An empirical study of large language models for type and call graph analysis in Python and JavaScript

Venkatesh, Ashwin Prasad Shivarpatna; Sunil, Rose; Sabu, Samkutty; Mir, Amir M.; Reis, Sofia; Bodden, Eric

DOI
10.1007/s10664-025-10704-3

Publication date
2025

Document Version
Final published version

Published in
Empirical Software Engineering

Citation (APA)
Venkatesh, A. P. S., Sunil, R., Sabu, S., Mir, A. M., Reis, S., & Bodden, E. (2025). An empirical study of large language models for type and call graph analysis in Python and JavaScript. *Empirical Software Engineering*, *30*(6), Article 167. https://doi.org/10.1007/s10664-025-10704-3

Empirical Software Engineering (2025) 30:167
https://doi.org/10.1007/s10664-025-10704-3

An empirical study of large language models for type and call graph analysis in Python and JavaScript

Ashwin Prasad Shivarpatna Venkatesh[4] · **Rose Sunil**[5] · **Samkutty Sabu**[5] · **Amir M. Mir**[2] · **Sofia Reis**[3] · **Eric Bodden**[1]

Accepted: 15 July 2025

Abstract
Large Language Models (LLMs) are increasingly being explored for their potential in software engineering, particularly in static analysis tasks. In this study, we investigate the potential of current LLMs to enhance call-graph analysis and type inference for Python and JavaScript programs. We empirically evaluated 24 LLMs, including OpenAI's GPT series and open-source models like LLaMA and Mistral, using existing and newly developed benchmarks. Specifically, we enhanced TypeEvalPy, a micro-benchmarking framework for type inference in Python, with auto-generation capabilities, expanding its scope from 860 to 77,268 type annotations for Python. Additionally, we introduce SWARM-CG and SWARM-JS, comprehensive benchmarking suites for evaluating call-graph construction tools across multiple programming languages. Our findings reveal a contrasting performance of LLMs in static analysis tasks. For call-graph generation, traditional static analysis tools such as PyCG for Python and Jelly for JavaScript consistently outperform LLMs. While advanced models like mistral-large-it-2407-123b and gpt-4o show promise, they still struggle with completeness and soundness in call-graph analysis across both languages. In contrast, LLMs demonstrate a clear advantage in type inference for Python, surpassing traditional tools like HeaderGen and hybrid approaches such as HiTyper. These results suggest that, while LLMs hold promise in type inference, their limitations in call-graph analysis highlight the need for further research. Our study provides a foundation for integrating LLMs into static analysis workflows, offering insights into their strengths and current limitations.

1 Introduction

In recent years, the field of Software Engineering (SE) has witnessed a paradigm shift with the integration of Large Language Models (LLMs), bringing new capabilities and enhancements to traditional software development processes (Rasnayaka et al. 2024; Huang et al. 2024a; Hou et al. 2023; Fan et al. 2023; Zhang et al. 2023; Zheng et al. 2023). LLMs, with

Communicated by: Massimiliano De Penta, Xin Xia, David Lo, Xing Hu.

Extended author information available on the last page of the article

their ability to understand and generate human-like text, are reshaping various SE tasks, such as code generation and bug detection.

Static analysis (SA), a foundational technique in SE, focuses on evaluating code without executing it, allowing developers to detect potential errors, maintain code quality, and identify security vulnerabilities early in the development lifecycle. Historically, SA tools have faced challenges, such as high rates of false positives, difficulty of scaling to large codebases, and limited ability to handle ambiguous or incomplete code. Models such as BERT (Devlin et al. 2019), T5 (Raffel et al. 2023), and GPT (Radford et al. 2019) have demonstrated potential in automating complex SA tasks (Zhang et al. 2023).

Recent works have shown how different SA tasks can benefit from LLMs, such as false-positives pruning (Li et al. 2023a), improved program behavior summarization (Li et al. 2023b), type annotation (Seidel et al. 2023), and general enhancements in precision and scalability of SA tasks (Li et al. 2023b; Mohajer et al. 2024), both fundamental issues of SA.

Goal This study positions itself at the intersection of SA and LLMs, examining the effectiveness of LLMs in SA within SE. It aims to **evaluate the accuracy of LLMs in performing specific SA tasks in Python and JavaScript programs**, such as call-graph analysis and type inference. We focus on Python and JavaScript as they are dynamically typed languages, making them inherently challenging for static analysis due to the absence of explicit type annotations and the high degree of runtime flexibility. *Call-graph analysis* helps in understanding the relationships and interactions between different components of a program, while *type inference* aids in identifying potential type errors and improving code reliability.

Methodology We performed a comprehensive analysis of the capabilities of 24 different LLMs across different SA tasks, using data from micro-benchmarks and customized prompts for each task. This evaluation enables one to make direct comparisons with the existing capabilities of traditional approaches in SA. To assess the performance of LLMs, we use the PyCG (Salis et al. 2021) and HeaderGen (Venkatesh et al. 2023b) micro-benchmarks for call-graph analysis in Python, and a newly created *SWARM-JS* micro-benchmark for JavaScript. For type inference, we use the TypeEvalPy (Venkatesh et al. 2023a) micro-benchmark. The use of micro-benchmarks in evaluating the performance of LLMs in our study is grounded in the following key considerations:

- Micro-benchmarks are designed to target specific aspects of the features under test and various characteristics of the programming language involved. This helps highlight the models' strengths and weaknesses, allowing for a more nuanced understanding of their capabilities in SA tasks.
- Micro-benchmarks development involves rigorous manual inspection and adherence to scientific methods, ensuring reliability and accuracy in evaluation. Conversely, obtaining large-scale, real-world data that can serve as ground truth is often a challenging endeavor. Where such data is available, it is susceptible to human errors, which can skew the results (Di Grazia and Pradel 2022).

The insights from this study are intended to offer a preliminary understanding of the role of LLMs in SA for the call-graph analysis and type inference tasks, contributing to the Artifi-

cial Intelligence for Software Engineering (AI4SE) and Software Engineering for Artificial Intelligence (SE4AI) fields.

Results The results of our study show that static analysis tools like PyCG and Jelly (Laursen et al. 2024) significantly outperform LLMs in call-graph generation for Python and JavaScript, respectively. LLMs, while showing some promise, especially with models like mistral-large-it-2407-123b and gpt-4o, struggled with completeness and soundness in both Python and JavaScript.

Contrarily, in the case of type inference, LLMs demonstrated a clear advantage over traditional static analysis tools like HeaderGen and hybrid approaches such as HiTyper. While OpenAI's gpt-4o initially performed best in the micro-benchmark, mistral-large-it-2407-123b surpassed it on the larger autogen-benchmark, indicating that some open-source models can outperform proprietary ones.

Contributions The primary contributions of this paper are as follows:

- Performed an empirical evaluation of 24 LLMs across Python and JavaScript for call-graph inference.
- Conducted an empirical evaluation of 24 LLMs for type inference in Python.
- Enhanced TypeEvalPy with auto-generation capabilities, expanding its benchmarking scope for type inference from 860 to 77,268 annotations.
- Introduced SWARM-CG, a comprehensive benchmarking suite for evaluating call-graph construction tools across multiple programming languages, starting with Python and JavaScript, to enable cross-language comparisons and consistent analysis evaluations.
- Developed SWARM-JS, a call-graph micro-benchmark for JavaScript.

Structure The structure of the paper is as follows: in Section 2 we discuss the necessary background information, including tools and baselines used in the study. In Section 3 we discuss the related work. The research questions are outlined in Section 4. In Section 5, we describe the micro-benchmarks used for evaluation, while Section 6 describes our methodology. The results are presented in Section 7 and subsequently discussed in Section 8. Section 9 addresses the threats to validity. Finally, the paper is concluded by outlining future research directions in Section 10.

Availability

- TypeEvalPy is published on GitHub as open-source software: https://github.com/secure-software-engineering/TypeEvalPy
- SWARM-CG is published on GitHub as open-source software: https://github.com/secure-software-engineering/SWARM-CG
- The raw outputs and analysis data are published on Zenodo at: https://zenodo.org/records/15045642

2 Background

Program analysis techniques, such as *type inference* and *call-graph analysis*, are essential for static analysis tools that allows them to reason about program correctness before execution, especially in dynamically typed languages. Type inference determines the types of variables without explicit annotations, while call-graph analysis tracks function calls and their relationships within a program.

Type inference Type inference is the process of deducing the types of variables based on available program information, such as function signatures and variable assignments. In dynamically typed languages like Python, where variable types can change at runtime, type inference helps predict potential type mismatches and enforce consistency in operations.

A static analyzer with type inference capabilities examines assignments, function calls, and operations to deduce the type of each variable at different points in the program. By performing this analysis, type inference can detect errors such as type mismatches before execution, preventing runtime failures.

Call-graph Analysis A call-graph is a representation of function call relationships in a program. Call-graph analysis helps in understanding control flow, tracking function dependencies, and identifying unreachable or unused code. It enables optimizations, refactoring, and bug detection by revealing function relationships.

Flow-insensitive vs. Flow-sensitive Analysis Flow-insensitive and flow-sensitive analyses differ in how they account for the order of program execution. Flow-*insensitive* analysis disregards the sequence of statements, treating all potential assignments to a variable as if they occur simultaneously. This lack of ordering leads to an over-approximation of possible program states, potentially reducing the precision of the analysis. In contrast, flow-*sensitive* analysis explicitly considers the order in which statements are executed, allowing it to track variable values and types throughout the program. As a result, flow-sensitive approaches often yield more precise analysis outcomes.

2.1 Motivating Example

In the following code, the create_str function returns a string, the variable func_ref is assigned with function references at lines 4 and 8, and x is assigned the value result + 1 at lines 6 and 10.

```
1  def create_str(x):
2      return x.upper()
3
4  func_ref = create_str
5  result = func_ref("Hello!")
6  x = result + 1 # Type mismatch!
7
8  func_ref = len
9  result = func_ref("Hello!")
10 x = result + 1 # Works
11
12 x = eval("create_str('Hello!')")
13 x = x + 1 # Type mismatch!
```

Type Inference A static analyzer with type inference capabilities can resolve that the variable result at line 5 is a string, while the variable result at line 9 is an integer. Using this, the static analyzer can raise a type error at line 6 even before executing it. However, static analyzers struggle with dynamically evaluated expressions that obscure type information at analysis time, such as the eval function in line 12. If a variable's value is determined through eval, reflection, or user input, static analyzers cannot reliably infer its type before execution. This further highlights the challenges of type inference in dynamically typed languages, where runtime behavior can introduce unexpected type inconsistencies.

Callgraph The complete call-graph for the snippet is as follows:

$$main \rightarrow create\_str() \rightarrow upper()$$
$$main \rightarrow len()$$

A flow-sensitive analysis can further resolve exactly where these calls are made. For instance, it can resolve that at line 5 the variable func_ref points to the function create_str while at line 9 func_ref points to the function len.

2.2 Type Inference Tools

In this subsection, we briefly describe the existing type inference approaches, namely, Type4Py, HiTyper, and HeaderGen.

Type4Py Type4Py (Mir et al. 2022) is a deep similarity learning-based type inference model for Python that addresses the limitations of previous ML type inference methods. Unlike earlier models trained on potentially incorrect human-provided annotations, Type4Py uses a type-checked dataset, ensuring higher accuracy in predicting variable, argument, and return types. It maps programs into type clusters in a high-dimensional space, leveraging a hierarchical neural network model to differentiate between similar and dissimilar types. This approach allows Type4Py to handle an unlimited type vocabulary. It improves upon the state-of-the-art models Typilus (Allamanis et al. 2020) and TypeWriter (Pradel et al. 2020), achieving a Mean Reciprocal Rank (MRR) of 77.1%, an 8.1%, and 16.7% improvement over these models, respectively.

To aid developers in retrofitting type annotations, Type4Py incorporates identifiers, code context, and visible type hints as features for type prediction. Its deep similarity learning methodology enables the model to learn from a wide range of data, making it particularly effective for real-world usage. Type4Py is also deployed as a Visual Studio Code extension, offering machine learning-based type auto-completion for Python, thereby enhancing developer productivity by easing the process of adding type annotations to existing codebases.

HiTyper HiTyper (Peng et al. 2022) is a hybrid type inference approach that combines static type inference with deep learning (DL) to address the challenges of type inference in dynamic languages like Python. It builds on the observation that static inference can accurately predict types with static constraints, while deep learning models are effective for cases with dynamic features but lack type correctness guarantees. HiTyper introduces a type dependency graph (TDG) to encode type dependencies among variables in a function.

By leveraging TDGs, HiTyper integrates static inference rules and type rejection rules to filter incorrect neural predictions, conducting an iterative process of static inference and DL-based type prediction until the TDG is fully inferred.

HiTyper's key advantage lies in its ability to combine the precision of static inference with the adaptability of learning-based predictions. By focusing on *hot type slots*, variables at the beginning of data flow with dependencies, HiTyper invokes DL models only when static inference is insufficient. Its similarity-based type correction algorithm supplements DL predictions, particularly for user-defined and rare types, which are challenging for traditional DL models. The results show that HiTyper outperforms state-of-the-art DL models like Typilus and Type4Py, achieving 10-12% improvements in overall type inference accuracy and significant gains in inferring rare and complex types.

HeaderGen HeaderGen (Venkatesh et al. 2023b) is a static analysis-driven tool that analyzes Python code in Jupyter Notebooks to create structure and enhance the comprehensibility of the notebook. HeaderGen uses a flow-sensitive call-graph analysis technique to extract fully qualified function names of all invoked function calls in the program and use this information as context to add structural headers to notebooks. To facilitate flow-sensitive call-graph construction, the underlying analysis first constructs an assignment graph that captures the relationship between program identifiers. HeaderGen augments this assignment graph with type information during its fixed-point iterations to infer types of program identifiers. The evaluation of HeaderGen on micro-benchmarks shows a precision of 95.6% and a recall of 95.3%.

2.3 Call-graph Construction Tools

In this subsection, we briefly describe the existing call-graph generation approaches, namely, PyCG, HeaderGen, TAJS, and Jelly.

PyCG PyCG (Salis et al. 2021) is a static call-graph construction technique for Python. It works by computing assignment relations between program, such as functions, variables, classes, and modules, through an inter-procedural analysis. PyCG is capable of handling complex Python features such as modules, generators, lambdas, and multiple inheritance. PyCG is evaluated on both micro-benchmarks and real-world Python packages. PyCG outperforms other tools in terms of precision and recall, achieving a precision of 99.2% and a recall rate of around 69.9%.

HeaderGen As previously discussed, HeaderGen uses a flow-sensitive analysis to extract all invoked function calls within a program. HeaderGen extends the assignment graph of PyCG to include flow-sensitive information, thereby increasing the precision of the call-graph construction algorithm.

TAJS - Type Analyzer for JavaScript. TAJS (Jensen et al. 2009) is a static analysis tool designed for JavaScript that performs type inference and constructs call-graphs. It fully supports ECMAScript 3rd edition and provides partial support for ECMAScript 5, including its standard library, HTML DOM, and browser APIs. However, TAJS does not support features

introduced in ECMAScript 6 (ECMA 2015), such as classes, arrow functions, and modules, which limits its effectiveness in analyzing modern JavaScript applications.

TAJS offers a command-line option to export these call-graphs as DOT files, which can be converted into JSON for further analysis or integration with other tools.

Jelly Jelly (Laursen et al. 2024) is a hybrid JavaScript call-graph analysis tool that combines static and dynamic analysis to improve accuracy. Jelly's analysis consists of two main steps. First, a dynamic pre-analysis is conducted to gather runtime hints regarding variable values and object structures. The second step uses these hints in a SA phase, refining the constructed call-graph and improving soundness. This method allows Jelly to outperform traditional SA tools, particularly in handling modern ECMAScript features and multi-file JavaScript programs. Evaluations show that Jelly outperforms tools like TAJS (Jensen et al. 2009) and ACG (Feldthaus et al. 2013), making it more accurate for real-world JavaScript analysis.

3 Related Work

This section reviews prior work at the intersection of Large Language Models and static analysis, highlighting existing approaches, their limitations, and how our study advances the state of the art.

3.1 Traditional Static Analysis for Python and JavaScript

Static Analysis Basics Static analysis (SA) tools examine code without executing it to find bugs, type errors, or security issues. In Python and JavaScript, traditional static analyzers include linters and type checkers. In Python, tools like **PyLint** and **Flake8** catch style issues and simple bugs, while **MyPy** and Facebook's **Pyre** perform static type checking using type hints. JavaScript relies on linters (e.g., **ESLint**) and optional type systems (Flow or TypeScript's compiler) to detect errors. Academic tools also exist: e.g., **PyCG** constructs call-graphs for Python using a context- and flow-insensitive analysis (Salis et al. 2021), and **Type4Py** uses deep learning to predict Python types for better static type inference (Mir et al. 2022). These tools improve code quality but face well-known challenges with dynamic language features. Python and JavaScript allow dynamic typing, runtime reflection, and polyglot patterns that make static reasoning difficult. As a result, static analyzers often miss issues or report false alarms due to incomplete information. For instance, static taint analyzers depend on manually provided specifications of library APIs (sources/sinks), which are often missing or outdated, leading to missed vulnerabilities (Li et al. 2025). They also tend to over-approximate program behaviors, yielding many false positives that developers must triage (Li et al. 2025). In short, traditional SA tools are powerful but limited by dynamic features and scalability issues, motivating the exploration of learning-based approaches to augment or replace them.

Advances in Static Analysis (pre-LLM) Before the recent LLM surge, researchers began injecting machine learning into static analysis. Early deep neural network models trained

on code graphs or tokens could predict types or detect certain bugs, but they lacked broad language understanding. For example, **DeepTyper** (Hellendoorn et al. 2018) and **Typilus** (Allamanis et al. 2020) learned to suggest variable types in Python, and **HiTyper** combined static inference with a neural network to improve Python type predictions (Peng et al. 2022). These approaches showed that learned models can complement rule-based analysis, but they were narrow in scope and struggled with long-range dependencies in code.

3.2 LLMs Enter Static Analysis: Early Experiments

The emergence of code LLMs, such as OpenAI Codex, GPT-3, GPT-4 and Code LLaMa, prompted researchers to ask how well these models can perform classic static analysis tasks and where they fall short. Initial studies found that **LLMs easily handle basic syntax and code summarization but struggle with deeper program analysis** (Sun et al. 2023), e.g., pointer analysis or detailed code behavior reasoning. Another survey noted that even large code models failed to reliably perform multi-step reasoning needed for vulnerability detection, achieving only 55% accuracy on such tasks without assistance (Steenhoek et al. 2025). In other words, LLMs are not ready to replace a static analyzer for complex analysis, as they often **hallucinate** facts or miss subtle relationships, especially when whole-program context is required.

On the positive side, researchers observed that LLMs have strong general knowledge of programming and can interpret code more semantically than traditional tools. Li et al. (2023b) argued that LLMs can be integrated into program analysis pipelines ("LLift") to compensate for static analysis blind spots (e.g., LLMs might naturally summarize what a code segment does, helping an analyzer decide if a warning is a true issue). Li et al. (2023a) took a first step in this direction by empirically testing ChatGPT as an assistant to static analysis. Overall, early investigations converged on the view that **LLMs are promising but have clear limitations**: they can understand intent and context in code, yet need careful prompting or fine-tuning to handle precise static analysis tasks. This realization spurred a new wave of techniques combining traditional static analysis strengths with LLMs' flexibility. In this study, we investigate whether LLMs can effectively perform type inference and call-graph construction.

3.3 LLM-Augmented Type Inference and Call Graph Construction

Recent work has explored the application of LLMs to traditional static analysis tasks such as **type inference** and **call-graph construction**, particularly in dynamic languages like Python and JavaScript.

Venkatesh et al. (2024b) conducted a comprehensive evaluation of 26 models, including GPT-3.5, GPT-4 and Code LLaMA, on Python benchmarks targeting two core static analysis tasks: **type inference** and **call-graph construction**. They found that **LLMs greatly outperformed a traditional analyzer on type inference accuracy, but lagged on call-graph construction**. In fact, GPT-4 inferred types in Python with higher accuracy than static methods, thanks to its learned knowledge of APIs and naming conventions. However, the same model struggled to predict dynamic function call targets, missing many edges in call-graphs. This suggests that LLMs excel at tasks requiring knowledge of common coding patterns (e.g. inferring that len() returns an int) but have trouble with exhaustive

program structure analysis. The authors note that fine-tuning LLMs specifically for call-graph tasks or integrating them with algorithmic analysis might be necessary to overcome these limitations. Similarly, Seidel et al. (2023) (CodeTIDAL) focused on TypeScript and trained a Transformer to predict missing type annotations, effectively learning from code context to enhance dataflow analysis. These efforts show that **LLMs can learn type and flow rules** in practice, often outperforming purely static approaches on inferring types, but they may need augmentation (e.g. external reasoning steps) for full program understanding. Venkatesh et al. (2023b) built **HeaderGen** to improve Python notebook analysis by adding flow-sensitive and type-aware reasoning on top of PyCG. HeaderGen demonstrated that adding semantic analysis and basic inference layers can improve static call-graph accuracy in Jupyter notebooks, even before full LLM integration. This underscores the potential of enhancing static tools with **LLM-augmented hybrid approaches**. This study extends (Venkatesh et al. 2024b) analysis to the JavaScript language.

3.4 Micro-benchmark Suites for Python and JavaScript

Static Call Graph Analysis Benchmarks in Python Salis et al. (2021) introduced one of the first modern micro-benchmark suites for Python call-graph analysis as part of the PyCG study. This suite contains minimal Python programs covering a wide range of language constructs, organized into different feature-focused categories (e.g., simple function calls, decorators, generators, etc). The PyCG benchmarks provided a standardized way to compare static analyzers on Python's dynamic features (e.g. lambdas, closures, dynamic dispatch via lists/dicts of functions) in a controlled setting. A strength of this suite is its breadth of coverage across Python 3's core features and the inclusion of expected outputs for each test, which improves reproducibility and fair comparison. However, by design it uses only small, self-contained programs, this micro-scale yields clarity but may not capture interactions present in large codebases (e.g. extensive module interplay or runtime reflection beyond basic imports). Subsequent work built on PyCG's benchmark to improve coverage and address its limitations. Huang et al. (2024b) extended the suite with more snippets in their Jarvis call-graph analysis (adding 23 new tests on top of PyCG's original 112). These additions include flow-sensitive scenarios, alongside other cases written by experienced Python developers to cover features that PyCG's suite lacked. Venkatesh et al. (2023b) similarly created a micro-benchmark for their HeaderGen tool by adopting PyCG's full suite and augmenting it with new snippets focused on flow-sensitive call sites. Both Jarvis's and HeaderGen's benchmarks remain centered on Python, inheriting the original PyCG test design and ground truths.

JavaScript Call Graph Analysis Benchmarks The SunSpider benchmark (WebKit 2010), a collection of JS programs originally meant for performance testing, has been used to compare call-graph tools, although it did not provide official ground-truth call-graph. Researchers had to manually inspect whether the edges produced by a tool match the actual calls in the source, a tedious process that focuses on precision of found edges and neglects recall (missing edges). Antal et al. (2023) followed this approach in a comparative study, manually validating tool outputs on SunSpider. Such ad hoc methods are labor-intensive and error-prone, and they struggled to exercise modern JavaScript features beyond the aging SunSpider suite. Notably, a static analyzer (TAJS) showed high precision on classic benchmarks

but failed to handle many ES6+ features, leading to underperformance on contemporary code.

The lack of structured and standardized call-graph benchmarks across diverse programming languages poses several challenges in evaluating and comparing call-graph construction tools. This gap makes cross-language comparisons difficult and unreliable, hindering consistent assessments of different analysis techniques. This study enables call-graph construction tools evaluation across multiple programming languages (SWARM-CG).

Python Type Inference Benchmarks Researchers either used large-scale corpora of real code with optional type hints (e.g. the ManyTypes4Py dataset and Type4Py) or relied on each tool's own set of examples, making it difficult to compare results across studies. Recognizing this gap, Venkatesh et al. (2023a) proposed TypeEvalPy, a micro-benchmarking framework for Python type inference tools. Categories cover dynamic typing constructs (uses of Python's dynamic features that affect types, e.g. changing a variable's type or using reflection) and external library calls, which simulate inferring types when third-party code is involved. While TypeEvalPy greatly improved standardization in evaluating Python type inference, it initially had some limitations in scope. The 154 snippets were designed to be representative but necessarily cannot cover all possible Python idioms. To address this, TypeEvalPy was augmented with an auto-generation extension that massively scaled up the benchmark's coverage (Venkatesh et al. 2024b). This study describes the methodology to expand the synthetic test cases with 77k type annotations that increase the diversity of types and scenarios.

4 Research Questions

We focus on the following research questions to evaluate the effectiveness of LLMs using micro-benchmarks in static analysis tasks:
RQ1: *What is the accuracy of LLMs in performing callgraph analysis against micro-benchmarks?*
RQ2: *What is the accuracy of LLMs in performing type inference against micro-benchmarks?*

5 Micro-benchmarks

In this study, we utilize a diverse set of benchmarks to evaluate the effectiveness and performance of call-graph generation and type inference tools. To support these evaluations, we extended existing frameworks and developed new benchmarks as required to ensure comprehensive testing.

To answer *RQ1*, we choose two benchmarks designed to evaluate callgraph analysis performance, PyCG (Salis et al. 2021) and HeaderGen (Venkatesh et al. 2023b). Furthermore, we evaluate the effectiveness of LLMs across programming languages by creating a new micro-benchmark for JavaScript.

To answer *RQ2*, we choose the micro-benchmark from TypeEvalPy (Venkatesh et al. 2023a), a general framework for evaluating type inference tools in Python. TypeEvalPy contains a micro-benchmark with 154 code snippets and 860 type annotations as ground truth. Additionally, we extend the TypeEvalPy with auto-generation capability and synthetically scale the micro-benchmark to include a wide spectrum of types. The TypeEvalPy *autogen-benchmark* now contains 7,121 test cases with 77,268 type annotations.

In the following sections, we outline the micro-benchmarks used for both call-graph generation and type inference.

5.1 PyCG: Call-graph Micro-benchmark

The PyCG micro-benchmark suite offers a standardized set of test cases for researchers to evaluate and compare call-graph generation techniques. It includes 112 unique and minimal micro-benchmarks, each designed to cover different features of the Python Language. These benchmarks are grouped into 16 categories, ranging from simple function calls to more complex constructs like inheritance schemes.

Each category comprises multiple tests, with each test providing: (1) the source code, (2) call-graph in JSON format, and (3) a brief description of the test case. The tests are structured to be easy to categorize and expand, with each focusing on a single execution path, without the use of conditionals or loops. This design ensures that the generated call-graph accurately reflects the execution of the source code.

To ensure completeness and quality, the authors had two professional Python developers review the suite, providing feedback on feature coverage and overall quality. Based on their recommendations, the authors refactored and further enhanced the suite.

In this study, we additionally include 14 new test cases to the PyCG benchmark based on the benchmark used in Jarvis (Huang et al. 2024b). These additions include 4 in *args* category, 4 in *assignments*, 5 in *direct_calls*, and 1 in *imports*.

5.2 HeaderGen: Flow-sensitive Call-sites Micro-benchmark

The micro-benchmark in HeaderGen is created by adopting the PyCG micro-benchmark. HeaderGen adds flow-sensitive call-graph information, i.e., line number information indicating where in the program the call is originating from. Furthermore, since HeaderGen performs a flow-sensitive analysis, eight new test cases specifically targeting flow-sensitivity are added.

5.3 SWARM-CG: Swiss Army Knife of Call Graph Benchmarks

The lack of structured and standardized call-graph benchmarks across diverse programming languages poses several challenges in evaluating and comparing call-graph construction tools. This gap makes cross-language comparisons difficult and unreliable, hindering consistent assessments of different analysis techniques.

To address this issue, we developed the *Swiss Army Knife of Call Graph Benchmarks* (SWARM-CG), a benchmarking suite designed to provide a standardized platform for evaluating call-graph construction tools across multiple programming languages. The primary goal of SWARM-CG is to create a unified environment that facilitates consistent comparisons and promotes further research in the field of call-graph analysis, especially in the current land-

scape, where ML models are being explored as alternatives to traditional static analysis. ML models often lack the transparency and verifiability that static analysis provides. As researchers investigate these models in call-graph construction, having a standardized framework is essential for accurately comparing their effectiveness with established methods. SWARM-CG fulfils this need by offering a well-organized, comprehensive set of call-graph benchmarks with ground truth annotations for each code snippet, enabling reliable and consistent evaluations.

Furthermore, each tool that SWARM-CG supports is dockerized to make the evaluation process straightforward. As a proof of concept, we have added support for the following tools: (1) PyCG, (2) HeaderGen, (3) Transformers, (4) Ollama, (5) TAJS, and (6) Jelly.

SWARM-CG supports multiple programming languages, starting with Python and JavaScript, with ongoing efforts to integrate Java and plans to extend to additional languages. The suite is designed to be community-driven, encouraging contributions from both static analysis experts and enthusiasts, making it a dynamic and evolving resource for the research community.

5.4 SWARM-JS: JavaScript Micro-benchmark

Despite the increasing importance of JavaScript analysis, the availability of well-defined benchmarks tailored for JavaScript call-graph construction remains limited. Existing benchmarks, such as SunSpider (WebKit 2010), part of the WebKit browser engine, are primarily designed to test the performance aspects of JavaScript engines rather than facilitating program analysis. SunSpider includes single-file JavaScript examples that represent real-world scenarios, but it does not provide explicit ground truth for call-graphs.

In a recent study by Antal et al. (2023), the authors assessed static call-graph techniques using the SunSpider benchmark by manually comparing the call-graphs generated by the tools with the source code. Precision was measured by verifying whether specific edges in the graph were accurately identified. However, this manual approach limits the scope of the evaluation and limits the extensibility of the respective research. Furthermore, the lack of attention to recall in this manual evaluation process results in an incomplete understanding of the tools' performance.

To address these limitations, we developed a new JavaScript micro-benchmark, SWARM-*JS*, tailored specifically for call-graph construction. Inspired by call-graph micro benchmarks in Python, such as PyCG and Jarvis, our benchmark aims to provide a systematic and comprehensive set of test cases that reflect the diverse language-specific constructs of JavaScript.

To construct the benchmark, we followed a methodology similar to that used by the authors of the PyCG (Salis et al. 2021) call-graph benchmark for Python. Their process consists of three main steps: (1) identifying a diverse set of language features relevant to call-graph construction, (2) designing minimal test scripts inspired by real-world uses of these features, and (3) conducting expert review to validate correctness and representativeness.

Applying this methodology to JavaScript, we began by surveying the ECMAScript specification (ECMA 2015) and the existing SunSpider (WebKit 2010) JavaScript benchmark to identify essential language features and edge cases. A comparative analysis with Python call-graph benchmarks, such as PyCG and Jarvis, helped determine which test scenarios could be adapted to JavaScript. Test cases were constructed by re-implementing the intent of PyCG's benchmark scenarios in JavaScript while maintaining feature isolation. For instance, Python's lambda expressions were mapped to JavaScript's arrow functions, given their similar semantics. In contrast, JavaScript-specific constructs such as prototypes, dynamic property access, and mixins were created additionally.

Validity of SWARM-JS To ensure correctness and reliability, all test cases and their ground truths were manually reviewed and refined through multiple iterations. A JavaScript expert independently validated a randomly selected subset of 25 test cases to verify the accuracy and correctness of the ground truth. Based on the expert's feedback, we revised the benchmark to correct ground truth annotations. This iterative review process improved the overall validity of the benchmark.

The resulting benchmark, SWARM-JS, comprises 126 JavaScript code snippets, organized into 18 feature categories. Table 1 presents the complete list of categories along with the number of test cases and their descriptions. Each snippet in the benchmark is accompanied by a corresponding ground truth file, which provides the expected call-graph. The ground truth schema follows the PyCG benchmark, allowing for a consistent framework for evaluating call-graph accuracy across different languages. The code snippets and ground truth information were manually inspected and iteratively refined to ensure correctness.

An example code snippet is shown in Listing 1 and its corresponding ground truth is given in Listing 2.

```
function paramFunc() {}


function func(a) {
    a();
}


func(paramFunc);
```

Listing 1: Code snippet of main.js

```
{
    "main.func": [
      "main.paramFunc"
    ],
    "main.paramFunc": [],
    "main": [
      "main.func"
    ]
}
```

Listing 2: Ground truth for main.js

Table 1 Distribution of 126 JavaScript code snippets into 18 feature categories in SWARM-JS micro-benchmark

Category	# Cases	Description
Args	10	Positional and default argument passing.
Assignments	8	Variable assignments and reassignments.
Builtins	3	Built-in functions and objects.
Classes	21	Class definitions, methods, and inheritance.
Decorators	7	Use of function and class decorators.
Objects	12	Object creation, property access, and manipulation.
Direct Calls	9	Focuses on direct function and method calls.
Dynamic	1	Dynamic code injection and method access.
Exceptions	3	Exception handling.
Functions	4	Function declarations and expressions.
Generators	6	Generator functions and yield behavior.
Imports	15	Module imports and exports.
Kwargs	3	Keyword arguments and related constructs.
Arrow Functions	5	Arrow function syntax and behavior.
Arrays	8	Array manipulation and iteration.
Inheritance	4	Prototype-based and class-based inheritance.
Mixins	3	Mixin patterns for object composition.
Returns	4	Functions that return other functions.

5.5 TypeEvalPy Autogen Extension

The micro-benchmark that is part of the TypeEvalPy framework is constrained by its limited representation of Python base types, covering only 860 types derived from 154 code snippets. This limited coverage has implications in the context of evaluating LLMs. Since a large proportion of type annotations in the TypeEvalPy benchmark are str, LLMs might have exhibited high exact matches due to overrepresentation, rather than a genuine understanding of diverse type usages. This narrow focus undermines the applicability and robustness of the evaluation results, as the models are not thoroughly tested on a wider variety of base types available in Python.

To address this limitation, we extend TypeEvalPy by integrating auto-generation capabilities aimed at broadening the type diversity in the micro-benchmark. This enhancement is realized through a systematic process of template-based code generation. We first designed templates for the existing code snippets, introducing placeholders, such as $< value1 >$, which are dynamically replaced by different types during code generation. Additionally, associated configuration files were created to map these placeholders to various possible type values. For example, in Listing 3, the code snippet includes placeholders at specific locations, and the corresponding code generated is shown in Listing 4. Type annotation ground-truth and values are generated based on the configuration rules outlined in Listing 5. As an illustration, line 2 in Listing 3 aligns with lines 15 and 31 in Listing 5, which define the relevant type mappings.

The auto-generation process systematically computes all permutations of types for the placeholders. For example, with four configured types and two placeholders, the generator produces 12 unique programs based on the formula $P(n, r) = n!/(n - r)!$ where n is the total number of configured types, and r is the number of placeholders in a given template. Each program is generated with a unique arrangement of types, such as (str, float) and (str, int). This method enables the creation of a comprehensive range of programs with different type configurations, enhancing the diversity of the benchmark. Note that the values are generated randomly for each of these placeholders according to their data types. For instance, an example of the generated test case for this template is shown in Listing 4 and its associated ground truth in Listing 6.

Special cases, such as lists and dictionaries, require additional handling to ensure that every element within these data structures is correctly annotated. Similarly, imported code segments demand careful modelling to avoid inconsistencies in the generated programs. These complexities were addressed within the generator, which was carefully designed to ensure that all edge cases were correctly handled.

Once the programs are generated, each is executed to verify its correctness. If the program executes without errors, it is retained in the benchmark. In contrast, programs that fail due to type incompatibility, such as attempting to add a string to a float, are discarded. This filtering ensures that only valid test cases are included in the final benchmark.

The creation of the auto-generated benchmark was a collaborative effort. The first author was responsible for creating the initial templates, while the second author verified the generated programs for correctness, iteratively fixing errors and ensuring the accuracy of type annotations. Furthermore, note that the programs have a single execution path, therefore avoiding ambiguities in the ground-truth.

The auto-generation capability expands the TypeEvalPy benchmark with 7,121 Python files, containing a total of 77,268 type annotations. This increase in both the quantity and variety of annotated types ensures a more comprehensive framework for evaluating the performance and generalizability of LLMs in type inference tasks.

```python
def func1():
    return <value1>


def func2():
    return <value2>

a = func1()
b = func2()
```

Listing 3: Template for main.py

```python
def func1():
    return 34


def func2():
    return 53.24

a = func1()
b = func2()
```

Listing 4: Generated main.py

```
{
"replacement_mode": "Complex",
"type_replacements": [
    "int",
    "float",
    "str",
    "bool"
],
"ground_truth": [{
    "file": "main.py",
    "line_number": 1,
    "col_offset": 5,
    "function": "func1",
    "type": [
        "<value1>"
    ]},
    {
    "file": "main.py",
    "line_number": 5,
    "col_offset": 5,
    "function": "func2",
    "type": [
        "<value2>"
    ]},
    {
    "file": "main.py",
    "line_number": 8,
    "col_offset": 1,
    "variable": "a",
    "type": [
        "<value1>"
    ]},
    {
    "file": "main.py",
    "line_number": 9,
    "col_offset": 1,
    "variable": "a",
    "type": [
        "<value2>"
    ]}]
}
```

Listing 5: Autogen configuration for main.py

```
[{
    "file": "main.py",
    "line_number": 1,
    "col_offset": 5,
    "function": "func1",
    "type": [
        "int"
    ]
},
{
    "file": "main.py",
    "line_number": 5,
    "col_offset": 5,
    "function": "func2",
    "type": [
        "float"
    ]
},
{
    "file": "main.py",
    "line_number": 8,
    "col_offset": 1,
    "variable": "a",
    "type": [
        "int"
    ]
},
{
    "file": "main.py",
    "line_number": 9,
    "col_offset": 1,
    "variable": "a",
    "type": [
        "float"
    ]
}]
```

Listing 6: Generated ground truth for main.py

6 Methodology

We next describe the experimental setup, the model selection criteria, the prompt design, and the metrics used to investigate these RQs.

6.1 Model Selection

In this extension study, we selected LLMs for evaluation by focusing on organizations that are actively conducting research and releasing state-of-the-art models on the Hugging Face platform.[1] We shortlisted five prominent organizations from Hugging Face that are building foundational models: Alibaba, Google, Meta, Microsoft, and Mistral. Apart from the models with open weights, we chose OpenAI as the proprietary service provider to compare against open models.

We selected a total of *24* LLMs across all organizations. From the organizations we shortlisted, we included all the instruction-tuned models, which are fine-tuned for following user instructions, across all available parameter sizes. This included multiple variations of the models, such as 7B, 13B, and larger configurations, allowing for a comprehensive evaluation across different scales. In addition to general-purpose models, we also included specialized code models, which are optimized for code understanding and related tasks, as these models are expected to perform better on code-specific benchmarks.

Two closed-source models from OpenAI, gpt-4o and gpt-4o-mini, were included due to their superior performance in general-purpose tasks, providing a benchmark for comparison against open-source models. We limited the number of proprietary models we test to optimize costs and chose OpenAI's GPT models due to their popularity.

The list of models evaluated in this study is listed in the Table 2.

6.2 Prompt Design

To optimize prompt design, we adopted an iterative and experimental approach (Chen et al. 2023; Schulhoff et al. 2024). Initial efforts focused on enhancing the prompt by including detailed task descriptions and specifying the expected response format. Notably, we used a one-shot prompting technique, embedding an example question and answer within the prompt. The one-shot prompt example was designed using the simplest program that encapsulates key aspects of the expected output for the given task. For instance, in the type inference task, the example included variables of different types to ensure variety. The decision to use a simple example was primarily to ensure that the model's responses adhered to the desired format, enabling reliable parsing of the results. Additionally, using a complex example in a one-shot setting does not always improve performance. Prior research by Chen et al. (2023) indicates that for sufficiently complex models, like those used in this study, a well-structured zero-shot prompt can be as effective as, or even preferable to, a complex few-shot prompt.

Despite these refinements, we encountered challenges with the LLM's ability to produce *structured* outputs. Our experiments revealed that even with explicit instructions to generate outputs in JSON format, models struggled to deliver results that could be reliably parsed.

[1] https://huggingface.co/

Table 2 Selected Models and Parameter Sizes

Organization	Model Name	Parameter Size (billion)
Alibaba	Qwen/Qwen2-7B-Instruct	7B
	Qwen/Qwen2-72B-Instruct	72B
Google	google/gemma-2-2b-it	2B
	google/gemma-2-9b-it	9B
	google/gemma-2-27b-it	27B
GPT	gpt-4o	-
	gpt-4o-mini	-
Meta	meta-llama/CodeLlama-7b-Instruct-hf	7B
	meta-llama/CodeLlama-13b-Instruct-hf	13B
	meta-llama/CodeLlama-34b-Instruct-hf	34B
	meta-llama/Meta-Llama-3.1-8B-Instruct	8B
	meta-llama/Meta-Llama-3.1-70B-Instruct	70B
	TinyLlama/TinyLlama-1.1B-Chat-v1.0	1.1B
Microsoft	microsoft/Phi-3-small-128k-instruct	7.3B
	microsoft/Phi-3-medium-128k-instruct	14B
	microsoft/Phi-3.5-mini-instruct	3.8B
	microsoft/Phi-3.5-MoE-instruct	41.9B
	microsoft/Phi-3-mini-128k-instruct	3.8B
Mistral	mistralai/Mixtral-8x22B-Instruct-v0.1	141B
	mistralai/Mixtral-8x7B-Instruct-v0.1	46.7B
	mistralai/Mistral-7B-Instruct-v0.3	7B
	mistralai/Mistral-Nemo-Instruct-2407	12.2B
	mistralai/Mistral-Large-Instruct-2407	123B
	mistralai/Codestral-22B-v0.1	22B

To address this, we explored a question-answer based method, querying the model and then translating its natural-language responses back into a structured JSON format.

To further improve reliability, we analyzed the initial output to refine the prompt, particularly for cases where models failed to generate accurate results in response to a simple prompt. For instance, we noted that the aliases of program variables were not being considered in the final output. Therefore, we introduced generic instructions to ensure alias tracking in the program. Note that the same prompt is used to evaluate all models, including code models.

In the following sections, we discuss the prompts for type inference and call-graphs tasks in detail.

6.2.1 Type-inference Prompts

The prompt design employed in this study follows a structured two-part approach to guide the LLM through the task. The first part provides a detailed description of the task necessary to conduct the analysis, ensuring clarity in the expected operations. This is followed by the second part, which includes an example input-output pair in line with the one-shot prompting technique. Additionally, instructions on the format of the output are explicitly

provided to direct the model's responses. Finally, the code relevant to the task is added to the prompt. Note that for test cases with file imports, all the relevant file contents are added to the prompt with relative file names to indicate the file structure of the test case.

Despite the careful structuring, we encountered difficulties in the initial attempts to generate valid JSON output using this approach. Specifically, the model often failed to consistently produce JSON in the required format. The primary issues observed were missing keys or the inclusion of unexpected keys, attributed to the LLM's inability to adhere to the complex output schema. The underlying complexity of the type annotations schema of the TypeEvalPy framework presented additional challenges for LLMs.

To address these limitations, the task complexity was reduced by breaking the task down into a series of *question-answer* pairs and using the one-shot prompting technique. This approach simplifies the requirement to follow specific output schema and enhances its ability to follow the prompt more accurately. For example, as shown in Listing 9, three specific questions were generated based on the variables declared in the one-shot code example. These questions include the name of the variable and the location of its declaration. Additionally, placeholders were introduced for each question, with sequential numbers to indicate where the model should provide responses.

The actual questions were generated using ground-truth data. By iterating over the variables and functions listed in the ground-truth, appropriate questions were formulated. However, in a practical setting, this task could be automated using the program's abstract syntax tree (AST). For this study, the available ground truth data was used to simplify the implementation.

To clarify, consider the full prompt in Listing 10. In this case, from the ground-truth, we know that five program identifiers require type inference. The five questions in the prompt are generated by iterating over the ground-truth data and extracting the identifier names, along with their corresponding line and column numbers. In theory, this information could be obtained by parsing the AST of the program.

Finally, the model's responses were parsed using regular expressions, which enabled the correct mapping of answers back to the original questions. This method allowed for generating JSON outputs that adhered to the TypeEvalPy schema, which were then used for the evaluation. To demonstrate this in practice, we have listed an example with the source code, ground-truth, model response, and parsed JSON in Listing 12.1.

6.2.2 Call-graph Prompts

The design of prompts for call-graph analysis follows an approach similar to the one described in the previous section. Initially, a detailed description of the task is provided, which is followed by an example input-output pair according to the target language. The task description outlines the specific requirements for analyzing the call-graph, and instructions for formatting the output are included to ensure consistency in the model's responses, as shown in Listing 11.

To generate questions within these prompts, a method akin to the one used previously is used. The first question typically addresses function calls at the module level, followed by questions regarding each individual call made within function definitions as illustrated in Listing 12.

In practical scenarios, these questions can be generated by iterating through the AST of the program. By identifying function definitions and call nodes within the AST, the necessary information can be extracted. However, for this study, ground truth data was used to formulate the questions, allowing for a more straightforward implementation.

Additionally, for flow-sensitive call-graph analysis, the prompts were adjusted to accommodate the location of the call site. Listings 13 and 14 present the specific prompts used for constructing flow-sensitive call-graphs.

Note on Context Length The maximum prompt size encountered across both the call-graph and type inference benchmarks was 1,287 tokens, as measured by the gpt-4o tokenizer. This means that the prompts used in this study were comfortably within the context limits of all the LLMs evaluated. The model with the largest context size, gpt-4o, supports up to 128,000 tokens, while the smallest context size was offered by TinyLlama-1.1b, which has a limit of 2,048 tokens.

For reference, the cumulative size of the prompts from the entire TypeEvalPy micro-benchmark amounts to 69,563 tokens. Even in this case, the total prompt size remains well below the maximum context length of most models evaluated, ensuring that the models had enough capacity to process the full input without truncation.

6.3 Evaluation Metrics

In this study, we measured completeness, soundness, and exact matches to assess both flow-insensitive callgraph construction and flow-sensitive call-site extraction. Furthermore, we use the exact matches metric to evaluate type inference performance.

Completeness and Soundness In this study, we use the terms completeness and soundness as they have been pre-established in call-graph research (Salis et al. 2021; Venkatesh et al. 2024a). The terms completeness and soundness are closely related to the precision and recall metrics.

Precision is directly tied to completeness, as it measures the proportion of correctly identified call edges relative to all edges produced by the model. A complete call-graph will have perfect precision, as it contains no false positives. This terminology can be a bit confusing at first because it implies that a call-graph that is "incomplete" in the above sense is not one that misses call edges but one that has spurious edges. The reader shall keep that in mind.

Recall is closely related to soundness, as it measures the proportion of true call edges that are correctly identified. A sound call-graph will demonstrate perfect recall by including all true call edges, without omitting any.

Here, completeness and soundness are measured at the individual test case level within the benchmark. A test case is considered complete if there are no false positives in the generated call-graph for that specific case. Similarly, it is considered sound if there are no false negatives. This means that if even a single false positive or false negative is detected in the responses generated for a test case, it is marked as a failure in terms of completeness or soundness, respectively.

However, precision and recall have specific implications when evaluated at the level of individual test cases, particularly in a micro-benchmark setting. Rather than measuring how precise or recall-efficient a system is overall, it is more insightful to determine whether a test case is fully complete or sound with respect to the specific feature being tested. This binary evaluation, either complete or sound, provides clearer insights into whether specific features are fully captured, without the ambiguity that partial correctness metrics like precision or recall might introduce. This evaluation approach mirrors the methodologies used in previous studies, specifically in PyCG (Salis et al. 2021) and HeaderGen (Venkatesh et al. 2023b).

Exact Matches The exact-matches metric for the call-graph measures the number of function calls that exactly match the ground truth. To compute this, we compare the expected calls for each node in the ground truth with those produced by the model. For nodes where both lists are non-empty, we count exact matches when every element in the generated list appears in the ground truth. For nodes with empty lists, an exact match is counted if the model also produces an empty list.

Furthermore, aligning with the literature (Allamanis et al. 2020; Mir et al. 2022; Peng et al. 2022; Venkatesh et al. 2023a, 2024b), for type-inference evaluation, we use exact matches as the metric as well.

Time Time measurements were taken on open models, as they were all executed on the same hardware using identical parameters for model loading and inference. To ensure uniformity in the testing setup, all models were loaded using 4-bit quantization, with a batching size of 12. To ensure a fair comparison, we applied the same batching size across all models. While smaller models could, in practical scenarios, process more prompts per batch due to lower memory requirements, we chose to standardize the testing conditions. This approach prevents smaller models from having an advantage and allows for a fair assessment.

The time recorded represents the total time needed to process all benchmark test cases. Time measurements for OpenAI models were omitted, as they were inferred using a batch API that returns results after 24 hours at a 50% lower cost. Given that these models were not run on our hardware, a direct comparison with the open models would not be appropriate.

6.4 Implementation Details

For the implementation of our experiments, we used the Hugging Face transformers (Wolf et al. 2020) Python interface to run LLMs on our hardware. This interface provides a flexible and efficient environment to manage inference tasks across multiple models. The models were loaded using 4-bit quantization, with a batch size of 12, and configured to use greedy search. Greedy search was chosen to always select the most probable next token, ensuring deterministic outputs across all runs.

To conduct the type-inference experiments, we extended the existing TypeEvalPy framework. This allowed for seamless integration with our testing pipeline. For the call-graph experiments, we built a custom adaptor within the SWARM-CG framework.

All experiments were run on the following hardware configuration: one NVIDIA H100-80GB GPU, 16 Intel(R) Xeon(R) Platinum 8462Y+ processors, and 78 GB of memory.

Note on Quantization To optimize resource utilization, we chose to load models using 4-bit quantization, allowing large models to be efficiently deployed on a single H100 GPU with 80GB of memory. This approach significantly reduces computational and memory requirements while maintaining the feasibility of running extensive experiments. Furthermore, prior research indicates that quantization has minimal impact on the accuracy of large models, making it a viable strategy for balancing efficiency and performance (Lang et al. 2024; Jin et al. 2024; Dettmers et al. 2023).

7 Results

We next address the research questions and highlight the key results from our different analysis.

7.1 RQ1: Accuracy of Callgraph Analysis

The results of our experiments for flow-insensitive call-graph analysis for Python and JavaScript are presented in Tables 3 and 4, respectively. The Python results are based on the PyCG micro-benchmark suite, while the JavaScript results use the SWARM-JS micro-benchmark. Additionally, Table 6 provides the results for flow-sensitive call-graph analysis based on the HeaderGen micro-benchmark. In the following sections, we discuss each of these results in detail.

7.1.1 Flow-insensitive Call-graph Analysis

This section reports the results obtained in the **flow-insensitive call-graph analysis** for Python and JavaScript programs, separately.

Python Programs The results of our evaluation, presented in Table 3, highlight the superior performance of the static analysis tool PyCG compared to LLMs in terms of completeness, soundness, exact matches, and processing time. Specific rows and values that are discussed in the text are highlighted in the table for clarity.

In a benchmark of 126 test cases, PyCG achieved 84.9% completeness and 87.3% soundness. This means that for the majority of test cases, PyCG produced no false positives (completeness) and missed very few valid function calls (soundness). These results significantly surpass those of the closest competing model, mistral-large-it-2407-123b, which attained 60.3% completeness and 62.6% soundness. Additionally, PyCG produced 569 exact matches out of 599, outperforming mistral-large-it-2407-123b by 51 matches.

The model mistral-large-it-2407-123b shows moderate performance in both completeness and soundness. However, it achieved a high exact match score of 86.4%, indicating that while it correctly identifies many function-call relations, it also introduces false positives and misses valid ones. This leads to failures in both completeness and soundness across many test cases, suggesting that the model lacks support for certain Python language features.

Table 3. Comparative analysis across LLMs for flow-insensitive call-graph analysis on the PyCG Python micro-benchmark

80-100%, 60-80%, 40-60%, 20-40%, 0-20%

Model	Complete	Sound	Exact Matches	Time (s)
	126 cases		599 cases	
PyCG	**107**	**110**	**569**	**0.41**
mistral-large-it-2407-123b	76	79	518	534.01
gpt-4o	63	75	486	n/a
qwen2-it-72b	35	67	427	286.17
llama3.1-it-70b	37	63	424	277.69
gpt-4o-mini	47	47	397	n/a
mistral-nemo-it-2407-12.2b	50	44	358	**59.35**
gemma2-it-27b	29	43	344	180.94
llama3.1-it-8b	2	40	179	185.52
mixtral-v0.1-it-8x22b	**2**	65	**173**	1106.46
phi3.5-moe-it-41.9b	9	25	166	**3335.89**
phi3-small-it-7.3b	3	10	150	73.92
phi3-medium-it-14b	3	29	142	140.55
codellama-it-34b	64	23	130	545.96
qwen2-it-7b	2	42	128	52.74
mistral-v0.3-it-7b	4	25	122	74.74
codestral-v0.1-22b	34	24	120	582.78
tinyllama-1.1b	35	5	91	507.21
gemma2-it-9b	**120**	4	**63**	**1587.85**
mixtral-v0.1-it-8x7b	9	19	52	1221.64
codellama-it-13b	20	17	43	619.04
codellama-it-7b	7	4	31	281.14
phi3-mini-it-3.8b	1	7	24	85.48
gemma2-it-2b	**117**	1	**10**	**731.7**
phi3.5-mini-it-3.8b	2	6	9	81.3

gpt-4o ranks third, performing behind mistral-large-it-2407-123b, which suggests that open-source models may be catching up to closed-source models. However, most other open-source models underperformed significantly.

The model gemma2-it-9b displayed a notable discrepancy between completeness (95%) and soundness (3%), suggesting that while it rarely introduces false positives, it misses a vast number of valid function calls, leading to numerous test cases failing the soundness criterion. The poor exact match score of 10.5% reflects this imbalance. Furthermore, its runtime of 1587 seconds makes it surprising given that the model is relatively small with 9 billion parameters.

The poor performance of mixtral-v0.1-it-8x22b, especially for a model with 141 billion parameters, demonstrates its limitations in handling the test cases. On the contrary, tinyllama-1.1b, despite being a smaller model, took significant time to process and performed poorly across all metrics.

Table 4 Comparative analysis across LLMs for **flow-insensitive** call-graph analysis on the SWARM-JS JavaScript micro-benchmark

80-100%, 60-80%, 40-60%, 20-40%, 0-20%

Model	Complete	Sound	Exact Matches	Time (s)
	126 cases		596 cases	
Jelly	**49**	**85**	**490**	643.51
mistral-large-it-2407-123b	**51**	**54**	**458**	537.86
gpt-4o	44	**64**	451	n/a
qwen2-it-72b	24	51	406	367.87
llama3.1-it-70b	28	34	357	251.91
gpt-4o-mini	32	30	347	n/a
mistral-nemo-it-2407-12.2b	44	23	311	56.5
gemma2-it-27b	14	14	280	188.88
llama3.1-it-8b	4	36	212	85.28
codestral-v0.1-22b	35	19	145	496.08
phi3.5-moe-it-41.9b	3	12	145	**3344.09**
phi3-small-it-7.3b	2	2	141	83.26
mistral-v0.3-it-7b	6	19	132	1297.24
phi3-medium-it-14b	2	17	130	145.18
codellama-it-34b	74	14	103	540.06
mixtral-v0.1-it-8x22b	2	33	94	90.21
qwen2-it-7b	2	26	87	43.3
TAJS	**119**	14	83	**135.65**
gemma2-it-9b	**118**	2	54	**1593.44**
phi3-mini-it-3.8b	2	2	40	77.61
tinyllama-1.1b	14	1	37	499.58
codellama-it-7b	14	0	30	272.88
codellama-it-13b	3	9	21	598.55
phi3.5-mini-it-3.8b	4	3	20	89.2
gemma2-it-2b	**116**	1	13	**721.01**
mixtral-v0.1-it-8x7b	3	1	8	578.66

To clarify how the results are parsed and evaluated, we provide an example in the appendix, showcasing the source code, ground truth, raw LLM response, and parsed call-graph JSON for both the top-performing model, mistral-large-it-2407-123b, and the least-performing model, phi3.5-mini-it-3.8b, for the same test case in our benchmark. These examples can be found in Sections 12.2 and 12.3, respectively.

JavaScript Programs The results from analyzing the JavaScript benchmark (SWARM-JS) are presented in Table 4.

Jelly, a hybrid static analysis tool, demonstrated strong performance in our evaluation. When executed with its approximate interpretation feature enabled (Laursen et al. 2024), Jelly achieved a completeness score of 38.8%, soundness of 67.4%, and a high exact match

rate of 82.2%. This configuration incorporates dynamic execution hints into the static analysis process, improving the tool's ability to resolve function calls accurately.

In contrast, TAJS, which was previously shown to perform well in call-graph generation by Antal et al. (2023), performed poorly in our evaluation. Although its results appeared to exhibit a low false-positive rate, further inspection revealed that this was due to widespread failures: 102 out of 126 SWARM-JS test cases resulted in analysis errors and produced empty outputs. This is because TAJS only supports ECMAScript 3rd edition, whereas the SWARM-JS benchmark includes features from ECMAScript 6th edition, such as classes and arrow functions.

The mistral-large-it-2407-123b model achieved 40.4% completeness and 42.8% soundness, making it the top-performing model overall, with an exact match score of 76.8%. Its runtime of 537.86 seconds, while not the fastest, is expected for a model of its size. gpt-4o achieved 34.9% completeness and 50.7% soundness, with a total of 451 exact matches, performing closely to mistral-large-it-2407-123b for capturing valid function calls.

Models gemma2-it-9b and gemma2-it-2b showed high completeness scores (118 and 116, respectively), but very low soundness (2 and 1, respectively). This indicates that although these models generated few false positives, they missed nearly all valid function calls, leading to largely empty call-graphs. Furthermore, gemma2-it-9b had a very high runtime of 1593.44 seconds, making it both inefficient and ineffective.

Comparative Analysis of Python and JavaScript Results In this section, we discuss the performance of LLMs across flow-insensitive call-graph evaluation for Python and JavaScript. Table 5 compares the top 10 performing LLMs based on exact match rates for Python and JavaScript programs. Overall, the models exhibited stronger performance in Python than in JavaScript programs. The leading model, mistral-large-it-2407-123b, achieved an exact match rate of 86.6% in Python, outperforming its JavaScript results, where it reached 76.8%. This performance gap is consistent across other models, all of which show a noticeable decline in accuracy across metrics when evaluated on JavaScript.

Table 5 Percentage comparison of models across Python and JavaScript **flow-insensitive call-graph** evaluations

80-100%, 60-80%, 40-60%, 20-40%, 0-20%

Model	Python (%)			JavaScript (%)		
	Comp.	Sound	EM.	Comp.	Sound	EM.
mistral-large-it-2407-123b	60.32	62.70	86.64	40.48	42.86	76.85
gpt-4o	50.00	59.52	81.14	34.92	50.79	75.67
qwen2-it-72b	27.78	53.17	71.29	19.05	40.48	68.12
llama3.1-it-70b	29.37	50.00	70.78	22.22	26.98	59.90
gpt-4o-mini	37.30	37.30	66.28	25.40	23.81	58.22
mistral-nemo-it-2407-12.2b	39.68	34.92	59.93	34.92	18.25	52.18
gemma2-it-27b	23.02	34.13	57.43	11.11	11.11	46.98
llama3.1-it-8b	1.59	31.75	29.88	3.17	28.57	35.57
mixtral-v0.1-it-8x22b	1.58	51.58	28.88	1.58	26.19	15.77
phi3.5-moe-it-41.9b	7.14	19.84	27.71	2.38	9.52	24.33

7.1.2 Flow-sensitive Call-graph Analysis

This section reports the results obtained in the **flow-sensitive call-graph analysis** for Python programs.

Python Programs Table 6 presents the results of flow-sensitive call-graph analysis on the HeaderGen micro-benchmark, comparing the performance of various LLMs and the static analysis tool HeaderGen.

HeaderGen outperforms LLMs by achieving a completeness of 90.9% and soundness of 91.8%. HeaderGen had 326 exact matches out of 357, achieving a score of 91.3%. This demonstrates that HeaderGen ensures low false positives and false negatives rates. Additionally, it achieves this in 10.94 seconds, highlighting its efficiency compared to LLMs.

Among the LLMs, mistral-large-it-2407-123b stands out as the best-performing model, although it still falls significantly short of HeaderGen. It achieved 31.1% completeness and 26.2% soundness, with 38 complete and 32 sound cases. Its 28.5% exact match score (102 out of 357 cases) further highlights its limitations in capturing all function calls correctly.

All the other models underperformed in every metric, indicating that they failed to accurately capture the majority of function calls in the benchmark. When comparing these results to flow-insensitive analysis, the performance of LLMs further deteriorates. The increased complexity of flow-sensitive analysis, which requires specificity about the location of function calls, poses additional challenges for LLMs. This seems to significantly reduce their ability to capture correct relationships, further highlighting the limitations of LLMs in handling more complex, context-specific analysis tasks.

> Static analysis tools such as PyCG, Jelly, and HeaderGen generally outperformed LLMs in flow-insensitive and flow-sensitive call-graph analysis for both Python and JavaScript. While `mistral-large` was the top-performing LLM, its overall accuracy consistently fell short of dedicated static analysis tools, highlighting the challenges LLMs face with complex code analysis.

7.2 RQ2: Accuracy of Type Inference

7.2.1 TypeEvalPy Micro-benchmark

Table 7 shows the results of LLMs, HeaderGen, and HiTyper considering the exact-match performance on the TypeEvalPy micro-benchmark. Note that the hybrid analysis tool HiTyper is configured with Type4Py (Mir et al. 2022). The results highlight that LLMs, particularly recent and larger models, significantly outperform previous approaches like HeaderGen and HiTyper. Among the models evaluated, OpenAI's gpt-4o emerges as the best-performing model, correctly inferring 806 of the total 860 type annotations. This aligns with expectations, as gpt-4o is known for its extensive parameter count and advanced capabilities. However, its performance comes at the potential cost of speed and computational expense, factors crucial for practical deployment in real-world applications.

Table 6 Comparative analysis across LLMs for **flow-sensitive** call-graph analysis on the HeaderGen micro-benchmark

80-100%, 60-80%, 40-60%, 20-40%, 0-20%

Model	Complete	Sound	Exact Matches	Time (s)
	122 cases		357 cases	
HeaderGen	**111**	**112**	**326**	**10.94**
mistral-large-it-2407-123b	38	32	102	**581.87**
mixtral-v0.1-it-8x22b	23	21	75	1123.6
gpt-4o	31	26	**74**	n/a
qwen2-it-72b	22	19	70	289.96
codestral-v0.1-22b	14	13	65	369.05
gemma2-it-27b	22	15	55	173.85
llama3.1-it-70b	19	17	54	362.55
gpt-4o-mini	17	12	**36**	n/a
mixtral-v0.1-it-8x7b	11	11	32	1000.04
llama3.1-it-8b	14	11	31	356
phi3-medium-it-14b	10	9	31	131.57
phi3-mini-it-3.8b	14	11	31	63.38
mistral-nemo-it-2407-12.2b	18	11	26	54.06
phi3.5-mini-it-3.8b	8	7	21	296.41
codellama-it-34b	14	10	18	320.42
qwen2-it-7b	11	9	16	48.82
mistral-v0.3-it-7b	7	7	15	58.1
tinyllama-1.1b	11	8	12	99.89
phi3-small-it-7.3b	9	8	11	324.57
phi3.5-moe-it-41.9b	8	6	8	2771.88
codellama-it-13b	8	7	6	256.57
gemma2-it-9b	6	6	5	1558.06
codellama-it-7b	6	6	0	222.17
gemma2-it-2b	6	6	0	716.7

Notably, the mistral-large-it-2407-123b model closely follows gpt-4o, correctly predicting 804 type annotations, showing how large open-source models are closing the performance gap with proprietary LLMs. This is significant because it implies that with proper tuning and architecture, open-source models can rival closed-source models, providing a potentially more accessible and cost-effective alternative for type inference tasks. Furthermore, specialized models like CodeLLaMA, particularly the 13B-instruct variant, shows good performance with 728 exact matches, suggesting that fine-tuning models specifically for code-related tasks offers a distinct advantage over general-purpose LLMs like vanilla LLaMA. In contrast, smaller models such as TinyLlama (1.1B parameters) exhibit poor performance, correctly predicting only 102 annotations, implying that model size is a critical factor for complex tasks like type inference.

From the inference speed perspective, there is a noticeable trade-off between model size, accuracy, and efficiency. For instance, while larger models like phi3.5-moe-it-41.9b achieve

relatively high accuracy, they incur significant inference times (3,574.35 seconds). In contrast, mid-sized models such as Codellama-it-13b strike a better balance, delivering decent performance with 728 exact matches in a considerably shorter time frame (92.81 seconds). This suggests that when selecting models for type inference in practice, one must consider not only accuracy but also the computational resources and speed required, especially for large-scale projects or environments with limited hardware.

7.2.2 TypeEvalPy Autogen Benchmark

In Table 8, we present the results of the same models on the significantly larger and extended TypeEvalPy autogen benchmark. Additionally, in Table 9, we list the differences in performance based on the total exact matches between the TypeEvalPy micro and autogen benchmarks.

Models with Consistent Performance Models in this category demonstrate a maximum delta of 5% between the micro-benchmark and autogen-benchmark scores. Notably, gpt-4o and mistral-large-it-2407-123b maintained high exact match across both benchmarks, with deltas of 2.38% and 3.48%, respectively. The close alignment of these results suggests these models are robust across different testing scenarios, crucial for real-world applications, where model performance needs to generalize across varied datasets. gpt-4o-mini and codestral-v0.1-22b showed a slight decline of 3.77% and 2.31% respectively. However, these models remained within the acceptable variance threshold, suggesting they are still usable for the type inference tasks. Additionally, HeaderGen, with a delta of just 0.26%, demonstrates the robustness of static analysis tools.

Models that Improved Three models showed improvements of more than 5% in exact matches from the micro-benchmark to the autogen-benchmark. mixtral-v0.1-it-8x22b improved by 7.18%, and qwen2-it-72b and mistral-v0.3-it-7b increased by 8.89% and 9.61%, respectively.

Models that Deteriorated Conversely, several models showed significant performance declines between the two benchmarks. mixtral-v0.1-it-8x7b and phi3.5-moe-it-41.9b exhibited the largest declines, with mixtral-v0.1-it-8x7b deteriorating by a -75.05% and phi3.5-moe-it-41.9b by -67.13%. The decline in performance indicates a possible overfitting to the *string* datatype that is primarily found in the micro-benchmark.

> LLMs, particularly larger models like gpt-4o and `mistral-large`, demonstrated superior performance in Python type inference compared to traditional static analysis tools such as HeaderGen and HiTyper, on the TypeEvalPy micro-benchmark. These top-performing LLMs also showed strong consistency when evaluated on the larger TypeEvalPy Autogen benchmark.

8 Discussion

In this section, we discuss the implications of the empirical results observed in the study. We first analyze call-graph construction in Python and JavaScript, highlighting strengths and weaknesses in different scenarios. We then discuss type inference performance in Python,

Table 7 Exact match comparison of LLMs for **type inference** on TypeEvalPy micro-benchmark

80-100%, 60-80%, 40-60%, 20-40%, 0-20%

FRT: Function return type, **FPT:** Function parameter type, **LVT:** Local variable type

Model	FRT	FPT	LVT	Total	Time (s)
Total	239	88	533	860	
gpt-4o	**217**	**81**	**508**	**806**	n/a
mistral-large-it-2407-123b	222	80	502	**804**	626.46
gpt-4o-mini	212	80	491	783	n/a
llama3.1-it-70b	215	70	485	770	279.73
codestral-v0.1-22b	209	82	469	760	242.05
mixtral-v0.1-it-8x22b	196	68	492	756	674.79
gemma2-it-27b	202	73	479	754	175.14
codellama-it-13b	193	77	458	**728**	**92.81**
qwen2-it-72b	202	70	456	728	303.35
codellama-it-34b	192	67	464	723	196.09
phi3-medium-it-14b	198	77	429	704	106.57
mistral-nemo-it-2407-12.2b	196	71	433	700	59.75
mixtral-v0.1-it-8x7b	181	71	434	686	302.86
phi3-small-it-7.3b	180	66	406	652	81.51
mistral-v0.3-it-7b	177	78	387	642	51.51
llama3.1-it-8b	175	69	394	638	170.32
phi3.5-moe-it-41.9b	158	68	389	**615**	3,574.35
phi3-mini-it-3.8b	175	57	372	604	131.15
phi3.5-mini-it-3.8b	171	55	344	570	111.32
headergen	186	56	321	563	18.25
qwen2-it-7b	167	58	338	563	56.61
codellama-it-7b	164	56	338	558	137.01
hityperdl	163	27	177	367	268.4
gemma2-it-9b	22	16	72	110	1,521.89
tinyllama-1.1b	42	7	53	**102**	416.63
gemma2-it-2b	21	10	49	80	680.03

comparing LLMs with traditional tools. Subsequently, we explore differences in LLM performance between type inference and call-graph analysis, followed by an examination of cross-language disparities, general discussions, and propose avenues for future research.

8.1 Call Graph Construction in Python: LLMs vs Static Analysis Tools

In Python, the static analysis tool PyCG consistently outperformed LLMs in constructing call-graphs, with mistral-large-it-2407-123b (mistral-large) ranking highest among the evaluated LLMs. Table 10 compares PyCG and mistral-large across selected categories from the PyCG micro-benchmark, chosen specifically for similarities and differences in tool performance. Furthermore, in Table 11, we list specific patterns in which LLMs struggled.

Within the *returns* category, both PyCG and mistral-large accurately resolved cases involving direct function returns and imported functions. However, mistral-large failed to handle scenarios with indirect imports through intermediate modules correctly, introducing

Table 8 Exact match comparison of LLMs for **type inference** on TypeEvalPy autogen benchmark

80-100%, 60-80%, 40-60%, 20-40%, 0-20%

FRT: Function return type, **FPT:** Function parameter type, **LVT:** Local variable type

Model	FRT	FPT	LVT	Total	Time (s)
Total	17,998	896	58,374	77,268	
mistral-large-it-2407-123b	**16,701**	**728**	**57,550**	**74,979**	**28,435**
gpt-4o	16,804	737	56,716	74,257	n/a
mixtral-v0.1-it-8x22b	16,424	514	56,550	73,488	22,959
qwen2-it-72b	16,488	629	55,160	72,277	15,877
llama3.1-it-70b	16,648	580	54,445	71,673	14,945
gpt-4o-mini	16,628	643	50,162	67,433	n/a
gemma2-it-27b	16,342	599	49,772	66,713	9,121
codestral-v0.1-22b	16,456	706	49,379	66,541	7,437
codellama-it-34b	15,960	473	48,957	65,390	10,983
mistral-nemo-it-2407-12.2b	16,221	526	48,439	65,186	3,227
mistral-v0.3-it-7b	16,686	472	47,935	65,093	2,589
phi3-medium-it-14b	16,802	467	45,121	62,390	6,038
llama3.1-it-8b	16,125	492	44,313	60,930	3,168
codellama-it-13b	16,214	479	43,021	59,714	4,485
phi3-small-it-7.3b	16,155	422	38,093	54,670	4,400
qwen2-it-7b	15,684	313	38,109	54,106	4,483
headergen	14,086	346	36,370	50,802	114
phi3-mini-it-3.8b	15,908	320	30,341	46,569	3,506
phi3.5-mini-it-3.8b	15,763	362	28,694	44,819	3,631
codellama-it-7b	13,779	318	29,346	43,443	5,528
hityperdl	15,765	61	5,365	21,191	6,564
gemma2-it-9b	1,611	66	5,464	7,141	57,828
tinyllama-1.1b	1,514	28	2,699	4,241	13,819
mixtral-v0.1-it-8x7b	3,235	33	377	3,645	78,215
phi3.5-moe-it-41.9b	3,090	25	273	3,388	121,617
gemma2-it-2b	1,497	41	1,848	3,386	23,637

false positives and omissions, while PyCG resolved these cases correctly. In complex multi-level return structures, as illustrated by the *return_complex* test case, mistral-large missed call edges, resulting in unsoundness, whereas PyCG successfully identified all call relationships.

Complex return constructs, particularly those involving Python's generator feature using yield statements, are common in real-world projects. Yield-based generator returns are the third most frequent functional feature in the dataset consisting of over 3.1 million Python files from 51,493 GitHub repositories created by Yang et al. (2022). This highlights the real-world significance of accurately handling complex return constructs. The *dicts* category demonstrated stronger performance by mistral-large, which achieved perfect completeness, soundness, and exact match rates, surpassing PyCG. Particularly notable was mistral-large's correct handling of dictionary updates using the *update()* method, a scenario where PyCG incorrectly missed a call edge.

Python dictionary features contribute to efficient and flexible data storage. Beyond direct method calls like update(), dictionary comprehensions serve as an efficient way to create and manipulate dictionaries in real-world code. The study by Yang et al. (2022) categorizes

Table 9 Exact matches comparison between micro-benchmark and autogen-benchmark percentages

80-100%, 60-80%, 40-60%, 20-40%, 0-20%

Model	Micro (%)	Autogen (%)	Diff. (%)
Models with Consistent Performance (5% Delta)			
gpt-4o	93.72	96.10	**2.38**
mistral-large-it-2407-123b	93.49	96.97	**3.48**
gpt-4o-mini	91.05	87.28	**-3.77**
llama3.1-it-70b	89.53	92.75	3.22
codestral-v0.1-22b	88.37	86.06	**-2.31**
gemma2-it-27b	87.67	86.28	-1.39
codellama-it-34b	84.07	84.64	0.57
phi3-medium-it-14b	81.86	80.74	-1.12
mistral-nemo-it-2407-12.2b	81.40	84.38	2.98
llama3.1-it-8b	74.19	78.89	4.70
qwen2-it-7b	65.47	70.02	4.55
HeaderGen	65.47	65.73	**0.26**
gemma2-it-9b	12.79	9.24	-3.55
Models that Improved (Minimum 5% Increase)			
mixtral-v0.1-it-8x22b	87.91	95.09	7.18
qwen2-it-72b	84.65	93.54	8.89
mistral-v0.3-it-7b	74.65	84.26	9.61
Models that Deteriorated (Minimum 5% Decrease)			
codellama-it-13b	84.65	77.26	-7.39
mixtral-v0.1-it-8x7b	79.77	4.72	**-75.05**
phi3-small-it-7.3b	75.81	70.76	-5.05
phi3.5-moe-it-41.9b	71.51	4.38	**-67.13**
phi3-mini-it-3.8b	70.23	60.27	-9.96
phi3.5-mini-it-3.8b	66.28	57.99	-8.29
codellama-it-7b	64.88	56.22	-8.66
hityperdl	42.67	27.43	-15.24
tinyllama-1.1b	11.86	5.49	-6.37

dictionary comprehension constructs as functional features and recorded 81,763 occurrences in their dataset, ranking them as the fifth most frequent functional feature.

In contrast, within the *functions* category, both PyCG and mistral-large demonstrated perfect accuracy, indicating comparable performance in resolving direct calls, variable assignments, and cross-module function imports.

> In constructing call-graphs for Python, PyCG consistently outperformed LLMs overall, with `mistral-large` showing competitive performance in specific cases such as dictionary updates, although it exhibited unsoundness in complex return structures and indirect imports where PyCG remained accurate.

8.2 Call Graph Construction in JavaScript: LLMs vs Static Analysis Tools

In JavaScript, the static analysis tool Jelly outperformed LLMs in terms of soundness and exact match rates, whereas the TAJS static analysis tool produced poor results due to its lack of support for modern JavaScript features and inactivity in recent years. Table 12 compares Jelly and mistral-large on the SWARM-JS micro-benchmark across categories chosen for their distinct and overlapping tool behaviours.

Table 13 lists specific challenging patterns where LLMs failed. In the *arguments* category, Jelly achieved perfect soundness, identifying all valid call edges, and completeness in 4 out of 10 test cases, while mistral-large matched Jelly's completeness but was sound in only half of the cases. Both tools effectively handled direct function passing and default arguments, though mistral-large struggled with indirect argument flows and cross-file imports.

Modern JavaScript features related to argument handling, such as default parameters and spread arguments, are widely adopted, appearing in over 56% and 60% of projects, respectively, in a study of 158 open-source systems (Lucas et al. 2025). Arrow Function Declaration, which offer concise argument syntax, is a highly popular feature, present in nearly 88% of projects in this dataset. The prevalence of these features underscores the importance of correctly resolving call graphs involving diverse argument patterns.

The *classes* category showed Jelly's clear superiority, achieving soundness in all test cases, whereas mistral-large showed significant challenges, particularly with inheritance, chained attribute references, and destructured assignments. In tests involving inheritance and method assignment, such as *base_class_calls_child*, Jelly reliably resolved call edges, contrasting mistral-large's frequent misses. The adoption of class syntax in JavaScript is substantial. A study by Nishiura et al. (2024) on 636 GitHub projects found that over half of the projects

Table 10 Category-wise call-graph construction performance on Python micro-benchmark

80-100%, 60-80%, 40-60%, 20-40%, 0-20%

Category	PyCG			mistral-large-it-2407-123b		
	Complete	Sound	E.M.	Complete	Sound	E.M.
returns	4/4	4/4	24/24	3/4	2/4	22/24
dicts	9/12	11/12	40/41	12/12	12/12	41/41
functions	4/4	4/4	9/9	4/4	4/4	9/9

Table 11 Challenging patterns that affect tool performance on Python benchmark

Sub-category	Concise example
	Returns

return_complex	`def func3():`
	`  return func2`
	
	`def func4(a):`
	`  return func3()`
	
	`...`
	
	`func4()()`

	Dicts

update	`def func1():`
	`  pass`
	
	`def func2():`
	`  pass`
	
	`d = {"a": func1}`
	
	`d.update({"a": func2})`
	`d["a"]()`

use class syntax, indicating a shift towards class-based programming. Furthermore, class inheritance (extends) is widely utilized, appearing in nearly 70% of projects that use classes.

Within the *objects* category, both Jelly and mistral-large effectively managed direct object access and calls via parameter returns. However, mistral-large surpassed Jelly in cases involving dynamically derived object keys from function parameters or external modules. Nevertheless, Jelly correctly handled a test case involving type coercion, which mistral-large did not, thus missing a call edge. A study by Lucas et al. (2025) found that object features like object destructuring are common, appearing in nearly 69% of projects in a dataset of 158 open-source JavaScript projects.

> In JavaScript call-graph construction, Jelly consistently outperformed LLMs, including `mistral-large`, particularly in classes and complex argument flows, while `mistral-large` demonstrated competitive handling of dynamic object keys but showed significant unsoundness in inheritance and indirect data flows.

8.3 Type Inference in Python: LLMs vs Static Analysis Tools

Table 14 lists the type-inference performance of mistral-large and HeaderGen on Micro and Autogen benchmarks. The analysis concentrates on three language constructs: assignments, decorators, and generators, which show a large performance difference.

HeaderGen 's errors arise primarily from the absence of modelling edge-case language constructs. Table 15 shows complex cases where HeaderGen failed to infer the types correctly. In assignments, it lacks rules for augmented updates, star unpacking, and tuples that are repeatedly unpacked and repacked, where HeaderGen falls back to the generic type Any. In decorators, HeaderGen misses the decorators that change a function's signature or return type. In generators, it does not track the return type of a user-defined __next__ method to the type produced by the function that is yielding.

Developing and maintaining such fine-grained modelling of language constructs is laborious. Static analysers, therefore, default to conservative approximation as Any. In contrast, an LLM acquires these behaviors implicitly through large-scale exposure to real-world repositories, learning that int += int remains an int and that a wrapper can replace a function's return type, it therefore preserves precise types where HeaderGen widens to Any.

> In Python type inference, `mistral-large` surpassed the static analysis tool HeaderGen by accurately preserving precise types in complex constructs such as augmented assignments, decorators, and generators, whereas HeaderGen defaulted to conservative approximations due to limited modelling of edge-case language constructs.

8.4 LLM Performance Differences: Type Inference vs. Call Graph Analysis

The empirical results demonstrate that LLMs show notably stronger performance in type inference tasks compared to call-graph analysis. However, explaining this behavior of LLMs is challenging, as their performance often emerges from complex interactions between training data, model architecture, and task formulation. Nonetheless, a likely explanation lies in the nature of LLM training: Python type annotations are embedded directly within source code and naturally align with next-token prediction objectives, enabling models to learn type patterns during pretraining. Although the micro-benchmark used in this study was newly created, lacked in-code annotations, and was unlikely to have been seen during pretraining, the LLMs were still able to generalize and perform well. Type inference is also benefited in certain instances, such as local variable assignments, where complex understanding of global program behavior is not always necessary, and types can often be inferred from the nearby context. By contrast, call-graph construction requires reasoning about control flows and structural relationships, which are harder to infer from token sequences alone, presenting greater challenges for LLMs. These capabilities are less likely to emerge purely from scale and pretraining on language-like sequences (Berti et al. 2025). Additionally, O'Brien et al.

Table 12 Category-wise call-graph construction performance on JavaScript micro-benchmark

● 80-100%, ◕ 60-80%, ◑ 40-60%, ◔ 20-40%, ○ 0-20%

Category	Jelly			mistral-large-it-2407-123b		
	Complete	Sound	E.M.	Complete	Sound	E.M.
args	4/10 ◔	10/10 ●	37/37 ●	4/10 ◔	5/10 ◑	30/37 ●
classes	5/21 ◔	20/21 ●	87/92 ●	1/21 ○	3/21 ◔	55/92 ◑
objects	7/12 ◑	9/12 ◕	37/41 ●	11/12 ●	11/12 ●	40/41 ●

Table 13 Challenging patterns impacting call-graph construction on JavaScript benchmark

Sub-Category	Example Snippet
	Arguments
nested_call	`function paramFunc(a) { a(); } function func(a) { a(function nestedFunc() {...}); } const b = paramFunc; const c = func; c(b);`
imported_call	`import { func } from "./to_import.js"; function paramFunc() { } func(paramFunc);`
	Classes
base_class_calls_child	`class A { func() { if (this.child) { this.child(); } } } class B extends A { constructor() { super(); this.child = this.func2; } func2() {...} } ... const b = new B(); b.func();`
	Objects
type_coercion	`function func1() {...} function func2() {...} let d = {1: func1, "1": func2}; d[1]();`

(2024) found no emergent improvements for software engineering tasks such as bug fixing or code analysis with increased model scale, suggesting that such tasks demand reasoning mechanisms not easily captured by current LLM architectures or pretraining regimes.

LLMs demonstrated stronger performance in type inference than call-graph analysis, likely due to the alignment of type information with token prediction objectives during pretraining, whereas call-graph construction requires complex global reasoning that is less accessible through local token patterns.

8.5 Cross-language Performance Disparities

A consistent observation throughout our experiments is that LLM performance is notably better on Python code than on JavaScript. Early studies have indicated that LLMs perform better on code generation tasks in Python compared to JavaScript (Buscemi 2023), likely due to Python's simpler syntax and more uniform structure. This has been further speculated by prior work (Chen et al. 2021), which highlights the influence of cleaner semantics and stronger conventions in Python. Despite these insights, a more systematic investigation is required to understand the underlying causes of this performance gap.

8.6 Implications for Type Inference in Dynamic Languages

Our findings suggest that LLMs are surpassing traditional tools in type inference tasks, especially in dynamically typed languages like Python. This has significant implications for large codebases, where manual annotation is often infeasible. Accurate type inference can substantially improve code readability, enable better tooling (e.g., code completion, static analysis), and facilitate the gradual adoption of type annotations in legacy projects.

Models such as gpt-4o and mistral-large-it-2407-123b demonstrate superior accuracy in inferring types. This capability suggests a potential shift in how type information is extracted and utilized within development workflows, moving from static, rule-based systems toward data-driven, context-aware assistants. However, the deployment of these models is not without challenges. Their computational requirements, including high memory usage and inference latency, can make LLMs difficult to integrate into resource-constrained environments such as continuous integration pipelines or lightweight IDE plugins. Furthermore, concerns

Table 14 Category-wise type-inference performance across Micro and Autogen benchmarks

80–100, 60–80, 40–60, 20–40, 0–20

FRT: Function-return type, **FPT:** Function-parameter type, **LVT:** Local-variable type

	Ground Truth			mistral-large-it-2407-123b			HeaderGen		
Category	FRT	FPT	LVT	FRT (%)	FPT (%)	LVT (%)	FRT (%)	FPT (%)	LVT (%)
Micro-benchmark									
assignments	22	4	56	95.5	100.0	100.0	68.2	25.0	58.9
decorators	29	17	12	86.2	70.6	58.3	37.9	35.3	16.7
generators	13	8	49	84.6	100.0	98.0	69.2	50.0	34.7
Autogen Benchmark									
assignments	8,308	8	25,357	91.3	100.0	99.9	77.6	50.0	62.6
decorators	748	269	494	77.5	82.9	75.7	36.1	25.7	6.1
generators	49	24	186	89.8	100.0	91.9	79.6	79.2	34.4

Table 15 Challenging patterns that trigger HeaderGen failures

Sub-category	Concise example
	Assignments
augmented	`a += 3`
starred	`a, *b, c = f1, f2, f3`
nested_unpack	`(x, (y, z)) = (1, (2, 3))`
recursive_tuple	`t1 = (1, 2); (a, b) = t1`
	Decorators
classes	`@decorator` `class C: ...`

nested_decorators	`@d1`
	`@d2`
	`def g(): ...`

	Generators

yield_function	`def f(): return`
	
	`def g(n):`
	`  for _ in range(n):`
	`  yield 5`
	
	`for fn in g(10):`
	`  print(fn())`

yield_next	`def squares():`
	`  n = 1`
	`  while True:`
	`  yield n * n`
	`  n += 1`
	
	`gen = squares()`
	`for _ in range(5):`
	`  a = next(gen)`

around determinism, explainability, and security (particularly with closed-source models) must also be considered when using LLMs in production tooling.

> LLMs show significant performance improvements in type inference for dynamic languages like Python, indicating a potential toward data-driven analysis approaches in development workflows, although practical deployment faces challenges related to resource demands, determinism, and security.

8.7 Trade-offs Between Model Accuracy and Efficiency

Interestingly, mid-sized models like codellama-it-13b and codestral-v0.1-22b offer a more balanced trade-off, achieving competitive accuracy with lower inference time. These results imply that specialized fine-tuning and architectural choices can lead to performance levels comparable to, or even better than, general-purpose proprietary models. Conversely, the notably poor performance of lightweight models like tinyllama-1.1b suggests that there is a lower bound on model complexity necessary for robust type inference. Results indicate that these smaller models lack the representational capacity needed to capture the complex code patterns that type inference demands, particularly in dynamically typed languages where explicit type hints are sparse. This observation suggests that while lightweight models may be attractive for extremely resource-constrained settings, they may not yet be viable replacements when inference precision is critical. In real-world applications, smaller models with moderate accuracy and faster inference times may be more appropriate in iterative development environments.

> Mid-sized models such as `codellama-it-13b` and `codestral-v0.1-22b` achieve a balance between accuracy and efficiency, whereas smaller models like `tinyllama-1.1b` lack the capacity for reliable type inference, indicating that a minimum model complexity is necessary for precision in dynamic languages.

8.8 Scalability and Deployment Considerations

Most LLMs evaluated in this study have over seven billion parameters, which typically require multi-GPU setups or specialized hardware (e.g., high-memory A100 or H100 nodes) to perform inference at acceptable speeds. This makes them impractical for deployment on standard single-GPU machines commonly used by individual developers. In contrast, traditional tools such as PyCG and HeaderGen can be executed efficiently in such environments, making them more viable for integration into developer workflows where hardware resources are limited. This gap points to the need for either lighter, more optimized LLM variants specifically designed for developer tooling or hybrid approaches that combine traditional static analysis with targeted LLM augmentation only when necessary.

Due to their high hardware demands, current LLMs are impractical for standard single-GPU deployments, highlighting the need for lighter LLM variants or hybrid approaches that combine static analysis with selective LLM augmentation to support resource-constrained developer environments.

8.9 Towards Hybrid Analysis: LLMs as Enhancers of Static Tools

Given their success in type inference, LLMs could serve as auxiliary tools to enrich traditional static analysis pipelines rather than as replacements. Accurately inferred types could enhance call-graph construction, especially in cases involving dynamic dispatch or polymorphism. By integrating inferred type annotations into SA pipelines, one could improve the precision and recall of downstream analyses. This hybrid approach–combining LLMs' contextual understanding with the rigor of SA tools–presents a promising direction for future work. However, realizing this vision will require careful system design. Issues such as calibration of confidence thresholds for LLM outputs, handling conflicting inferences, and maintaining transparency and auditability within SA workflows must be addressed.

LLMs could supplement rather than replace static analysis tools by providing inferred types to improve downstream analyses like call-graph construction, though achieving effective hybrid integration requires addressing challenges related to confidence calibration and transparency.

9 Threats to Validity

We acknowledge the following limitations and threats to the validity of our study:

- We applied the same prompt to all models, which may not have optimized performance for each individual model. Tailored prompts could potentially extract better results from specific models.
- Open-source models frequently deviated from the expected output formats provided in the prompt. To mitigate this, we manually identified response patterns and added a preprocessing step to standardize the format. However, this approach may not account for all variations, further underscoring the challenge of consistently generating structured data with LLMs.
- While we tested several prompts iteratively, our approach did not focus exclusively on optimizing prompt engineering. A dedicated experiment to explore different prompting strategies could lead to better results. Our modular framework can serve as a foundation for future research aimed at refining prompts to improve performance.
- We used greedy search for token prediction, always selecting the highest-probability token. Future research could explore higher temperature settings and incorporate a voting mechanism to identify the best output, potentially yielding better results.
- While micro-benchmarks are useful for isolating and evaluating specific aspects of system performance, they may miss the complexity and variability of real-world workloads

and use cases. Therefore, we extended TypeEvalPy with auto-generation capabilities to improve the type diversity of the micro-benchmark. It's hard to ensure that all variations were considered in this study and that the results will generalize, but we made efforts to extend considerably the amount of use cases previously available.
– Extending TypeEvalPy with auto-generation capabilities required significant human effort to create the initial templates. While two authors reviewed these templates, they may still be subject to human error. However, we believe that any minor mistakes in the templates are unlikely to have a significant impact on the overall results of this study.

10 Conclusion

This study provides a comprehensive evaluation of LLMs in static analysis tasks, particularly call-graph construction and type inference, using enhanced micro-benchmarks across Python and JavaScript programs. Our results reaffirm that while, LLMs offer promising capabilities in various software engineering tasks, for Python traditional static analysis methods remain more effective for call-graph construction. Similar to findings in previous studies, LLMs have yet to surpass the efficiency of static tools like PyCG and Jelly for this task.

Interestingly, our analysis also highlights a notable performance difference between LLMs' handling of Python and JavaScript code, with LLMs generally performing better on Python. One possible explanation is that LLMs may inherently handle Python more effectively due to the language's widespread use in LLM training datasets. Yet, further investigation is required to fully understand this performance gap.

In type inference tasks, LLMs demonstrated a clear advantage over traditional tools where models like gpt-4o and mistral-large-it-2407-123b excelled. However, their large computational demands limit their practicality in resource-constrained environments. Notably, smaller specialized models like codestral-v0.1-22b showed competitive performance, highlighting the potential for optimization.

This study demonstrates the potential of LLMs in software engineering tasks, while also emphasizing their limitations and the continued strengths of traditional methods. Future research should explore hybrid approaches that combine the strengths of LLMs and static analysis to further advance the field, for instance, by using type inference capabilities of LLMs with traditional static analysis techniques to improve call-graph construction, especially in handling dynamic dispatch and polymorphism.

Appendix A Prompts

A.1 Type Inference Prompts

```
You will be provided with the following information:
1. Python code. The sample is delimited with triple backticks.
2. Sample JSON containing type inference information for the
    Python code in a specific format.
3. Examples of Python code and their inferred types. The examples
    are delimited with triple backticks. These examples are to be
    used as training data.

Perform the following tasks:
1. Infer the types of various Python elements like function
    parameters, local variables, and function return types
    according to the given JSON format with the highest
    probability.
2. Provide your response in a valid JSON array of objects
    according to the training sample given. Do not provide any
    additional information except the JSON object.
3. {format_instructions}

Example Python Code:
```main.py
{code snippet}
```

Example JSON Response:
```
{example response in JSON format}
```
```

Listing 7: Prompt for type inference with JSON output format

```
## Task Description

**Objective**: Examine and identify the data types of various
    elements such as function parameters, local variables, and
    function return types in the given Python code.

**Instructions**:
1. For each question below, provide a concise, one-word answer
    indicating the data type.
2. For arguments and variables inside a function, list every data
    type they take within the current program context as a comma
    separated list.
3. Do not include additional explanations or commentary in your
    answers.
```

Listing 8: Prompt for type inference in question-answer format (part 1 of 2)

```
**Example Python Code**:
```python
a = 1
b = 1.0
c = "hello"
```

Example Questions:
1. What is the type of the variable 'a' at line 1, column 1? Reply
     in one word.
2. What is the type of the variable 'b' at line 2, column 1? Reply
     in one word.
3. What is the type of the variable 'c' at line 3, column 1? Reply
     in one word.

Example Answers:
1. int
2. float
3. str

Python Code Provided:
{code}

Questions:
{questions}

Format for Answers:
- Provide your answer next to each question number, using only one
     word.
- Example:
    1. int
    2. float
    3. str

Your Answers:
{answers}
```

Listing 9: Prompt for type inference in question-answer format (part 2 of 2)

````
## Task Description

**Objective**: Examine and identify the data types of various
    elements such as function parameters, local variables, and
    function return types in the given Python code.

**Instructions**:
1. For each question below, provide a concise, one-word answer
    indicating the data type.
2. For arguments and variables inside a function, list every data
    type they take within the current program context as a comma
    separated list.
3. Do not include additional explanations or commentary in your
    answers.

**Example Python Code**:
```python
a = 1
b = 1.0
c = "hello"
```

**Example Questions**:
1. What is the type of the variable 'a' at line 1, column 1? Reply
     in one word.
2. What is the type of the variable 'b' at line 2, column 1? Reply
     in one word.
3. What is the type of the variable 'c' at line 3, column 1? Reply
     in one word.

**Example Answers**:
1. int
2. float
3. str

**Python Code Provided**:
```main.py
def param_func():
    return "Hello from param_func"


def func(a):
    return a()


b = param_func
c = func(b)
```

**Questions**:
1. What is the return type of the function 'param_func' at line 1,
     column 5?
2. What is the return type of the function 'func' at line 5,
    column 5?
3. What is the type of the parameter 'a' at line 5, column 10,
    within the function 'func'?
4. What is the type of the variable 'b' at line 9, column 1?
5. What is the type of the variable 'c' at line 10, column 1?

**Format for Answers**:
- Provide your answer next to each question number, using only one
     word.
- Example:
    1. int
    2. float
    3. str

**Your Answers**:
1.
2.
3.
4.
5.
````

Listing 10: Example of an actual full prompt for type inference in question-answer format used in the study

A.2 Call-graph Prompts

```
## Task Description

**Objective**: Examine and identify the function calls in the
    given {language} code and answer the questions.

**Instructions**:
1. For each question below, provide a concise answer indicating
    the function calls.
2. List every function call as a comma separated list.
3. Do not include additional explanations or commentary in your
    answers.
4. Include both explicit and implicit function calls in your
    answers. An implicit function call is a function that is
    called as a result of another operation, such as the __init__
    method being called when an object is created.
5. If a function is called through an alias or a reference,
    identify and list the actual function that is called after
    resolving the alias.
6. If a passed argument is not invoked within the function, do not
     include the function call in the answer.
7. Example of {language} code, questions, and answers are given
    below. This example should be used as training data.
```

Listing 11: Prompt for call-graph task in question-answer format (Part 1 of 2)

```
**Format for Answers**:
- Provide your answer next to each question number, using only one
     word.
- Do not include the questions in your answer.
- Example:
    1. simple.func
    2. simple.examplefunc
    3. func

**Example Questions**:
1. What are the module level function calls in the file "main.py"?
2. What are the function calls inside the "main.return_func"
    function in the file "main.py"?
3. What are the function calls inside the "main.func" function in
    the file "main.py"?

**Example Answers**:
1. main.func, main.return_func
2. main.func
3.

**{language} Code Provided**:
{code}

**Questions**:
{questions}

**Answers**:
{answers}
```

Listing 12: Prompt for call-graph task in question-answer format (Part 2 of 2)

```
## Task Description

**Objective**: Examine and identify the fully qualified names of
    function calls in the given {language} code, including class
    methods with both the class name and the method name.

**Instructions**:
1. For each question below, provide a concise answer indicating
    the fully qualified name of function call in that line,
    including class name for methods.
2. List every function call in that line as a comma separated list
    .
3. Do not include additional explanations or commentary in your
    answers.
4. Example of {language} code, questions, and answers are given
    below. This example should be used as training data.
```

Listing 13: Prompt for flow-sensitive call-graph task in question-answer format (Part 1 of 2)

```
**Format for Answers**:
- Provide your answer next to each question number, using only one
     word.
- Add the line number of where the function is defined in the
    program to the answer separated by ":"
- Count the line number from 1
- Make sure to include the question number in your response
- Do not include the questions in your answer.
- Example:
    1. simple.func:1
    2. simple.examplefunc:2
    3. func:5

**Example {language} Code**:
```main.{language_extension}
{example code snippet}
```

Example Questions:
1. What are the fully qualified function calls at line 7 in file '
    main.py'?
2. What are the fully qualified function calls at line 8 in file '
    main.py'?

Example Answers:
1. main.simple_function:1
2. main.another_function:4

{language} Code Provided:

{code}

Questions:
{questions}

Answers:
{answers}
```

Listing 14: Prompt for flow-sensitive call-graph task in question-answer format (Part 2 of 2)

Appendix B Example Responses

B.1 Type Inference Output of mistral-large-it-2407-123b for test case args/assigned_call

```python
def param_func():
    return "Hello!"


def func(a):
    return a()


b = param_func
c = func(b)
```

Listing 15: Source code

```
1. str
2. str
3. function
4. function
5. str
```

Listing 16: Raw LLM response

```
[{
    "file": "main.py",
    "line_number": 1,
    "col_offset": 5,
    "function": "param_func",
    "type": ["str"]
},
{
    "file": "main.py",
    "line_number": 5,
    "col_offset": 5,
    "function": "func",
    "type": ["str"]
},
{
    "file": "main.py",
    "line_number": 5,
    "col_offset": 10,
    "parameter": "a",
    "function": "func",
    "type": ["callable"]
},
{
    "file": "main.py",
    "line_number": 9,
    "col_offset": 1,
    "variable": "b",
    "type": ["callable"]
},
{
    "file": "main.py",
    "line_number": 10,
    "col_offset": 1,
    "variable": "c",
    "type": ["str"]
}]
```

Listing 17: Ground truth


```
[{
    "file": "main.py",
    "line_number": 1,
    "col_offset": 5,
    "function": "param_func",
    "type": ["str"]
},
{
    "file": "main.py",
    "line_number": 5,
    "col_offset": 5,
    "function": "func",
    "type": ["str"]
},
{
    "file": "main.py",
    "line_number": 5,
    "col_offset": 10,
    "parameter": "a",
    "function": "func",
    "type": ["callable"]
},
{
    "file": "main.py",
    "line_number": 9,
    "col_offset": 1,
    "variable": "b",
    "type": ["callable"]
},
{
    "file": "main.py",
    "line_number": 10,
    "col_offset": 1,
    "variable": "c",
    "type": ["str"]
}]
```


Listing 18: Parsed JSON response

B.2 Callgraph Output of mistral-large-it-2407-123b for test case args/assign_return

```python
def return_func(args1):
    return args1


def func1():
    pass


def func2():
    pass

return_func(func1)
return_func(func2)()
```

Listing 19: Source code

```
1. main.return_func, main.func2
2.
3.
4.
```

Listing 21: Raw LLM response

```
{
  "main": [
    "main.return_func",
    "main.func2"
  ],
  "main.func1": [],
  "main.func2": [],
  "main.return_func": []
}
```

Listing 20: Ground truth

```
{
    "main": [
        "main.return_func",
        "main.func2"
    ],
    "main.func1": [],
    "main.func2": [],
    "main.return_func": []
}
```

Listing 22: Parsed JSON response

B.3 Callgraph Output of phi3-mini-it-3.8b for test case args/assign_return

```python
def return_func(args1):
    return args1


def func1():
    pass


def func2():
    pass

return_func(func1)
return_func(func2)()
```

Listing 23: Source code

```
1. func1, func2
2. func1
3. func2
4. return_func
```

Listing 25: Raw LLM response

```
{
  "main": [
    "main.return_func",
    "main.func2"
  ],
  "main.func1": [],
  "main.func2": [],
  "main.return_func": []
}
```

Listing 24: Ground truth

```
{
    "main": [
        "func1",
        "func2"
    ],
    "main.func1": [
        "func1"
    ],
    "main.func2": [
        "func2"
    ],
    "main.return_func": [
        "return_func"
    ]
}
```

Listing 26: Parsed JSON response

Acknowledgements Funding for this study was provided by the Ministry of Culture and Science of the State of North Rhine-Westphalia under the SAIL project with the grand no NW21-059D

Author Contributions Ashwin Prasad Shivarpatna Venkatesh: First author, ideation, implementation, and execution of the whole idea.
Rose Sunil: Worked on the SWARM-JS part of the paper and its analysis.
Samkutty Sabu: Worked on the TypeEvalPy implementation and analysis of micro-benchmark results.
Amir M. Mir: Machine learning expert and worked on the analysis and gathering insights from observed data.
Sofia Reis: Static analysis expert and worked on the analysis and gathering insights from observed data.
Eric Bodden: Static analysis expert and worked on the analysis and gathering insights from observed data.

Funding Open Access funding enabled and organized by Projekt DEAL. Funding for this study was provided by the Ministry of Culture and Science of the State of North Rhine-Westphalia under the SAIL project with the grant no NW21-059D.

Data Availability Data to reproduce experiments in this study, along with the source code, are published on GitHub at: https://github.com/secure-software-engineering/TypeEvalPy and https://github.com/secure-software-engineering/SWARM-CG.
The raw outputs and analysis data is published on Zenodo at: https://zenodo.org/records/15045642

Declarations

Conflict of Interest The authors declare that they have no conflict of interest.

Clinical Trial Number Clinical trial number: not applicable.

References

Allamanis M, Barr ET, Ducousso S, Gao Z (2020) Typilus: Neural Type Hints (PLDI 2020). Association for Computing Machinery, New York, NY, USA, pp 91–105. https://doi.org/10.1145/3385412.3385997

Antal G, Hegedűs P, Herczeg Z, Lóki G, Ferenc R (2023) Is JavaScript call graph extraction solved yet? A comparative study of static and dynamic tools. IEEE Access 11(2023):25266–25284. https://api.semanticscholar.org/CorpusID:257480090

Berti L, Giorgi F, Kasneci G (2025) Emergent abilities in large language models: a survey. https://doi.org/10.48550/arXiv.2503.05788

Buscemi A (2023). A comparative study of code generation using ChatGPT 3.5 across 10 programming languages. arXiv:2308.04477

Chen B, Zhang Z, Langrené N, Zhu S (2023) Unleashing the Potential of Prompt Engineering in Large Language Models: A Comprehensive Review. arxiv:2310.14735

Chen M, Tworek J, Jun H, Yuan Q, de Oliveira Pinto HP, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A, Puri R, Krueger G, Petrov M, Khlaaf H, Sastry G, Mishkin P, Chan B, Gray S, Ryder N, Pavlov M, Power A, Kaiser L, Bavarian M, Winter C, Tillet P, Such FP, Cummings D, Plappert M, Chantzis F, Barnes E, Herbert-Voss A, Guss WH, Nichol A, Paino A, Tezak N, Tang J, Babuschkin I, Balaji S, Jain S, Saunders W, Hesse C, Carr AN, Leike J, Achiam J, Misra V, Morikawa E, Radford A, Knight M, Brundage M, Murati M, Mayer K, Welinder P, McGrew B, Amodei D, McCandlish S, Sutskever I, Zaremba W (2021) Evaluating Large Language Models Trained on Code. arxiv:2107.03374

Dettmers T, Pagnoni A, Holtzman A, Zettlemoyer L (2023) QLoRA: efficient finetuning of quantized LLMs. https://doi.org/10.48550/arXiv.2305.14314

Devlin J, Chang M-W, Lee K, Toutanova K (2019). BERT: pre-training of deep bidirectional transformers for language understanding. arxiv:1810.04805

Di Grazia L, Pradel M (2022) The evolution of type annotations in python: an empirical study. In Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering (ESEC/FSE 2022). Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3540250.3549114

ECMA (2015) ECMA-262. https://ecma-international.org/publications-and-standards/standards/ecma-262/

Fan A, Gokkaya B, Harman M, Lyubarskiy M, Sengupta S, Yoo S, Zhang JM (2023) Large language models for software engineering: survey and open problems. arxiv:2310.03533v4

Feldthaus A, Schäfer M, Sridharan M, Dolby J, Tip F (2013) Efficient construction of approximate call graphs for JavaScript IDE services. In 2013 35th International conference on software engineering (ICSE). pp 752–761. https://doi.org/10.1109/ICSE.2013.6606621. ISSN: 1558-1225

Hellendoorn VJ, Bird C, Barr ET, Allamanis M (2018) Deep learning type inference. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering (Lake Buena Vista, FL, USA) (ESEC/FSE 2018). Association for Computing Machinery, New York, NY, USA, pp 152–162. https://doi.org/10.1145/3236024.3236051

Hou X, Zhao Y, Liu Y, Yang Z, Wang K, Li L, Luo X, Lo D, Grundy J, Wang H (2023) Large language models for software engineering: a systematic literature review. https://doi.org/10.48550/arXiv.2308.10620

Huang K, Yan Y, Chen B, Tao Z, Peng X (2024b) Scalable and precise application-centered call graph construction for python. https://doi.org/10.48550/arXiv.2305.05949

Huang Y, Chen Y, Chen X, Chen J, Peng R, Tang Z, Huang J, Xu F, Zheng Z (2024a) Generative software engineering. https://doi.org/10.48550/arXiv.2403.02583

Jensen SH, Møller A, Thiemann P (2009) Type analysis for JavaScript. In Proceedings of the 16th international symposium on static analysis (Los Angeles, CA) (SAS '09). Springer-Verlag, Berlin, Heidelberg, pp 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

Jin R, Du J, Huang W, Liu W, Luan J, Wang B, Xiong D (2024) A comprehensive evaluation of quantization strategies for large language models. In: Ku L-W, Martins A, Srikumar V (Eds) Findings of the association for computational linguistics: ACL 2024. Association for Computational Linguistics, Bangkok, Thailand, pp 12186–12215. https://doi.org/10.18653/v1/2024.findings-acl.726

Lang J, Guo Z, Huang S (2024) A comprehensive study on quantization techniques for large language models. In 2024 4th International conference on artificial intelligence, robotics, and communication (ICAIRC). pp 224–231. https://doi.org/10.1109/ICAIRC64177.2024.10899941

Laursen MR, Xu W, Møller A (2024) Reducing static analysis unsoundness with approximate interpretation. Proc ACM Program Lang 8(PLDI):1165–1188. https://doi.org/10.1145/3656424

Li H, Hao Y, Zhai Y, Qian Z (2023a) Assisting static analysis with large language models: a ChatGPT experiment. In Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering (San Francisco, CA, USA) (ESEC/FSE 2023). Association for Computing Machinery, New York, NY, USA, 2107–2111. https://doi.org/10.1145/3611643.3613078

Li H, Hao Y, Zhai Y, Qian Z (2023b) The hitchhiker's guide to program analysis: a journey with large language models. https://doi.org/10.48550/arXiv.2308.00245

Li Z, Dutta S, Naik M (2025) IRIS: LLM-assisted static analysis for detecting security vulnerabilities. arxiv:2405.17238

Lucas W, Nunes R, Bonifácio R, Carvalho F, Lima R, Silva M, Torres A, Accioly P, Monteiro E, Saraiva J (2025) Understanding the adoption of modern Javascript features: an empirical study on open-source systems. Empirical Softw Eng 30(42):1382–3256. https://doi.org/10.1007/s10664-025-10663-9

Mir AM, Latoškinas E, Proksch S, Gousios G (2022) Type4Py: practical deep similarity learning-based type inference for python. In: Proceedings of the 44th international conference on software engineering (ICSE '22). Association for Computing Machinery, New York, NY, USA, 2241–2252. https://doi.org/10.1145/3510003.3510124

Mohajer MM, Aleithan R, Harzevili NS, Wei M, Belle AB, Pham HV, Wang S (2024) Effectiveness of ChatGPT for static analysis: how far are we?. In: Proceedings of the 1st ACM international conference on AI-powered software (AIware 2024). Association for Computing Machinery, New York, NY, USA, pp 151–160. https://doi.org/10.1145/3664646.3664777

Nishiura K, Misawa S, Monden A (2024) Analyzing class usage in javascript programs. In Proceedings of the 2024 the 6th world symposium on software engineering (WSSE) (WSSE '24). Association for Computing Machinery, New York, NY, USA, pp 139–143. https://doi.org/10.1145/3698062.3698081

O'Brien C, Rodriguez-Cardenas D, Velasco A, Palacio DN, Poshyvanyk D (2024) Measuring emergent capabilities of LLMs for software engineering: how far are we? https://doi.org/10.48550/arXiv.2411.17927

Peng Y, Gao C, Li Z, Gao B, Lo D, Zhang Q, Lyu M (2022) Static inference meets deep learning: a hybrid type inference approach for python. In: Proceedings of the 44th international conference on software engineering (ICSE '22). Association for Computing Machinery, New York, NY, USA, pp 2019–2030. https://doi.org/10.1145/3510003.3510038

Pradel M, Gousios G, Liu J, Chandra S (2020) TypeWriter: neural type prediction with search-based validation. In Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering (ESEC/FSE 2020). Association for Computing Machinery, New York, NY, USA, pp 209–220. https://doi.org/10.1145/3368089.3409715 Number of pages: 12 Place: Virtual Event, USA

Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I (2019) Language models are unsupervised multitask learners

Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou Y, Li W, Liu PJ (2023) Exploring the limits of transfer learning with a unified text-to-text transformer. https://doi.org/10.48550/arXiv.1910.10683

Rasnayaka S, Wang G, Shariffdeen R, Iyer GN (2024) An empirical study on usage and perceptions of LLMs in a software engineering project. In Proceedings of the 1st international workshop on large language models for code (LLM4Code '24). Association for Computing Machinery, New York, NY, USA, 111–118. https://doi.org/10.1145/3643795.3648379

Salis V, Sotiropoulos T, Louridas P, Spinellis D, Mitropoulos D (2021) PyCG: practical call graph generation in python. In 2021 IEEE/ACM 43rd international conference on software engineering (ICSE). pp 1646–1657. https://doi.org/10.1109/ICSE43902.2021.00146

Schulhoff S, Ilie M, Balepur N, Kahadze K, Liu A, Si C, Li Y, Gupta A, Han H, Schulhoff S, Dulepet PS, Vidyadhara S, Ki D, Agrawal S, Pham C, Kroiz G, Li F, Tao H, Srivastava A, Da Costa H, Gupta S, Rogers ML, Goncearenco I, Sarli G, Galynker I, Peskoff D, Carpuat M, White J, Anadkat S, Hoyle A, Resnik P (2024) The prompt report: a systematic survey of prompting techniques. arXiv:2406.06608

Seidel L, Effendi SDB, Pinho X, Rieck K, van der Merwe B, Yamaguchi F (2023) Learning type inference for enhanced dataflow analysis. arxiv:2310.00673

Steenhoek B, Rahman MdM, Roy MK, Alam MS, Tong H, Das S, Barr ET, Le W (2025). To Err is machine: vulnerability detection challenges LLM reasoning. arxiv:2403.17218

Sun W, Fang C, You Y, Miao Y, Liu Y, Li Y, Deng G, Huang S, Chen Y, Zhang Q, Qian H, Liu Y, Chen Z (2023) Automatic code summarization via ChatGPT: how far are we? arxiv:2305.12865

Venkatesh APS, Sabu S, Chekkapalli M, Wang J, Li L, Bodden E (2024) Static analysis driven enhancements for comprehension in machine learning notebooks. Empir Softw Eng 29(5):1573–7616. https://doi.org/10.1007/s10664-024-10525-w

Venkatesh APS, Sabu S, Mir AM, Reis S, Bodden E (2024b) The emergence of large language models in static analysis: a first look through micro-benchmarks. arXiv:2402.17679

Venkatesh APS, Sabu S, Wang J, Mir AM, Li L, Bodden E (2023a) TypeEvalPy: a micro-benchmarking framework for python type inference tools. https://doi.org/10.48550/arXiv.2312.16882

Venkatesh APS, Wang J, Li L, Bodden E (2023b) Enhancing comprehension and navigation in jupyter notebooks with static analysis. In 2023 IEEE international conference on software analysis, evolution and reengineering (SANER). IEEE Computer Society, pp 391–401. https://doi.org/10.1109/SANER56733.2023.00044

WebKit (2010) WebKit/PerformanceTests/SunSpider/tests/sunspider-1.0.2 at main · WebKit/WebKit. https://github.com/WebKit/WebKit/tree/main/PerformanceTests/SunSpider/tests/sunspider-1.0.2

Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, Cistac P, Ma C, Jernite Y, Plu J, Xu C, Le Scao T, Gugger S, Drame M, Lhoest Q, Rush AM (2020) Transformers: state-of-the-art natural language processing. https://www.aclweb.org/anthology/2020.emnlp-demos.6 Pages: 38–45 original-date: 2018-10-29T13:56:00Z

Yang Y, Milanova A, Hirzel M (2022) Complex Python features in the wild. In Proceedings of the 19th international conference on mining software repositories (Pittsburgh, Pennsylvania) (MSR '22). Association for Computing Machinery, New York, NY, USA, pp 282–293. https://doi.org/10.1145/3524842.3528467

Zhang Q, Fang C, Xie Y, Zhang Y, Yang Y, Sun W, Yu S, Chen Z (2023) A survey on large language models for software engineering. https://doi.org/10.48550/arXiv.2312.15223

Zheng Z, Ning K, Chen J, Wang Y, Chen W, Guo L, Wang W (2023) Towards an understanding of large language models in software engineering tasks. https://doi.org/10.48550/arXiv.2308.11396

Authors and Affiliations

Ashwin Prasad Shivarpatna Venkatesh[4] · Rose Sunil[5] · Samkutty Sabu[5] · Amir M. Mir[2] · Sofia Reis[3] · Eric Bodden[1]

✉ Ashwin Prasad Shivarpatna Venkatesh
ashwin.prasad@upb.de

Rose Sunil
rose10@mail.uni-paderborn.de

Samkutty Sabu
samkutty@mail.uni-paderborn.de

Amir M. Mir
s.a.m.mir@tudelft.nl

Sofia Reis
sofia.o.reis@tecnico.ulisboa.pt

Eric Bodden
eric.bodden@upb.de

[1] Heinz Nixdorf Institut & Fraunhofer IEM, Paderborn University, Paderborn, Germany

[2] Delft University of Technology, Delft, Netherlands

[3] IST, University of Lisbon & INESC-ID, Lisbon, Portugal

[4] Heinz Nixdorf Institut, Paderborn University, Paderborn, Germany

[5] Paderborn University, Paderborn, Germany