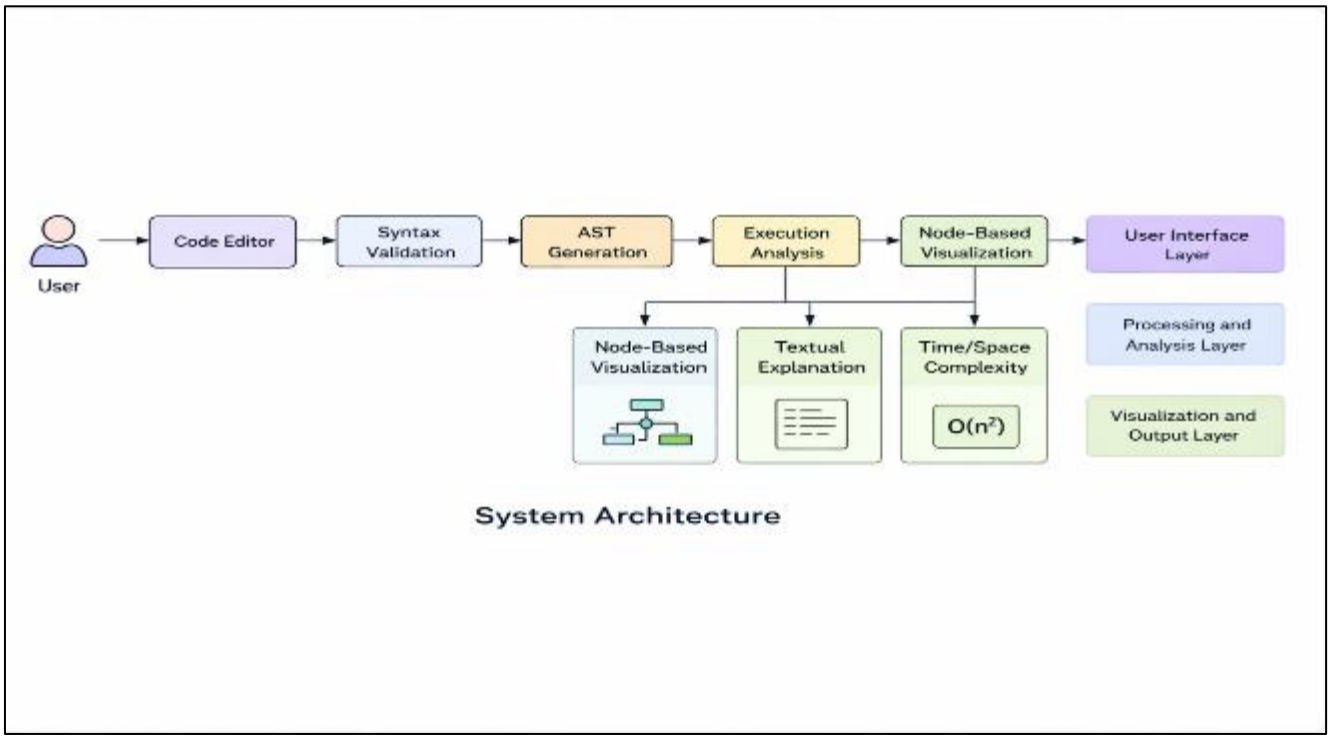

Figure 1 System Architecture of the Universal Java Code Execution Tracer

4. Methodology

4.1. System Overview

The proposed Java Code Visualizer follows a structured workflow that converts user-written Java code into an interactive visual representation. The system consists of three main components: code input, execution analysis, and visualization output. The process begins with user input and ends with graphical and textual representation along with complexity analysis.

4.2. Code Input and Validation

The user enters Java code in the editor interface. The system ensures that the visualization is generated only after the code is completely written and syntactically correct. A validation mechanism checks for syntax errors, and only valid code is processed further to maintain accuracy and reliability.

4.3. Code Parsing using AST

Once validated, the input code is parsed into an Abstract Syntax Tree (AST). The AST represents the structural hierarchy of the program, including statements, loops, conditions, and function calls. This step is essential for understanding program logic and enabling further analysis.

4.4. Execution Flow Generation

The system traverses the AST to simulate program execution step-by-step. Each operation, such as variable initialization, condition checking, and loop iteration, is extracted and stored as an execution step. These steps form the foundation for both textual and graphical visualization.

4.5. Node-Based Visualization

Each execution step is converted into a node, and nodes are connected using directed edges to represent the flow of execution. The visualization is displayed on the right side of the interface, allowing users to clearly observe how the program progresses.

4.6. Textual Execution Representation

In addition to graphical visualization, the system provides a step-by-step textual explanation of execution in the middle panel. This helps users understand the logic in a descriptive manner alongside the visual flow.

4.7. Complexity Analysis

The system analyzes the parsed code to detect loops, nested structures, and recursion. Based on these patterns, it estimates time complexity (e.g., O(n), $O(n^2)$) and space complexity. The results are displayed at the top of the interface.

4.8. Output Generation

- Node-based execution graph
- Step-by-step textual explanation
- Time and space complexity

5. Implementation

The proposed Java Code Visualizer is implemented as a web-based application using modern frontend technologies to provide an interactive and responsive user experience. The system is primarily developed using

React.js with TypeScript, which enables efficient component-based design and state management. The code editor functionality is integrated using the Monaco Editor, which provides features such as syntax highlighting, code formatting, and error detection for Java programs. Users can input their code in the editor, and the system ensures that the visualization process is triggered only after the code is completely written and syntactically correct.

The core logic of the system is handled through a custom analysis module, which processes the input Java code and generates execution steps. This module identifies key programming constructs such as variable declarations, loops, and conditional statements, and simulates the execution flow accordingly. Based on this analysis, the system constructs a structured representation of program execution.

For visualization, the system uses a graph-based approach implemented with a library such as React Flow, where each execution step is represented as a node and connected through directed edges to illustrate the flow of control. This node-based visualization is dynamically rendered on the interface, allowing users to clearly observe how the program executes step-by-step.

In addition to graphical representation, a textual explanation module is implemented to display step-by-step execution details. This provides users with a descriptive understanding of how variables change and how control flows through the program.

The system also includes a complexity analysis component that evaluates the structure of the code. By detecting loops, nested loops, and iterative constructs, the system estimates time complexity (e.g., $O(n)$, $O(n^2)$) and space complexity, which are displayed alongside the visualization.

The application is developed and tested using Visual Studio Code, with Node.js and npm used for dependency management and execution. The build process is handled using modern tools, ensuring fast performance and smooth rendering.

Overall, the implementation integrates code input, validation, execution analysis, visualization, and complexity estimation into a unified system, providing an effective platform for understanding Java program execution.

6. Result

The proposed Java Code Visualizer was evaluated using different categories of Java programs to assess its effectiveness in execution tracing and complexity analysis. The system successfully generated node-based graphical representations, step-by-step textual explanations, and accurate time and space complexity for all tested cases. The visualization clearly illustrated control flow, including variable initialization, condition checks, and iterative processes. The results demonstrate that the system provides an intuitive and reliable way to understand program execution for both basic and advanced programming scenarios.

6.1. Result for simple program without loops and conditional structures

A basic Java program without loops or complex control structures was tested to evaluate the system's handling of linear execution. The visualizer generated a sequential node-based graph representing each statement in the order of execution. The textual explanation panel accurately described each step, including variable initialization and output statements. The system correctly identified the time complexity as O(1) and space complexity as O(1). This confirms that the system effectively handles simple programs with straightforward execution flow.

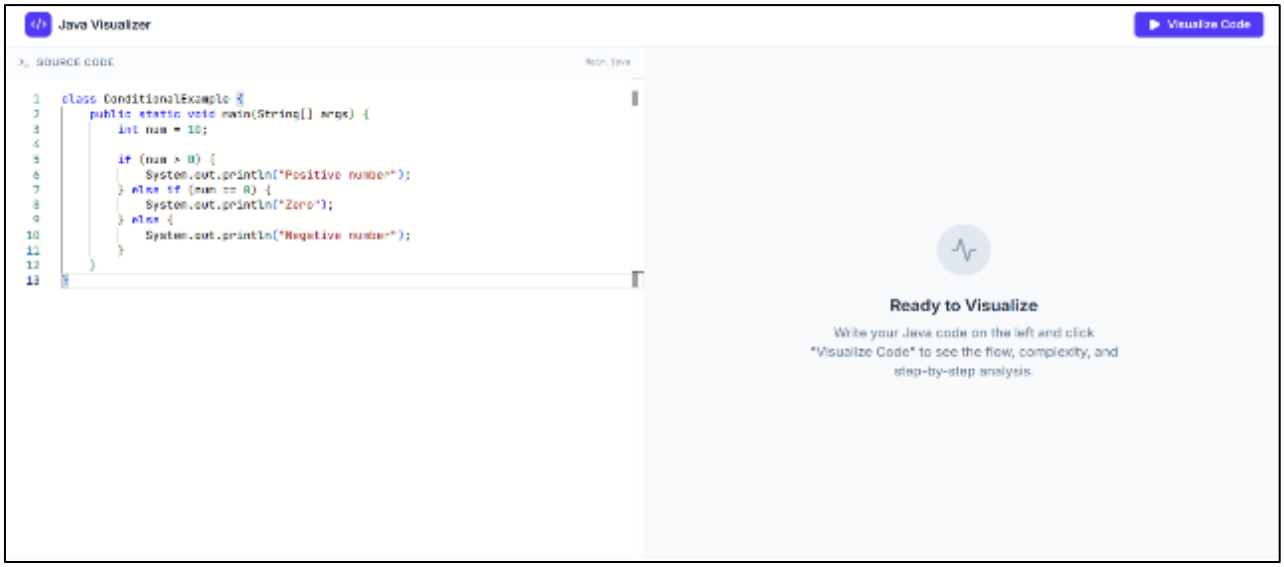

Figure 2 Output Visualization of a Simple Java Program without Loops and conditional structures

6.2. Result for simple loop

A Java program containing a single loop was tested to analyze iterative behavior. The system successfully visualized the loop as a repeating structure in the node-based graph, clearly showing initialization, condition checking, and increment operations. The textual explanation provided a detailed step-by-step breakdown of each iteration. The complexity analyzer correctly determined the time complexity as O(n) and space complexity as O(1). This demonstrates the system's ability to represent and analyze single-loop constructs effectively.

Figure 3 Output Visualization of a Java Program with a Single Loop

6.3. Results for nested loops

A program with nested loops was tested to evaluate the system's ability to handle multiple levels of iteration. The visualization displayed a multi-layered node structure, clearly distinguishing between outer and inner loops. Each iteration was represented accurately, and the execution flow was easy to trace. The system correctly identified the time complexity as $O(n^2)$, reflecting the nested nature of the loops. The textual explanation further enhanced understanding by describing the interaction between the loops. This result confirms the system's capability to analyze complex iterative structures.

Figure 4 Output Visualization of a Java Program with Nested Loops

6.4. Result for Data Structures and Algorithms (DSA) problem

The system was also tested with a basic Data Structures and Algorithms (DSA) problem, such as searching or array traversal. The visualizer successfully generated a node-based representation of the algorithm, showing each step involved in processing the data. The textual explanation provided insights into how the algorithm progresses, including comparisons and updates. The complexity analyzer accurately determined the time complexity based on the algorithm used (e.g., O(n^2) for bubble sort). This demonstrates that the system is capable of supporting DSA-based programs and analyzing their performance effectively.

Figure 6 Visualization of Data Structure Execution in Java Program

7. Conclusion

The proposed Java Code Visualizer provides an effective solution for understanding program execution through interactive and intuitive visualization. By converting Java code into a node-based execution graph and providing step-by-step textual explanations, the system enables users to clearly observe the flow of control and behavior of programs. The integration of automatic time and space complexity analysis further enhances the system by allowing users to evaluate the efficiency of their code.

A key strength of the system is its ability to handle both simple Java programs and Data Structures and Algorithms (DSA), including iterative constructs such as loops and nested loops. The validation mechanism ensures that visualization is generated only for syntactically correct code, thereby improving accuracy and reliability.

The results demonstrate that the system successfully generates meaningful visualizations and accurate complexity estimations, making it a valuable tool for learning, debugging, and analyzing programs. Overall, the proposed system bridges the gap between theoretical concepts and practical implementation, and serves as an effective platform for improving programming understanding and problem-solving skills.

Compliance with ethical standards

Disclosure of conflict of interest

The authors declare that they have no conflict of interest.

References

[1] P. J. Guo, “Online Python Tutor: Interactive Program Visualization Tool,” Proceedings of the ACM Technical Symposium on Computer Science Education, 2013.

[2] M. Kuittinen, M. Rajala, T. Laakso, “Jeliot 3: A Program Visualization System for Java,” Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education, 2004.

[3] S. Halim, “VisuAlgo: Visualizing Data Structures and Algorithms through Animation,” ACM Inroads, vol. 4, no. 2, pp. 58–63, 2013.

[4] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd ed., Pearson, 2006.

[5] M. M. T. da Silva, “Program Visualization: A Systematic Literature Review,” IEEE Transactions on Education, vol. 60, no. 1, pp. 1–8, 2017.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd ed., MIT Press, 2009.