UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Secure Software Engineering

Doctoral Thesis

Submitted to the Secure Software Engineering Research Group
in Partial Fulfillment of the Requirements for the Degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

Advances in Python Call-Graph Construction and Type Inference: A Benchmark-Driven Approach

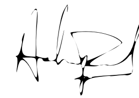
by
Ashwin Prasad Shivarpatna Venkatesh

Thesis Supervisor:
Prof. Dr. Eric Bodden

Paderborn, February 9, 2026

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig, ohne fremde Hilfe und ohne Verwendung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser nicht als Teil einer Prüfungsleistung angenommen worden ist. Alle Stellen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Generative KI-Softwaretools wurden ausschließlich zur sprachlichen Überarbeitung sowie zur Verbesserung von Klarheit, Textfluss und Lesbarkeit eingesetzt; Inhalt und wissenschaftliche Schlussfolgerungen stammen vollständig von mir selbst.



Paderborn, 09.02.2026	
Ort, Datum	Unterschrift

Abstract. The rapid growth of data science and machine learning has made Python a central language in modern research, particularly in computational environments such as Jupyter Notebooks. While these notebooks facilitate exploration, their interactive nature often leads to unstructured and undocumented code that hinders reproducibility and comprehension. Compounding this issue, existing tools for code navigation and refactoring primarily rely on static analysis. However, these methods struggle with Python's dynamic nature, often yielding precision and recall rates that are too low for practical application. This dissertation addresses these challenges by introducing novel static analysis approaches and systematic benchmarking frameworks for Python, ultimately facilitating the semantic structuring of notebooks, reliable evaluation of type inference systems, and a critical analysis of Large Language Models (LLMs) for static analysis tasks.

To address the lack of structure in notebooks, we develop a heuristics-based call-graph analysis for Python that models the unique behavior of machine-learning libraries and leverages type hints from external dependencies. Using this enhanced code understanding, we introduce HeaderGen, a tool that automatically segments undocumented Jupyter Notebooks into navigable, semantically labeled narratives. Our evaluation on real-world notebooks demonstrates that the underlying analysis achieves high precision and recall for both call-graph construction and the classification of notebook cells according to their semantic role in the machine-learning workflow. Furthermore, a user study confirms that the resulting structure significantly accelerates code comprehension and navigation for data-science practitioners.

Beyond the development of individual tools, advancing the field requires reliable methods to measure the capabilities of underlying analysis techniques, yet the current evaluation landscape remains fragmented and reliant on unverified datasets. To address this gap, we introduce TypeEvalPy, a comprehensive benchmarking framework for Python type inference with 154 test cases that provides a controlled environment with manually curated code snippets and verified ground truth. Our empirical study of six representative tools reveals significant performance variations. While HeaderGen exhibits the most balanced results, open-source tools such as *Pyright* excel in ecosystem integration but lack consistency across scenarios. Conversely, learning-based and hybrid systems demonstrate that probabilistic predictions can augment static analysis, however, they still remain constrained by training distributions.

Finally, recognizing the paradigm shift towards generative AI, we extend our rigorous evaluation methodology to assess the capabilities of LLMs. To stress-test generalizability, we augment TypeEvalPy with an auto-generation engine that scales the benchmark to 7,121 test cases and introduce SWARM_{CG}, a multi-language suite for call-graph analysis. Our evaluation of 24 LLMs reveals a performance divergence: LLMs excel at type inference but struggle with call-graph construction, an area where traditional analyzers remain superior. These results suggest that the future of Python tooling lies not in replacement, but in a hybrid integration where static analysis provides the structural foundation that learning-based methods typically lack.

Acknowledgment

The completion of this thesis would not have been possible without the support, guidance, and encouragement of many individuals who accompanied me throughout this journey.

First and foremost, I would like to thank my wife, Bhargavi, whose love and support have guided me throughout this journey. Her discipline, perseverance, and consistency had a strong and lasting influence on me, extending beyond her own life and shaping my approach to mine. I am deeply grateful that I met her at the right time. I would also like to express my sincere gratitude to my parents and my parents-in-law for their constant encouragement, trust, and belief in me.

I gratefully acknowledge the guidance I received during the final phase of my Master's studies from Hadi, Sevil, and Professor Holger Karl. Their mentorship ignited my interest in research, strengthened my confidence in my ability to conduct meaningful scientific work, and played a key role in paving the way toward my PhD.

My deepest thanks go to my PhD supervisor, Eric. Over the past five years, his trust, guidance, and support were central to my development as a researcher. I truly enjoyed working with him and greatly value his supervisory style, which emphasizes intellectual independence and constructive criticism towards success.

I would like to thank my colleagues for making the entire doctoral journey both productive and enjoyable. Jonas deserves special mention for acting as a liaison within the group and for helping to bind us together as a cohesive team. I would also like to thank my first office roommates, Mugdha and Kadiray. The early stages required all of us to persevere in identifying our research directions, and it has been rewarding to see each of us succeed. I thank Philip, Michael, and Stefan Schott for maintaining a competitive and motivating research environment. I am grateful to Marcus Hüwe for being a thoughtful listener, for his analytical sharpness, and for his distinctive sense of humor. I also thank Martin Mory for his open-mindedness, cultural inclusivity, and for helping me feel welcomed and integrated into the society here. I thank Ingo for being a constant comrade and for always playing along. I am grateful to Markus Schmidt, Oshando, and Ranjith for bringing humor into everyday work. I thank Sriteja and Fabian for helping bridge collaboration between HNI and Fraunhofer. I also thank Linghui and Goran for their support and guidance in career-related decisions.

I would like to acknowledge my collaborators Jiawei and Professor Li for the focused and high-impact discussions we shared, which significantly shaped the direction and outcomes of this thesis. The trip to Macau for SANER had a strong academic and professional impact. During this time, I met Amir from TU Delft and Sergey from JetBrains. I greatly enjoyed collaborating with Amir both professionally and personally, and our collaboration resulted in several joint publications. My introduction to Sergey later led to a fruitful internship at JetBrains, which I thoroughly enjoyed.

Finally, I would like to thank everyone involved in my internship at AWS in New York City,

which had a substantial impact on my career. I greatly enjoyed the high-intensity research environment and the work carried out there. I sincerely thank Pranav for providing me with this opportunity. I also thank Yaro for the enjoyable and engaging collaboration on projects, we had too much fun with agents, GPUs, and billions of tokens!

Publications

This dissertation presents original research contributions that have previously been published in peer-reviewed software engineering journals, conferences, and workshops. Specifically, this dissertation includes material from the following publications, for which the author of this dissertation is the lead author.

Conference and Workshop Papers

[1] **Ashwin Prasad Shivarpatna Venkatesh**, Jiawei Wang, Li Li, and Eric Bodden. *Enhancing Comprehension and Navigation in Jupyter Notebooks with Static Analysis.* SANER 2023. [SVWLB23a] 🏆 ***Distinguished Paper Award***

Discussed in Chapter 3.

Artifact published at `https://github.com/secure-software-engineering/HeaderGen`

[2] **Ashwin Prasad Shivarpatna Venkatesh**, Samkutty Sabu, Jiawei Wang, Amir M. Mir, Li Li, and Eric Bodden. *TypeEvalPy: A Micro-Benchmarking Framework for Python Type Inference Tools.* ICSE Tool Demo 2024. [SVSW+24]

Discussed in Chapter 4.

Artifact published at `https://github.com/secure-software-engineering/TypeEvalPy`

[3] **Ashwin Prasad Shivarpatna Venkatesh**, Samkutty Sabu, Amir M. Mir, Sofia Reis, and Eric Bodden. *The Emergence of Large Language Models in Static Analysis: A First Look through Micro-Benchmarks.* Short Paper at FORGE 2024. [SVSM+24]

Discussed in Chapter 5.

Journal Articles

[1] **Ashwin Prasad Shivarpatna Venkatesh**, Samkutty Sabu, Mouli Chekkapalli, Jiawei Wang, Li Li, and Eric Bodden. *Static Analysis Driven Enhancements for Comprehension in Machine Learning Notebooks.* Empirical Software Engineering, 2024. [SVSC+24]

Discussed in Chapter 3 and Chapter 4.

[2] **Ashwin Prasad Shivarpatna Venkatesh**, Rose Sunil, Samkutty Sabu, Amir M. Mir, Sofia Reis, and Eric Bodden. *An Empirical Study of Large Language Models for Type and Call Graph Analysis in Python and JavaScript.* Empirical Software Engineering, 2025. [SVSS+25]

Discussed in Chapter 5.

Artifact published at `https://github.com/secure-software-engineering/SWARM-CG`

Contents

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Contributions . . . 2
1.3 Structure . . . 3

2 Background **5**
2.1 Static Program Analysis Techniques . . . 5
2.2 Dynamic Nature of Python and Its Analysis Challenges . . . 7
2.2.1 Dynamic Typing . . . 7
2.2.2 Dynamic Data Structures . . . 8
2.2.3 Dynamic Evaluation . . . 8
2.2.4 Complex Builtin Library . . . 8
2.2.5 Import Aliasing . . . 9
2.2.6 Chained Function Calls . . . 10
2.2.7 Variable Reuse . . . 10
2.2.8 Scarcity of Type Annotations . . . 10
2.2.9 CPython as the De Facto Reference . . . 11
2.3 Unstructured and Undocumented Jupyter Notebooks . . . 11
2.3.1 Lack of Structure . . . 11
2.3.2 Insufficient Documentation . . . 12
2.3.3 Limited Support for Structuring and Documentation . . . 12

3 HeaderGen: Call-Graph Based Structuring of Python Notebooks **15**
3.1 Motivating Example . . . 18
3.2 HeaderGen's Static Analyzer . . . 20
3.2.1 Extended Assignment Graph . . . 21
3.2.2 Flow-sensitive Callsite Extraction . . . 26
3.2.3 Jupyter Notebook Annotation . . . 26
3.3 Evaluation . . . 31
3.4 Benchmarks . . . 32
3.4.1 Jupyter Notebook Micro-benchmark . . . 32
3.4.2 Real-world Benchmark . . . 33
3.4.3 Extended Real-world Benchmark . . . 33
3.5 RQ1: Comprehension and Navigation Study . . . 34
3.5.1 Study Design . . . 34
3.5.2 Participants . . . 35
3.5.3 Metrics . . . 36

3.5.4 Results 36
3.6 RQ2: Accuracy of Callsite Recognition 37
3.7 RQ3: Accuracy of Generated Headers 38
3.8 RQ4: Comparison with Existing Tools 40
3.9 Related Work 41
3.10 Limitations & Future work 43
3.11 Conclusion 43

4 Type Inference Benchmarking with TypeEvalPy 45
4.1 TypeEvalPy Framework 46
4.1.1 Micro-benchmark 46
4.1.2 Runner 49
4.1.3 Translator 50
4.1.4 Result Analyzer 50
4.2 Research Question 51
4.3 Tool Selection 51
4.4 Results 52
4.5 Discussion 55
4.5.1 HeaderGen 56
4.5.2 Jedi 56
4.5.3 Pyright 56
4.5.4 Scalpel 56
4.5.5 Type4Py 57
4.5.6 HiTyper and HiTyper-DL 57
4.5.7 Outlook 57
4.6 Limitations 57
4.7 Conclusion 58

5 Large Language Models for Static Analysis 61
5.1 Related Work 62
5.1.1 Traditional Static Analysis for Python and JavaScript 62
5.1.2 LLMs Enter Static Analysis: Early Experiments 63
5.1.3 Micro-Benchmark Suites for Python and JavaScript 64
5.2 Research Questions 65
5.3 Call-Graph Micro-Benchmarks 65
5.3.1 PyCG: Call-graph Micro-Benchmark 66
5.3.2 HeaderGen: Call-sites Micro-Benchmark 66
5.3.3 SWARM-JS: JavaScript Micro-benchmark 66
5.4 Type Inference Micro-benchmarks 67
5.4.1 Manual Baseline (TypeEvalPy) 68
5.4.2 TypeEvalPy$_{\text{AutoGen}}$ Extension 68
5.5 Methodology 69
5.5.1 Type Inference Tools 70
5.5.2 Call-graph Construction Tools 70
5.5.3 Large Language Model Selection 72
5.5.4 Prompt Design 72
5.5.5 Evaluation Metrics 75
5.5.6 Implementation Details 76
5.6 Results 76
5.6.1 RQ1: Accuracy of Call-graph Analysis 76

5.6.2 RQ2: Accuracy of Type Inference . . . 82
5.7 Discussion . . . 84
5.7.1 Call Graph Construction in Python: LLMs vs Static Analysis Tools . . . 84
5.7.2 Call Graph Construction in JavaScript: LLMs vs Static Analysis Tools . . 88
5.7.3 Type Inference in Python: LLMs vs Static Analysis Tools . . . 89
5.7.4 LLM Performance Differences: Type Inference vs. Call Graph Analysis . 90
5.7.5 Cross-language Performance Disparities . . . 91
5.7.6 Implications for Type Inference in Dynamic Languages . . . 91
5.7.7 Trade-offs Between Model Accuracy and Efficiency . . . 92
5.7.8 Scalability and Deployment Considerations . . . 92
5.7.9 Towards Hybrid Analysis: LLMs as Enhancers of Static Tools . . . 93
5.8 Threats to Validity . . . 93
5.9 Conclusion . . . 94

6 Conclusion and Future Work **97**

A PyCG Intermediate Representation **99**
A.1 Running Example Source Code . . . 99
A.2 Analysis State Domains . . . 99
A.2.1 Namespace Domain . . . 100
A.2.2 Scope Domain . . . 100
A.2.3 Class Hierarchy Domain . . . 100
A.2.4 Assignment Graph Domain . . . 101
A.3 Definition–Use Chains and Flow Sensitivity . . . 101
A.3.1 DUC Analysis Output . . . 101
A.3.2 Flow-Sensitive Node Construction . . . 102

B Prompts **103**
B.1 Type Inference Prompts . . . 103
B.1.1 Call-graph Prompts . . . 106
B.2 Example Responses . . . 107
B.2.1 Type inference Output of mistral-large for test case args/assigned_call . . 107
B.2.2 Callgraph Output of mistral-large-it-2407-123b for test case args/assign_return109
B.2.3 Callgraph Output of phi3-mini-it-3.8b for test case args/assign_return . . 110

Bibliography **110**

1 Introduction

Software engineering as a discipline is fundamentally concerned with building reliable, maintainable, and comprehensible software systems. To support these goals, a wide range of methods has been developed for reasoning about program behavior, improving software quality, and enabling effective collaboration. Among these methods, static analysis has become an important foundation. By examining programs without executing them, static analysis can detect potential errors [KNR+17], identify security vulnerabilities [DDD+25], and support the construction of advanced development tools [NSB22, Jet26]. In statically typed languages such as Java, type information and well-defined semantics provide a strong foundation for precise analysis. Static analysis techniques have significantly improved software quality and remain a cornerstone of modern development practices.

The increasing adoption of dynamically typed languages has introduced new challenges for this established body of work. Among these languages, Python has become especially influential. Its simple syntax, rich ecosystem of libraries, and community support have made it the dominant language in data science and machine learning domains [RPN20]. Python is widely used for rapid prototyping, exploratory analysis, and experimental research. These are contexts in which flexibility and expressiveness are often prioritized over rigid structure and performance. However, these characteristics complicate static analysis. Features such as dynamic type reassignment, dynamic evaluation, flexible data structures, etc., limit the applicability of traditional methods that rely on static type information [NG25, CCY+24, OGRG25, LKF20]. Consequently, existing analysis tools cannot easily achieve the same precision for Python that they provide for statically typed languages.

In parallel, the rise of Jupyter Notebooks has reshaped how Python code is written, shared, and consumed by data science practitioners [WLZ20]. By combining source code, natural language text, and visualizations in a single environment, notebooks embody the idea of literate programming [Knu84], where a program is written as an explanation of its logic in natural language intermixed with snippets of source code. They have become the de-facto medium for education, data science, and computational research, reducing entry barriers and promoting the exchange of knowledge. Yet empirical studies reveal that many notebooks fail to realize their potential as explainable, reproducible, and maintainable research artifacts [PMBF19, RTH18, WLZ20, WKLZ20b]. Common issues include insufficient documentation, ad-hoc organization, and dependence on non-linear execution [WLZ20]. These practices hinder comprehension, complicate collaboration, and reduce the long-term value of notebooks as scientific or educational resources [PMBF19, WKLZ20a].

This dissertation addresses these dual challenges: the difficulty of performing precise static

analysis on Python programs and the shortcomings of Jupyter Notebooks as reliable, structured artifacts.

1.1 Problem Statement

The growing adoption of Python and Jupyter Notebooks within data science and machine learning has introduced both significant advantages and persistent challenges. Python offers a highly adaptable programming environment supported by an extensive ecosystem of libraries, and Jupyter Notebooks have become the primary medium for combining code, explanatory text, and visualization. Together, they promote rapid experimentation and foster collaboration across academic and industrial settings.

At the same time, two central problems emerge that limit the reliability and long-term usefulness of Python-based workflows:

1. **Analytical complexity of Python.** In contrast to statically typed languages such as Java, Python lacks explicit type information at compile time and exhibits a high degree of runtime flexibility. Variables may change type across execution contexts, data structures often contain heterogeneous elements, and constructs such as `eval()` allow dynamic evaluation that obscures control flow. In addition, Python's expressive language design introduces further modeling difficulties: complex built-in libraries employ iterators, generators, and asynchronous functions; import aliasing conceals symbol origins; and chained method calls, or variable reuse, complicate dependency tracking. These characteristics limit the application of established static analysis techniques, leaving a gap in the availability of precise and scalable program analysis techniques for Python.

2. **Lack of structure and documentation in Jupyter Notebooks.** While Jupyter Notebooks were intended to promote literate programming, they are frequently employed as ad hoc exploratory scratchpads. Many lack explanatory text or coherent organization, and their flexible execution model allows variable redefinitions and out-of-order cell execution. Such practices reduce comprehension and diminish the potential of notebooks as long-term scientific and educational artifacts.

A detailed discussion of these challenges is deferred to Chapter 2, where the complexity of Python and the structural limitations of Jupyter Notebooks are examined in depth.

1.2 Contributions

This dissertation addresses the two challenges outlined in the previous section by developing an improved analysis infrastructure and applying it to enhance the comprehensibility of Jupyter Notebooks. To achieve this, the work makes three primary contributions.

Contribution 1: Enhancing Notebook Structure via Call-Graph Analysis [SVWLB23a]
To address the prevalence of undocumented and unstructured Jupyter Notebooks, we introduce HeaderGen, a tool that automatically imposes a narrative structure by annotating code cells based on a taxonomy of machine-learning operations. To realize this, we enhanced the underlying static analysis infrastructure by introducing flow-sensitivity and a novel mechanism for resolving external library return types, a capability where existing tools often fall short. Evaluation on real-world notebooks demonstrates that this enhanced analysis yields high precision, while a user study confirms that the resulting annotations significantly accelerate developer comprehension and navigation.

Contribution 2: Benchmarking Type Inference Capabilities [SVSC+24, SVSW+24] Recognizing that accurate static analysis for Python relies fundamentally on type inference, we address the community's lack of a standardized evaluation process. We introduce TypeEvalPy, a comprehensive micro-benchmarking framework containing a diverse set of code snippets targeting specific Python features. By automating the execution of containerized tools and normalizing their outputs, the framework produces comparable metrics. We utilized this infrastructure to systematically assess state-of-the-art tools, revealing critical performance gaps and establishing a rigorous baseline for future optimizations in Python type inference.

Contribution 3: Evaluating Large Language Models for Program Analysis [SVSS+25] Finally, we investigate the potential of Large Language Models (LLMs) to overcome the limitations of traditional analysis. To enable a large-scale empirical study, we enhanced TypeEvalPy with auto-generation capabilities, significantly expanding the ground truth, and introduced SWARM_{CG}, a new benchmark for call-graph construction. Our evaluation of a wide range of LLMs reveals a distinct contrast in performance: while LLMs often surpass traditional tools in type inference accuracy, they struggle with the global reasoning required for call graph completeness and soundness. These findings suggest that the future of Python tooling lies not in replacing static analysis, but in a hybrid integration.

1.3 Structure

The remainder of this dissertation is organized as follows. Chapter 2 establishes the theoretical foundation for the work by introducing key static analysis techniques and elaborating on the specific complexities posed by Python's dynamic behavior and the limitations of Jupyter Notebooks. Chapter 3 presents our work on HeaderGen, detailing the enhancements made to flow-sensitive analysis and the development of heuristics for resolving external library calls. Chapter 4 introduces the TypeEvalPy framework, describing the design of the micro-benchmarks and discussing the comparative performance of existing static analysis tools. Chapter 5 extends this investigation to Large Language Models, presenting the $\text{TypeEvalPy}_{\text{AutoGen}}$ and SWARM_{CG} benchmarks alongside an empirical study contrasting LLM capabilities with traditional analysis. Finally, Chapter 6 concludes the dissertation and outlines directions for future research.

2 Background

This chapter establishes the theoretical and technical foundations necessary for understanding the contributions of this dissertation. We begin by introducing fundamental concepts in static program analysis, including type inference, call-graph construction, and call-site analysis. Building on these definitions, we then examine the specific challenges posed by Python's dynamic nature and how they complicate the application of traditional analysis techniques. Finally, we discuss the context of Jupyter Notebooks, highlighting the deficits that necessitate the development of specialized analysis tools like HeaderGen.

2.1 Static Program Analysis Techniques

Program analysis techniques are essential for static analysis tools, enabling them to reason about program correctness and structure before execution. To achieve this, especially in the context of dynamically typed languages, tools rely on several complementary techniques. *Type inference* is used to determine variable types without execution, *call-graph analysis* maps relationships between functions, and *call-site analysis* resolves the specific targets of individual invocation points. Together, these techniques form the backbone of the analysis infrastructure developed in this work.

To illustrate these analysis concepts in practice, consider the code snippet in Listing 2.1. 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.

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 tracks the data flow to resolve variable types by examining assignments, function calls, and operations at different points in the program. In the code snippet in Listing 2.1, such an analyzer determines that the variable `result` at line 5 is a string (return value of `create_str`), while the variable `result` at line 9 is an integer (return value of `len`). By performing this analysis, type inference can detect errors such as type mismatches before execution and correctly raise a type error at line 6, preventing runtime failures.

```
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!
```

Listing 2.1: Python code snippet illustrating dynamic call sites, type inference challenges across variable reassignments, and the impact of dynamic evaluation via eval

However, static analyzers struggle with dynamic evaluation features like `eval` (line 12), which obscure the code structure. Since `eval` executes code that is only constructed at runtime, static analyzers cannot reliably determine the control flow or variable effects without executing the program. Therefore, dynamic constructs like `eval` are typically not parsed by static analyzers during code analysis; the type of `x` remains unknown, and a subsequent error at line 13 may go undetected. This further highlights the challenges of type inference in dynamically typed languages, where runtime behavior can introduce unexpected type inconsistencies.

Call-graph Analysis. A call graph is a representation of function-call relationships in a program that captures the structural relationships among functions, specifically mapping callers to their potential callees. It provides a global view of the system by determining which functions can invoke others, thereby helping understand control flow, track function dependencies, and identify unreachable or unused code.

The complete call graph for the snippet in Listing 2.1 is shown in Figure 2.1, revealing high-level function relationships (e.g., "function A calls function B") to enable optimizations, refactoring, and bug detection. Here, the main function acts as the entry point, invoking `main.create_str`, `builtins.eval`, and `builtins.len`. While the solid line demonstrates a clear static relationship where `main.create_str` calls `builtins.str.upper`, the dashed line labeled dynamic from `builtins.eval` to `main.create_str` highlights a critical challenge. Because dynamic constructs like `eval` are typically not parsed by static analyzers, they obscure the code structure and make it difficult to resolve potential callees, ultimately showing how runtime behavior can introduce unexpected dependencies that a standard analysis might miss.

Call-site Analysis. While a call graph provides a global summary of relationships, call-site analysis focuses on the specific locations in the source code where functions are invoked. In dynamic languages, a single call site (e.g., `obj.method()`) may invoke different functions at different times depending on the runtime type of `obj`; therefore, precise call-site analysis aims to determine the set of potential target functions for every individual invocation point. To analyze this code precisely, the tool must differentiate multiple assignments to the variables `x` and `result`, representing potential targets for each invocation point based on the current program state rather than just a global summary.

The list of call sites for the snippet in Listing 2.1 is as follows:

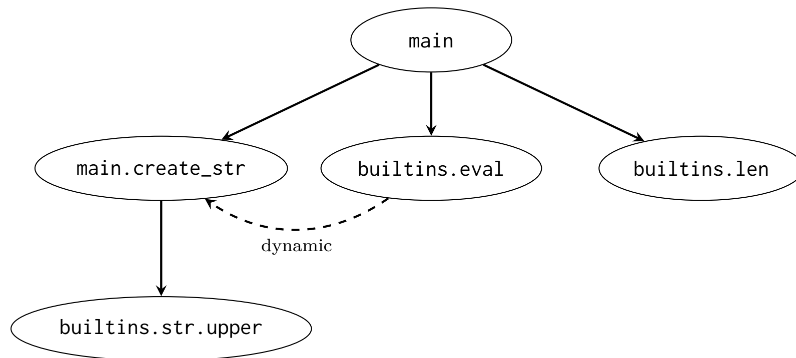


Figure 2.1: Call Graph for code snippet in Listing 2.1

- **Call-Site 1 (Line 2):** `builtins.str.upper`.
- **Call-Site 2 (Line 5):** `create_str`.
- **Call-Site 3 (Line 9):** `builtins.len`.
- **Call-Site 4 (Line 12):** `builtins.eval`.

2.2 Dynamic Nature of Python and Its Analysis Challenges

Programming languages serve as the fundamental bridge between human logic and machine execution, typically categorized by their execution models and type systems. At one end of the spectrum lie low-level, statically typed languages like C or Java, which enforce rigid structures to ensure performance and predictability. At the other end are high-level, interpreted languages designed to prioritize developer productivity and flexibility. Python has emerged as the leading language in this category.

To understand why traditional static analysis often fails or loses precision when applied to Python, one must examine the specific linguistic features that complicate the extraction of program semantics. In the following subsections, we detail the primary hurdles encountered during analysis.

2.2.1 Dynamic Typing

Dynamic typing constitutes a major challenge in Python analysis. Variables are not declared with fixed types and may assume different types during execution, a feature often referred to as "duck typing." As a result, static analysis must depend on inference rather than explicit type information, which limits accuracy in detecting inconsistencies and reasoning about program behavior.

In Listing 2.2, the same variable `x` is used with values of different types, each invoking different built-in operations. Because the type of `x` changes during execution, a static analyzer must infer all possible types that `x` may assume and determine the corresponding set of valid operations. Accurately performing such inference requires reasoning about control flow, variable reassignments, and runtime behavior.

```python
x = 10
print(abs(x)) # Works for integers -> 10

x = 'Hello'  # Type changes from int to string
print(x.upper()) # Works for strings -> 'HELLO'
```

Listing 2.2: Dynamic typing in Python

2.2.2 Dynamic Data Structures

Python features dynamic built-in data structures, such as lists and dictionaries, that can hold heterogeneous types. These high-level built-in data structures pose a challenge for static analysis because their contents are often not explicitly defined. Python's dynamic typing means that variable types are not fixed at compile time and can change during execution, making type inference difficult.

In Listing 2.3, the list `my_list` contains an integer, a string, a floating-point number, and a boolean value. To analyze this structure statically, the analyzer must identify all possible element types, represent the list as a union of these types, and ensure that subsequent operations on the list are compatible with each element type. Performing this reasoning requires propagating type information across heterogeneous collections, a process that adds considerable complexity to static analysis in Python.

```python
my_list = [1, 'apple', 3.14, True]  # List with heterogeneous types
```

Listing 2.3: Dynamic data structures in Python

2.2.3 Dynamic Evaluation

The use of `eval()` or `exec()` to execute arbitrary strings as code obfuscates control flow and makes static analysis challenging. Such dynamic evaluation can lead to analyses that are incomplete or unsound because the side effects of the executed code depend on the runtime context.

In Listing 2.4, the variable `code` contains a string that defines and evaluates a new variable assignment at runtime. When `exec()` is called without optional dictionary arguments, the code is executed in the current scope. If this occurs at the module level, `x` becomes a global variable. However, if executed inside a function, variables defined dynamically via `exec` are not accessible as regular local variables because Python determines local variable scope at compile time.

```python
code = "x = 5 + 10"
exec(code)  # Executes the string as Python code
```

Listing 2.4: Dynamic evaluation in Python

2.2.4 Complex Builtin Library

Python's vast standard library provides powerful tools for developers but presents additional challenges for analysis. Static analysis tools must support a wide range of library functions, thereby increasing both the scope and complexity of the analysis. Moreover, Python's control

flow can be intricate due to the use of high-level iterators, generators, and asynchronous functions, all of which contribute to a more dynamic and less predictable flow than in typical C or Java programs.

In Listing 2.5, the function `squares()` is a generator that yields values lazily, and the built-in function `map()` applies a transformation to each yielded element. To analyze this behavior, a static analyzer requires abstract representations of generator states and iterators, increasing the complexity of control-flow and data-flow analysis in Python.

```
1 # Example using iterators and generators
2 def squares(n):
3     for i in range(n):
4         yield i * i
5
6 # The built-in map function applies a lambda to each element
7 result = map(lambda x: x + 1, squares(5))
8
9 print(list(result))  # Produces [1, 2, 5, 10, 17]
```

Listing 2.5: Iterators and generators in Python's builtin library

2.2.5 Import Aliasing

Python provides several mechanisms for importing external libraries, including importing complete modules, selectively importing individual functions or classes, and introducing aliases that rename modules or symbols. While this flexibility supports concise and expressive programming, it poses significant challenges for static analysis. Wildcard imports (`from module import *`) obscure the origin of identifiers, making it difficult to determine which functions or classes are actually in use. Similarly, the introduction of aliases (e.g., `import numpy as np`) conceals the original library names, thereby requiring additional resolution steps for accurately identifying dependencies and references. As a consequence, static analysis tools must incorporate alias tracking and disambiguation techniques not only for program variables but also for identifiers in general, including modules, packages, and imported symbols, to achieve sound and precise results.

In Listing 2.6, the module `numpy` is imported under the alias `np`, while the wildcard import from `math` introduces all available mathematical functions without explicit qualification. To analyze this code accurately, a static analyzer must resolve the alias `np` to its corresponding module `numpy` and determine the origins of the functions `sin` and `sqrt` introduced by the wildcard import.

```
1 import numpy as np
2 from math import *  # Wildcard import obscures what is being used
3 array = np.array([1, 2, 3])
4 result = sin(array[0]) + sqrt(array[1])  # The source of sin and
      sqrt is unclear due to wildcard import
```

Listing 2.6: Import aliasing in Python

2.2.6 Chained Function Calls

Consider the function call `my_dataframe.values[:1].astype(int)` in the code snippet in Listing 2.7. Here, the variable `my_dataframe` is of type DataFrame from the Pandas library. The attribute access `.values` returns a NumPy array, followed immediately by an operator-based slice `[:1]`, before finally invoking `.astype(int)`. This structure highlights a distinct challenge in Python analysis: chains frequently mix standard method calls with operator overloading (e.g., slicing via `__getitem__`), all of which rely on dynamic attribute resolution. Although such expressions can be lowered into intermediate representations, the heavy reliance on library-defined semantics means that failing to resolve a single link causes imprecision to propagate through the entire chain, rendering subsequent analysis inaccurate.

```
1 import pandas as pd
2 my_dataframe = pd.DataFrame({'numbers': [1.2, 2.3, 3.3]})
3 values = my_dataframe.values[:1].astype(int)  # Chained function
      calls
```

Listing 2.7: Chained function calls in Python

2.2.7 Variable Reuse

Variable reuse is another common challenge in Python, particularly in Jupyter Notebooks. It is common for the same variable name to be reused in different cells for entirely different purposes. For example, in Listing 2.8, a variable named `model` could be assigned to a `Sequential` object in one part of the notebook and a `LogisticRegressionCV` object in another. To accurately analyze such scenarios, the analyzer must distinguish between different lifetimes of the same variable name across the program. This can be achieved through flow-sensitive analysis that takes into account the execution order and tracks variable states over time. However, in many practical cases, a commonly used workaround is to split variable live ranges, as done in static single assignment (SSA) form, which assigns distinct versions to successive redefinitions without requiring complex dataflow analysis.

```
1 from sklearn.linear_model import LogisticRegressionCV
2 model = LogisticRegressionCV()  # First use of `model'
3
4 # Later in another cell
5 from tensorflow.keras.models import Sequential
6 model = Sequential()  # Reuse of `model' for a different object type
```

Listing 2.8: Variable reuse in Python

2.2.8 Scarcity of Type Annotations

Although Python supports optional type hints (PEP 484), a vast majority of real-world code remains unannotated [DGP22a]. Crucially, even when present, these annotations are *not enforced* by the Python interpreter; they serve only as metadata. In the absence of such hints, static analysis lacks explicit guarantees regarding input and output types. Furthermore, even when developers do provide annotations, many existing static analysis tools are not equipped to integrate this metadata into their analysis pipeline. Consequently, valuable semantic information is

often ignored, causing tools to default to less precise inference methods despite the availability of explicit types.

In Listing 2.9, the function `load_data` is defined without any type information. In contrast, `load_data_typed` includes explicit type hints for both the argument `path` (expected to be a string) and the return value (expected to be a `pd.DataFrame`).

```python
# Without annotation: Return type is unknown
def load_data(path):
    return pd.read_csv(path)

# With annotation: Return type is explicit
def load_data_typed(path: str) -> pd.DataFrame:
    return pd.read_csv(path)
```

Listing 2.9: Function with and without type annotations

2.2.9 CPython as the De Facto Reference

The primary implementation of Python, *CPython*, serves as the de facto reference for Python behavior. Since Python lacks a formal ISO/ECMA standard, developers rely on the behavior exhibited by CPython to understand the syntax and semantics of Python. This means that any specific behavior, bug, or quirk in CPython can become the reference for how Python code should work, even if other implementations such as PyPy or Jython may behave differently. This reliance can lead to inconsistency or confusion among developers, as behaviors that are considered standard in CPython might not be replicated in other environments, resulting in divergent behavior [Dif].

2.3 Unstructured and Undocumented Jupyter Notebooks

Jupyter Notebooks are widely adopted for data science and machine learning projects due to their interactive and flexible environment. However, the flexibility often leads to notebooks becoming unstructured and poorly documented, making them challenging for both novice and experienced users to understand, maintain, and reproduce [PMBF19, RTH18, WKLZ20a, WLZ20, WWD+22, WKLZ20b]. Unlike traditional software development practices, which often emphasize organization, modularization, and extensive documentation, Jupyter Notebooks frequently fall short in these areas.

2.3.1 Lack of Structure

The lack of structure in Jupyter Notebooks is largely attributed to the exploratory nature of their usage. Data scientists use notebooks as interactive scratchpads for experimenting with data, model training, and visualization. As a result, the code tends to be linear, without modular functions or reusable components, leading to notebooks that are difficult to navigate and maintain. Variables and functions are often defined in arbitrary cells, and the execution order may vary significantly, resulting in dependencies that are difficult to track. This unstructured approach makes it challenging to reuse code or understand the logical flow of the notebook, particularly when revisiting it or when sharing it with collaborators.

Unlike traditional programming, Jupyter notebooks do not encode control flow explicitly in the program structure. Cells can be executed in arbitrary order while sharing a persistent

global state, so program behavior depends on execution history rather than textual order. This implicit, user-driven execution model complicates reasoning about dependencies and program behavior.

2.3.2 Insufficient Documentation

Markdown cells, designed to provide explanations and context, are often not used. Even when markdown cells are present, they frequently do not provide a sufficient explanation of the code logic or its intended purpose. Studies have shown that 30.93% of publicly available Jupyter Notebooks do not contain markdown cells [RTH18, PMBF19]. Without sufficient documentation, understanding the purpose of individual cells or the overall flow of the notebook is challenging.

2.3.3 Limited Support for Structuring and Documentation

The lack of built-in mechanisms to enforce structure or encourage documentation exacerbates these problems. Current notebook environments offer minimal guidance for organizing code, maintaining consistency, or integrating narrative explanation. To address these shortcomings, tools that automatically suggest cell organization, insert contextual comments, or promote modularization are required.

#Gen

3 HeaderGen: Call-Graph Based Structuring of Python Notebooks

In the evolving landscape of machine learning (ML) and data science, Jupyter Notebooks have emerged as the de-facto platform for creating ML solutions within the ML community. Jupyter notebooks resonate with the paradigm of *literate programming* postulated by Donald E. Knuth [Knu84]. This approach advocates the integration of code, comprehensive documentation, and visual representations within a unified document to promote understanding and facilitate the sharing of complex solutions. The underlying principles of literate programming include: (1) Augmenting code with descriptive text and illustrative diagrams, (2) Imposing a coherent narrative by separating code cells with pertinent headers, and (3) Logically segmenting and labeling the program's reusable modules.

Within notebooks, code is written in executable *code cells* while accompanying documentation is written in *markdown cells*. Augmenting code segments with explanatory textual content enhances the overall comprehensibility of notebooks and further promotes collaboration [WFM+22]. Moreover, a markdown-to-code cell ratio of 2, as posited by Wagemann et al. [WFM+22], serves as an indication of adherence to literate programming principles. This assertion finds further support in the work of Samuel et al. [SM22], who report that a higher markdown-code cell proportion correlates with better reproducibility, a vital metric in scientific studies.

Need for Automated Support. Despite the fact that the inherent capabilities of Jupyter notebooks resonate with literate programming principles, real-world adoption often diverges from this ideal. Empirical studies consistently report that publicly available notebooks often lack documentation and contain poor structural organization [KRA+18, WLZ20, PMBF19, QCL22]. Many practitioners describe notebooks as personal scratchpads and "messy" rather than carefully curated artifacts, with more than 30% of published notebooks containing no markdown annotations [RTH18, PMBF19]. This scarcity of annotations reveals the prevailing negligence towards best practices. Such omissions are especially detrimental in platforms like Kaggle[1], where subpar practices risk propagating to the next generation of ML practitioners. Therefore, there exists an imperative for the software engineering research community to develop tools that address this problem.

Early Prototype and Limitations. To address the lack of documentation in notebooks, we initially developed a prototype [SVB21] designed to automatically generate headers. This

[1] `https://www.kaggle.com/`

prototype relied on an abstract syntax tree (AST)-based approach to identify call sites within Jupyter notebooks and resolve them to their corresponding libraries. While this demonstrated the potential of static analysis for improving notebook comprehension, it also revealed significant technical hurdles inherent to analyzing Python code. Specifically, the prototype failed to address the following challenges:

1. **Limited AST-based analysis:** the prototype relied on call-node extraction from the AST without alias resolution, robust handling of complex Python syntax, or support for inter-procedural flows, leading to reduced accuracy.

2. **Absence of robust program analysis techniques:** meaningful structural augmentation requires precise identification of all function calls within a notebook. However, existing static analysis techniques for Python proved insufficient. Python's features, such as dynamic typing, dynamic execution, and reflection, pose inherent challenges to static reasoning [SSL+21a, KPSB21]. Compared with languages such as Java, Python lacks mature tool support for advanced static analysis techniques [YMH22a]. Most available tools perform only rudimentary AST-based analysis, and the lack of precise type inference further undermines accuracy (see Section 2.2.1 on dynamic typing).

3. **Limitations of state-of-the-art call-graph tools:** PyCG [SSL+21a] represents a significant step forward, as it builds call graphs from an intermediate representation derived from AST and supports several complex Python features. However, it does not analyze calls to external libraries and operates in a flow-*insensitive* manner. Flow sensitivity, which considers the order of program statements and tracks variable states at different points, is crucial for accurate analysis (see Section 2.2.6 and 2.2.7). Its absence restricts the applicability of PyCG in real-world notebooks.

4. **Lack of fully qualified names:** function calls were resolved to their library of origin but not to their fully qualified definitions. For example, functions such as `seaborn.load_dataset()` are defined in internal modules (e.g., `seaborn/utils.py`) that are not referenced directly in the import statements. Because the prototype relied solely on surface-level import information and AST traversal, it lacked the module-level resolution required to trace functions through internal package structures.

5. **Manual function-to-phase mapping:** the classification of functions into machine learning workflow phases (e.g., preprocessing, training, evaluation) depended on a manually curated mapping. This required frequent updates to accommodate new libraries and APIs, reducing scalability.

Overview of HeaderGen. Building on the lessons learned from the prototype, this dissertation's first major contribution introduces HeaderGen, a static analysis tool that augments Python-based Jupyter Notebooks with structural annotations. HeaderGen addresses the twofold challenge of inadequate static analysis and unstructured notebooks by implementing a sophisticated analysis pipeline. Unlike the prototype, HeaderGen extends the analysis in PyCG to include flow-sensitivity (to accurately track variable states at specific program locations) and external library resolution (to identify return types from libraries like pandas and scikit-learn).

Using this enhanced infrastructure, the tool extracts function calls from notebook code cells and classifies them according to a taxonomy of machine learning operations [WWD+22]. The taxonomy, proposed by Wang et al. [WWD+22], outlines the sequence of activities typical in ML workflows, beginning with data preparation, followed by feature engineering, and model building and training (see Figure 3.1). This workflow defines an implicit narrative structure

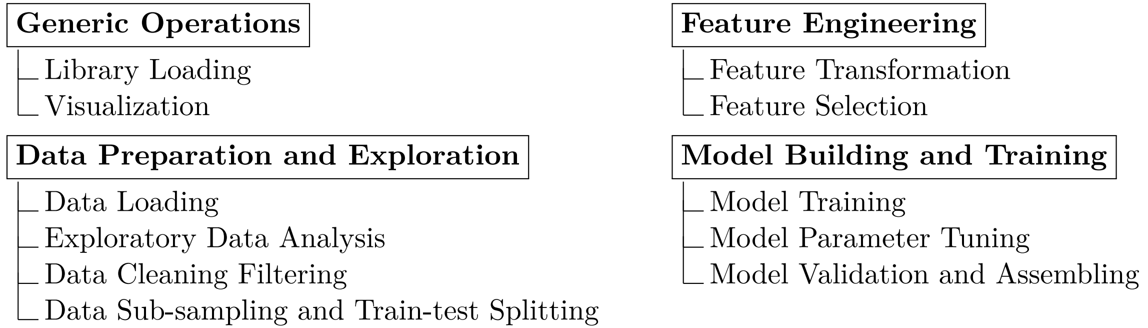


Figure 3.1: Taxonomy of machine learning operations based on Wang et al. [WWD+22].

that HeaderGen aims to recover automatically. Based on these classifications, HeaderGen constructs a narrative structure that improves notebook navigation and comprehension by generating: (1) a global index of machine learning operations placed at the beginning of the notebook, and (2) markdown headers annotating individual code cells to reflect the operations they perform.

To do so, the tool identifies function calls with high accuracy, assigns them to the appropriate ML workflow categories, and uses the resulting classifications to build a structural map of the notebook. This map is rendered as an “Index of ML operations” at the top of the notebook and is complemented by cell-level annotations that summarize the operations performed within each code cell (see example on page 30).

Addressing Key Challenges. In summary, HeaderGen addresses the identified technical hurdles by replacing the prototype’s limited AST-based approach with a robust static analysis infrastructure. To mitigate the specific challenges of dynamic typing, the system employs a heuristic-based approach to external library return-type resolution, ensuring method calls are correctly identified even when object types are not explicitly declared in the program. Addressing the limitations of existing tools like PyCG, HeaderGen introduces flow-sensitivity to accurately track variable states. Furthermore, it resolves the ambiguity of function definitions by implementing a hybrid module-level resolution, enabling the retrieval of fully qualified names that were previously inaccessible via surface-level imports. Finally, to overcome the scalability issues of manual mapping, HeaderGen incorporates an ML-based approach to automatically classify function calls into their respective workflow phases.

Evaluation. To evaluate the performance of HeaderGen’s static function call analyzer, we employed an enhanced version of PyCG’s micro-benchmark, complemented by a real-world benchmark consisting of 15 notebooks sourced from Kaggle. On the real-world benchmark, HeaderGen achieved 95.6% precision and 95.3% recall, outperforming PyCG and other function call analyzers based on off-the-shelf tools such as *pyright*[2] and *Jedi*[3]. On the same benchmark, we also evaluated HeaderGen for header annotation and achieved 85.7% precision and 92.8% recall. Building on the initial publication of these results [SVWLB23b], the benchmark suite was subsequently extended with an additional 15 Jupyter notebooks to further assess HeaderGen across a broader spectrum of real-world notebooks. HeaderGen achieves an average of 94.4% precision and 91.6% recall. These results are consistent with the findings from the initial benchmark, confirming the robustness of HeaderGen across a wider range of notebook styles and domains.

[2] `https://github.com/microsoft/pyright`
[3] `https://github.com/davidhalter/jedi`

Additionally, through a user study involving eight data science practitioners, we gathered evidence indicating that HeaderGen significantly enhances navigation speed and improves comprehension.

Contributions. The primary contributions of the work presented in this chapter are as follows:

- A static analysis technique for Python Jupyter Notebooks that enhances them with structural and explanatory annotations, supporting the principles of literate programming.
- An improved function call extraction method that achieves high precision and recall on real-world benchmarks.
- An open-source implementation of HeaderGen, available under the Apache 2.0 license at `https://github.com/secure-software-engineering/HeaderGen`

Why Not Large Language Models? Given the recent surge in Generative AI, a modern reader might question why LLMs were not employed to solve these classification and structuring tasks. First, it is important to note that the core research and development of HeaderGen predated the widespread availability of capable LLMs such as GPT or Llama. Second, traditional deep-learning-based solutions rely heavily on large-scale, manually annotated datasets, which are scarce for our use-case of code annotation.

Structure. The remainder of this chapter is organized as follows: We present challenges in statically analyzing Python code with a motivating example in the Section 3.1, followed by detailing our design in Section 3.2. We then present the research questions in Section 3.3. We discuss the details of our micro-benchmark and real-world benchmark in Section 3.4. We address the research questions from Section 3.5 to Section 3.8, and discuss related work in Section 3.9. The limitations of HeaderGen are discussed in Section 3.10, and finally, the paper is concluded in Section 3.11.

3.1 Motivating Example

As a motivating example, consider the notebook in Figure 3.2. It consists of one markdown cell, which is rendered as an HTML header, and five code cells that can be identified by the comments in the first line of each code cell. The example notebook in Figure 3.2 is a concise version of a real-world notebook containing a machine learning based solution.

In cell 1, various ML libraries are imported. In cell 2, a sample dataset called "iris" from the seaborn library is loaded and processed through basic feature selection to retain only the essential columns. The selected values are cast to the NumPy `float64` type, and the dataset is checked for null entries. In cell 3, the data is split into training and test subsets. Cells 4 and 5 then define, train, and evaluate two different machine learning models using the processed dataset: cell 4 applies a logistic regression model, while cell 5 uses a deep learning–based sequential model.

Note that this notebook is undocumented and does not contain any explanatory text or structural headers as markdown cells, thereby violating the principles of literate programming. Empirical studies show that approximately one in three publicly available notebooks lacks any markdown content [PMBF19]. The absence of explanatory text or structural organization forces machine learning practitioners, particularly beginners, to invest considerable effort in navigating and understanding the notebook. This difficulty is further amplified by the fact that nearly one-third of real-world notebooks contain at least 50 cells [PMBF19].

On the other hand, the example notebook poses several challenges to SA, including:

Figure 3.2: Example of a Jupyter notebook containing a machine learning solution.

- **Import aliasing:** different ways of importing libraries, and importing libraries with aliases as shown in cell 1. This is further complicated with wildcard imports of the form "`from MODULE import *`". (see Section 2.2.5)

- **Dynamic typing:** in cell 2, the type of the variable `iris_dataset` is not known statically, i.e., the return-type of the function `load_dataset()` is not known statically unless the developer manually annotates the function definition. Unfortunately, widespread adoption of type annotation is still lacking in practice (see Section 2.2.8). As a result, subsequent statements that involve the variable `iris_dataset` cannot be resolved, i.e., in cell 2 lines 4–7. (see Section 2.2.1)

- **Chained function calls:** in cell 2 line 4, the expression `iris_dataset.values[].astype()` spans multiple libraries. Here, `iris_dataset.values` yields a NumPy array through the Pandas *DataFrame* API, and `astype()` is a NumPy method. Although the syntactic chain can be parsed, accurate modeling requires the analyzer to propagate type information across all steps. Any imprecision accumulates along the chain, and current tools cannot resolve these dependencies statically (see Section 2.2.6).

- **Variable reuse:** the same variable `model` is reused in cells 4 and 5, for different model objects, i.e., `Sequential` and `LogisticRegressionCV` objects. Reuses of the same variable names are common in notebooks. Therefore, for precise annotation of code cells, the analyzer should know the type of an object at a specific location in the notebook. In other words, the analysis should be flow-*sensitive*. (see Section 2.2.7)

Note that in general, analyzing dynamic programming languages such as Python poses several other challenges as discussed in Section 2.2. Features such as dynamic evaluation, where a string can be evaluated as a code fragment at runtime, and complex control-flow structures with generators also pose challenges to SA.

In summary, for HeaderGen to accurately classify code cells based on function calls, the static analyzer needs to: (1) handle complex Python features, (2) statically resolve return-types of external library calls, and (3) be flow-sensitive.

To address these issues, HeaderGen incorporates a static analysis pipeline tailored to the notebook setting. The pipeline integrates enhanced call-graph construction, type resolution, and flow-sensitive analysis, enabling the tool to recover accurate function call information even in the presence of aliasing, chained calls, and variable reuse. The output of this analysis is subsequently combined with a taxonomy-based classification model to generate structural annotations for the notebook.

3.2 HeaderGen's Static Analyzer

A high-level overview of HeaderGen is shown in Figure 3.3. The initial pre-processing step involves transforming a notebook into a standard Python script. This process aids in eliminating metadata from the notebook, ensuring that only pertinent information remains for subsequent analysis. Post this conversion, in phase one, HeaderGen proceeds to analyze the resultant Python script to create an Extended Assignment Graph (EAG). Leveraging the EAG, HeaderGen extracts flow-sensitive callsite information in the second phase. The final phase involves augmenting the notebook with structural annotations corresponding to the callsites by using an ML-based taxonomy classifier.

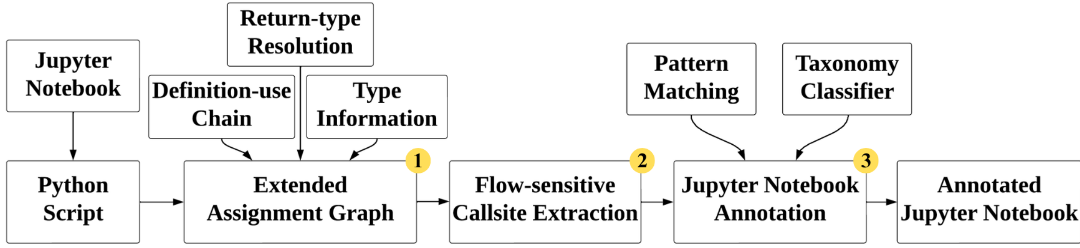


Figure 3.3: High-level overview of HEADERGEN.

The construction of the EAG and the methodology for extracting flow-sensitive callsite information are presented in Sections 3.2.1 and 3.2.2. Section 3.2.3 describes the ML-based taxonomy classifier, and Section 3.2.3 details the procedure for annotating notebooks using the output produced by the analyzer.

3.2.1 Extended Assignment Graph

Motivation and Departure from PyCG. The EAG extends the assignment graph introduced by PyCG [SSL+21a]. PyCG models a Python program as an inter-procedural fixed-point analysis that constructs assignment relations among program identifiers such as functions, variables, classes, and modules. Each node in the original assignment graph represents an identifier, and directed edges capture value assignments, or reference flows between identifiers. This is conceptually similar to the Pointer Assignment Graph (PAG) in the Spark framework [LH03]. However, PyCG adapts this model to address Python-specific semantics by abstracting away low-level memory operations. Specifically, whereas a PAG employs distinct load/store edges and field dereference nodes, PyCG operates on high-level identifiers by treating modules and functions as first-class objects and resolving attribute accesses directly to their defining namespaces via the Method Resolution Order (MRO). PyCG's assignment graph enables the subsequent extraction of a call graph by resolving all identifiers that a callable variable may reference. Here, a callable denotes any Python object that can be invoked using parentheses. However, two main limitations restrict its applicability to notebook-based ML programs.

The first limitation concerns flow-*insensitivity*. PyCG represents every program variable as a single node, regardless of how many times the variable is redefined. This design propagates imprecise information across unrelated program states. In notebooks, variables are frequently redefined across cells, often holding objects of incompatible types. When such redefinitions occur, PyCG merges all instances into one node, obscuring the temporal order of assignments and the specific data-flow paths among cells.

The second limitation involves the incomplete handling of external libraries. PyCG intentionally omits analysis of functions defined in external modules. Although this decision improves scalability, it prevents the tool from reasoning about the behavior of library functions and their return types. In notebook environments where most computation relies on external ML libraries, this omission constrains the accuracy of call-graph extraction and subsequent program comprehension tasks.

The EAG addresses these limitations through three targeted extensions:

1. Flow-sensitive definition-use chains: Each definition of a variable is indexed by its program location, producing distinct nodes for every redefinition and preserving execution order.

2. Canonical resolution of external functions: Calls to library functions are resolved to their fully qualified names (FQNs) through controlled reflection, allowing links between notebook code and the actual implementation sites within external packages.

3. Return-type resolution for external APIs: Each external definition stores an abstract type obtained from resolving the return types of external API calls through type-stub lookup (see Section 2.2.8). These external return types are integrated into the EAG, with literal and user-defined types added when available, enabling type-aware reasoning.

These extensions retain the semantics of PyCG's assignment relations while improving precision in a dynamic and modular notebook environment. Figure 3.4 illustrates the difference: PyCG collapses all definitions of the variable `model` in our motivating example (c.a., Figure 3.2) into a single node, whereas the EAG distinguishes definitions in cells 4 and 5 and connects them respectively to constructors imported from external libraries: scikit-learn and keras. The EAG forms the basis for accurate call-site resolution and structural notebook annotation in later stages of HeaderGen.

model

⇒

model:(C4,2)

sklearn.linear_model.
_logistic.
LogisticRegressionCV

model:(C5,2)

keras.engine.
sequential.Sequential

(a) Assignment Graph

(b) Extended Assignment Graph

Figure 3.4: Generated assignment graphs for the variable "model" in the motivating example, left in PyCG (empty), right in HeaderGen (flow-sensitive).

Construction of the Extended Assignment Graph. The construction of EAG in HeaderGen extends the inter-procedural fixed-point analysis of PyCG [SSL+21a]. Before detailing the extensions introduced by HeaderGen, it is necessary to summarize the intermediate representation (IR) defined by PyCG [SSL+21a]. This IR models Python programs in terms of four categories of identifiers: functions (*func*), variables (*var*), classes (*cls*), and modules (*mod*). Each identifier is paired with its category, yielding a definition of the form $(x, \tau) \in \mathit{Identifier} \times \mathit{IdentType}$. A definition, together with the namespace in which it appears, constitutes an object in the IR, following the Obj construction. Namespaces are represented as ordered sequences of definitions and ensure that objects with identical identifiers but different scopes remain distinguishable. This representation captures the structural relationships among modules, classes, functions, and variables in Python programs.

HeaderGen preserves PyCG's IR and its transition rules while extending it with flow-sensitive node creation and explicit modeling of external functions. For clarity, the four domains that constitute the IR are briefly defined below; a concrete example of these domains applied to a code snippet is provided in Appendix A.

- **Namespace:** The namespace domain records the defining context of each identifier and distinguishes objects and attribute accesses based on where definitions occur (see Listing A.2).
- **Scope:** The scope domain represents the nesting structure of definitions and determines which identifiers are visible within specific namespaces (see Listing A.3).
- **Class hierarchy:** The class-hierarchy domain models inheritance relations among classes and supports the resolution of attributes and methods defined in base and ancestor classes (see Listing A.4).

- **Assignment graph:** The assignment-graph domain captures potential value flow between objects by recording assignment relations across variables, parameters, and return values (see Listing A.5).

These domains constitute the IR that HeaderGen preserves and extends. Algorithm 1 summarizes the EAG construction process, and the subsequent sections describe the integration of definition-use chains for flow-sensitivity and external return-type resolution.

Definition–Use Chains for Flow Sensitivity

During analysis, each expression in the source code is evaluated by repeatedly applying PyCG's transition rules until convergence. Flow sensitivity is achieved by indexing variable definitions with their program locations. This is realized through a definition–use chain (DUC) analysis over the notebook's Python AST, computed using *Beniget* [ser22]. The DUC captures, for every assignment, the reachable uses without intervening redefinitions, respecting lexical scope and Python's name-resolution rules.

Given the DUC index, HeaderGen creates flow-sensitive nodes during EAG construction; a concrete example of this mechanism is provided in Appendix A.3. When an assignment $x := y$ at source location ℓ is reduced by PyCG's transition rules, HeaderGen constructs versioned nodes (x, ℓ) and (y, ℓ) and inserts the edge $(x, \ell) \rightarrow (y, \ell)$ into E. This mechanism prevents the merging of redefinitions that would occur in PyCG's flow-insensitive assignment graph, allowing the EAG to preserve evaluation order and support precise propagation of facts.

Return-type Resolution for External Library Calls

A large fraction of code in notebook-based machine learning workflows consists of calls to external library functions. These functions often produce complex structured outputs such as `pandas.DataFrame`, `numpy.ndarray`, or neural network objects that form the backbone of later computation. Accurate recovery of these return types is therefore essential for constructing a precise, type-annotated EAG, because all subsequent attribute accesses and method invocations depend on the types of the variables produced by such calls. However, achieving this via static analysis presents significant difficulties.

Challenges in resolving external libraries. Two primary obstacles prevent the direct inference of return types for external calls: computational limitation and namespace aliasing:

1. Analytical Constraints and Tool Limitations: Directly analyzing modern machine learning libraries at the source level is computationally infeasible because they are large, heterogeneous, and dynamic. They intermix Python and native code and rely extensively on metaprogramming. Experiments with PyCG on entire packages like *TensorFlow*, *NumPy*, or *Pandas* confirmed that exhaustive analysis fails to terminate or exhausts memory. Even when restricted to a subset of modules, the prevalence of runtime dispatch, decorator-based indirection, and dynamic imports prevents reliable inference of return types through conventional static methods. Furthermore, conventional static analyzers (e.g., *pytype*, *pyre*) depend on explicit annotations which do not generalize to dynamically exported functions, while learning-based approaches (e.g., Typilus [ABDG20a] and Type4Py [MLPG22a]) often lack awareness of specific library semantics.

2. The Canonicalization Problem: Given that direct analysis is infeasible, another viable alternative is to rely on pre-annotated type summaries. However, this approach introduces a new challenge: identifying the *canonical* name of the function being called. Python libraries often expose re-exported or aliased entry points. Top-level imports in Python can re-export symbols from

Algorithm 1: EAG Construction by Augmenting PyCG

Input : Script: P, line↔cell mapping: `map`, DUC index: DUC, summary store: Σ
Output: EAG = (V, E), final PyCG state σ

```
1  Initialize PyCG state σ ← (π = ∅, s, n, h)  // assignment graph, scope, namespace,
     class hierarchy
2  V ← ∅, E ← ∅                                            // EAG nodes and edges
3  foreach top-level expression or statement e ∈ P in evaluation order do
4      while e is not an object do
5          (σ, e) ← PyCG_Step(σ, e)          // apply one transition rule of PyCG
       // Flow-sensitive assignment
6      if e reduced from an assignment x := y at source location ℓ then
7          ℓ_x ← defSite(x, ℓ, DUC); ℓ_y ← useSite(y, ℓ, DUC)
8          v_x ← (x, ℓ_x); v_y ← (y, ℓ_y)
9          V ← V ∪ {v_x, v_y}; E ← E ∪ {(v_x → v_y)}
10         if typeOf(y, σ) ≠ ⊥ then
11             attachType(v_x, typeOf(y, σ))
                                                        // propagate abstract type
       // External call, canonicalization, and abstract return type
12     if e is a call expression f(a⃗) at location ℓ then
13         if isExternal(f, σ) then
14             fqn ← CanonicalResolve(f, σ)                          // Algorithm 2
15             τ ← LookupReturnType(fqn, Σ)
16             if τ ≠ ⊥ then
17                 v_r ← (ret_f, ℓ);                        // virtual return variable
18                 V ← V ∪ {v_r};
19                 attachType(v_r, τ)
20 return ((V, E), σ)
```

nested submodules, and decorators may wrap the original callable. For instance, in the motivating example (see Figure 3.2), the expression `sns.load_dataset()` refers to a symbol in the top-level `seaborn` package, but the actual implementation resides in `seaborn.utils.load_dataset`. Without resolving `sns.load_dataset` to its canonical name, the analyzer cannot locate the correct type definition in any static summary store.

Methodology: Canonical resolution through controlled reflection. To overcome these challenges, HeaderGen employs a lightweight strategy. Rather than expanding external packages, it identifies the canonical name of the function and looks up its return type in a pre-annotated summary store. HeaderGen introduces a resolution routine that uses *controlled reflection* to obtain the fully qualified name (FQN) of an external callable without invoking it. The process involves three steps:

1. Evaluate a reference to the callable using Python's `eval(<func>)` without execution, i.e., by stripping the call operator `()`, thereby avoiding execution.

2. Apply `inspect.unwrap()` to remove decorators and wrapper layers, exposing the underlying implementation object.

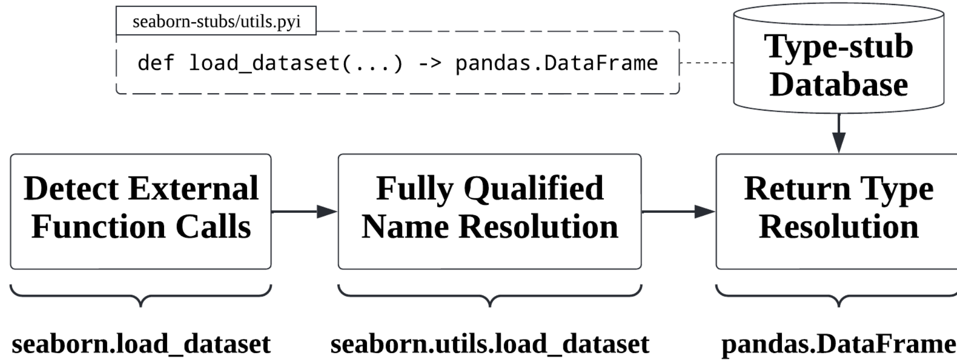


Figure 3.5: Workflow of imported library function return-type resolution.

3. Extract the `module` and `qualname` attributes from the unwrapped object and combine them into a canonical identifier.

Summary-based return type approximation. Once the canonical name is established, HEADERGEN fetches the return type by consulting Σ, a static Type Summary Store. Each entry in Σ associates a canonical function name with an abstract return type (e.g., mapping `seaborn.utils.load_dataset` to `pandas.DataFrame`). We constructed these annotations by aggregating existing developer-annotated `.pyi` stubs, parsing library documentation, and performing manual verification for common ML libraries. When a match is found in Σ, the inferred type is attached to the EAG node corresponding to the call's return value.

Figure 3.5 summarizes this workflow. In the motivating example, this process transforms `sns.load_dataset()` into the canonical identifier `seaborn.utils.load_dataset`, enabling a correct lookup in the summary store. This enables the analyzer to infer that `iris_dataset` holds a `pandas.DataFrame`, allowing subsequent operations at (C2,4-7) to be resolved. Algorithm 2 formalizes this procedure.

Algorithm 2: Canonicalization of External Function Calls

Input : Callee symbol f, PYCG state σ
Output: Canonical fully qualified name fqn or $\bot$

```
1  Function CanonicalResolve(f, σ):
2      if not isExternal(f, σ) then
3          return ⊥
4      m ← importModuleOf(f, σ)                          // import top-level module
5      o ← eval(m +' .' + f)          // evaluate reference to callable, not invocation
6      u ← inspect.unwrap(o)                    // remove decorators and wrapper layers
7      mod ← u.__module__;   name ← u.__qualname__
8      if mod ≠ ⊥ and name ≠ ⊥ then
9          fqn ← mod +' .' + name                           // compose canonical path
10         return fqn
11     else
12         return ⊥
```

Integration into EAG construction. The canonicalization procedure described in Algorithm 2 is incorporated into the EAG construction process at line 14 in Algorithm 1. During the reduction of a call expression, Algorithm 1 invokes the canonical resolution routine to obtain the callee's fully qualified name, thereby linking the surface-level symbol used in the notebook to its underlying implementation site. This canonical name is then used to query the summary store Σ, which provides the return type associated with the external function. When a summary entry is available, the inferred type is attached to the EAG node that represents the call's return value.

Following PyCG, return values are modeled using the virtual variable `ret` introduced by the [RETURN] rule [SSL+21a]. HeaderGen reuses this mechanism by creating a node $(\texttt{ret}, \ell)$ for each external call at its program location ℓ and attaching the corresponding abstract return type obtained from Σ.

Implications. Through this combination of canonical resolution and summary-based approximation, HeaderGen attains type-aware flow sensitivity without the need for full library analysis when type annotations exist. The resulting EAG integrates both user-defined and library-derived variables into a single, annotated representation. This enriched graph forms the foundation for downstream notebook annotation and taxonomy classification, discussed in the following sections.

3.2.2 Flow-sensitive Callsite Extraction

The EAG constructed in the previous phase serves as the foundation for identifying and localizing executable function invocations within the notebook. Using the call-graph construction mechanism of PyCG, HeaderGen performs a flow-sensitive traversal over the intermediate representation of the program. Callable objects are resolved according to the definitions and type information recorded in the EAG. This approach ensures that targets are resolved based on the variable state valid at each specific call site, allowing the analysis to distinguish between multiple invocations even when variables are redefined or types change during execution.

Each callsite is then mapped to its corresponding notebook cell using the line↔cell mapping established during notebook-to-script conversion. This mapping ensures that every function invocation, including those occurring transitively through user-defined functions, can be attributed to the notebook cell initiating the call. For example, when a user-defined function `x()` defined in one cell invokes other library routines, HeaderGen attributes these nested invocations to the calling cell that executes `x()`, thereby constructing a transitive closure over the call graph.

This stage outputs a comprehensive mapping between notebook cells and their associated call sites, serving as the basis for annotation and classification in the final phase.

3.2.3 Jupyter Notebook Annotation

The final stage of the pipeline converts the results of the static analysis into structured annotations intended to enhance the comprehensibility of complex notebooks. This stage integrates a Taxonomy Classifier that maps resolved function calls to high-level ML operations and a Pattern Matcher that identifies operations implied by Python control flow or data-manipulation constructs rather than by explicit library calls.

Taxonomy Classifier

Within the Python ecosystem, machine learning tasks are implemented through diverse APIs spanning numerous libraries. In this context, an API refers to any callable entity, such as a

Table 3.1: Overview of Kaggle competitions and notebooks

Kaggle Competition	**Teams**	**Notebooks**
Predict Future Sales	15,656	833
Titanic - Machine Learning From Disaster	14,560	1,089
Santander Customer Transaction Prediction	8,751	907
Home Credit Default Risk	7,176	900
IEEE-CIS Fraud Detection	6,351	820
M5 Forecasting - Accuracy	5,558	608
Total Notebooks		6,698

function, method, or class constructor, that is exposed by an external library. These APIs can be systematically categorized according to the ML operation they realize, such as data preparation, model training, or evaluation. In HeaderGen, the callsite information derived from the flow-sensitive extraction stage provides the foundation for such categorization.

Earlier versions of HeaderGen relied on a manually curated mapping between APIs and ML operation categories [SVWLB23b]. Such a static mapping is not sustainable given the rapid evolution of ML libraries. To overcome this limitation, a simple supervised ML model is introduced as a proof-of-concept that automatically assigns each API call to one or more ML operation categories by analyzing its documentation string (docstring).

Dataset Preparation. In the process of constructing a supervised machine learning model, the initial requisite is a well-labeled dataset. Given the absence of any publicly accessible datasets that maps API calls with ML operations, our first step was to establish such a dataset. To accomplish this, we used Kaggle, a platform that offers a wide range of resources such as datasets and notebooks. The ML notebooks on Kaggle serve as a valuable resource for practical code implementations with API calls from diverse libraries. We retrieved 6,698 notebooks from Kaggle's top six competitions, leveraging its API. The rationale behind selecting these specific competitions was their widespread popularity based on the number of submissions and participating teams. A breakdown of these competitions and notebooks is listed in Table 3.1.

We used HeaderGen to analyze the selected notebooks. Our primary objective was to extract the fully qualified names associated with all the API calls in the notebook, as well as their corresponding docstrings. This data forms the foundational basis for our dataset.

Empirical Analysis. HeaderGen identified a total of 141,657 API call-sites to 12 different ML libraries. After removing the duplicates, we were left with 2,553 unique API calls. The breakdown of this is listed in Table 3.2.

Data Annotation. To ensure the accuracy and reliability of our annotations, we engaged six experts specializing in machine learning. These experts were identified through our professional network on LinkedIn. Given the vastness of the dataset, and in consideration of the potential for annotator fatigue, we opted to select a subset comprising 400 APIs from various libraries. This strategic selection was made to ensure a comprehensive coverage across different functionalities while at the same time optimizing the use of our resources.

The chosen APIs are representative of diverse use cases, ensuring a holistic understanding of the domain. For example, libraries such as *Matplotlib*, *Seaborn*, and *Plotly* provide an extensive array of APIs dedicated to plotting functionalities. On the other hand, *Pandas* and *NumPy* are predominantly known for their data manipulation capabilities. Similarly, the *Sklearn*, *Keras*, and *Torch* libraries are renowned for their APIs that facilitate model building in ML. This

Table 3.2: Library API utilization overview

Library	Total API calls	Unique API calls
Pandas	61,949	393
Sklearn	27,314	571
Matplotlib	16,762	224
Numpy	12,355	273
Seaborn	6,531	64
Keras	6,136	239
LightGBM	3,100	34
XGBoost	2,321	37
Tensorflow	1,664	345
Plotly	1,482	98
Torch	1,449	188
Statsmodels	594	87
Total	141,657	2,553

structured approach in API selection ensured that our annotations captured the broad spectrum of functionalities inherent in the ML domain.

We developed an annotation tool designed to display the API together with its associated docstring. This feature was implemented to aid annotators in efficiently identifying and selecting the relevant machine learning operation. It is important to note that, to ensure accuracy, every API was reviewed by multiple annotators. To assess the consistency of these annotations, we calculated the inter-annotator agreement and obtained a Cohen's kappa coefficient [Coh60] of 0.80. This score indicates a substantial level of agreement among the annotators.

Data Preprocessing. Docstrings often contain special characters, additional reference texts, and example content that can make ML pattern recognition challenging. To tackle this, we use Natural Language Processing (NLP) methods to preprocess these strings. Note that, during the annotation process, the unmodified doc strings were presented to the annotators.

The steps involved are outlined as follows:

- **Data cleaning:** removal of LaTeX and markdown formatting strings. Further, we eliminate version numbers, statements indicating deprecation, URLs, punctuation, and other special characters to ensure a cleaner dataset.
- **Stop word removal:** commonly used words such as "the", "and", and "is" are removed.
- **Lemmatization:** applied to reduce words to their base form.

Model Training. We used the final dataset with a train-test split of 80%-20% to train and test a series of multi-label classification techniques. This ratio was selected as it represents a standard heuristic in machine learning literature, providing sufficient data to the model for learning robust patterns while reserving a large enough subset (20%) to ensure statistically significant evaluation metrics [GKK18]. The nature of our classification problem is multi-label because a single API can correspond to multiple ML Operations. For instance, the API `numpy.ndarray.reshape` can be simultaneously categorized under both "Data Preparation and Exploration" and "Feature Engineering".

Table 3.3: Performance of classifiers for different text vectorization techniques

A: Accuracy, **P:** Precision, **R:** Recall

Classifier	**TF-IDF**			**Word2Vec**			**CountVec**		
	A.	P.	R.	A.	P.	R.	A.	P.	R.
LogisticRegression	0.86	1.0	0.86	0.60	0.95	0.59	0.89	0.98	0.89
RandomForest	0.84	0.99	0.86	0.80	0.98	0.80	0.88	0.99	0.88
DecisionTree	0.76	0.89	0.82	0.57	0.74	0.68	0.69	0.81	0.88
GaussianNB	0.83	0.92	0.86	0.72	0.81	0.83	0.75	0.84	0.88
SVM	**0.94**	**1.0**	**0.94**	0.83	0.89	0.91	0.77	0.93	0.80
GradientBoosting	0.81	0.96	0.82	0.79	0.92	0.81	0.85	0.95	0.89

As ML Models operate on numerical data, we first convert docstrings into a numerical format, in a process termed as text vectorization. We evaluated three prominent text vectorization methodologies: Word2Vec, TF-IDF, and CountVectorizer. With data from these techniques, we subsequently trained a range of classical ML models listed as follows: Logistic Regression, Random Forest, Decision Tree, GaussianNB, Support Vector Machines (SVM), and Gradient Boosting. Note that, driven by the constraints posed by our small dataset size, we abstained from incorporating neural networks in our experimentation.

Model Selection. The performance of various classifier models is shown in Table 3.3. In comparison with other classifiers and across vectorization methods, SVM stands out particularly with TF-IDF, showcasing a balance of accuracy, precision, and recall. Therefore, within HeaderGen we integrated the SVM classifier with TF-IDF vectorization for classifying API calls.

Pattern Matching

Notebooks may contain code cells that implement machine-learning operations without relying on explicit function calls. Instead, these cells often employ Python constructs that transform objects. In such cases, and in the absence of a function call, HeaderGen applies AST-based pattern matching to identify machine learning operations.

The process is as follows: Initially, HeaderGen identifies code cells devoid of any function calls by analyzing the flow-sensitive CG constructed in the preceding phase of the analysis. Subsequently, it traverses the AST of these code cells to detect the presence of specific patterns outlined in Table 3.4. However, given that ASTs lack type information, HeaderGen consults the EAG using line number and node identifier from the AST node, `Name`, to retrieve type information for AST elements identified within the code cell. Finally, if a pattern aligns with one of the supported patterns by HeaderGen, the corresponding code cell is annotated accordingly.

For illustration, consider the first pattern in Table 3.4, which captures a *feature engineering* operation of the form `df['xy'] = df.x * df.y`. This assignment introduces a new column `xy` into the DataFrame `df` by multiplying the existing columns `x` and `y`. HeaderGen inspects the `BinOp` AST node representing the operator `*` and checks whether both operands correspond to DataFrame attribute accesses. The AST alone does not provide sufficient information to determine the type of `df`, therefore HeaderGen retrieves the type of `df` from the EAG. Once confirmed as a DataFrame access, the statement is classified as a feature engineering operation. Table 3.4 lists the DataFrame access patterns currently supported by HeaderGen.

Index of ML Operations

- **Imported Libraries**
- ~~Visualization~~ (no calls found)
- Data Preparation
  - **Exploratory Data Analysis**
  - ~~Data Cleaning Filtering~~ (no calls found)
  - **Data Sub-sampling and Train-test Splitting**
- Feature Engineering
  - **Feature Transformation**
  - **Feature Selection**
- Model Building and Training
  - **Model Training** (A)
    - View All "Model Training" Calls
    - Cell # 4
    - Cell # 5 (B)
      goto cell # 5
      - **keras**
        - keras.engine.sequential.Sequential | **(See Args)** (C)
        - keras.layers.core.dense.Dense | **(See Args)**
        - keras.layers.core.activation.Activation | **(See Args)**
        - keras.engine.training.Model.compile | **(See Args)**
          - **Args:** [] | **Kwargs:** {'loss': 'categorical_crossentropy', 'optimizer': 'adam'}

(D)
```
Configures the model for training.
```

(a) Index of ML operations for our motivational example. Ⓐ ML operation category "Model Training" is expanded to view all code cells that are performing model training operations. Ⓑ Cell # 5 is expanded to view all function calls in the cell. Ⓒ Fully qualified function names are displayed. Ⓓ Expanded view showing the arguments used and their docstring.

3. Data Preparation

back to top

View function calls

- **sklearn**
  - sklearn.model_selection._split.train_test_split

```
# Cell 3
train_X, test_X, train_y, test_y = \
                              model_selection.train_test_split(X, y)
```

(b) Annotated version of cell # 3 in our motivational example.

Figure 3.6: Snapshots of the output notebook generated by HeaderGen for the motivational example.

Table 3.4: Mapping of Dataframe usage patterns to ML operations

ID	Pattern	ML Operation
1	`df['xy'] = df.x * df.y`	Feature Engineering
2	`df.x = 1`	Feature Transformation Data Preparation
3	`df.x[df.x == 1] = 1`	Feature Transformation Data Preparation
4	`x = df.x[['f1', 'f2']]`	Feature Selection
5	`print(df[0:20])`	Exploratory Data Analysis
6	`for i in df:`	Data Preparation
7	`len(df)`	Exploratory Data Analysis
8	`df.shape`	Exploratory Data Analysis

Text Annotation Generation

Based on the classification and pattern matching information from previous phases, the following annotations are added to the notebook: (1) Index of ML Operations, (2) Code cell headers, and (3) Table of contents.

1) Index of ML Operations: The index provides a clickable and nested list of all function calls in the notebook classified according to the taxonomy of ML operations shown in Figure 3.1. Figure 3.6a shows the index of ML operations generated for our motivating example. The index is displayed on top of the notebook using HeaderGen's notebook plugin. If no functions are found for a particular ML operation category, the category is displayed struck out. Each ML operation category and cell list can be expanded or collapsed as required. Function calls are organized based on the library, as seen in the figure. Additionally, different areas of the notebook are hyperlinked, which makes it easy for the user to explore the notebook back-and-forth. For instance, cell 5 can be quickly visited by pressing *"goto cell # 5"* and back to the index again by pressing *"back to top"*.

2) Code cell headers: High-level ML operation categories from the taxonomy are added as headers for individual code cells. Note that when code cells contain ML operations from more than one category, all of these are added to the header. Figure 3.6b shows the annotated version of code cell 3 from our motivating example (Figure 3.2). The headers can be further extended to see all the functions used in the following code cell, along with the docstrings that were fetched during analysis time from the source code.

3) Table of contents: Code cell headers are attached with anchors that allow in-page navigation. Using this information, the table of contents combines the headers of all code cells and adds an anchor-link to each entry. This simplifies access to relevant sections of the notebook based on the taxonomy.

3.3 Evaluation

The evaluation of HeaderGen focuses on four research questions designed to assess its static analysis accuracy, classification capability, and support for comprehension and navigation:

RQ1: *Does HeaderGen improve comprehension and navigation of undocumented Jupyter Notebooks?*

RQ2: *How accurate is HeaderGen's callsite recognition?*

RQ3: *How accurately can HeaderGen classify code cells using callsites?*

RQ4: *How does HeaderGen compare to other tools?*

First, in Section 3.4, we introduce the benchmarks constructed to assess HeaderGen's analytical performance in both controlled and real-world settings. Then, in the following Sections 3.5 to 3.8, we present the research questions and the corresponding evaluation results.

3.4 Benchmarks

A rigorous evaluation of HeaderGen requires datasets that represent both the structural organization of Jupyter notebooks and the semantic characteristics of Python code embedded within them. Jupyter notebooks differ from conventional Python modules by allowing interleaved, stateful execution, which amplifies challenges related to variable scope and callsite resolution. Therefore, the selected benchmarks must not only capture the cell-oriented execution model of notebooks but also include a diverse range of Python language constructs such as nested function definitions, class hierarchies, decorators, comprehensions, higher-order functions, and dynamic features.

Existing Python call-graph benchmark accompanying PyCG [SSL+21a] primarily focus on flow-insensitive, file-level analyses and are thus insufficient for evaluating tools that operate in notebook environments. To ensure analytical validity, three benchmarks are developed for this study:

1. Micro-benchmark containing 121 notebooks,
2. Real-world benchmark containing 15 notebooks from Kaggle, and
3. Extended real-world benchmark containing 15 notebooks from the wild.

Each benchmark captures a complementary aspect of HeaderGen's evaluation requirements: the micro-benchmark emphasizes syntactic coverage and flow sensitivity, the real-world benchmark targets undocumented machine learning notebooks, and the extended benchmark evaluates generalizability using diverse notebooks annotated by domain experts. The following subsections describe each benchmark in detail.

3.4.1 Jupyter Notebook Micro-benchmark

We evaluate HeaderGen by adopting the call-graph benchmark created by Salis et al. [SSL+21a] as part of PyCG. While the original benchmark evaluates flow-insensitive call-graph construction, HeaderGen requires flow-sensitive callsite information, including precise mappings between function calls and their line numbers within notebook cells. To enable such evaluation, the Python scripts from PyCG's benchmark were converted into notebooks, and the ground truth was manually created by associating callsites with line numbers. Furthermore, eight new test cases were introduced to explicitly assess flow-sensitive scenarios.

Our micro-benchmark consists of 121 minimal and focused test cases categorized into distinct Python language features. The micro-benchmark used in this study includes 121 test cases in the following 17 categories: *Arguments, Assignments, Builtins, Classes, Decorators, Dicts, Direct Calls, Flow Sensitive, Exceptions, Functions, Generators, Imports, Kwargs, Lambdas, Lists, Mro, Returns.*

3.4.2 Real-world Benchmark

To assess HeaderGen in real-world usage contexts, a real-world benchmark comprising 15 undocumented notebooks from Kaggle was constructed. Kaggle hosts numerous machine learning competitions where data scientists publicly share their notebooks. Many of these notebooks, despite being widely viewed and upvoted, lack documentation in the form of markdown cells. We found that 99 of the top 500 notebooks submitted to the most popular competition on Kaggle contained no markdown cell. Therefore, we base our real-world benchmark on these undocumented notebooks containing no markdown cells but are still being viewed by many (cf. Table 3.5).

To encourage variation in the benchmark, we selected notebooks from three different and most popular competitions on Kaggle based on the number of submissions: (1) Titanic - Machine Learning from Disaster, (2) Predict Future Sales, and (3) Santander Customer Transaction Prediction. We downloaded the top 30 notebooks according to votes for each competition with the search term "Keras", since Keras[4] is a popular ML library among novices. We used the Kaggle API to search and download notebooks. All 30 notebooks from each competition were further filtered to target those without any markdown cells. Finally, we selected the top five most-viewed notebooks from each competition. The selected notebooks in our benchmark are listed in Table 3.9. These notebooks have a median of 20 code cells, compared to 13 cells that are found in real-world notebooks as reported by Pimentel et al. [PMBF19]. Note that these undocumented notebooks still have 248 upvotes and 19,518 views as of October 2023.

The callsites ground truth was created manually by inspecting code cells in each notebook and listing the fully qualified names of all function calls. Notebooks were executed cell-by-cell and dynamically analyzed using Python's reflection module `inspect` to gather the fully qualified names. Multiple iterations were carried out to avoid errors in the ground truth.

Further, the ground truth for headers was created using experts. 15 notebooks from the benchmark were divided and assigned to four data scientists working in the industry for manual annotation of each code cell. Notebooks were distributed such that each notebook was seen by at least two reviewers. Based on the taxonomy of ML operations, each annotator inspected and classified each code cell into relevant categories. The inter-rater reliability score, as measured by Cohen's kappa coefficient [Coh60], was improved by conducting follow-up interviews with all four reviewers. Finally, a score of 0.89 was achieved, which signals an almost perfect agreement.

3.4.3 Extended Real-world Benchmark

Building upon our initial publication [SVWLB23b], we extended our benchmark suite by incorporating an additional 15 Jupyter notebooks. This extension aims to further assess the utility of HeaderGen across a broader spectrum of ML notebooks found in real-world contexts.

The annotation of notebooks with metadata describing call sites and headers is a resource-intensive task. To mitigate this challenge, the DASWOW dataset introduced by Ramasamy et al. [RSBB22] was utilized as a foundational source. The dataset comprises 470 notebooks annotated by domain experts, although its classification framework differs from the taxonomy employed by HeaderGen.

Since accurate evaluation of HeaderGen requires ground truth for both callsites and headers, a subset of notebooks from DASWOW was selected to balance analytical depth with feasibility. The selection process first filtered notebooks lacking markdown cells, resulting in 124 candidates. From these, the 15 notebooks containing the highest number of code cells were retained, ensuring diversity and complexity in code structure. The resulting collection exhibits a median of 27 code cells.

[4] `https://keras.io/`

Table 3.5: Details of notebooks included in the real-world benchmark evaluation

ID	Name	Votes	Views
1	bulentsiyah/keras-deep-learning-to-solve-titanic	69	2,244
2	hongdnghuy/relu-sigmoid	13	743
3	vaidicjain/titanic-easy-deeplearning-acc-78	9	489
4	tanvikurade/complete-analysis-of-titanic	19	326
5	alexanderbader/mytitanic	10	113
6	econdata/predicting-future-sales-with-lstm	7	3,363
7	lhavanya/predict-future-sales	3	500
8	elvinagammed/stacked-lstm-top-5-4-mae	9	523
9	ashishkapasiya/prediction-future-sales-with-keras	4	525
10	the0electronic0guy/keras-begineer-friendly	12	290
11	higepon/starter-keras-simple-nn-kfold-cv	20	4,387
12	vishesh17/keras-nn-with-scaling-and-regularization	32	3,205
13	christofhenkel/nn-with-magic-augmentation	21	1,573
14	naivelamb/multibranch-nn-baseline-magic	10	652
15	miklgr500/nn-embedding	10	585
	Total	**248**	**19,518**

The creation of ground truth for headers involved aligning the DASWOW taxonomy with our own, leveraging the partial mappings provided by the authors in their work [RSBB22]. This was followed by a manual verification of header annotations by the first author, to ensure accuracy and to add any missing labels. Furthermore, the callsites for notebooks were determined through a manual review process, adhering to the methodology described in the preceding Section 3.4.2.

3.5 RQ1: Comprehension and Navigation Study

The goal of HeaderGen is to enhance comprehension and navigation in undocumented notebooks. To quantitatively assess these improvements, a user study was conducted comparing the performance of participants when using HeaderGen against undocumented notebooks.

3.5.1 Study Design

The study is aimed at recreating the exploration of notebooks that ML practitioners routinely do. The study is designed as a within-subject study where the participants were given two notebooks from our real-world benchmark and asked to complete five comprehension tasks on each notebook, one after the other. To minimize learning effects, we chose a Latin-square design: Participants were divided into two groups. While participants in group-1 were given the undocumented notebook first, followed by the HeaderGen annotated version, participants in group-2 saw the annotated notebook first. Each study was conducted in a one-on-one online session lasting about one hour using a video-conferencing tool. First, an overview of the study-protocol was presented to the participant, including a walk-through of HeaderGen. Next, participants were provided access to the remote Jupyter instance along with a questionnaire

Table 3.6: Comprehension tasks

Id	Question
Q1	What are the deep learning layers used in the model?
Q2	What are the different data cleaning & data preparation operations?
Q3	Which of the following cells are used for model building and model training?
Q4	Select ML and visualization libraries that are used in the notebook
Q5	What are the different visualizations used in the notebook?
Q6	How is the dataset split into test and train subsets?

Table 3.7: Statements concerning the perception of usefulness

Id	Statement
S1	The classification of cells according to ML phases and headers helped me navigate the undocumented notebook.
S2	The generated list of functions used in the notebook helped me understand the notebook better.
S3	The header annotations added to the notebook are rather hindering the understanding of the notebook.
S4	I would install HeaderGen if it is made available as a plugin.

containing step-wise instructions on how to proceed. Before proceeding to the study, participants were instructed to examine an example notebook annotated with HeaderGen in order to get them comfortable with the features. The entire session was recorded with the consent of the participant for further analysis. Upon completion of the comprehension tasks, participants were asked to fill a Likert-scale questionnaire to understand the participants' perception of improvements provided by HeaderGen. Finally, participants were asked if they had any general comments about the tool.

Comprehension Tasks. A set of tasks was developed to reflect common exploratory questions that arise when data scientists inspect unfamiliar notebooks. The tasks were finalized after discussions with a data-science expert. For each task, participants were expected to select the right answers from all the choices given to them. Overall, six comprehension tasks were created, as listed in Table 3.6. For each notebook given to the participant, five tasks from the table were assigned to them based on the relevance to the notebook.

Likert-scale Questionnaire. Following the completion of the session, participants were asked to rate the level of agreement with statements about the usefulness of HeaderGen. The level of agreement was based on a 5-point Likert scale, where "1" is *Strongly disagree* and "5" is *Strongly agree*. The statements given to the participants are listed in Table 3.7.

3.5.2 Participants

The study involved eight participants: three master's students in computer science, three professional data scientists, and two academic researchers. Students were recruited through faculty contacts in the data-science research department, professionals were identified via LinkedIn[5]

[5] `https://www.linkedin.com/`

based on their job titles, and researchers were selected based on their publications in relevant areas. Participation was voluntary and conducted without monetary incentives. Due to privacy considerations, no identifying information about participants is disclosed.

3.5.3 Metrics

(1) Time: The total time taken to complete all five tasks per notebook.

(2) Accuracy: Inspired by a similar comprehension study by Adeli et al. [ANC+20], the accuracy is measured using the F1-score that takes into account both precision and recall.

(3) Navigability: The perceived navigability based on responses to Likert scale questions.

(4) Usefulness: The perceived usefulness is based on responses to Likert scale questions.

3.5.4 Results

The study yielded 80 accuracy measurements (8 participants × 5 tasks × 2 treatments) and 16 time measurements (8 participants × 2 treatments). We compare accuracy and time measurements between treatments using the non-parametric two-sided Wilcoxon Signed Rank (WSR) test, as the measurements between treatments are paired and the sample size is small. In addition, all measurements are analyzed based on descriptive statistics. Figure 3.7 shows the box-plot of accuracy scores, time measurements, and perception ratings.

Time. Both the mean and median completion times were lower for annotated notebooks (*mean = 336.6s, median = 328.5s*) than for undocumented ones (*mean = 486.4s, median = 464.5s*). The WSR test confirmed this difference as statistically significant (*p-value = 0.025, statistic = 34.0*). The large difference in completion time for the undocumented variant is associated with the back-and-forth navigation in the notebook, trying to find relevant areas. These findings indicate that participants completed comprehension tasks significantly faster when using HeaderGen.

Accuracy. The mean accuracy was higher for annotated notebooks across all tasks except T6, where performance was equal. Furthermore, accuracy variance across tasks was approximately three times higher for undocumented notebooks, suggesting greater consistency when using HeaderGen. Median accuracy was higher for the annotated treatment in tasks T4 and T5. However, the WSR test did not reveal statistically significant differences between the two conditions (*p-value = 0.106, statistic = 55.0*). Notably, the study was not time-limited, allowing participants to spend more time achieving correct answers in the undocumented condition, which likely mitigated accuracy differences.

Navigability and Usefulness. Responses to the statements in Table 3.7 indicated that participants found HeaderGen substantially beneficial for navigation and comprehension. None of the participants disagreed with statements S1, S2, or S4, and none agreed with the negative statement S3. All participants expressed willingness to install HeaderGen if released as a plugin.

Qualitative Results. Participants noted that HeaderGen would be especially useful when dealing with very large, undocumented notebooks, as it provides a "map" of the notebook. Participants also found the function documentation to be useful, given that the libraries are continuously evolving and that they would often come across methods that they have not seen before. Furthermore, minor recommendations to improve the taxonomy categories were noted and added to the final version.

Threats to Validity. The study we conducted is prone to some common limitations of conducting user studies. Due to the small number of participants, it may not be representative of

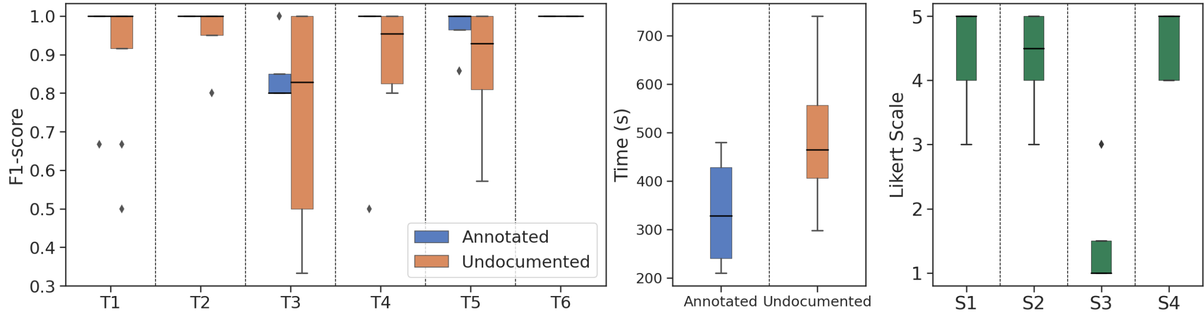


Figure 3.7: **Left:** Box plots of accuracy for participant responses grouped by treatment. **Center:** Box plots of time measurements for two treatments. **Right:** Box plots of responses to likert-questions about perception.

a larger population. However, participants were selected from all fields: students, professionals, and academics, to get inputs from different perspectives. Furthermore, since the study follows a within-subject design, the order of tasks and treatments can have an effect on the outcome. Therefore, to limit the learning effect, we use latin-square design to randomize the order of treatments, tasks, and multiple choices. Although the participants were experienced working with the default notebook environment, HEADERGEN adds additional interfaces that might seem confusing at first. As a result, some participants did not make full use of HEADERGEN's capabilities.

3.6 RQ2: Accuracy of Callsite Recognition

Micro-benchmark Results. The accuracy of HEADERGEN was first evaluated using the micro-benchmark to assess the completeness and soundness of its callsite recognition. Table 3.8 summarizes the results. The analysis is considered complete when no false positives occur and sound when no false negatives are present. Across 121 test cases, HEADERGEN achieved soundness and completeness in 113 cases each.

The lack of soundness in eight cases arose primarily from unimplemented handling of certain challenging Python constructs, particularly decorators. Of the eight incomplete cases, three were similarly due to unsupported language features, whereas the remaining five resulted from the analysis being context-insensitive, leading to over-approximation in specific scenarios.

Note that we do not perform a direct comparison of HEADERGEN with PyCG because the micro-benchmark does not pose specific challenges to flow-sensitivity, except for the new *flow_sensitive* category with eight test cases that we added. When manually compared to PyCG for the *flow_sensitive* category, as expected, PyCG is incomplete for all eight test cases. Furthermore, note that PyCG does not output line numbers in its analysis, and therefore, a direct automated comparison is not possible.

Real-world Benchmark Results. Table 3.9 lists the precision and recall values of HEADERGEN for real-world notebooks. HEADERGEN achieves an average of 95.6% precision and 95.3% recall. Note that in four instances, the analysis achieves 100% precision and recall.

Losses in precision are primarily attributable to over-approximations in the type-stub database used to infer return types. For example, a call such as `x.isnull()` can refer to either `Series.isnull` or `DataFrame.isnull`, depending on the type of `x`. Since this distinction requires advanced data-flow reasoning that goes beyond current capabilities, the analysis may produce multiple valid interpretations. Conversely, reductions in recall occur when HEADERGEN encounters complex Python constructs not yet supported by the current implementation.

Table 3.8: Micro-benchmark results for completeness and soundness of callsite recognition in HeaderGen.

Category	Complete	Sound
args	6/6	6/6
assignments	3/4	3/4
builtins	3/3	1/3
classes	21/22	22/22
decorators	6/7	5/7
dicts	10/12	11/12
direct_calls	4/4	4/4
exceptions	3/3	3/3
flow_sensitive	**8/8**	**8/8**
functions	5/5	5/5
generators	6/6	6/6
imports	14/14	14/14
kwargs	2/3	3/3
lambdas	4/5	5/5
lists	7/8	7/8
mro	7/7	6/7
returns	4/4	4/4
Total	**113/121**	**113/121**

Extended Real-world Benchmark Results. The evaluation using the extended real-world benchmark produced comparable outcomes, as summarized in Table 3.10. Across 15 additional notebooks, HeaderGen achieved an average precision of 94.4% and recall of 91.6%, with two notebooks reaching 100% precision and recall. These results confirm that the accuracy achieved in the primary real-world benchmark generalizes well to notebooks from different sources.

A notable exception occurs in notebook *nb_111547*, where recall drops to 66.7%. This reduction is attributed to extensive use of the `DataFrame.apply` and `Series.apply` methods, which pass function references as arguments to operate across data structures. Although HeaderGen can partially analyze these calls, the precise behavior of such methods depends on the runtime data types within the DataFrame, information that is not statically available. This case illustrates a known limitation of the approach, namely the difficulty of resolving callsites whose semantics depend on dynamically determined data structures based on input data.

3.7 RQ3: Accuracy of Generated Headers

HeaderGen uses identified function calls within code cells to automatically generate descriptive headers, which are categorized according to the taxonomy of machine learning operations. To evaluate the accuracy of these generated headers, precision and recall are measured against manually annotated ground truth in the real-world and extended real-world benchmarks.

Results. The precision and recall values are presented on the right-hand side of Table 3.9 and Table 3.10. Each generated header was evaluated based on its correspondence to the high-level categories of the taxonomy illustrated in Figure 3.1. Across the real-world benchmark, HeaderGen achieved a precision of 85.7% and a recall of 92.8%. For the extended real-world benchmark, results were comparable, with a precision of 84.8% and a recall of 91.2%. The

Table 3.9: Real-world benchmark evaluation for callsite recognition and header annotation

Id	Callsite Recognition		Header Annotation	
	Precision	**Recall**	**Precision**	**Recall**
1	100	90.0	71.4	100
2	100	100	84.2	100
3	95.8	95.8	100	87.5
4	100	100	75.8	95.9
5	93.8	97.4	83.3	90.9
6	86.0	97.4	100	100
7	94.4	91.9	83.3	88.2
8	100	100	94.1	76.2
9	90.9	97.2	83.6	92.7
10	100	100	85.0	90.7
11	97.1	85.0	68.8	100
12	96.0	96.0	94.7	94.7
13	94.3	94.4	100	100
14	94.4	93.5	74.1	87.0
15	91.3	91.3	87.5	87.5
Average	**95.6**	**95.3**	**85.7**	**92.8**

Table 3.10: Extended benchmark evaluation for callsite recognition and header annotation

Id	Callsite Recognition		Header Annotation	
	Precision	**Recall**	**Precision**	**Recall**
nb_111547	95.4	**66.7**	82.5	91.7
nb_12952	98.1	98.7	73.4	100
nb_13973	100	100	92.0	76.7
nb_2004	88.9	94.1	95.8	74.2
nb_24325	97.2	100	95.5	91.3
nb_27915	92.0	95.8	74.5	100
nb_44070	85.7	88.9	81.8	93.1
nb_50450	89.2	87.9	79.0	89.1
nb_5433	97.7	80.8	92.5	100
nb_6005	84.4	85.7	56.4	91.7
nb_62235	95.8	90.7	93.9	91.2
nb_62337	100	100	100	97.5
nb_79900	97.2	89.7	92.9	96.3
nb_87231	99.1	99.1	85.7	81.1
nb_96079	95.6	95.6	76.3	93.5
Average	**94.4**	**91.6**	**84.8**	**91.2**

Table 3.11: Comparison with existing tools on our real-world benchmark

Tool	Callsite Recognition		Header Annotation	
	Precision	**Recall**	**Precision**	**Recall**
HeaderGen	96.4	95.9	82.2	96.8
Pyright	96.7	87.2	83.8	82.7
Jedi	84.6	65.8	85.1	69.8
PyCG	41.7	23.3	84.6	26.2

similarity in performance across both datasets indicates that the header generation process generalizes effectively to diverse notebooks. Losses in precision are primarily attributed to APIs that correspond to multiple machine learning operations within the taxonomy, leading to overlapping classifications.

Impact of Pattern Matching. To examine the influence of pattern matching on header annotation accuracy, HeaderGen was executed with the pattern matching feature disabled, and the resulting performance was compared to the default configuration. Results show that enabling pattern matching substantially increases recall by 10.6% for the real-world benchmark and 14.8% for the extended real-world benchmark. However, this improvement comes at the cost of a reduction in precision, as broader pattern associations occasionally over-approximate matches. Specifically, precision decreased by 2.3% for the real-world benchmark and 4.6% for the extended real-world benchmark. These results suggest that pattern matching enhances HeaderGen's ability to identify relevant operations, but with a trade-off in precision.

3.8 RQ4: Comparison with Existing Tools

We compare HeaderGen in terms of callsite recognition and header annotation with *PyCG*, *pyright*, and *Jedi* using our real-world benchmark. Since *pyright* is designed for type checking and *Jedi* for auto-completion, we added helper functions to output type information and callsite information as required by HeaderGen. In addition, to ensure a fair comparison, both *pyright* and *Jedi* were supplied with the same type-stub database of machine learning libraries used by HeaderGen.

Results. The comparative precision and recall results are summarized in Table 3.11. Because header annotation in HeaderGen depends on the underlying callsite recognition, tools with higher recall in callsite detection correspondingly achieve higher recall in header annotation. HeaderGen achieves the highest recall of 95.9%, which leads to a 96.8% recall in header annotation of code cells. However, *pyright* is close with 87.2% recall for callsite recognition, which leads to 82.7% recall for header annotation.

The loss of precision is attributed to the over-approximation of return-types in our type-stub database, as discussed earlier. Nevertheless, the results indicate that HeaderGen offers the most balanced trade-off between precision and recall while maintaining strong consistency across both tasks.

Modeling of Pandas Behavior. To further understand the comparative differences between the tools, we examined their handling of data manipulation operations in the Pandas library, which is widely used in machine learning workflows. Listing 3.1 presents representative code patterns drawn from the real-world benchmark, and Table 3.12 summarizes the inferred types for each variable.

The results show that both *pyright* and *Jedi* fail to infer the correct return types for variables `x1` through `x6`. In contrast, HeaderGen accurately identifies all corresponding types

```python
import pandas as pd

df = pd.read_csv("./input.csv")
x1 = df["a"].map(lambda x: x + 1.0)
x2 = df.iloc[[False]].reset_index().copy()
x3 = df.a.fillna(0)
x4 = df.groupby(["a"])[["b"]].agg({"b": ["min"]})
x5 = df[["b", "c"]]
x6 = df.c.values.astype(int)
```

Listing 3.1: Common uses of Pandas DataFrame that existing tools fail to infer.

Table 3.12: Comparison of type inference by existing tools for listing 3.1

Var	Actual	HeaderGen	Pyright	Jedi
df	DataFrame	DataFrame	DataFrame	DataFrame
x1	Series	Series	Any	Any
x2	DataFrame	DataFrame	Any	Any
x3	Series	Series	Any	Any
x4	DataFrame	DataFrame	Any	Any
x5	DataFrame	DataFrame	Any	Any
x6	Ndarray	Ndarray	Any	Any

by modeling Pandas' internal data access patterns. For example, the dot notation access `df.a` (line 6) is ignored by the other tools but correctly interpreted by HEADERGEN as a `Series`. This capability enables HEADERGEN to reason about complex chained operations that are common in data preprocessing pipelines.

3.9 Related Work

Tool-support for Jupyter Notebooks. A considerable body of research has examined coding practices and software quality in Jupyter notebooks [KRA+18, RTH18, PMBF19, KES20, WLZ20, EWDD22, QCL22, GTS+22]. These studies consistently indicate a deficiency in the quality of notebooks, signaling a need for greater attention from the software engineering community. Despite these findings, there is a noticeable gap in research efforts focusing on tools that can address the identified issues.

In a step towards addressing this gap, Wang et al. [WWD+22] introduced *Themisto*. This tool prompts data scientists to document their code cells. It employs a deep learning method to auto-generate code documentation in natural language and then suggests to users whether to integrate or directly apply this documentation. Notably, *Themisto* relies on the Abstract Syntax Tree (AST) of Python code for model training, without incorporating static analysis methods to extract more contextual information from the code. We posit that the analytical capabilities of HEADERGEN might enhance deep learning approaches, potentially yielding better outcomes.

In another study, Pimentel et al. [PMBF19] studied 1.4 million notebooks for features that affect reproducibility and suggested a set of best practices. Following this, Wang et al. [WKLZ20a]

propose *Osiris*, a tool-based approach to restore reproducibility in notebooks by using AST parsing for data-flow analysis to find dependencies of variables between code cells. Furthermore, Yang et al. [YBLK22] design a static analysis approach to detect data leakage in notebooks. In contrast, our contribution to this domain focuses on the automatic annotation of code cells, offering tool-support for literal programming.

Static Analysis for Python. Python has grown to be one of the leading programming languages, but the field still has a significant lack of static analysis tools [YMH22a]. Yang et al. [YMH22a] emphasize that Python's distinct characteristics make it difficult to use traditional analysis methods developed from existing scientific research. One reason for this is Python's dynamic features, like duck typing, which while beneficial for rapid prototyping, complicate its analysis.

A major advancement in Python static analysis came with the introduction of *PyCG* [SSL+21a], a method for constructing call graphs. Yet, this method does not account for the flow of values and is not tailored for Jupyter notebooks. Furthermore, Python still lacks a comprehensive static analysis framework to produce data flow intermediate representations. The most related work in this area is the Scalpel project [LWQ22], but it too falls short, particularly in inferring return types for external functions and considering notebook cells.

In addition, Monat et al. [MOM20a] delve into using static abstract interpretation to aid Python type analysis. This approach covers a broad range of constructs and precisely combines domains, allowing sound knowledge of nominal and structural types and exceptions raised in the program. Building on this foundation, Monat et al. [MOM20b] developed *MOPSA*, a prototype tool that integrates value analysis. However, MOPSA focuses specifically on analyzing Python code that is combined with C code. In this work, HeaderGen contributes to this research line by providing practical methods for resolving return types of external API calls and extracting flow-sensitive callsites through def-use analysis across notebook cells. However, MOPSA has the potential to provide more reliable type stubs from C modules, which can benefit our work and other analyses. We suggest future research to study this more closely.

Code classification. Code classification is fundamental for various tasks, such as determining code authorship, identifying the programming language, and understanding source file content. While significant work exists in this domain, studies that specifically focus on Jupyter notebooks are limited. Several machine learning techniques have been instrumental in the progress of this field. Methods like Max-entropy [ZH17], decision tree [UKG02], K-nearest neighbor (KNN), and Naive Bayes [BGG14] are noteworthy. They are proficient in tasks like predicting the programming language of source code and identifying the topics within a document. Additionally, recent advancements in deep learning have introduced weakly supervised transformer-based architectures for the classification and tagging of source code, as highlighted by Zhang et al., [ZML+22].

Within the notebook domain, Ramasamy [RSBB22] introduced a supervised framework for classifying data-science-related code cells. Their work framed the problem as topic classification, supporting both single-label (one label per cell) and multi-label setups to accommodate cells containing multiple operations. However, there are inherent limitations with their supervised classification approach. As noted by the authors, the classification of notebook cells, particularly those associated with evaluation, prediction, and visualization tasks, can achieve lower F1-scores due to the skewed distribution of their training dataset, where certain labels are underrepresented. In contrast, our method bypasses the need for an extensive pre-training process. By leveraging API usage patterns to infer the functional intent of cells, HeaderGen provides a more lightweight and adaptable solution suited to diverse real-world notebook scenarios.

3.10 Limitations & Future work

To enhance the accuracy of function name resolution, we leverage Python's reflection mechanism. While this approach enables precise introspection, it also restricts API coverage, as the analysis is dependent on the specific library versions installed in the environment. Our current analysis predominantly focuses on machine learning applications. Nonetheless, the architecture of the framework we designed is not confined to this domain. HeaderGen is equipped to annotate notebooks, accommodating domain-specific return-type stubs and library classifications. Lastly, the present version of HeaderGen utilizes an ML taxonomy that comprises three main categories. Recent studies on notebooks have introduced a more detailed taxonomy, and future iterations of HeaderGen will adopt this refined classification.

The taxonomy classification model has been developed using a small dataset consisting of 400 function calls, which represents a constraint of our methodology. The process of curating a dataset is costly and requires the involvement of several experts. Future work should investigate the use of large language models to automate and extend the classification process.

Additionally, input from HeaderGen can be used to automatically restructure code cells in notebooks for better readability. It can achieve this by reorganizing complex code cells, particularly those encompassing multiple ML operations, into a sequential arrangement. Finally, the flow-sensitive and accurate callsite recognition capabilities of HeaderGen open avenues for large-scale empirical studies and mining of Python notebook code bases, facilitating new research in code comprehension, reuse, and evolution.

3.11 Conclusion

This chapter presented HeaderGen, a static analysis technique developed to improve the comprehension and navigability of machine learning notebooks through automatic annotation. By integrating flow-sensitive call-graph construction, return-type resolution, and semantic classification of notebook cells, HeaderGen combines the precision of program analysis with the interpretability required for real-world data science workflows. The approach addresses the dual challenge of reasoning about Python's dynamic semantics while restoring structural organization to undocumented notebooks.

The evaluation demonstrated that HeaderGen performs reliably across both controlled and real-world environments. It achieved high precision and recall in callsite recognition and header annotation, outperforming existing static analysis tools such as *PyCG*, *pyright*, and *Jedi*. The user study further confirmed that practitioners experienced measurable gains in comprehension and navigation when interacting with annotated notebooks, reinforcing the value of automated structural guidance in exploratory computational work.

Beyond its immediate contributions, HeaderGen highlights a broader methodological gap: the absence of standardized frameworks for evaluating the precision and recall of type inference systems that underpin such analyses. Type inference constitutes a fundamental component of static analysis for Python, as it enables the approximation of runtime behavior in a language that lacks explicit type declarations. Accurate type information is essential for resolving the targets of function calls, identifying dependencies, understanding object hierarchies, etc.

Consequently, type inference serves as the foundational layer upon which static reasoning about Python programs is built. Establishing standardized methodologies for evaluating its accuracy is therefore crucial, not only for assessing existing inference tools but also for ensuring the validity of derived analyses such as call-graph generation. The next chapter addresses this need through the introduction of TypeEvalPy, a benchmarking framework designed to systematically evaluate and compare type inference systems.

TYPE_EVALPY

4 Type Inference Benchmarking with TypeEvalPy

The development of type inference tools for Python has surged in recent years, driven by interest from both academic researchers and the open-source community. Existing work introduces a diverse range of techniques, varying from rule-based static analysis to advanced machine learning approaches. Despite this progress, the evaluation landscape remains fragmented. Research prototypes such as *Type4Py* [MLPG22a] and *HiTyper* [PGL+22] are typically evaluated on large-scale, real-world datasets, whereas open-source tools are often validated against narrow, tool-specific test suites. This lack of a controlled, reproducible benchmarking environment prevents a uniform comparison of state-of-the-art systems.

Current evaluation practices suffer from three primary methodological constraints:

1. **Inconsistent Datasets:** Studies frequently use different datasets, making direct comparisons of tool performance impossible [MLPG22a, PGL+22, HBBA18, ABDG20b].

2. **Unverified Ground Truth:** Large-scale real-world datasets are not human-verified. They often contain inaccurate labels or inconsistent annotations, thereby compromising the validity of the reported metrics [DGP22b].

3. **Lack of Granularity:** Evaluations rely on aggregate scores that mask performance on specific language features. For instance, a global accuracy score fails to reveal how a tool handles specific language features such as iterator types, decorators, or type propagation in higher-order functions.

To address these issues, this dissertation's second major contribution introduces TypeEvalPy, a micro-benchmark and evaluation framework that provides a controlled, feature-specific, and reproducible assessment of Python type inference systems. Leveraging this framework, we perform an extensive evaluation of six leading tools, providing the first granular comparison of how rule-based and learning-based approaches handle dynamic Python constructs.

Structure. The remainder of this chapter is organized as follows: Section 4.1 details the framework design and the construction of the micro-benchmark. Section 4.2 states the research question, while Section 4.3 introduces the six inference tools selected for the study. The empirical comparison is presented in Section 4.4, followed by a discussion of the findings in Section 4.5. We address the limitations in Section 4.6 and, finally, conclude the chapter in Section 4.7.

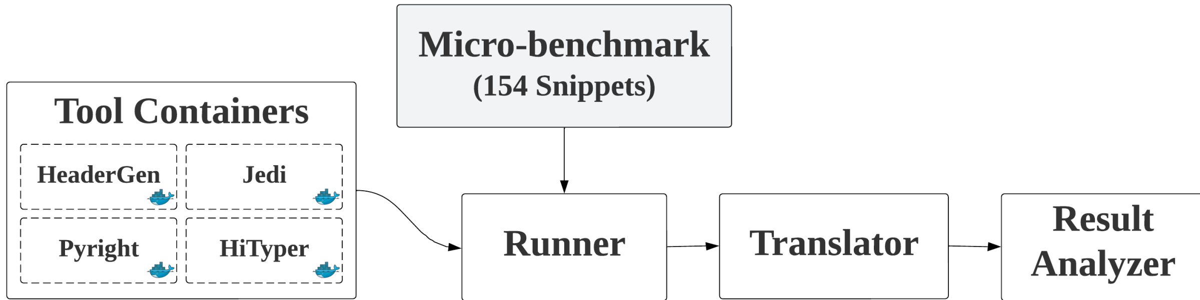


Figure 4.1: Overview of benchmarking workflow in TypeEvalPy

4.1 TypeEvalPy Framework

The TypeEvalPy framework establishes a uniform environment for executing type inference tools and normalizing their outputs. The benchmark consists of 154 manually curated snippets, each targeting a distinct Python construct with verified ground-truth annotations. Furthermore, the framework standardizes tool execution through containerized environments and uniform interfaces, ensuring that diverse inference systems can be analyzed under identical conditions and a standardized evaluation metric.

As illustrated in Figure 4.1, the workflow integrates four primary components:

1. **Micro-benchmark:** 154 manually curated snippets with ground-truth annotations.
2. **Runner:** A containerized execution engine.
3. **Translator:** An interface to normalize tool outputs.
4. **Result Analyzer:** A module for computing evaluation metrics.

The following sections describe each component and its role in the evaluation process.

4.1.1 Micro-benchmark

The core of the evaluation framework is the micro-benchmark, a collection of 154 code snippets designed to isolate and test specific Python language features. As outlined in Table 4.1, these snippets are organized into 18 categories adapted from the taxonomy established by the PyCG [SSL+21a] call-graph benchmark. Each snippet is accompanied by a manually curated set of ground-truth data, totaling 845 individual type annotations.

Schema for Ground-Truth Annotations. To ensure a tool-agnostic evaluation, TypeEvalPy normalizes ground-truth types into a uniform JSON-based schema. This schema encodes essential metadata for each annotation, including the target entity, precise program location (line number and column offset), and type category. To illustrate this structure, Listing 4.1 presents a sample Python snippet, while Listing 4.2 details its corresponding ground-truth representation under the TypeEvalPy schema.

The schema defines three distinct annotation categories:

- **Function Return Type (FRT):** Identifies the set of possible return types for a given function.
- **Function Parameter Type (FPT):** Enumerates all types passed to a specific parameter of a function across all the observed calling contexts.

```python
def id_func(arg):
    x = arg
    return x

result = id_func("String")
result = id_func(1)
```

Listing 4.1: simple_code.py


```json
[{
    "file": "simple_code.py",
    "line_number": 1,
    "col_offset": 1,
    "function": "id_func",
    "type": ["int", "str"]
},
{
    "file": "simple_code.py",
    "line_number": 1,
    "col_offset": 13,
    "function": "id_func",
    "parameter": "arg",
    "type": ["int", "str"]
},
{
    "file": "simple_code.py",
    "line_number": 2,
    "col_offset": 5,
    "function": "id_func",
    "type": ["int", "str"],
    "variable": "x"
},
{
    "file": "simple_code.py",
    "line_number": 5,
    "col_offset": 1,
    "type": ["str"],
    "variable": "result"
},
{
    "file": "simple_code.py",
    "line_number": 6,
    "col_offset": 1,
    "type": ["int"],
    "variable": "result"
}]
```


Listing 4.2: Annotations for simple_code.py

Figure 4.2: Example code snippet and associated ground-truth annotations

Table 4.1: Categories in the TypeEvalPy micro-benchmark

Category	Count	Description
args	8	Snippets involving positional arguments and argument unpacking.
assignments	8	Assignment patterns, including chained and multiple assignments.
builtins	7	Use of Python built-in functions and operations.
classes	26	Class definitions, attribute access, methods, and instantiation.
decorators	8	Decorator usage and its effect on function signatures.
dicts	15	Dictionary construction, updates, iteration, and key-value access.
direct_calls	6	Direct function invocation without intermediate indirection.
dynamic	3	Dynamic features such as runtime attribute creation or modification.
exceptions	2	Exception raising, handling, and propagation constructs.
external	7	Interactions with external library calls.
functions	9	Function definitions, parameter handling, and return behavior.
generators	6	Generator functions, `yield` semantics, and iteration.
imports	14	Import statements and module-level namespace resolution.
kwargs	4	Keyword arguments, default values, and dictionary-based unpacking.
lambdas	6	Lambda expressions and their use as first-class functions.
lists	10	List creation, indexing, slicing, and mutation.
mro	7	Method resolution order, inheritance hierarchies, and overriding.
returns	8	Return statements and multiple-return-type behavior.

- **Local Variable Type (LVT):** Describes types associated with variables defined within a function scope or at the module level.

The ground-truth annotation in Listing 4.2 illustrates these categories. The first entry defines the return type of `id_func` as the union `['int', 'str']`, matching the type assigned to the parameter `arg` in the second entry, aggregated from the two invocations (`"String"` and `1`). The remaining entries capture local variable behavior: the third annotation tracks the function-local variable `x`, also typed as `['int', 'str']`. Finally, the last two annotations document the module-level variable `result` at two distinct program points; notably, the type shifts from `['str']` at line 5 to `['int']` at line 6, capturing the reassignment of the variable.

Design Requirements.

- **Requirement 1: Capturing Dynamic Behavior.** Unlike static languages, where types are bound to variable declarations, Python variables are dynamic. Therefore, the ground truth must reflect the *dynamic* type a variable holds at specific execution points.
- **Requirement 2: Compatibility with Standard Practices.** The framework must evaluate tools ranging from strict static analyzers to probabilistic ML models. Since most existing tools align their outputs with PEP-484 [1] (Python Type Hints), the ground truth

[1] https://peps.python.org/pep-0484/

schema must be compatible with these standard practices (e.g., using `Union` types) rather than requiring complex representations that most tools cannot produce.

- **Requirement 3: Granular Verification.** To enable precise error analysis, the evaluation must decouple complex type structures. For instance, considering Python containers can have heterogeneous types, verifying a container's type should be distinct from verifying the types of its elements individually.

Design Considerations for Ground-Truth Construction. The construction of the ground truth is governed by a set of methodological principles designed to ensure consistency, clarity, and suitability for automated comparison. The benchmark adopts the following design choices:

- **Location-Specific Typing:** Annotations are strictly bound to specific program coordinates (line number and column offset). This granularity allows the benchmark to capture variable reassignments, recording distinct types for the same identifier as its value evolves during execution.
- **Context-Insensitive Aggregation:** The benchmark records the *union* of all observed types for function parameters, return values, and local variables. This captures the accumulated behavior across all calling contexts without distinguishing between individual call paths. While distinct call traces would offer higher precision, we aggregate these into standard PEP-484 `Union` types to ensure compatibility with the majority of static analysis and ML tools, which do not typically output context-sensitive types.
- **Generic Container Abstraction:** To mitigate ambiguity in complex nested structures, container annotations omit element type parameters. For instance, a list of integers is recorded simply as `List` rather than `List[int]`. Instead, the elements contained within are annotated individually at their respective program points.
- **Exclusion of the `Any` Type:** The benchmark avoids the use of `Any`. When a function returns multiple concrete types, all observed types are explicitly enumerated.
- **Explicit `NoneType`:** Functions lacking an explicit return statement are annotated with `NoneType` to represent the actual return value.

4.1.2 Runner

The `runner` component orchestrates the execution of type inference tools under controlled and reproducible conditions. By encapsulating the entire lifecycle of a tool's execution, the runner ensures consistent dependencies and uniform access to the benchmark.

Operationally, the `runner` manages the workflow for each tool: it builds the corresponding Docker image, configures the execution environment, and transfers the benchmark snippets into the container. The component then invokes the tool's analysis script and streams execution logs for diagnostic purposes. Once the inference completes, the `runner` extracts the raw results from the container and stores them in the designated host directory.

Following extraction, the `runner` coordinates the post-processing phase by invoking the `translator` to convert tool-specific outputs into the canonical TypeEvalPy schema. Once the data is normalized, the `runner` automatically triggers the `result analyzer`.

To accommodate the specific requirements of different tools, the framework employs specialized subclasses. These extensions allow for tool-specific configurations, such as adjusting runtime parameters, enabling GPU acceleration, or modifying the invocation command, while retaining full compatibility with the general execution protocol.

4.1.3 Translator

The `translator` component is responsible for standardizing the heterogeneous outputs produced by individual type inference tools. Since these tools differ substantially in how they report inferred types, program locations, and identifier names, normalization is a prerequisite for fair comparison. This component bridges the gap, ensuring that all tool outputs are uniformly ingested by the `result analyzer`.

To achieve this consistency, the translator performs three core normalization tasks:

1. **Entity Alignment:** It maps tool-specific identifiers to the corresponding entities defined in the benchmark ground truth. This step is essential to resolve naming discrepancies, such as when tools output paths that have a different naming convention or symbols.

2. **Type Normalization:** It converts native type representations into the TypeEvalPy schema (e.g., standardizing `None` to `NoneType`). This aligns inconsistent naming conventions across tools, ensuring that semantically equivalent types (e.g., `builtins.int` vs. `int`) are correctly matched.

3. **Filtering:** It discards predictions that cannot be matched to valid program entities or lack required metadata. This eliminates noise by excluding internal tool artifacts.

While translators of all supported tools in TypeEvalPy adhere to this common interface, their internal logic is tailored to the complexity of the target tool. For tools like *HiTyper*, which use distinct identifiers for functions and classes, the translator reconstructs fully qualified names to resolve ground-truth matches and sanitizes type strings by removing unnecessary module prefixes. Conversely, for tools like *Pyright*, which produce outputs structurally similar to the benchmark schema, the translator functions primarily as a validator, filtering incomplete entries and correcting minor formatting inconsistencies to ensure strict adherence to the schema.

4.1.4 Result Analyzer

The `result analyzer` quantifies the performance of each tool by comparing normalized predictions against the ground truth. The analysis relies on a deterministic matching procedure that aligns inferred facts with ground-truth entities based on file name, program location, and category.

Based on this alignment, the analyzer computes a comprehensive set of quantitative metrics:

- **Exact matches:** The proportion of inferred types that match the ground truth exactly.

- **Precision:** The ratio of correctly inferred types to the total number of types reported by the tool.

- **Recall:** The ratio of correctly inferred types to the total number of types present in the ground truth.

- **Soundness:** A measure of the tool's ability to identify all possible types specified in the code, ensuring no valid types are omitted.

- **Completeness:** A measure of the tool's ability to report *only* the types that are present, avoiding incorrect or extraneous predictions.

- **Top-n Accuracy:** For probabilistic (ML-based) tools, the frequency with which the correct type appears within the top-n predictions.

These metrics are computed at two levels of granularity. Category-level analysis aggregates metrics for specific Python features (e.g., decorators, generators), enabling a fine-grained inspection of a tool's strengths and weaknesses. Benchmark-level analysis consolidates results across all categories to provide high-level statistics for cross-tool comparison.

In addition to quantitative metrics, the analyzer generates actionable diagnostic reports. These reports explicitly enumerate missing types (valid types missed by the tool) and mismatched types (inferred types that contradict the ground truth), facilitating detailed failure analysis.

4.2 Research Question

To validate the utility of TypeEvalPy and assess the current state of the art, this section anchors its empirical evaluation to the following research question:

How do rule-based, learning-based, and hybrid type inference systems compare against each other when applied to Python constructs?

Answering this question requires a comparative analysis of tools that exemplify these distinct paradigms. Consequently, the following section details the selection of representative systems used to conduct this evaluation.

4.3 Tool Selection

To address the proposed research question, we selected six inference systems that represent the full spectrum of techniques currently employed in Python type analysis.

To ensure a comprehensive evaluation, the selection spans three distinct paradigms:

1. **Static Analysis:** Tools relying on rule-based control-flow and data-flow analysis (*HeaderGen* [SVWLB23b], *Jedi* [Hal22], *Pyright* [Sta22], *Scalpel* [LWQ22]).
2. **Machine Learning:** Data-driven approaches utilizing vector embeddings and deep learning (*Type4Py* [MLPG22a]).
3. **Hybrid Approaches:** Systems that combine rule-based static analysis with learning-based approaches (*HiTyper* [PGL+22]).

The following overview summarizes these approaches:

HeaderGen HeaderGen [SVWLB23b], introduced in Chapter 3, is a system originally developed to improve the readability of Jupyter notebooks. It serves as a representative example of type inference grounded in pure static analysis. Its analysis pipeline constructs a flow-sensitive assignment graph that tracks relationships between identifiers. Through fixed-point iteration, *HeaderGen* propagates type information to resolve variables, parameters, and return values.

Jedi *Jedi* [Hal22] is a widely used open-source library designed primarily for IDE code completion and navigation. It represents a heuristic approach to static inference. Unlike whole-program analyzers, *Jedi* employs a lazy evaluation model, resolving only the expressions required for the current context. Built upon the `parso`[2] parser, it applies name resolution across scopes and imports to infer possible values on demand.

[2] https://github.com/davidhalter/parso

Pyright *Pyright* [Sta22] is a high-performance static type checker maintained by Microsoft. It performs type inference within the constraints of the PEP 484 type system, supporting union types, generics, conditional narrowing, and a wide range of modern typing features. *Pyright* analyzes code by combining control-flow reasoning with structural type rules, enabling it to refine inferred types across branches and propagate annotations through assignment and call chains. Its architecture includes both a command-line checker and a language server, making it suitable for integration with development tools while maintaining high performance on large codebases.

Scalpel *Scalpel* [LWQ22] is a static analysis framework that provides foundational abstractions such as control-flow graphs, static single-assignment forms, and alias analysis. While primarily a framework for building custom analyzers, its built-in type inference module utilizes backward data-flow analysis to refine types across assignments and interprocedural edges. It serves as an example of academic static analysis frameworks adapted for type inference tasks.

Type4Py *Type4Py* [MLPG22a] represents the state-of-the-art in machine-learning-based type inference. It employs a deep similarity learning model to predict types for variables, parameters, and return values. The system projects code fragments and type representations into a shared vector space using a hierarchical neural architecture. Inference is then performed via nearest-neighbor search within this embedding space. Notably, *Type4Py* is trained on a type-checked corpus to reduce noise, and its ability to produce ranked lists of candidates allows for evaluation under top-n conditions.

HiTyper *HiTyper* [PGL+22] is a hybrid system that bridges rule-based static analysis and ML-based prediction. Its core abstraction is the Type Dependency Graph (TDG), which models relationships among program entities. *HiTyper* iterates between applying static propagation rules to the TDG and invoking a neural network model to suggest types where static constraints are insufficient. By employing rejection rules to filter inconsistent predictions, *HiTyper* leverages the complementary strengths of logical consistency (static) and pattern recognition (ML).

4.4 Results

We now present a systematic empirical evaluation of the Python type inference landscape. To isolate the impact of the machine-learning component, *HiTyper* is evaluated in two configurations: *HiTyper (Static):* The baseline configuration relying solely on rule-based propagation and *HiTyper-DL (Hybrid):* The full configuration augmenting static analysis with neural predictions from Type4Py.

The evaluation assesses the performance of type inference systems using three standard metrics: exact matches, soundness, and completeness. Additionally, for ML-based systems, we analyze the predictions under a top-n setting.

Static Analysis Tools

Table 4.2 details the performance of the static analysis tools across the 18 benchmark categories.

HeaderGen emerged as the top-performing tool overall, achieving 564 exact matches out of 845 annotations (66%). It demonstrated consistent performance across all annotation types, Function Returns (FRT), Function Parameters (FPT), and Local Variables (LVT), with particular strength in resolving local variable types.

Following this, *Jedi* and *Pyright* achieved 415 (49%) and 405 (47%) exact matches, respectively. While their aggregate performance is comparable, their specific strengths diverge. *Pyright* excels in the `builtins` category (45 matches vs. 21 for *Jedi*), reflecting its rigorous adherence to standard library typing stubs. In contrast, *Jedi* proves more effective in the `external` category (8 matches vs. 2 for *Pyright*), likely due to its heuristic-based approach to third-party library resolution. However, both tools exhibit a conservative design philosophy regarding function parameters. Unless explicit type annotations are available and matched, they default to `Any`, resulting in negligible accuracy for FPT annotations (0 for *Jedi* and 8 for *Pyright*).

Conversely, *Scalpel* achieved 193 exact matches (22%). This lower score reflects its narrower feature coverage compared to the industry-standard tools. *Scalpel* struggles significantly with categories involving dynamic constructs, higher-order functions, or complex inheritance, limiting its effectiveness as a general-purpose inference tool.

Table 4.2: Comparison of exact matches, sound, and complete values of type inference tools for micro-benchmark categories (HeaderGen, Jedi, Pyright, Scalpel). The benchmark comprises **154** test cases with **845** type annotations. Abbreviations: **LVT** (Local Variable Type), **FRT** (Function Return Type), **FPT** (Function Parameter Type).

Category	**HeaderGen**			**Jedi**			**Pyright**			**Scalpel**		
	FRT	FPT	LVT	FRT	FPT	LVT	FRT	FPT	LVT	FRT	FPT	LVT
args	17	9	12	12	0	9	8	1	8	8	7	0
assignments	15	1	33	20	0	21	20	0	25	20	2	1
builtins	0	0	26	0	0	**21**	1	0	**45**	0	0	0
classes	39	7	67	0	0	57	1	0	46	25	0	0
decorators	11	6	2	10	0	8	7	0	3	16	3	0
dicts	23	3	60	21	0	34	19	2	50	19	2	1
direct_calls	10	3	8	6	0	7	3	0	6	5	1	0
dynamic	1	0	2	1	0	2	1	0	2	1	0	0
exceptions	0	0	2	0	0	1	0	0	1	0	0	0
external	0	0	3	0	0	**8**	0	0	**2**	0	1	0
functions	8	9	12	5	0	14	5	2	13	6	5	1
generators	9	4	17	5	0	23	4	3	18	6	1	3
imports	3	0	11	1	0	16	3	0	20	3	0	0
kwargs	8	5	5	7	0	4	4	0	5	4	4	0
lambdas	3	7	4	6	0	11	2	0	1	2	4	0
lists	14	1	26	17	0	27	13	0	25	16	1	0
mro	14	0	16	0	0	13	0	0	14	13	0	0
returns	11	1	16	11	0	17	9	0	13	11	1	0
Total	**186**	**56**	**322**	**122**	**0**	**293**	**100**	**8**	**297**	**155**	**32**	**6**
	564/845			**415/845**			**405/845**			**193/845**		
Sound	**68/154**			**24/154**			**21/154**			**0/154**		
Complete	**55/154**			**30/154**			**91/154**			**81/154**		

Machine-Learning-Based Tools

Table 4.3 details the performance metrics for the machine-learning-based systems.

The benefit of augmenting static analysis with neural inference is evident in the performance gap between *HiTyper* and *HiTyper-DL*. The hybrid configuration (*HiTyper-DL*) achieved 369 exact matches (43%), significantly outperforming the static-only baseline, which obtained 250 matches (29%). This augmentation proves particularly effective in categories where static analysis traditionally struggles with ambiguity, such as `classes` (70 overall matches vs. 47) and `dicts` (45 overall matches vs. 38). These results suggest that neural predictions successfully bridge critical information gaps in the static type dependency graph that *HiTyper* generates, resolving types that rule-based propagation fails to capture.

In contrast, *Type4Py*, which operates as a purely data-driven system, achieved 157 exact matches (18%). This lower performance highlights the difficulty of generalizing learned patterns to out-of-distribution code snippets without the grounding provided by static analysis constraints.

Table 4.3: Comparison of exact matches, sound, and complete values of type inference tools for micro-benchmark categories (HiTyper, HiTyper-DL, Type4Py). The benchmark comprises **154** test cases with **845** type annotations. Abbreviations: **LVT** (Local Variable Type), **FRT** (Function Return Type), **FPT** (Function Parameter Type).

Category	**HiTyper**			**HiTyper-DL**			**Type4Py**		
	FRT	FPT	LVT	FRT	FPT	LVT	FRT	FPT	LVT
args	8	0	0	12	0	6	11	2	6
assignments	20	0	5	21	4	9	0	2	5
builtins	1	0	17	1	2	18	1	2	8
classes	24	0	23	**27**	**2**	**41**	0	0	17
decorators	7	0	0	8	0	3	7	0	3
dicts	20	2	16	**20**	**3**	**22**	2	3	18
direct_calls	2	0	0	3	2	4	2	2	4
dynamic	1	0	2	1	0	5	0	0	5
exceptions	0	0	1	0	0	1	0	0	0
external	0	0	2	1	0	4	1	0	1
functions	3	2	1	5	5	5	1	3	4
generators	10	3	11	10	5	11	1	1	3
imports	3	0	0	3	0	11	3	0	10
kwargs	4	0	0	7	0	0	3	0	0
lambdas	3	0	5	3	0	7	0	0	2
lists	13	0	13	16	3	16	2	3	4
mro	13	0	6	15	0	11	0	0	4
returns	9	0	0	10	1	5	5	1	5
Total	**141**	**7**	**102**	**163**	**27**	**179**	**39**	**19**	**99**
	250/845			**369/845**			**157/845**		
Sound	**3/154**			**18/154**			**5/154**		
Complete	**135/154**			**32/154**			**11/154**		

Table 4.4: top-n exact matches comparison with ML tools

Tool	top-n	FRT	FPT	LVT	Total
HeaderGen	1	186	56	322	564
HiTyper-DL	1	163	27	179	369
	3	173	37	225	435
	5	175	37	229	441
Type4Py	1	39	19	99	157
	3	103	31	167	301
	5	109	31	174	314

Soundness and Completeness

Soundness and completeness are evaluated using strict, test-case-level criteria. A tool is considered *sound* for a given test case if it produces zero false negatives (i.e., it misses no valid types). Conversely, it is considered *complete* if it produces zero false positives (i.e., it reports no erroneous types). Under this strict binary standard, a single error in a test case fails the respective metric.

HeaderGen achieved the most balanced profile, demonstrating soundness on 68 of 154 snippets (44%) and completeness on 55 (35%). Among the ML-based tools, *HiTyper* showed poor soundness with 3/154, but the hybrid *HiTyper-DL* improved this to 18/154. While this increase is modest, it confirms that ML-based augmentation helps to capture types that static analysis misses. Notably, the purely static *HiTyper* configuration appears highly complete (87%), but this metric is inflated by the tool's failure to produce any predictions for 34 test cases; since an empty result set contains no false positives, these cases are technically marked as complete. *Type4Py* exhibited low scores for both soundness and completeness, further reflecting the stochastic nature of pure ML-based prediction on this benchmark.

Top-n Evaluation

Table 4.4 compares the top-n accuracy of the machine-learning-based tools against the best-performing static tool, *HeaderGen*.

HiTyper-DL shows significant improvement when considering top-5 predictions, reaching 441 exact matches (78.2% of *HeaderGen*'s score). However, the marginal gain between top-3 and top-5 is minimal, indicating that the majority of correct predictions fall within the first three candidates. *Type4Py* benefits immensely from this relaxed constraint; its exact matches nearly double from top-1 to top-3. Yet, similar to *HiTyper-DL*, its performance plateaus after the top-3 types.

Overall, while the top-n results demonstrate the promise of probabilistic models, particularly for suggesting likely types in an IDE context, they still trail the precision of robust static analysis even when granted a margin of error.

4.5 Discussion

This section interprets the empirical findings to analyze how architectural choices shape trade-offs among precision, coverage, and adaptability, and reflects on broader implications for Python type inference research.

4.5.1 HeaderGen

HeaderGen demonstrated the most consistent performance across the benchmark, balancing precision with broad feature coverage. Its analysis, built upon PyCG, enables the robust propagation of type information through assignments and control structures. Its primary advantage stems from the construction of an Extended Assignment Graph, which augments the underlying PyCG framework with Definition-Use Chains, enabling *HeaderGen* to index variable definitions by their specific program location, preventing the imprecise merging of reassignments. This allows the tool to model the sequential evolution of data types, enabling accurate type inference.

However, the tool exhibits clear limitations in the `builtins` and `external` categories, indicating restricted support for leveraging type stubs. While HeaderGen can incorporate external type information, it currently lacks the logic to handle advanced stub features, such as overloads or context-sensitive type definitions, which limits its ability to reason about richly annotated third-party libraries. Despite these constraints, HeaderGen remains the most reliable static analysis tool evaluated, balancing soundness and completeness more effectively than the other baselines.

4.5.2 Jedi

Jedi, an open-source community-driven static analysis tool designed for IDEs, provides broad support for Python's feature set. Its ability to incorporate external modules and partially reason about library-provided behavior contributes to its strong performance in the `builtins` and `external` categories.

However, diagnostic analysis using TypeEvalPy identified a recurring semantic error: when a variable receives a function as an argument, *Jedi* frequently infers the function's return type rather than the type `Callable`. This behavior, observed in 18 cases, suggests a gap in *Jedi*'s modeling of higher-order functions. While *Jedi* excels at the latency-sensitive name resolution required for code completion, its underlying inference model shows limitations when subjected to a rigorous semantic benchmark.

4.5.3 Pyright

Pyright stands out for its precise integration of `Typeshed` [3] and third-party typestubs. This integration is directly reflected in its top-tier performance for the `builtins` and `external` categories. Compared to *Jedi*, *Pyright* offers superior inference for function parameters and local variables, although it remains conservative, defaulting to `Any` in the absence of explicit constraints.

A significant barrier to using *Pyright* in research contexts, however, is its architecture. Implemented in TypeScript, it does not expose a native API for accessing intermediate inference results. To evaluate it within TypeEvalPy, we had to develop a custom Language Server Protocol (LSP) client to query the server for every program element. This workaround highlights a missed opportunity: providing official programmatic interfaces would expand *Pyright*'s applicability within the static analysis research community.

4.5.4 Scalpel

Scalpel proves effective for function-level analysis, ranking second among static tools for return (FRT) and parameter (FPT) types. However, its analysis is currently incomplete; it lacks support for inferring local variable (LVT) types, resulting in only six LV matches across the entire benchmark.

[3] https://github.com/python/typeshed

This omission severely impacts its overall utility. Additionally, *Scalpel* does not attempt to model types from external libraries, thereby limiting its relevance to real-world, dependency-heavy projects. Nevertheless, *Scalpel* is designed as a modular framework rather than a user-facing tool. Its support for control-flow graph construction, alias analysis, and call-graph generation makes it a valuable foundation for building custom analyzers. Improving its type inference component represents a clear pathway for future development.

4.5.5 Type4Py

Type4Py illustrates the trade-offs inherent in pure machine learning approaches. It uses deep similarity learning to infer types from high-dimensional embeddings of code and identifiers. Because its predictions are constrained to the distribution of types encountered during training, it struggles with categories that require structural reasoning, such as `classes`.

The model relies heavily on lexical features like variable names, signatures, and usage contexts. This limits its ability to generalize to rare or structurally complex types. Interestingly, it performs comparatively better on local variables than on function signatures, likely reflecting the prevalence of local variable assignments in its training corpus. Given its probabilistic nature, *Type4Py* benefits substantially from top-n evaluation; considering the top-3 or top-5 predictions nearly doubles its accuracy. This suggests that its primary utility lies not in serving as a precise verifier, but as an assistive recommendation engine for developers.

4.5.6 HiTyper and HiTyper-DL

HiTyper represents a hybrid architecture that attempts to fuse static analysis with neural predictions. The evaluation reveals that its static component is highly unsound, despite its attempt to be highly complete.

The hybrid variant, *HiTyper-DL*, improves accuracy by integrating predictions from Type4Py and employing “rejection rules” to filter inconsistent candidates. However, the rejection rules frequently discard valid neural candidates to avoid false positives, effectively lowering the system’s recall. Furthermore, the hybrid approach imposes considerable computational overhead without a proportional gain in soundness; only 15 of the 154 sound results originated from the ML component. These results illustrate that while hybrid inference is conceptually promising, balancing the rigor of static analysis with the fuzziness of ML-based predictions remains an open challenge.

4.5.7 Outlook

Our evaluation indicates that no single tool excels across all dimensions of Python type inference. While *HeaderGen* delivers the most balanced and logically consistent performance, it currently lags behind industry-standard tools such as *Pyright* and *Jedi* in support for the *Typeshed* ecosystem. Conversely, *HiTyper-DL* demonstrates that hybridizing static and learned inference can outperform either approach in isolation, though significant integration challenges remain. Consequently, the most promising path for future research lies in synergistic combinations, specifically, augmenting *HeaderGen*’s robust analysis with *Type4Py*’s probabilistic suggestion engine to create a system that is both semantically sound and lexically aware.

4.6 Limitations

While TypeEvalPy provides a rigorous and verified baseline for evaluating type inference, the methodology is subject to constraints inherent to manual curation. These limitations primarily

stem from the trade-off between precision (verified ground truth) and scale.

Insufficient Ground-Truth Volume. The current micro-benchmark comprises 154 code snippets. While this volume is sufficient to identify specific failure modes in deterministic static analysis tools (e.g., *HeaderGen* or *Pyright*), it is statistically insufficient for evaluating probabilistic models. For data-driven approaches like using Large Language Models, a small test set may reflect the memorization of common patterns rather than genuine reasoning capabilities.

High Annotation Costs. The creation of verified ground truth is labor-intensive, requiring expert domain knowledge to manually trace types and resolve ambiguities. This high annotation cost creates a bottleneck that prevents the benchmark from scaling to thousands of examples.

Limited Diversity of Base Types. Critically, the manual benchmark exhibits a skewed distribution of Python base types. Common types such as `str` and `int` are over-represented, while other primitives appear less frequently. This imbalance allows learning-based models to achieve artificially high exact-match scores by biasing their predictions toward the most frequent types, rather than demonstrating a genuine understanding of the variable's content.

These limitations highlight the necessity for scalable, automated alternatives, a challenge we address in the subsequent chapter through the introduction of synthetic benchmark generation.

4.7 Conclusion

This chapter introduced TypeEvalPy, a micro-benchmarking framework designed to enable a controlled, reproducible, and fine-grained evaluation of Python type inference systems. By standardizing tool execution, normalizing inference outputs, and providing rigorous ground-truth metrics, TypeEvalPy establishes a unified environment for comparing static, learning-based, and hybrid inference approaches.

Using this framework, we conducted an empirical study of six representative tools. The results reveal significant variation in the capabilities of existing techniques. Among static approaches, *HeaderGen* achieved the most balanced performance, delivering high exact-match accuracy alongside strong soundness and completeness. *Jedi* and *Pyright* demonstrated superior handling of external libraries and built-in types, largely due to their integration with the *Typeshed* ecosystem, but suffered from a systemic inability to infer function parameter types. *Scalpel*, while effective at the function level, was constrained by its lack of local variable analysis.

The learning-based systems highlighted a different set of trade-offs. *Type4Py* showed promise in local variable inference but was limited by its dependence on training distribution and its inability to reason about unseen types. The hybrid system *HiTyper-DL* successfully demonstrated that ML-based predictions can augment static analysis, yet it also revealed the inherent difficulty of integrating probabilistic suggestions with strict static constraints without compromising soundness.

The limitations uncovered in this study point to a growing need for methods that can reason more flexibly about dynamic patterns, ambiguous contexts, and complex dependencies, areas where traditional static analyses struggle. Recent advances in LLMs offer a potential solution: a new paradigm capable of deducing code semantics through statistical, contextual, and learned representations rather than rigid rule systems.

The next chapter explores this emerging direction in depth. We examine how LLMs can complement, extend, or even surpass conventional static analyses in tasks such as type inference and call-graph construction, building directly on the evaluation methodology established here.

SWARM-CG

5 Large Language Models for Static Analysis

The history of static analysis (SA) has been defined by the struggle to balance analysis accuracy with the dynamic nature of modern programming languages. Traditional rule-based tools, while effective in static contexts, inherently lack the flexibility to handle runtime behaviors found in Python or JavaScript. As the field moves toward probabilistic, data-driven methods, Large Language Models (LLMs) have emerged as a transformative force in Software Engineering [RWSI24, HCC+24, HZL+23, FGH+23, ZFX+23, ZNC+23].

Models such as BERT [DCLT19], T5 [RSR+23], and GPT [RWC+19] have demonstrated potential in automating complex SA tasks [ZFX+23]. Recent works have shown how different SA tasks can benefit from LLMs, such as false-positives pruning [LHZQ23a], improved program behavior summarization [LHZQ23b], type annotation [SEP+23], and general enhancements in precision and scalability of SA tasks [LHZQ23b, MAH+24], both fundamental issues of SA.

Despite these advances, the application of LLMs to core SA tasks, specifically *Type Inference* and *Call-Graph Construction*, remains significantly under-explored. Comprehensive evaluation in this domain is currently hindered by the methodological constraints identified in Chapter 4: **insufficient ground-truth volume**, **high annotation costs**, **limited semantic diversity**, and the **inherent inconsistency of developer-annotated code**. These factors restrict evaluation to common cases, failing to stress-test models against diverse edge cases. To rigorously evaluate whether LLMs can replace or augment traditional tools, it is imperative to overcome these barriers through the development of scalable and diverse evaluation methodologies.

Consequently, this dissertation's third major contribution introduces two advancements to the evaluation landscape:

1. **Scaling Type Evaluation via Auto-Generation:** The manual TypeEvalPy benchmark [SVSW+23] is constrained by a limited ground-truth size, covering only 860 type annotations. We introduce $\text{TypeEvalPy}_{\text{AutoGen}}$ to automatically generate and scale the evaluation to 77,268 type annotations, ensuring models are tested on generalizability and diverse type usages.

2. **Multi-Language Call-Graph Benchmarking:** To evaluate structural reasoning beyond Python, we introduce SWARM_{CG}, a comprehensive benchmarking suite for evaluating call-graph construction tools across multiple programming languages. This includes the development of $\textit{SWARM}_{\textit{JS}}$, a specialized micro-benchmark for JavaScript, enabling consistent cross-language comparison of call-graph construction capabilities.

Leveraging these advancements, we conducted a comprehensive empirical study to rigorously benchmark the state of the art. We evaluated **24** distinct models, ranging from proprietary systems like OpenAI's GPT-4o to open-source models such as LLaMA and Mistral. This evaluation utilizes our suite of controlled micro-benchmarks, specifically TypeEvalPy$_{\text{AutoGen}}$ for type inference and SWARM$_{\text{JS}}$ alongside PyCG for call-graph analysis, to isolate specific language features and ensure rigorous ground-truth validation.

Our decision to rely on these micro-benchmarks rather than large-scale real-world repositories is deliberate. While real-world data provides scale, it is frequently susceptible to human error and noise [DGP22b]. In contrast, our micro-benchmarks allow for rigorous manual inspection and the isolation of specific language characteristics, ensuring that observed failures are due to model limitations rather than ground-truth ambiguities.

Results. The results of our study show that static analysis tools like PyCG and Jelly [LXM24] 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, suggesting a narrowing gap between open-weight and proprietary approaches.

Structure. The remainder of this chapter is organized as follows: Section 5.1 reviews the related work. Section 5.2 defines the research questions, while Section 5.3 and Section 5.4 detail the benchmarks we used to address them, including the auto-generation methodology for TypeEvalPy$_{\text{AutoGen}}$ and the design of the SWARM$_{\text{CG}}$ suite. Section 5.5 outlines the experimental setup, including model selection and prompt engineering. The empirical results are presented in Section 5.6, followed by a discussion of their implications for the future of LLM-driven static analysis in Section 5.7. Finally, threats to validity are discussed in Section 5.8 and Section 5.9 concludes the chapter.

5.1 Related Work

5.1.1 Traditional Static Analysis for Python and JavaScript

Static Analysis Basics: Static analysis tools examine code without execution to identify bugs, type errors, or security vulnerabilities. In the context of Python and JavaScript, traditional analyzers primarily consist of linters and type checkers. For Python, tools such as **PyLint**[1] and **Flake8**[2] address style inconsistencies and simple bugs, while **MyPy**[3] and Meta's **Pyre** perform static type checking utilizing type hints. Similarly, JavaScript ecosystems rely on linters (e.g., **ESLint**[4]) and optional type systems (such as Flow or the TypeScript compiler) for error detection.

Beyond industry-standard tools, academic solutions have also emerged. For instance, **PyCG** constructs call graphs for Python using a context- and flow-insensitive analysis [SSL+21b],

[1] https://github.com/pylint-dev/pylint
[2] https://github.com/PyCQA/flake8
[3] https://github.com/facebook/pyre-check
[4] https://github.com/eslint/eslint

while **Type4Py** uses deep learning to predict Python types, thereby enhancing static inference [MLPG22b].

Despite their utility, these tools face significant challenges when applied to dynamic languages. Python and JavaScript's dynamic features complicate static reasoning. 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 [LDN25]. They also tend to over-approximate program behaviors, yielding many false positives that developers must triage [LDN25]. 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** [HBBA18] and **Typilus** [ABDG20b] learned to suggest variable types in Python, and **HiTyper** [PGL+22] combined static inference with a neural network to improve Python type predictions. These approaches showed that learned models can complement rule-based analysis, but they are narrow in scope and struggle with long-range dependencies in code.

5.1.2 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** [SFY+23], 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 [SRR+25]. 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. [LHZQ23b] 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. [LHZQ23a] 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. Seidel et al. [SEP+23] (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. In this study, we investigate whether LLMs can effectively perform type inference and call-graph construction.

5.1.3 Micro-Benchmark Suites for Python and JavaScript

Static Call Graph Analysis Benchmarks in Python: Salis et al. [SSL+21b] 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.

Subsequent work built on PyCG's benchmark to improve coverage and address its limitations. Huang et al. [HYC+24] 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. As part of our HeaderGen [SVWLB23c] tool, we similarly created a micro-benchmark by adopting PyCG's full suite and augmenting it with new snippets focused on 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 does not provide formally verified ground-truth call-graphs. 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 the precision of found edges and neglects recall (missing edges). Antal et al. [AHH+23] 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 struggle 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. Recently, a hybrid JavaScript call-graph analysis tool, Jelly [LXM24], addressed this problem with a tool that combines static and dynamic analysis to improve accuracy and support modern JavaScript features.

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 evaluation of call-graph construction tools across multiple programming languages with SWARM_{CG}.

Python Type Inference Benchmarks: Previous evaluations of type inference relied on either large-scale corpora of real-world code with optional type hints (e.g., the ManyTypes4Py dataset and Type4Py) or on each tool's own set of examples and test cases, making it difficult to compare results across studies. Recognizing this gap, we built TypeEvalPy, a micro-benchmarking framework for Python type inference tools as discussed in the previous chapter. While TypeEvalPy greatly improved standardization in evaluating Python type inference, it has some limitations in scope. The 154 snippets were designed to be representative but cannot cover all possible scenarios and combinations of data types. To address this, we augment TypeEvalPy with an auto-generation extension that massively scaled up the benchmark's coverage to $\approx 77k$ type annotations that increase the diversity of types and scenarios. Similar ideas of automatically generating benchmarks have also been explored in other domains, such as Android taint analysis with DroidGenBench [SP22], which uses synthesized apps to reduce manual benchmark construction effort.

5.2 Research Questions

This study is guided by two primary research questions aimed at evaluating the effectiveness and performance of call-graph generation and type inference tools:

RQ1: *How accurate are LLMs in constructing call graphs compared to traditional static analysis tools, and does this performance generalize across languages?*

RQ2: *Can LLMs outperform traditional tools in Type Inference for Python, and how does their performance scale when tested against diverse cases?*

Answering these research questions requires an evaluation strategy that balances precision with scale. Standard large-scale real-world benchmarks are often too large and noisy for isolating specific reasoning failures, while existing micro-benchmarks lack the scale required to statistically validate probabilistic models.

Consequently, we established a composite evaluation framework designed to stress-test LLMs beyond the limits of current datasets. This framework integrates proven baselines with our two novel contributions: an auto-scaled type inference benchmark and a new multi-language call-graph suite. We describe the design and construction of these benchmarks in the following sections.

5.3 Call-Graph Micro-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 landscape, 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,[5] (4) Ollama,[6] (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.

Within this framework, we have integrated and expanded benchmarks for two primary languages: Python and JavaScript.

[5] `https://huggingface.co/docs/transformers/en/index`
[6] `https://ollama.com/`

Python Baselines (PyCG & HeaderGen). For Python, we utilize two established micro-benchmarks that evaluate different aspects of structural analysis:

- **PyCG** [SSL+21b]: We use this benchmark to evaluate the accuracy of the overall call graph. It serves as the standard for validating the correctness of the graph topology (nodes and edges) across various language features.
- **HeaderGen** [SVWLB23c]: This benchmark extends the evaluation to call sites with specific program locations. HeaderGen's benchmark requires the tool to resolve the exact definition location of a function invoked at a specific line and column.

JavaScript Expansion ($\mathbf{SWARM_{JS}}$). To evaluate cross-language generalization, we introduce $\mathbf{SWARM_{JS}}$. As no modern ground truth existed for JavaScript call graphs, we developed this suite to mirror the rigor of PyCG. It isolates specific ES6+ features, such as arrow functions and prototypes, to determine if LLM reasoning is consistent across languages.

5.3.1 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 [HYC+24]. These additions include 4 in *args* category, 4 in *assignments*, 5 in *direct_calls*, and 1 in *imports*.

5.3.2 HeaderGen: Call-sites Micro-Benchmark

Derived from the PyCG suite, the HeaderGen benchmark introduces a layer of granular metadata. It annotates function calls with specific source coordinates (line and column numbers) to enable the evaluation of precise call-site resolution. Additionally, the suite is expanded with eight specialized test cases targeting flow-sensitive analysis.

5.3.3 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 [Web10], 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 [AHH+23], 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 [SSL+21b] 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 [ECM15] and the existing SunSpider [Web10] 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 5.1 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 5.1 and its corresponding ground truth is given in Listing 5.2.

5.4 Type Inference Micro-benchmarks

To evaluate type inference, we utilize the state-of-the-art TypeEvalPy framework, addressing the scalability of manual benchmarks through a novel auto-generation engine.

Table 5.1: 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.

5.4.1 Manual Baseline (TypeEvalPy)

The core TypeEvalPy benchmark [SVSW+23], introduced in Chapter 4, consists of 154 manually curated code snippets with 860 type annotations. These target specific capabilities, such as dynamic attributes and external library calls.

5.4.2 TypeEvalPy$_{\textbf{AutoGen}}$ Extension

The micro-benchmark within the TypeEvalPy framework is constrained by its limited representation of Python base types, covering only 860 types derived from 154 code snippets. This scarcity has significant implications for evaluating LLMs. Since the original benchmark exhibits a high prevalence of specific types, such as `str`, LLMs may achieve high exact-match scores due to statistical overrepresentation rather than a genuine understanding of type semantics. This narrow focus undermines the robustness of the evaluation, as models are not sufficiently tested against the wide variety of base types available in Python.

To address this limitation, we extended TypeEvalPy by integrating an auto-generation capability designed to broaden type diversity. This enhancement is realized through a systematic, template-based generation process. We first converted existing code snippets into templates containing placeholders, such as `<value1>`, which are dynamically replaced by different types during generation. Configuration files were created to map these placeholders to potential type values. For example, Listing 5.3 shows a template with placeholders, while Listing 5.4 shows the corresponding generated code. The ground truth and values are derived from the configuration

```javascript
function paramFunc() {}

function func(a) {
    a();
}

func(paramFunc);
```

Listing 5.1: Code snippet of main.js

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

Listing 5.2: Ground truth for main.js

rules in Listing 5.5. Specifically, line 2 in Listing 5.3 corresponds to the type mappings defined in lines 15 and 31 of Listing 5.5.

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, i.e., $P(n, r) = n!/(n - r)!$ where n is the total number of configured types, and r is the number of placeholders in a given template. Each generated program features a unique arrangement of types, such as `(str, float)` versus `(str, int)`, enhancing the benchmark's diversity. It is important to note that the concrete values for these variables are generated randomly according to their assigned data types. An example of a generated test case and its associated ground truth is provided in Listing 5.4 and Listing 5.6, respectively.

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 modeling to avoid inconsistencies in the generated programs. The generator was engineered to recursively annotate elements within data structures and correctly model external dependencies, ensuring edge cases were addressed.

Following generation, each program undergoes an execution check to verify correctness. Programs that execute without errors are retained, while those failing due to runtime type incompatibilities, such as attempting to add a `string` to a `float`, are automatically discarded. This filtering step ensures that only valid, runnable test cases are included in the final benchmark.

The development of the benchmark involved a rigorous quality assurance process. The initial templates were designed by the first author, while the second author verified the generated programs, iteratively correcting errors to ensure the accuracy of the type annotations. Furthermore, all programs are designed to have a single execution path, thereby eliminating ambiguity in the ground truth. Since its initial release, open-source contributions have further enhanced the robustness of the benchmark, expanding the template library and refining edge-case handling to ensure the dataset remains a reliable standard for the community.

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

5.5 Methodology

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

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


def func2():
    return <value2>

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

Listing 5.3: Template for main.py

```python
def func1():
    return 34


def func2():
    return 53.24

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

Listing 5.4: Generated main.py

5.5.1 Type Inference Tools

For this study, we select a focused subset of the tools evaluated in the previous chapter (Section 4.3) to serve as robust baselines against LLMs. Rather than re-evaluating the entire spectrum of tools, we choose the best-performing representatives from the static and hybrid paradigms:

1. **HeaderGen:** Selected as the static analysis baseline because it demonstrated the highest overall accuracy and balanced performance.

2. **HiTyper:** Selected as the representative for machine learning and hybrid approaches. Note that HiTyper integrates *Type4Py* as its internal model.

5.5.2 Call-graph Construction Tools

To evaluate analysis capabilities across languages, we select four representative tools. For Python, we employ *PyCG* and *HeaderGen* to assess call-graph analysis and call-site analysis, respectively. For JavaScript, we select *TAJS* as a traditional static baseline and *Jelly* as a state-of-the-art hybrid approach that combines static and dynamic analysis.

Python Baselines

We utilize the same Python analysis tools described in Chapter 4, distinguishing them by the granularity of their analysis and output:

1. **PyCG (Call-Graph Analysis):** *PyCG* [SSL+21b] serves as our baseline for call-graph analysis. As detailed in Section 3.2.1, it computes inter-procedural relations abstracted as an assignment graph and models complex Python features.

2. **HeaderGen (Call-Site Analysis):** *HeaderGen* [SVWLB23b] is employed to evaluate call-site analysis. As previously discussed in Section 3.2, it extends *PyCG*'s analysis with the ability to pinpoint the specific program location (line number and column offset) of every function call.

JavaScript Baselines

To extend our evaluation to JavaScript, we selected two tools representing distinct generations of static analysis capability:

```
{
"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 5.5: 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 5.6: Generated ground truth for main.py

1. **TAJS - Type Analyzer for JavaScript:** TAJS [JMT09] 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 [ECM15], such as classes, arrow functions, and modules, which limits its effectiveness in analyzing modern JavaScript applications.

2. **Jelly:** Jelly [LXM24] 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 static analysis phase, refining the constructed call-graph and improving soundness. This method allows Jelly to outperform traditional tools, particularly in handling modern ECMAScript. Evaluations show that Jelly outperforms tools like TAJS[JMT09] and ACG[FSS+13], making it more accurate for real-world JavaScript analysis.

5.5.3 Large Language Model Selection

We selected LLMs for evaluation by focusing on organizations that are actively conducting research and publishing state-of-the-art models on the Hugging Face platform.[7] 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 5.2.

5.5.4 Prompt Design

To optimize prompt design, we adopted an iterative and experimental approach [CZLZ23, SIB+24]. 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. [CZLZ23] indicates that for sufficiently complex models,

[7] `https://huggingface.co/`

Table 5.2: 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	*Not Public*
	gpt-4o-mini	*Not Public*
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

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. To address this, we explored a *question-answer* based method, querying the model with explicit questions and then converting 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 graph tasks in detail.

Type-Inference Prompts

The prompt design employed in this study follows a structured two-part approach. The first section, as shown in Listing B.2, provides a detailed task description to ensure clarity regarding the expected analysis. This is followed by a one-shot example (an input-output pair) and explicit instructions on the output format. Finally, the target code is appended to the prompt; for test cases involving imports, all relevant file contents and relative paths are included to preserve the project structure. Finally, the target code 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 this careful structuring, initial attempts to generate valid JSON outputs proved difficult. Models frequently failed to adhere to the complex TypeEvalPy output schema, resulting in JSON with missing or unexpected keys. To address these schema-compliance issues, we reduced the task complexity by shifting to a *question-answer* format. Rather than generating a complex JSON object directly, the model is asked to answer a series of specific questions. As shown in Listing B.3, each question targets a specific variable or function, identified by its name and declaration location, and includes a numbered placeholder for the model's response.

For the purpose of this study, the questions were generated using the benchmark's ground-truth data. We iterated over the variables and functions listed in the ground truth to formulate precise questions regarding their types. To clarify, consider the full prompt in Listing B.4. In this case, we know from the ground-truth 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. While we utilized ground truth to simplify implementation, in a practical application, this context could be derived by parsing the program's Abstract Syntax Tree (AST).

Finally, the model's responses were parsed using regular expressions to map the answers back to their corresponding questions. This reconstruction step allowed us to generate valid JSON outputs that strictly adhered to the TypeEvalPy schema for evaluation. A complete example of this workflow, including source code, raw model response, ground truth, and the parsed JSON, is provided in Section B.2.1.

Call-Graph Prompts

The prompt design for call-graph analysis mirrors the structure used for type inference. Each prompt begins with a detailed task description outlining the specific analysis requirements (Listing B.5), followed by a language-specific input-output example (Listing B.6). Explicit formatting instructions are included to ensure consistency in the model's responses.

Questions within the prompt are generated using a similar strategy to the one described in the previous section. Typically, the first question targets module-level function calls, while subsequent questions address calls made within specific function definitions (see Listing B.6). While these questions were formulated using ground-truth data to simplify the experimental implementation, in a practical deployment, they could be automatically generated by traversing the program's AST to identify function definitions and call nodes.

For call-site analysis, the prompts were further refined to request precise call-site locations. Listings B.7 and B.8 detail the specific prompt adjustments used to extract call-site information.

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 size is comfortably within the context window of all evaluated LLMs. The smallest model, `TinyLlama-1.1b`, supports a context of 2,048 tokens, while the largest, `gpt-4o`, supports up to 128,000 tokens. For perspective, even the cumulative size of all prompts in the TypeEvalPy micro-benchmark (69,563 tokens) fits well within the maximum context length of most models, ensuring that no input truncation occurred during evaluation.

5.5.5 Evaluation Metrics

To assess the performance of the evaluated models, we employ three primary metrics: *completeness*, *soundness*, and *exact matches*. While completeness and soundness are primarily used for call-graph construction, exact matches are applied to both call-graph and type inference tasks. Additionally, we report inference time for open-source models to provide a measure of efficiency.

Completeness and Soundness. In this study, we adopt the definitions established in prior call-graph research [SSL$^+$21b, SVSC$^+$24]. 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 [SSL+21b] and HeaderGen [SVWLB23c].

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 with empty lists, an exact match is counted if the model also produces an empty list. 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 **type inference**, we follow standard literature conventions [ABDG20b, MLPG22b, PGL+22, SVSW+23, SVSM+24], defining an exact match as a prediction that is textually identical to the ground-truth type annotation.

Time Efficiency. We report the total inference time for processing the full benchmark suite. To ensure a fair comparison, all open-source models were executed on identical hardware using standardized parameters: 4-bit quantization and a fixed batch size of 12. Although smaller models could theoretically support larger batch sizes, we maintained uniform conditions to prevent hardware-utilization differences from skewing the assessment.

Proprietary models accessed via OpenAI's API are excluded from the runtime comparison. As their execution occurs on remote infrastructure with unknown hardware configurations and variable network latency, a direct efficiency comparison with local models is not feasible.

5.5.6 Implementation Details

We implemented our experimental pipeline using the Hugging Face Transformers library [WDS+20], which provided a unified interface for managing inference across the diverse set of evaluated LLMs. To ensure reproducibility, all models were configured to use greedy search decoding, selecting the most probable next token to guarantee deterministic outputs.

Integration with our benchmarking frameworks was achieved through custom extensions. For type inference, we extended the TypeEvalPy framework to support the auto-generated test cases and LLM interaction. For call-graph construction, we developed a specialized adaptor within the SWARM_{CG} framework to facilitate model execution and result parsing.

All experiments were conducted on a high-performance computing node equipped with a single NVIDIA H100 GPU (80GB VRAM), 16 Intel(R) Xeon(R) Platinum 8462Y+ processors, and 78 GB of system memory.

Note on Quantization. To maximize resource efficiency, we employed 4-bit quantization for all models. This strategy significantly reduced memory footprint and computational overhead, enabling the deployment of large-scale models (up to 123B parameters) on a single GPU. We standardized this configuration across all open-source models (Batch Size: 12) to ensure a controlled comparison. Prior research confirms that post-training quantization has a minimal impact on the accuracy of large models, making it an effective trade-off for extensive empirical studies [JDH+24a, LGH24, JDH+24b, DPHZ23].

5.6 Results

We now present the empirical findings from our evaluation, addressing the research questions through a detailed analysis of model performance.

5.6.1 RQ1: Accuracy of Call-graph Analysis

The results of our experiments for call-graph analysis for Python and JavaScript are presented in Tables 5.3 and 5.4, respectively. The Python results are based on the PyCG micro-benchmark

suite, while the JavaScript results use the SWARM_{JS} micro-benchmark.

Table 5.6 presents the results for call-site analysis based on the HeaderGen micro-benchmark. In the following sections, we discuss each of these results in detail.

Call-graph Analysis

Python Programs The evaluation on Python programs (Table 5.3) reveals the distinct superiority of the static analysis tool PyCG over the evaluated LLMs across all key metrics. PyCG achieved 84.9% completeness and 87.3% soundness across 126 test cases, demonstrating a robust ability to avoid false positives (completeness) while missing very few valid function calls (soundness). It also secured 569 exact matches out of 599, significantly outperforming the closest LLM, *mistral-large-it-2407-123b*, by 51 matches.

Among the LLMs, *mistral-large-it-2407-123b* delivered the strongest performance, though it remained moderate with 60.3% completeness and 62.6% soundness. Its relatively high exact match score of 86.4% indicates competence in identifying many call relations, but frequent failures in soundness and completeness highlight gaps in handling specific Python language features.

gpt-4o ranked third, trailing *mistral-large*, which suggests that top-tier open-source models are becoming competitive with closed-source alternatives in this domain. However, the majority of other open-source models performed poorly. Notably, *gemma2-it-9b* exhibited an extreme imbalance: despite a high completeness score of 95%, its soundness was a negligible 3%, resulting in a very low exact match rate of 10.5%. This indicates a tendency to miss nearly all valid calls while rarely hallucinating incorrect ones.

Efficiency also varied drastically. PyCG completed the analysis in 0.41 seconds, whereas LLMs required orders of magnitude more time (e.g., 534 seconds for *mistral-large*). Some smaller models, like *gemma2-it-9b*, exhibited unexpectedly high runtimes (1587 seconds), further emphasizing the trade-off between model architecture and inference efficiency.

To clarify how the results are parsed and evaluated, diagnostic examples detailing the source code, ground truth, raw LLM response, and parsed call-graph JSON are provided in Appendices B.2.2 and B.2.3.

JavaScript Programs Table 5.4 summarizes the results for the JavaScript SWARM_{JS} benchmark. The hybrid analysis tool that combines static and dynamic analysis, *Jelly*, demonstrated robust performance, achieving 38.8% completeness, 67.4% soundness, and 82.2% exact matches when using its approximate interpretation feature [LXM24]. This configuration incorporates dynamic execution hints into the static analysis process, improving the tool's ability to resolve function calls accurately.

Conversely, the traditional static analyzer *TAJS* performed poorly. Despite a seemingly high completeness score (indicating few false positives), 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 failure stems from its lack of support for ECMAScript 6 features (e.g., classes, arrow functions), which are prevalent in the SWARM_{JS} benchmark.

Among LLMs, *mistral-large-it-2407-123b* led with 40.4% completeness and 42.8% soundness (76.8% exact matches). *gpt-4o* followed closely, achieving higher soundness (50.7%) but lower completeness (34.9%). As observed in the Python results, the *gemma2* family (9b and 2b) exhibited extremely high completeness but negligible soundness, resulting in largely empty call graphs. Furthermore, gemma2 models had a very high runtime of 1593.44 seconds, making it both inefficient and ineffective.

Table 5.3: Comparative analysis across LLMs for 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

Table 5.4: Comparative analysis across LLMs for call-graph analysis on the SWARM$_{JS}$ JavaScript micro-benchmark

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

Model	Complete (126 cases)	Sound (126 cases)	Exact Matches (596 cases)	Time (s)
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

Cross-Language Comparison Table 5.5 compares the top 10 LLMs across both languages. A consistent pattern emerges: models generally perform better on Python than on JavaScript. For instance, the top-performing *mistral-large* achieved 86.6% exact matches in Python compared to 76.8% in JavaScript. This performance gap is consistent across other models, all of which show a noticeable decline in accuracy across metrics when evaluated on JavaScript. This trend suggests that current LLMs may be biased towards Python, potentially due to its prevalence in training corpora.

Table 5.5: Comparison of models across Python and JavaScript **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

Call-site Analysis

Table 5.6 presents the evaluation of flow-sensitive call-site analysis on the HeaderGen micro-benchmark. This task is notably more difficult, requiring the model to identify specific call-sites with line numbers.

The static tool HeaderGen outperformed all LLMs by a wide margin, achieving 90.9% completeness and 91.8% soundness (91.3% exact matches) with an execution time of 10.94 seconds. This underscores the efficiency and accuracy of dedicated static analysis algorithms for this task.

Among LLMs, *mistral-large-it-2407-123b* was again the top performer but fell significantly short of the baseline, achieving only 31.1% completeness and 26.2% soundness. Its low exact match score of 28.5% highlights the struggle of LLMs to pinpoint precise call locations. The deterioration in performance compared to call-graph analysis confirms that increased complexity requiring specificity about the location of function calls severely impacts LLM reliability.

Table 5.6: Comparative analysis across LLMs for call-site analysis on the HeaderGen micro-benchmark

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

Model	Complete (122 cases)	Sound (122 cases)	Exact Matches (357 cases)	Time (s)
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

RQ1 Summary

Static analysis tools such as PyCG, Jelly, and HeaderGen consistently outperformed LLMs in both call-graph and call-site analysis across Python and JavaScript. While *mistral-large* emerged as the most capable LLM, its accuracy remained well below that of dedicated static tools, highlighting the significant challenges LLMs face in performing rigorous structural code analysis.

5.6.2 RQ2: Accuracy of Type Inference

We now examine the performance of LLMs on Python type inference, utilizing the *Exact Match* metric. This evaluation compares the models against the static baseline HeaderGen and the hybrid tool HiTyper (configured with Type4Py [MLPG22b]).

TypeEvalPy Micro-benchmark

Table 5.7 presents the exact match accuracy on the 860 annotations of the TypeEvalPy micro-benchmark. In stark contrast to the call graph results, where static analysis reigned supreme, here, LLMs demonstrate a decisive advantage.

LLM Dominance. 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.

Specialization vs. Scale. 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, show 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.

Efficiency Trade-offs. 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.

TypeEvalPy$_{\textbf{AutoGen}}$ Benchmark

To test whether the LLMs' strong performance on the micro-benchmark was due to overfitting or memorization, we evaluated them on the TypeEvalPy$_{\text{AutoGen}}$ benchmark. This dataset

Table 5.7: 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

scales the evaluation from 860 to 77,268 annotations, introducing a diverse array of generated types. Table 5.8 presents the results, while Table 5.9 analyzes the performance delta between the two 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.

RQ2 Summary

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 LLMs achieved near-perfect accuracy ($> 93\%$) and maintained robust consistency when scaled to the 77k-annotation $\text{TypeEvalPy}_{\text{AutoGen}}$ benchmark.

5.7 Discussion

This section analyzes the empirical findings of our study. We first examine call-graph construction performance in Python and JavaScript, identifying specific strengths and weaknesses. We then assess type inference capabilities in Python, contrasting LLMs with traditional static analysis tools. Subsequently, we explore the performance disparity between type inference and call-graph analysis, cross-language differences, and the broader implications for software analysis, before proposing directions for future research.

5.7.1 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 5.10 compares PyCG and `mistral-large` across selected categories from the PyCG micro-benchmark, chosen specifically for similarities and differences in performance. Furthermore, in Listings 5.7 and 5.8, we list specific failure patterns for LLMs.

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 false positives and omissions, while PyCG resolved these cases correctly. In complex multi-level return

Table 5.8: 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

Table 5.9: 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

Table 5.10: 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 ●

structures, as illustrated by the *return_complex* test case (see Listing 5.7, `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. [YMH22b]. 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 (see Listing 5.8), 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. [YMH22b] categorizes 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.

```python
def func3():
    return func2

def func4(a):
    return func3()

...

func4()()
```

Listing 5.7: Returns: higher-order returns

```python
def func1():
    pass

def func2():
    pass

d = {"a": func1}

d.update({"a": func2})
d["a"]()
```

Listing 5.8: Dicts: update and indirect call

Table 5.11: 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

Summary: Call Graph Construction in Python

PyCG demonstrated superior overall performance in Python call-graph construction. However, `mistral-large` exhibited competitive capabilities in specific dynamic scenarios, such as dictionary updates, despite struggling with complex return structures and indirect imports.

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

For JavaScript, the hybrid analysis tool *Jelly* surpassed LLMs in soundness and exact match rates. The traditional static tool, *TAJS*, proved ineffective, largely due to its lack of support for modern ECMAScript features. Table 5.11 contrasts the performance of *Jelly* and `mistral-large`. Specific failure patterns are shown in Listings 5.9–5.12.

In the *arguments* category, *Jelly* achieved perfect soundness. `mistral-large` matched *Jelly*'s completeness but was sound in only 50% of cases, struggling with indirect argument (see Listing 5.9) flows and cross-file imports (see Listing 5.10). Correctly resolving these patterns is essential, as features like default parameters and spread arguments appear in over 56% of open-source JavaScript projects [LNB+25].

In the *classes* category, *Jelly* demonstrated robustness, achieving near-perfect soundness. In contrast, `mistral-large` faced significant challenges with inheritance and chained attribute references (see Listing 5.11). Given that class syntax and inheritance are now widely adopted in JavaScript development [NMM24], this represents a notable limitation for current LLMs.

In the *objects* category, `mistral-large` outperformed *Jelly* in resolving dynamically derived object keys from function parameters or external modules. However, *Jelly* proved more reliable in handling type coercion (see Listing 5.12), which `mistral-large` failed to resolve. The ability to handle these constructs is critical; a study by Lucas et al. [LNB+25] found that object features such as destructuring are highly prevalent, appearing in nearly 69% of projects in a dataset of 158 open-source JavaScript systems.

Summary: Call Graph Construction in JavaScript

Jelly consistently outperformed LLMs in JavaScript call-graph construction, particularly in handling classes and complex argument flows. While `mistral-large` showed potential in resolving dynamic object keys, its utility is limited by failures in high-frequency patterns like inheritance and indirect data flows.

```
1  function paramFunc(a) {
2    a();
3  }
4
5  function func(a) {
6    a(function nestedFunc() { ... });
7  }
8
9  const b = paramFunc;
10 const c = func;
11 c(b);
```

Listing 5.9: Arguments: nested call

```
1  import { func } from "./to_import.js"
       ;
2
3  function paramFunc() { }
4
5  func(paramFunc);
```

Listing 5.10: Arguments: imported call

```
1  class A {
2    func() {
3      if (this.child) { this.child();
       }
4    }
5  }
6
7  class B extends A {
8    constructor() {
9      super();
10     this.child = this.func2;
11   }
12   func2() { ... }
13 }
14
15 ...
16
17 const b = new B();
18 b.func();
```

Listing 5.11: Classes: base class calls child

```
1  function func1() { ... }
2  function func2() { ... }
3
4  let d = {1: func1, "1": func2};
5  d[1]();
6
```

Listing 5.12: Objects: type coercion

5.7.3 Type Inference in Python: LLMs vs Static Analysis Tools

Table 5.12 presents the comparative performance of `mistral-large` and HEADERGEN on Micro and Autogen benchmarks. The analysis highlights significant performance divergences in language constructs such as assignments, decorators, and generators.

HEADERGEN's errors arise primarily from the absence of modeling edge-case language constructs. Listings 5.13–5.16 presents complex cases where HEADERGEN failed to infer the types correctly. In assignments (see Listing 5.13), 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 (see Listing 5.14). 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 (see Listings 5.15 and 5.16).

Developing and maintaining such fine-grained modeling of language constructs is laborious. Static analyzers, therefore, default to conservative approximation as `Any`. In contrast, an

Table 5.12: 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

Category	Ground Truth			mistral-large			HeaderGen		
	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 ◔

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`.

```python
# augmented assignment
a += 3

# starred assignment
a, *b, c = f1, f2, f3

# nested unpacking
(x, (y, z)) = (1, (2, 3))

# recursive tuple unpacking
t1 = (1, 2)
(a, b) = t1
```

Listing 5.13: Assignments

```python
# class decorator
@decorator
class C:
    ...

# nested decorators
@d1
@d2
def g():
    ...
```

Listing 5.14: Decorators

Summary: Type Inference in Python

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 modeling of edge-case language constructs.

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

The empirical results indicate that LLMs perform substantially better on type inference tasks than on call-graph analysis. However, explaining this behavior of LLMs is challenging, as their

```python
def f():
    return

def g(n):
    for _ in range(n):
        yield 5

for fn in g(10):
    print(fn())

```

Listing 5.15: Generators: yield in function

```python
def squares():
    n = 1
    while True:
        yield n * n
        n += 1

gen = squares()
for _ in range(5):
    a = next(gen)
```

Listing 5.16: Generators: next on iterator

performance often emerges from complex interactions between training data, model architecture, and task formulation. A likely explanation is the alignment between the task and the model's training objective. Python type annotations are locally scoped and embedded within the source code, aligning well with the next-token prediction objective of LLMs. This allows models to learn type patterns effectively during pre-training. Although the micro-benchmark used in this study was newly created, lacked in-code annotations, and was unlikely to have been seen during pre-training, the LLMs were still able to generalize and perform well. Type inference also benefits from certain scenarios, such as local variable assignments, where a 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. Recent studies suggest that such structural reasoning capabilities do not naturally emerge from scaling alone [BGK25, ORV^{+}24], limiting the effectiveness of LLMs for global analysis tasks.

Summary: Type Inference vs. Call Graph Analysis

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

5.7.5 Cross-language Performance Disparities

A consistent observation throughout our experiments is that LLM performance is notably better on Python code than on JavaScript. This aligns with prior findings [Bus23, CTJ^{+}21], which suggest that Python's simpler syntax and prevalence in training corpora contribute to better model performance. While these factors offer a plausible explanation, a systematic investigation is required to definitively isolate the causes of this cross-language performance gap.

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

Summary: Future of Type Inference

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

5.7.7 Trade-offs Between Model Accuracy and Efficiency

Mid-sized models, such as `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.

Summary: LLM Scale vs Accuracy Trade-offs

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.

5.7.8 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.

Summary: Practical Deployment Considerations

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.

5.7.9 Towards Hybrid Analysis: LLMs as Enhancers of Static Tools

Given their proficiency in type inference, LLMs are well-positioned to augment, rather than replace, traditional static analysis pipelines. Inferred types can improve the precision of downstream analyses, such as call-graph construction, particularly in the presence of dynamic dispatch or polymorphism. A hybrid approach that combines the semantic understanding of LLMs with the structural rigor of static analysis represents a promising research direction. Realizing this requires addressing challenges related to confidence calibration, conflict resolution, and the maintenance of transparency within analysis workflows.

Summary: Towards Hybrid Analysis

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.

5.8 Threats to Validity

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

- **Prompt Uniformity:** We utilized a standardized prompt across all models to ensure a controlled comparison. Tailored prompt engineering for specific architectures could potentially yield improved performance. Our modular framework can serve as a foundation for future research aimed at refining prompts to improve performance.

- **Output Formatting:** 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 pre-processing 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.

- **Decoding Strategy:** 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.

- **Micro-benchmark Limitations:** Micro-benchmarks isolate specific features but may not fully capture the complexity of real-world applications. We mitigated this by extending TypeEvalPy with auto-generation capabilities, significantly expanding the diversity of test cases, though full coverage of all real-world variations cannot be guaranteed.

- **Auto-generation Validity:** Extending TypeEvalPy with auto-generation capabilities required significant human effort to create the initial templates. Although these were reviewed by multiple authors, they remain subject to human error. We assess that any minor inaccuracies are unlikely to substantially impact the overall trends observed.

- **Lack of Auto-generated Call Graph Benchmarks:** In contrast to type inference, we did not scale the call-graph benchmark using auto-generation techniques. While the low performance observed on the existing benchmarks already indicates weaknesses of the evaluated models, scaling the call graph benchmarks may further amplify these issues. Addressing this limitation by developing large-scale, automatically generated call graph benchmarks is a direction for future work.

5.9 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 chapter investigated the potential of LLMs to overcome the inherent limitations of traditional static analysis. To rigorously assess this, we expanded the evaluation landscape by introducing $\text{TypeEvalPy}_{\text{AutoGen}}$, which scales type inference benchmarking to over 77,000 annotations, and SWARM_{CG}, a multi-language framework for call-graph analysis. Through comprehensive empirical experiments involving 24 LLMs and state-of-the-art static baselines, we uncovered insights into their performance.

In type inference tasks, LLMs demonstrated a clear superiority over traditional tools. Models like `gpt-4o` and `mistral-large` successfully resolved dynamic Python features, achieving near-perfect accuracy even on large-scale, auto-generated benchmarks. This confirms that LLMs possess an implicit understanding of type systems that allows them to succeed where rigid static rules fail.

Conversely, our results highlight that LLMs currently lack the global reasoning capabilities required for call-graph construction. Traditional static tools, PyCG for Python and *Jelly* for JavaScript, consistently outperformed LLMs. Furthermore, we observed a notable performance disparity across languages, with LLMs performing significantly worse on JavaScript than Python, likely reflecting biases in training data composition.

While LLMs offer semantic understanding, their deployment comes with high costs. The high inference latency and hardware requirements of top-tier models contrast sharply with the sub-second efficiency of static analyzers. However, the competitive performance of mid-sized, fine-

tuned models (e.g., `codestral`) suggests a path toward more efficient, domain-specific solutions.

This study demonstrates the potential of LLMs in software analysis tasks, while also emphasizing their limitations and the continued strengths of traditional methods. Consequently, the most promising path forward for software analysis lies in hybrid approaches. By leveraging LLMs to infer precise semantic properties and feeding these insights into rigorous static rule-based algorithms, we can move beyond the trade-offs that have historically constrained 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.

6 Conclusion and Future Work

Designing effective static analysis tools for dynamic languages like Python presents a unique set of challenges. While static analysis has long been a cornerstone of software reliability for languages like Java, its application to Python has historically lagged. This discrepancy is particularly critical given the dominance of Python in data science and machine learning domains, where code quality is often compromised by ad-hoc development practices. In this dissertation, we addressed these challenges by developing novel static analysis techniques, establishing rigorous benchmarking frameworks, and empirically evaluating emerging learning-based approaches.

Our first contribution addresses the prevalence of unstructured Jupyter Notebooks. While intended for literate programming, notebooks often end up being "messy" scripts that hinder collaboration. To mitigate this, we developed HeaderGen, which automatically augments code with semantic headers and documentation. This tool is built upon our novel Extended Assignment Graph, which introduces flow-sensitivity to resolve calls to external machine learning libraries. By mapping these calls to a high-level taxonomy of operations, HeaderGen restores a narrative structure to the code. Our evaluation demonstrates that this approach not only outperforms existing tools like PyCG in call-site recognition but also significantly improves navigation and comprehension for practitioners.

Second, we addressed the methodological gap in evaluating type inference, the foundational layer of Python analysis. To provide specificity, we introduced TypeEvalPy, a micro-benchmark grounded in manually curated snippets with 845 verified annotations. To achieve scale, we augmented this with an automated generation engine, thereby expanding the evaluation set to over 77,000 annotations. Using this unified pipeline, we assessed state-of-the-art static analyzers alongside learning-based models, including LLMs. Our results highlight the superior adaptability of LLMs: models like `mistral-large` successfully resolved precise types in complex constructs, such as decorators and generators, where static tools often defaulted to conservative approximations. While this performance signals a potential paradigm shift toward data-driven analysis, widespread deployment must still overcome barriers related to resource intensity, determinism, and security.

Third, we extended our evaluation to call-graph analysis by introducing SWARM$_{\text{CG}}$, a multi-language benchmark suite. Unlike our findings in type inference, here traditional static analyzers consistently outperformed LLMs, particularly in handling complex inheritance and indirect data flows. While models like `mistral-large` showed competitive utility in resolving specific dynamic behaviors, such as dictionary updates and dynamic object keys, they ultimately struggle with the global reasoning required for structural analysis. This contrast suggests that while LLM pre-training aligns well with local token prediction (types), it is less effective at

capturing the non-local dependencies necessary for accurate call-graph construction.

In conclusion, this work demonstrates that accurate call-graph construction and reliable type inference are essential for supporting developers in downstream tasks such as program comprehension. The tools and frameworks presented in this dissertation illustrate how empirical benchmarks can guide the refinement of analysis techniques and quantify their trade-offs. These findings highlight LLMs as a complementary paradigm, suggesting that the most effective solutions will combine the determinism of traditional static analysis with the flexibility of learning-based methods.

Future Work. Several directions for future research emerge from the findings of this dissertation:

- **Systematic Integration via Feedback Loops:** Integrating static analysis with LLMs requires moving beyond simple prompting to iterative feedback loops. In this architecture, static analysis can serve as a *validator* to constrain LLM hallucinations, while the LLM acts as an ambiguity resolver for dynamic features that static tools cannot easily model. This hybrid feedback loop targets the simultaneous improvement of precision, by strictly filtering invalid predictions, and recall, by uncovering dynamic behavior that rule-based systems miss.

- **Scalable and Leakage-Resilient Benchmarking:** As LLMs are increasingly trained on large corpora of open-source code, static benchmarks are at risk of contamination through data leakage, where evaluation instances or closely related variants appear in training data. Such overlap can inflate measured performance by rewarding memorization rather than genuine generalization. Future research should therefore pivot toward live, evolving evaluation frameworks that utilize synthetic code generation and systematic mutation. By continuously synthesizing novel test cases and transforming existing ones, these frameworks can produce streams of evaluation data that reduce direct overlap with training corpora. This shift from fixed datasets to continuous assessment better emphasizes reasoning and generalization capabilities, while acknowledging that complete immunity to memorization cannot be guaranteed.

- **Agentic Workflows:** In this dissertation, LLMs are evaluated in a single-message interaction setting. With the emergence of agentic workflows, future research should investigate agents capable of self-exploring a codebase, navigating files, and utilizing auxiliary tools like text manipulators to on-the-fly static analyzers to iteratively refine their understanding. This approach moves beyond static contexts, enabling the construction of comprehensive call graphs and type assignments in large-scale projects.

PyCG Intermediate Representation

This appendix provides a detailed breakdown of the Intermediate Representation (IR) used by PyCG [SSL+21a]. This IR serves as the foundation for the construction of the Extended Assignment Graph (EAG). To illustrate the concrete state of the analysis, we use the following Python code snippet as a running example throughout this chapter.

A.1 Running Example Source Code

Listing A.1 defines a simple class hierarchy involving inheritance and a method call. This minimal example is sufficient to demonstrate how PyCG models namespaces, static scoping, inheritance resolution, and data flow. Note that the `pass` keyword is a null operation that serves as a placeholder in the body of class `B`. It indicates that `B` does not introduce any new attributes or methods and inherits all behavior from its superclass `A`.

A.2 Analysis State Domains

After analyzing the source code, PyCG generates an internal state composed of four distinct domains: the Namespace, Scope, Class Hierarchy, and the Assignment Graph. The following sections visualize the concrete state for each domain corresponding to the running example.

```python
# module: main
class A:
    def func(self, x):
        return x

class B(A):
    pass

b = B()
res = b.func(10)
```

Listing A.1: Running Example: Python Source Code

A.2.1 Namespace Domain

The Namespace domain defines the program's identifiers and their fully qualified paths (objects). This disambiguates variable names based on their location (e.g., distinguishing a local variable `x` from a global one). Note that abstract values, such as the literal `10`, are also represented as object nodes.

```
1  // Identifiers are pairs of (name, type)
2  d_main = ("main", mod)
3  d_A = ("A", cls)
4  d_func = ("func", func)
5  d_x = ("x", var)
6  d_B = ("B", cls)
7  d_b = ("b", var)
8  d_res = ("res", var)
9
10 // Objects are fully qualified paths (sequences of definitions)
11 o_main = [d_main]
12 o_A = [d_main, d_A]
13 o_func = [d_main, d_A, d_func]
14 o_x = [d_main, d_A, d_func, d_x]
15 o_B = [d_main, d_B]
16 o_res = [d_main, d_res]
```

Listing A.2: Concrete State: Namespace Domain

A.2.2 Scope Domain

The Scope domain captures the static nesting structure of the program. It maps a definition to the set of definitions strictly contained within it.

```
1  // Maps a definition to the set of definitions strictly inside it
2  s = {
3      d_main ↦ {d_A, d_B, d_b, d_res},
4      d_A ↦ {d_func},
5      d_func ↦ {d_self, d_x, d_ret}, // implicit return var
6      d_B ↦ ∅
7  }
```

Listing A.3: Concrete State: Scope Domain

A.2.3 Class Hierarchy Domain

The Class Hierarchy domain models inheritance. It maps a class object to its Method Resolution Order (MRO), allowing the analysis to correctly resolve attribute lookups (like `b.func`) to the parent class `A`.

```
1  // Maps a class object to its Method Resolution Order (MRO)
2  h = {
3      o_B ↦ [o_A]   // B inherits from A
4  }
```

Listing A.4: Concrete State: Class Hierarchy Domain

A.2.4 Assignment Graph Domain

The Assignment Graph (π) captures the data flow. It maps an object to the set of objects assigned to it. This includes explicit variable assignments, argument passing, and return values.

```
1  // Maps an object to the set of objects assigned to it
2  π = {
3      // Instantiation: b points to an instance of B
4      o_b ↦ {o_B},
5
6      // Call "b.func(10)":
7      // 1. Argument passing: param x points to literal object 10
8      o_x ↦ {o_10},
9
10     // 2. Return flow: virtual return var points to x
11     o_func.ret ↦ {o_x},
12
13     // 3. Result assignment: res points to the function return
14     o_res ↦ {o_func.ret}
15 }
```

Listing A.5: Concrete State: Assignment Graph Domain

A.3 Definition–Use Chains and Flow Sensitivity

HEADERGEN achieves flow sensitivity by distinguishing between different assignments to the same identifier based on their source code location. This is realized through a Definition–Use Chain (DUC) analysis computed over the Python AST using *Beniget* [ser22].

Using the running example from Listing A.1 (Appendix A), we demonstrate how HEADERGEN identifies definitions and uses to construct precise graph nodes.

A.3.1 DUC Analysis Output

The DUC analysis identifies the definition site for every variable usage, respecting Python's lexical scoping and name resolution rules. Listing A.6 shows the chains extracted for the identifiers in the running example.

```
1  // Format: Definition(Variable, Line) -> [Use(Line), ...]
2
3  // 1. Class Definitions
4  Def(A, 2) -> [Use(6)]   // 'A' defined at L2, used as base class at L6
5  Def(B, 6) -> [Use(9)]   // 'B' defined at L6, instantiated at L9
6
7  // 2. Variable Definitions
8  Def(b, 9) -> [Use(10)]  // 'b' defined at L9, receiver of call at L10
9
10 // 3. Parameter Definitions
11 Def(x, 3) -> [Use(4)]   // 'x' defined as param at L3, used in return at L4
```

Listing A.6: Extracted Definition–Use Chains

A.3.2 Flow-Sensitive Node Construction

When constructing the Extended Assignment Graph (EAG), HeaderGen queries the DUC index. Instead of creating a generic node for an identifier (which would merge all values assigned to it globally), it creates *versioned nodes* indexed by the definition line number.

For the instantiation assignment `b = B()` at line 9:

1. The transition rule identifies the assignment to `b`.
2. The DUC is queried to find that this definition of `b` occurs at Line 9.
3. A versioned node `(b, 9)` is created (or retrieved).
4. The edge is drawn from the value source to this specific version.

Listing A.7 visualizes the resulting nodes and edges for this specific operation.

```
// Context: Assignment "b = B()" at Line 9

// 1. Node Creation (Identifier, Location)
n_{LHS} = (d_b, 9)       // Node for 'b' defined at L9
n_{RHS} = (d_B, 9)       // Node for 'B' usage at L9

// 2. Flow Edge Creation
// Captures that the specific version of 'b' at L9 points to class B
Edge: (d_b, 9) ← (d_B, 9)

// Note: If 'b' were redefined at Line 11 (e.g., b = 100),
// a new node (d_b, 11) would be created.
// Any subsequent usage of 'b' would link to either (d_b, 9) or (d_b, 11)
// depending on which definition reaches that use.
```

Listing A.7: Flow-Sensitive Nodes in EAG

B Prompts

B.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 B.1: 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 B.2: 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 B.3: 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 B.4: Example of an actual full prompt for type inference in question-answer format used in the study

B.1.1 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 B.5: 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 B.6: 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 B.7: 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 B.8: Prompt for flow-sensitive call-graph task in question-answer format (Part 2 of 2)

B.2 Example Responses

B.2.1 Type inference Output of mistral-large for test case args/assigned_call

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


def func(a):
    return a()


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

Listing B.9: Source code

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

Listing B.10: Raw LLM response

```json
[{
    "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 B.11: Ground truth

```json
[{
    "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 B.12: Parsed JSON response

B.2.2 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 B.13: Source code

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

Listing B.14: Ground truth

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

Listing B.15: Raw LLM response

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

Listing B.16: Parsed JSON response

B.2.3 Callgraph Output of phi3-mini-it-3.8b for test case args/assign_return

```
def return_func(args1):
    return args1


def func1():
    pass


def func2():
    pass

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

Listing B.17: Source code

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

Listing B.18: Ground truth

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

Listing B.19: Raw LLM response

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

Listing B.20: Parsed JSON response

Bibliography

[ABDG20a] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 91–105, London UK, June 2020. ACM.

[ABDG20b] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. PLDI 2020, pages 91–105, New York, NY, USA, 2020. Association for Computing Machinery.

[AHH+23] Gábor Antal, Péter Hegedűs, Zoltán Herczeg, Gábor Lóki, and Rudolf Ferenc. Is javascript call graph extraction solved yet? a comparative study of static and dynamic tools. *IEEE Access*, 11:25266–25284, 2023.

[ANC+20] Marjan Adeli, Nicholas Nelson, Souti Chattopadhyay, Hayden Coffey, Austin Henley, and Anita Sarma. Supporting Code Comprehension via Annotations: Right Information at the Right Time and Place. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–10, August 2020.

[BGG14] Vera Barstad, Morten Goodwin, and Terje Gjøsæter. Predicting source code quality with static analysis and machine learning. In *Norsk IKT-konferanse for forskning og utdanning*, 2014.

[BGK25] Leonardo Berti, Flavio Giorgi, and Gjergji Kasneci. Emergent Abilities in Large Language Models: A Survey, March 2025.

[Bus23] Alessio Buscemi. A Comparative Study of Code Generation using ChatGPT 3.5 across 10 Programming Languages, August 2023. arXiv:2308.04477 [cs].

[CCY+24] Zhifei Chen, Lin Chen, Yibiao Yang, Qiong Feng, Xuansong Li, and Wei Song. Risky Dynamic Typing-related Practices in Python: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.*, 33(6):140:1–140:35, June 2024.

[Coh60] Jacob Cohen. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46, April 1960.

[CTJ+21] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens

Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[CZLZ23] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering in Large Language Models: A comprehensive review, October 2023.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, May 2019.

[DDD+25] Doaa Dalaq, Kaniz Fatima Daya, Alaa Dalaq, Muhammed Nazmul Arefin, and Mahmood Khan Niazi. A Systematic Literature Review on Static Application Security Testing (SAST) Tools: Evaluation, Benchmarks, Challenges, and Future Directions. In *Proceedings of the 2025 29th International Conference on Evaluation and Assessment in Software Engineering Companion*, EASE Companion '25, pages 162–168, New York, NY, USA, December 2025. Association for Computing Machinery.

[DGP22a] Luca Di Grazia and Michael Pradel. 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*, pages 209–220, Singapore Singapore, November 2022. ACM.

[DGP22b] Luca Di Grazia and Michael Pradel. 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, pages 209–220, New York, NY, USA, November 2022. Association for Computing Machinery.

[Dif] Differences between PyPy and CPython — PyPy documentation.

[DPHZ23] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient Finetuning of Quantized LLMs, May 2023.

[ECM15] ECMA. ECMA-262, 2015.

[EWDD22] Will Epperson, April Wang, Robert DeLIne, and Steven Drucker. Strategies for Reuse and Sharing among Data Scientists in Software Teams. In *ICSE 2022*, May 2022.

[FGH+23] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large Language Models for Software Engineering: Survey and Open Problems. https://arxiv.org/abs/2310.03533v4, October 2023.

[FSS+13] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In

2013 35th International Conference on Software Engineering (ICSE), pages 752–761, May 2013. ISSN: 1558-1225.

[GKK18] Afshin Gholamy, Vladik Kreinovich, and Olga Kosheleva. Why 70/30 or 80/20 Relation Between Training and Testing Sets: A Pedagogical Explanation. *Departmental Technical Reports (CS)*, February 2018.

[GTS+22] Konstantin Grotov, Sergey Titov, Vladimir Sotnikov, Yaroslav Golubev, and Timofey Bryksin. A large-scale comparison of Python code in Jupyter notebooks and scripts. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, pages 353–364, New York, NY, USA, October 2022. Association for Computing Machinery.

[Hal22] Dave Halter. Jedi - an awesome autocompletion, static analysis and refactoring library for Python, September 2022.

[HBBA18] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 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*, ESEC/FSE 2018, page 152–162, New York, NY, USA, 2018. Association for Computing Machinery.

[HCC+24] Yuan Huang, Yinan Chen, Xiangping Chen, Junqi Chen, Rui Peng, Zhicao Tang, Jinbo Huang, Furen Xu, and Zibin Zheng. Generative Software Engineering, April 2024. arXiv:2403.02583 [cs].

[HYC+24] Kaifeng Huang, Yixuan Yan, Bihuan Chen, Zixin Tao, and Xin Peng. Scalable and Precise Application-Centered Call Graph Construction for Python, September 2024. arXiv:2305.05949 [cs].

[HZL+23] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review, September 2023.

[JDH+24a] Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, and Deyi Xiong. A Comprehensive Evaluation of Quantization Strategies for Large Language Models, June 2024. arXiv:2402.16775 [cs].

[JDH+24b] Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, and Deyi Xiong. A Comprehensive Evaluation of Quantization Strategies for Large Language Models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12186–12215, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[Jet26] JetBrains. JetBrains/intellij-community, February 2026. original-date: 2011-09-30T13:33:05Z.

[JMT09] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, page 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[KES20] A. P. Koenzen, N. A. Ernst, and M.-A. D. Storey. Code Duplication and Reuse in Jupyter Notebooks. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–9, August 2020.

[KNR+17] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. CogniCrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936, October 2017.

[Knu84] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, January 1984.

[KPSB21] Sriteja Kummita, Goran Piskachev, Johannes Späth, and Eric Bodden. Qualitative and Quantitative Analysis of Callgraph Algorithms for Python. In *2021 International Conference on Code Quality (ICCQ)*, pages 1–15, March 2021.

[KRA+18] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 1–11, New York, NY, USA, April 2018. Association for Computing Machinery.

[LDN25] Ziyang Li, Saikat Dutta, and Mayur Naik. Iris: Llm-assisted static analysis for detecting security vulnerabilities, 2025.

[LGH24] Jiedong Lang, Zhehao Guo, and Shuyu Huang. A Comprehensive Study on Quantization Techniques for Large Language Models. In *2024 4th International Conference on Artificial Intelligence, Robotics, and Communication (ICAIRC)*, pages 224–231, December 2024.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java Points-to Analysis Using Spark. In Görel Hedin, editor, *Compiler Construction*, pages 153–169, Berlin, Heidelberg, 2003. Springer.

[LHZQ23a] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 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*, ESEC/FSE 2023, page 2107–2111, New York, NY, USA, 2023. Association for Computing Machinery.

[LHZQ23b] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models, November 2023.

[LKF20] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated Unit Test Generation for Python. volume 12420, pages 9–24. 2020. arXiv:2007.14049 [cs].

[LNB+25] Walter Lucas, Rafael Nunes, Rodrigo Bonifácio, Fausto Carvalho, Ricardo Lima, Michael Silva, Adriano Torres, Paola Accioly, Eduardo Monteiro, and João Saraiva. Understanding the adoption of modern javascript features: An empirical study on open-source systems. *Empirical Softw. Engg.*, 30(4), May 2025.

[LWQ22] Li Li, Jiawei Wang, and Haowei Quan. Scalpel: The python static analysis framework, 2022.

[LXM24] Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. Reducing Static Analysis Unsoundness with Approximate Interpretation. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1165–1188, June 2024.

[MAH+24] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. Effectiveness of ChatGPT for Static Analysis: How Far Are We? In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware 2024, pages 151–160, New York, NY, USA, July 2024. Association for Computing Machinery.

[MLPG22a] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4Py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 2241–2252, New York, NY, USA, 2022. Association for Computing Machinery.

[MLPG22b] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4Py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 2241–2252, New York, NY, USA, July 2022. Association for Computing Machinery.

[MOM20a] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static type analysis by abstract interpretation of python programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[MOM20b] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Value and allocation sensitivity in static python analyses. In *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, pages 8–13, 2020.

[NG25] Dibri Nsofor and Ben Greenman. Toward a Corpus Study of the Dynamic Gradual Type, March 2025. arXiv:2503.08928 [cs] version: 1.

[NMM24] Kinari Nishiura, Shuto Misawa, and Akito Monden. Analyzing class usage in javascript programs. In *Proceedings of the 2024 The 6th World Symposium on Software Engineering (WSSE)*, WSSE '24, page 139–143, New York, NY, USA, 2024. Association for Computing Machinery.

[NSB22] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, pages 532–543, New York, NY, USA, July 2022. Association for Computing Machinery.

[OGRG25] Jonhnanthan Oliveira, Rohit Gheyi, Márcio Ribeiro, and Alessandro Garcia. Bugs in the Shadows: Static Detection of Faulty Python Refactorings, July 2025. arXiv:2507.01103 [cs].

[ORV+24] Conor O'Brien, Daniel Rodriguez-Cardenas, Alejandro Velasco, David N. Palacio, and Denys Poshyvanyk. Measuring Emergent Capabilities of LLMs for Software Engineering: How Far Are We?, November 2024.

[PGL+22] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: A hybrid type inference approach for python. In *Proceedings of the 44th International Conference on*

Software Engineering, ICSE '22, pages 2019–2030, New York, NY, USA, 2022. Association for Computing Machinery.

[PMBF19] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 507–517, Montreal, QC, Canada, May 2019. IEEE.

[QCL22] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. Eliciting Best Practices for Collaboration with Computational Notebooks. *Proceedings of the ACM on Human-Computer Interaction*, 6(CSCW1):87:1–87:41, April 2022.

[RPN20] Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine Learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence, March 2020. arXiv:2002.04803 [cs].

[RSBB22] Dhivyabharathi Ramasamy, Cristina Sarasua, Alberto Bacchelli, and Abraham Bernstein. Workflow analysis of data science code in public GitHub repositories. *Empirical Software Engineering*, 28(1):7, November 2022.

[RSR+23] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, September 2023.

[RTH18] Adam Rule, Aurélien Tabard, and James D. Hollan. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 1–12, New York, NY, USA, April 2018. Association for Computing Machinery.

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

[RWSI24] Sanka Rasnayaka, Guanlin Wang, Ridwan Shariffdeen, and Ganesh Neelakanta Iyer. 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, pages 111–118, New York, NY, USA, September 2024. Association for Computing Machinery.

[SEP+23] Lukas Seidel, Sedick David Baker Effendi, Xavier Pinho, Konrad Rieck, Brink van der Merwe, and Fabian Yamaguchi. Learning type inference for enhanced dataflow analysis, 2023.

[ser22] serge-sans-paille. Gast, Beniget!, April 2022.

[SFY+23] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, Hanwei Qian, Yang Liu, and Zhenyu Chen. Automatic code summarization via chatgpt: How far are we?, 2023.

[SIB+24] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli,

Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. The Prompt Report: A Systematic Survey of Prompting Techniques, July 2024. arXiv:2406.06608 [cs].

[SM22] Sheeba Samuel and Daniel Mietchen. Computational reproducibility of Jupyter notebooks from biomedical publications. *arXiv preprint arXiv:2209.04308*, 2022.

[SP22] Stefan Schott and Felix Pauck. Benchmark Fuzzing for Android Taint Analyses. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 12–23, October 2022. ISSN: 2470-6892.

[SRR+25] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Hengbo Tong, Swarna Das, Earl T. Barr, and Wei Le. To err is machine: Vulnerability detection challenges llm reasoning, 2025.

[SSL+21a] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. PyCG: Practical Call Graph Generation in Python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1646–1657, Madrid, Spain, May 2021. IEEE.

[SSL+21b] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. PyCG: Practical Call Graph Generation in Python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1646–1657, May 2021.

[Sta22] Pyright, static type checker for Python, September 2022.

[SVB21] Ashwin Prasad Shivarpatna Venkatesh and Eric Bodden. Automated cell header generator for Jupyter notebooks. In *Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis*, AISTA 2021, pages 17–20, New York, NY, USA, July 2021. Association for Computing Machinery.

[SVSC+24] Ashwin Prasad Shivarpatna Venkatesh, Samkutty Sabu, Mouli Chekkapalli, Jiawei Wang, Li Li, and Eric Bodden. Static Analysis Driven Enhancements for Comprehension in Machine Learning Notebooks, June 2024. arXiv:2301.04419 [cs].

[SVSM+24] Ashwin Prasad Shivarpatna Venkatesh, Samkutty Sabu, Amir M. Mir, Sofia Reis, and Eric Bodden. The Emergence of Large Language Models in Static Analysis: A First Look through Micro-Benchmarks, February 2024. Issue: arXiv:2402.17679 arXiv:2402.17679 [cs].

[SVSS+25] Ashwin Prasad Shivarpatna Venkatesh, Rose Sunil, Samkutty Sabu, Amir M. Mir, Sofia Reis, and Eric Bodden. An empirical study of large language models for type and call graph analysis in Python and JavaScript. *Empirical Software Engineering*, 30(6):167, September 2025.

[SVSW+23] Ashwin Prasad Shivarpatna Venkatesh, Samkutty Sabu, Jiawei Wang, Amir M. Mir, Li Li, and Eric Bodden. TypeEvalPy: A Micro-benchmarking Framework for Python Type Inference Tools, December 2023.

[SVSW+24] Ashwin Prasad Shivarpatna Venkatesh, Samkutty Sabu, Jiawei Wang, Amir M. Mir, Li Li, and Eric Bodden. TypeEvalPy: A Micro-benchmarking Framework for Python Type Inference Tools. In *Proceedings of the 2024 IEEE/ACM*

46th International Conference on Software Engineering: Companion Proceedings, ICSE-Companion ’24, pages 49–53, New York, NY, USA, May 2024. Association for Computing Machinery.

[SVWLB23a] Ashwin Prasad Shivarpatna Venkatesh, Jiawei Wang, Li Li, and Eric Bodden. Enhancing Comprehension and Navigation in Jupyter Notebooks with Static Analysis. pages 391–401. IEEE Computer Society, March 2023.

[SVWLB23b] Ashwin Prasad Shivarpatna Venkatesh, Jiawei Wang, Li Li, and Eric Bodden. Enhancing Comprehension and Navigation in Jupyter Notebooks with Static Analysis. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 391–401. IEEE Computer Society, March 2023.

[SVWLB23c] Ashwin Prasad Shivarpatna Venkatesh, Jiawei Wang, Li Li, and Eric Bodden. Enhancing Comprehension and Navigation in Jupyter Notebooks with Static Analysis. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 391–401. IEEE Computer Society, March 2023.

[UKG02] Secil Ugurel, Robert Krovetz, and C Lee Giles. What’s the code? automatic classification of source code archives. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 632–638, 2002.

[WDS+20] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-Art Natural Language Processing, October 2020. Pages: 38–45 original-date: 2018-10-29T13:56:00Z.

[Web10] WebKit. 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`, 2010.

[WFM+22] Julia Wagemann, Federico Fierli, Simone Mantovani, Stephan Siemen, Bernhard Seeger, and Jörg Bendix. Five guiding principles to make jupyter notebooks fit for earth observation data education. *Remote Sensing*, 14(14):3359, 2022.

[WKLZ20a] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. Assessing and restoring reproducibility of Jupyter notebooks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 138–149, Virtual Event Australia, December 2020. ACM.

[WKLZ20b] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. Restoring reproducibility of Jupyter notebooks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE ’20, pages 288–289, New York, NY, USA, June 2020. Association for Computing Machinery.

[WLZ20] Jiawei Wang, Li Li, and Andreas Zeller. Better code, better sharing: On the need of analyzing jupyter notebooks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER ’20, pages 53–56, New York, NY, USA, June 2020. Association for Computing Machinery.

[WWD+22] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael Muller, Soya Park, Justin D. Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks. *ACM Transactions on Computer-Human Interaction*, 29(2):17:1–17:33, January 2022.

[YBLK22] Chenyang Yang, Rachel A Brower-Sinning, Grace A Lewis, and Christian Kästner. Data leakage in notebooks: Static detection and better processes. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[YMH22a] Yi Yang, Ana Milanova, and Martin Hirzel. Complex Python Features in the Wild. 2022.

[YMH22b] Yi Yang, Ana Milanova, and Martin Hirzel. Complex python features in the wild. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 282–293, New York, NY, USA, 2022. Association for Computing Machinery.

[ZFX+23] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. A Survey on Large Language Models for Software Engineering, December 2023.

[ZH17] Shaul Zevin and Catherine Holzem. Machine learning based source code classification using syntax oriented features. *arXiv preprint arXiv:1703.07638*, 2017.

[ZML+22] Ge Zhang, Mike A Merrill, Yang Liu, Jeffrey Heer, and Tim Althoff. Coral: Code representation learning with weakly-supervised transformers for analyzing data analysis. *EPJ Data Science*, 11(1):14, 2022.

[ZNC+23] Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. Towards an Understanding of Large Language Models in Software Engineering Tasks, August 2023.