Graph-Grounded Verification of Large Language Model Reasoning in Code Analysis (MidTerm Report)

Selen Erdoğan

ABSTRACT The integration of Large Language Models (LLMs) into software development workflows has accelerated productivity but introduced significant challenges regarding reliability and trustworthiness. These models frequently generate plausible yet factually incorrect explanations about code structure and logic, a phenomenon known as hallucination. This project proposes the development of an automated pipeline to verify the factual accuracy of LLM-generated reasoning regarding Python source code. The proposed methodology involves parsing the source code into structural graphs, such as Abstract Syntax Trees (AST) and data-flow graphs, to establish a verifiable ground truth. Subsequently, the natural language claims derived from the LLM's reasoning are systematically mapped onto these graphs, and graph traversal algorithms are employed to validate the logical correctness of each step. By distinguishing between structurally valid reasoning and unsupported hallucinations, this system aims to provide a robust mechanism for error detection. The expected outcome is a reproducible evaluation tool that enhances transparency and accountability in AI-assisted software engineering, ensuring that developers can rely on model-generated code analysis.

INDEX TERMS Large Language Models, Code Analysis, Program Comprehension, Graph Theory, Natural Language Processing.

I. INTRODUCTION

The rise of powerful Large Language Models (LLMs) has transformed the landscape of software development by enabling automated code generation, debugging, and comprehensive code explanation. Developers increasingly depend on these models to navigate, understand, and maintain complex codebases, leading to significant improvements in productivity. However, the integration of generative AI into coding workflows has introduced critical challenges regarding reliability and trustworthiness.

A primary concern is the tendency of LLMs to produce "hallucinations"—statements that are linguistically fluent and plausible but factually incorrect. In the domain of code analysis, these models may invent non-existent functions, misstate library dependencies, or describe incorrect data flows between variables. Such errors not only erode user trust but can also mislead developers, potentially introducing bugs or vulnerabilities into software systems. Consequently, there is a growing need for automated mechanisms that can rigorously verify an LLM's reasoning against the actual, executable structure of the code.

To address this challenge, this project proposes a novel framework for the **graph-grounded verification** of LLM reasoning in code analysis. The core objective is to establish a verifiable "ground truth" by parsing Python source code into structural representations, specifically Abstract Syntax Trees (AST) and control flow graphs. By systematically mapping the natural language claims extracted from the model's output to these structural graphs, the system can traverse the graph to validate the logical correctness of each reasoning step.

The main contribution of this work is the development of a modular pipeline that distinguishes between structurally valid reasoning and unsupported hallucinations. By grounding model outputs in the actual program structure, the proposed system aims to enhance the transparency of AI-assisted coding tools, facilitate detailed error analysis, and provide a reproducible foundation for future evaluation frameworks.

A. Related Work

The verification of Large Language Model (LLM) outputs has become a critical area of research, particularly with the widespread adoption of Chain-of-Thought (CoT) prompting to elicit complex reasoning. Early approaches focused on evaluating the final output of models, but recent scholarship emphasizes the necessity of verifying the intermediate reasoning steps to ensure reliability.

Ling et al. introduced deductive verification methods for CoT reasoning, focusing on the logical consistency between steps in natural language tasks. Similarly, Vacareanu et al.

proposed general-purpose verification frameworks for prompting strategies, highlighting the trade-off between reasoning depth and hallucination rates.

More recently, research has shifted towards graph-based representations to ground LLM reasoning. Zhao et al. (2025) proposed verifying CoT reasoning via its computational graph, treating reasoning steps as nodes in a directed graph to check for coherence. Sistla et al. (2025) further explored verified code reasoning, suggesting that external solvers could be used to validate specific logic claims made by models.

However, existing approaches often treat code as generic text or focus on logical puzzles. This project differentiates itself by specifically leveraging **Program Comprehension** techniques. Unlike general text verification, our approach parses Python code into Abstract Syntax Trees (AST) and data-flow graphs. This allows for a deterministic comparison between the LLM's natural language explanations and the actual executable structure of the code, providing a more rigorous "ground truth" for software engineering tasks

III. Methodology

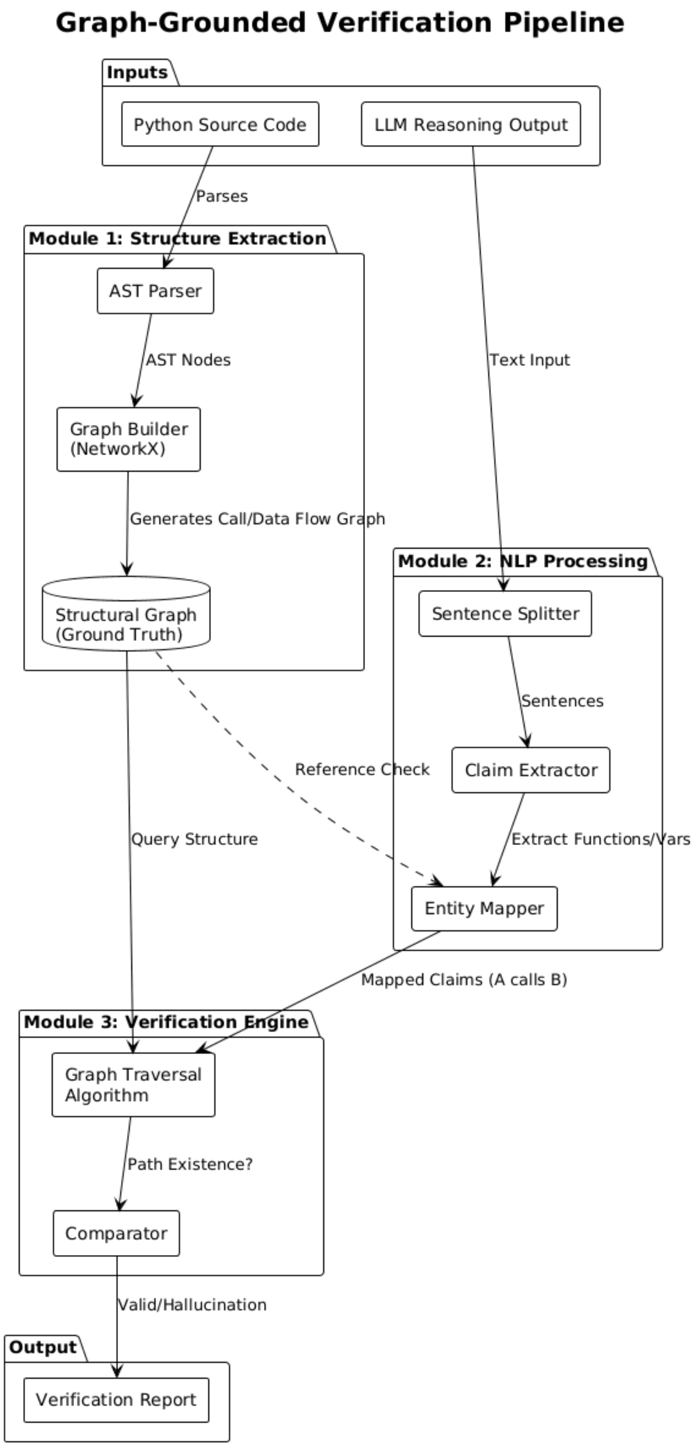

Figure 1

The proposed system integrates principles from compiler design, graph theory, and natural language processing to systematically evaluate the consistency of Large Language Model (LLM) reasoning over Python code. The methodology is executed through a modular pipeline consisting of three primary components: structural graph extraction, entity grounding, and graph-based verification. The overall system architecture is illustrated in Fig. 1.

A. CODE REPRESENTATION AS STRUCTURAL GRAPHS

The foundation of the verification system is the creation of a reliable "ground truth" derived directly from the source code10.

1) *AST Parsing:* The input Python projects are first parsed into Abstract Syntax Trees (AST) using Python's standard ast library[11]. This hierarchical structure represents the syntactic composition of the code.

2) *Graph Construction:* From the AST, the system derives higher-level graph representations, specifically Call Graphs and Data-Flow Graphs[12]. In this graph G = (V, E), vertices V represent code entities (functions, classes, variables), and edges E represent functional dependencies (e.g., *Function A calls Function B*) or data propagation. This graph serves as the structural reference against which textual claims are verified.

B. ENTITY GROUNDING AND MAPPING

To verify natural language statements, the system must bridge the gap between unstructured text and the structured graph. This module processes the LLM's reasoning trace to identify mentions of code elements.

- *Claim Extraction:* The LLM output is segmented into individual reasoning steps.
- *Mention Mapping:* Textual entities (e.g., "the calculate_total function") are mapped to their corresponding nodes v in V in the structural graph. This process utilizes a hybrid approach combining rule-based parsing and semantic similarity measures to ensure robust mapping between vague textual descriptions and actual code identifiers.

C. REASONING VERIFICATION VIA GRAPH TRAVERSAL

Once entities are grounded, the core verification logic is applied. The system translates the LLM's natural language claims into graph queries16. For instance, if the model claims "Function X depends on Variable Y," the system verifies this by traversing the graph from node X to node Y.

- *Traversal Algorithms:* Standard graph traversal algorithms are employed to check for the existence of edges or paths that validate the claimed relationship[17].
- *Classification:* Based on the traversal result, each reasoning step is classified as "Structurally Valid" (path exists) or "Hallucination" (no path exists).

D. EVALUATION METRICS

To quantify the performance of the verification process, the system computes reproducible metrics:

- *Hallucination Rate:* The proportion of reasoning steps that reference non-existent structures or false relationships.
- *Coverage:* The degree to which actual code structures are represented in the reasoning chain.
- *Step Validity:* The logical correctness of each individual reasoning step.

IV. Implementation

The implementation of the proposed verification system is being conducted in a Python-based environment, chosen for its extensive support for both code analysis and natural language processing tasks. The current phase of development focuses on establishing the core infrastructure and prototyping the initial structural extraction modules.

A. DEVELOPMENT ENVIRONMENT

The system relies on a set of robust open-source libraries to handle the distinct requirements of the pipeline:

- **Parsing:** The standard Python ast library is utilized to parse raw source code into Abstract Syntax Trees. This allows for a syntax-safe conversion of code into a traverseable tree structure.
- **Graph Processing:** The **NetworkX** library has been integrated to construct, manipulate, and visualize the structural graphs derived from the AST. This library serves as the engine for the graph traversal algorithms required for verification.
- **LLM Integration:** Interfaces are being developed to interact with Large Language Models (e.g., via OpenAI API) to retrieve reasoning traces for analysis.

B. PRELIMINARY PROGRESS

Initial implementation efforts have prioritized the "Code Representation" component described in the methodology. At this stage, a prototype parser has been developed that successfully accepts Python scripts and converts them into preliminary graph structures. Basic experiments are being conducted to calibrate the mapping between code identifiers and graph nodes. Simultaneously, the text processing module is under construction to enable the segmentation of LLM outputs into verifiable claims. The integration of these components is currently in progress.

V. Results & Discussion

In the initial phase, prototypes for the core components, specifically the parser and the graph builder modules, were developed and tested. Through tests performed on simple Python scripts, the process of converting source code into Abstract Syntax Trees (AST) and subsequently into basic call graphs was successfully executed. Using these generated graphs, the factual accuracy of textual claims produced by a Large Language Model (LLM) was examined.

In a controlled test scenario, a specific case was analyzed where the LLM referenced a function call that was not present in the source code. By employing the developed graph traversal algorithms, the connection claimed by the model was queried within the structural graph. Since no corresponding path was found, the statement was correctly flagged by the system as a "potential hallucination." This observation supports the findings of Zhao et al. (2025) [2], which suggest that computational graphs significantly contribute to the verification process. Unlike the text-based consistency checks discussed by Ling et al. (2023) [3], it was observed that structural analysis could yield more deterministic results. However, during the implementation process, certain challenges regarding the dynamic nature of the Python language were encountered. It was noted that there are limitations in statically analyzing variables whose types are determined at runtime. In line with the principles outlined by Sistla et al. (2025) [4], the future stages of the project aim to address these limitations and extend the system to cover more complex code structures.

VII. CONCLUSION

In this midterm report, the proposed graph-grounded verification framework for assessing Large Language Model reasoning in code analysis has been presented, alongside its current implementation status. The methodology relies on parsing Python source code into structural graphs to establish a deterministic ground truth for hallucination detection. To date, the prototypes for the core parsing and graph construction modules have been successfully implemented and validated against preliminary test cases. Initial findings indicate that the structural approach holds significant potential for identifying factual errors that purely text-based methods might overlook. For the remainder of the project timeline, efforts will be concentrated on the full integration of the entity mapping module, addressing the limitations regarding dynamic code features, and conducting comprehensive performance evaluations.

REFERENCES AND FOOTNOTES

A. REFERENCES

[1] Python Software Foundation, "The Python ast library – Abstract Syntax Trees," 2023. [Online]. Available: https://docs.python.org/3/library/ast.html.

[2] H. Zhao *et al.*, "Verifying Chain-of-Thought Reasoning via Its Computational Graph," 2025.

[3] Z. Ling *et al.*, "Deductive Verification of Chain-of-Thought Reasoning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[4] S. Sistla *et al.*, "Towards Verified Code Reasoning by LLMs," 2025.

[5] R. Vacareanu *et al.*, "General Purpose Verification for CoT Prompting," 2024.