Large Language Model-Aided Partial Program Dependence Analysis

Xiaokai Rong*
University of Texas at Dallas
Dallas, USA
xiaokai.rong@utdallas.edu

Aashish Yadavally*
University of Texas at Dallas
Dallas, USA
aashish.yadavally@utdallas.edu

Tien N. Nguyen
University of Texas at Dallas
Dallas, USA
tien.n.nguyen@utdallas.edu

Abstract

Dependence analysis (DA) plays a critical role in software engineering, from code optimization to debugging. It is traditionally limited to scenarios where entire source code is available. In practice, however, developers often encounter incomplete or partial code snippets, as in StackOverflow (S/O) forums or during modular development, where program constructs are missing. This presents challenges for DA tools, which rely on syntactic and semantic correctness to correctly identify dependencies. Thus, existing DA tools for partial code often face trade-offs in precision and recall.

In this work, we introduce LλMDA, a framework that addresses these limitations by leveraging large language models (LLMs) as context augmenters to enrich partial code snippets with the program elements required for enabling such analyses. Through our evaluation, we showed that LλMDA exhibits high correctness and completeness guarantees, yielding a higher recall than traditional approaches, and a higher precision than learning-based approaches. Overall, LλMDA improves over all baselines in partial program dependence analysis by 5%-265% and 16%-331% across S/O benchmarks. Moreover, we show LλMDA's effectiveness in providing exception handling suggestions as well as exception-flow analysis.

CCS Concepts

• **Computing methodologies** → **Neural networks**; • **Software and its engineering**;

Keywords

AI4SE, Large Language Models, Partial Program Analysis, Partial Program Dependence Analysis

ACM Reference Format:
Xiaokai Rong, Aashish Yadavally, and Tien N. Nguyen. 2026. Large Language Model-Aided Partial Program Dependence Analysis. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773119

1 Introduction

Dependence analysis (DA) is a fundamental technique in program analysis that examines the relationships between different program elements. Program dependence graph (PDG) [15] is a standard representation used to model such dependencies. They are useful in understanding program behaviors and serve as the basis for multiple applications, including optimization [15, 21], slicing [37], debugging [29], testing [42], and model checking [35].

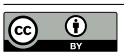
Typically, dependence analysis assumes access to the complete codebase, using syntactic and semantic information to construct an accurate model of dependencies. However, in many real-world scenarios, only partial programs (formally defined in Section 3.1) are available–whether due to modular development, privacy constraints, or coming from online forums (e.g., StackOverflow). Specifically, these partial programs often exhibit several challenges: (1) missing variable declarations, which result in incomplete variable-level dependencies (*e.g.*, def-use/use-def chains); (2) unresolved data types, which hinder type-sensitive analyses such as alias resolution and field access tracking; (3) unknown parameters, which can mislead dependence analysis; (4) missing `import` statements, leading to unresolved API elements and incomplete call/return flow information; and (5) absent exception flows, further reducing the accuracy of control and data flow modeling. In such cases, it is not always possible for compiler-based dependence analysis approaches to completely disambiguate the syntactic constructs. As a result, they are rendered *ineffective* for partial programs. Thus, it is pivotal to *enhance DA tools to build an accurate model of dependencies for partial programs* and broaden their utility in real-world scenarios.

Precision–Recall Conundrum. When applied to complete code, classical DA tools (*e.g.*, Joern [4]) correctly identify most or all dependencies, achieving a *high precision and recall*. We denote this in Figure 1 (*left*) with ♦. However, since DA tools prioritize caution over assumption, ambiguities arising from missing information in partial code weaken their ability to fully identify dependencies. As a result, some dependencies between program elements are *missed*, leading to a *high precision but lower recall* (denoted by ♦).

To address these challenges, recent work [23, 39] has leveraged large language models (LLMs) to "predict" the dependencies among program elements. The core principles driving these learning-based approaches are: *first*, that missing type-specific information in partial code, required for correctly identifying semantic dependencies, can *implicitly* be learned in the latent space during the (pre)training process. *Second*, that LLMs are useful in scenarios where *low* levels of imprecision are tolerable. By design, employing LLMs to directly analyze dependencies in partial code may yield a *higher recall* than DA tools. Nonetheless, this would be at the cost of a *lower precision*, as the LLMs do not provide correctness guarantees (denoted by ❄).

Our Approach. Addressing this precision-recall trade-off in partial code remains challenging yet critical. To this end, we advocate for a novel paradigm called **predictive dependence analysis**

*Both authors contributed equally to this research.



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773119

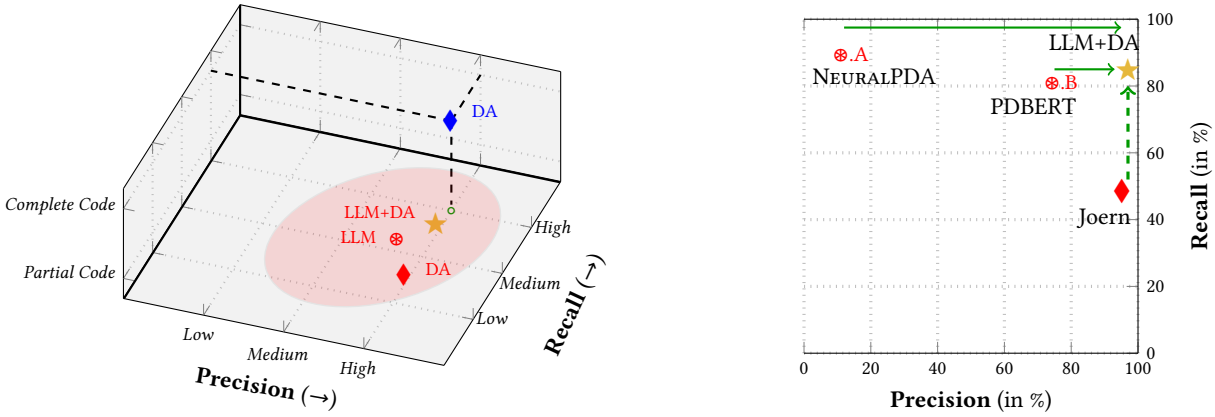


Figure 1: (*left*) A theoretical framework representing the efficacy of program dependence analysis approaches for *partial* and *complete* code, without LLM (⧫), and with LLM (⊛, ★). A high precision denotes the *correct* identification of dependence, and a high recall, the identification of *all dependencies* between program elements. (*right*) Our experimental results for partial Java programs in COSTER-SO benchmark in alignment with the theoretical framework (Section 6). Here, → and ⇢ denote improvements in Precision and Recall, respectively, with our proposed framework, LλMDA.

(LλMDA), which leverages LLMs in conjunction with the traditional DA tools. Rather than using LLMs to directly predict dependencies by *implicitly* inferring the missing program elements (as in LLM ⊛), LλMDA leverages an LLM as a *context augmenter* to *explicitly* fill them in and disambiguate partial code. We posit that by harnessing the complementary capabilities of the *LLMs for improving coverage* (i.e., *high recall)* and *DA techniques for correctness* (i.e., *high precision)*, we can achieve optimality in partial code settings (denoted by ★). With necessary information, ★ would theoretically converge to ◦, the spatial counterpart to applying DA tools on complete code.

LλMDA operates in two phases: *Context Augmentation* and *Analysis*. In the first phase, LλMDA uses feedback from a semantic verifier (in this case, a compiler) to guide the LLM to fill-in missing variable declarations, `import` statements, and type information. Note that for other analyses, the verifier in LλMDA can alternatively be replaced with model checkers, specification verifiers, *etc.* By augmenting the given partial program P with the suggested program elements, we transform it into a contextually augmented and syntactically complete variant. We refer to this as the *approximately-complete program* P_{AC}. In the second phase, traditional DA tools are subsequently applied to P_{AC} to obtain dependencies, which are then refined to retain only those relevant to P, *thereby aligning with the initial scope of the analysis despite the augmented context.* Moreover, each project has its own requirements and standards, and P_{AC} may not always resemble the developers' version P_D when incorporating P into their projects. However, P_{AC} is designed to capture the essential dependencies and syntactic structure necessary for preliminary analyses, allowing LλMDA to proceed with a reliable approximation of the program's intended functionality.

We implemented LλMDA framework using state-of-the-art LLMs, including Claude 3.5 Sonnet and GPT-4o, along with the advanced dependence analysis tool Joern [4]. The LLMs are enhanced with feedback from a semantic verifier, which we implemented using a compiler. LλMDA selectively retains *only* compiler messages helpful in disambiguating unknown/unresolved identifiers. Moreover, our feedback prompt guides the LLM to synthesize only *minimal, dependency-relevant* code needed for reliable DA. Our experimental results on partial program dependence analysis, illustrated in the Precision-Recall plot in Figure 1 (*right*), corroborate with the theoretical framework outlined in Figure 1 (*left*). LλMDA improves over the learning-based approaches (⊛.A and ⊛.B) in *precision* for the StatType-SO and COSTER-SO benchmarks by 18.6%–558.6% and 32.4%–733.6%, respectively. We also observed improvements in *recall* over traditional DA technique (⧫) by 26.2% and 71.3%. Overall, LλMDA improves over all baselines in partial program dependence analysis by 5%–265.3% and 15.9%–331% (Section 6). We also show LλMDA's applicability and effectiveness in two downstream tasks: (a) analyzing exception flows in partial program to handle such suggestions (Section 9.1), and (b) suggesting the exceptions to be handled in partial programs (Section 9.2).

Novelty. This work makes the following key contributions:

- *Predictive Dependence Analysis.* A paradigm for analyzing dependencies in (in)complete code with correctness and completeness.
- We carry out a *rigorous* evaluation using two StackOverflow benchmarks, employing multiple LLMs.
- *Usefulness.* We show LλMDA's effectiveness in exception flow analysis and exception handling suggestion.

2 Motivation and Key Ideas

2.1 Dependence Analysis in Incomplete Code

Developers frequently access online forums such as StackOverflow (S/O) for quick solutions to coding tasks, often using those online code examples. While such code reuse can accelerate development, it also introduces potential risks. For instance, these examples might be outdated [31], or possess vulnerabilities [34], and may inadvertently migrate to a codebase. Thus, a thorough analysis and scrutiny of such "toxic" code snippets is crucial for maintaining the integrity and robustness of the target codebase.

Figure 2 illustrates an example from an S/O answer (post #161801-30 [1]). This was originally copied from the `LineRecordReader` class in the Hadoop project on Github [2], intending to explain the usage of

```
1  if (codec != null) {
2    in = new LineReader(codec.createInputStream(fileIn),job);
3    end = Long.MAX_VALUE;
4  } else {
5     if ( start != 0 ) {
6       skipFirstLine = true;
7       --start;
8       fileIn.seek ( start );
9     }
10    in = new LineReader(fileIn,job);
11 }
12 if (skipFirstLine) {
13   start += in.readLine (new Text(),0,(int) Math.min((long)
          Integer.MAX_VALUE, end-start));
14 }
```

Figure 2: Incomplete code from S/O post #16180130

```
1  import java.io.IOException;
2  import java.io.InputStream;
3  import org.apache.hadoop.conf.Configuration;
4  import org.apache.hadoop.fs.FSDataInputStream;
5  import org.apache.hadoop.fs.FileSystem;
6  import org.apache.hadoop.fs.Path;
7  import org.apache.hadoop.io.LongWritable;
8  import org.apache.hadoop.io.Text;
9  import org.apache.hadoop.io.compress.CompressionCodec;
10 import org.apache.hadoop...CompressionCodecFactory;...
11 public class LineRecordReader implements ... { ...
12   public LineRecordReader(Configuration job, FileSplit split) throws
          IOException {
13    start = split.getStart();
14    end = start + split.getLength();
15    final Path file = split.getPath();
16    compressionCodecs = new CompressionCodecFactory(job);
17    final CompressionCodec codec = compressionCodecs.getCodec(file);
18    FileSystem fs = file.getFileSystem(job);
19    FSDataInputStream fileIn = fs.open(split.getPath());
20    boolean skipFirstLine = false;
21    if (codec != null) { // copied from Figure 1
22      in = new LineReader(codec.createInputStream(fileIn),job);
23      end = Long.MAX_VALUE;
24    } else {
25      if (start != 0) {
26        skipFirstLine = true;
27         --start;
28        fileIn.seek(start);
29      }
30      in = new LineReader(fileIn,job);
31    }
32    if (skipFirstLine) {
33      start += in.readLine(new Text(),0,(int) Math.min((long)
            Integer.MAX_VALUE,end-start));
34    }
35    this.pos = start;
```

Figure 3: A snapshot of `LineRecordReader` class in Hadoop project, from where the code snippet in Figure 2 was copied

the `FileSplit` object, which requires an offset of `-1` in some use cases and not in others. Notably, this has subsequently been modified to better handle such offsets in the Hadoop project itself (not shown here), significantly impacting its behavior and usage requirements. Although this change was introduced several years ago, the code in the S/O post remains outdated. Alarmingly, this outdated snippet has since been adopted by multiple new projects [31], which may encounter unexpected issues in their implementations.

Traditional DA tools have been shown to struggle with capturing dependencies effectively in incomplete code snippets [38]. As an illustration, we used Joern [4] to analyze the data dependencies of the potentially vulnerability-injecting code snippet in Figure 2. When input as is, we found that Joern missed all data dependencies. Next, we wrapped the code snippet in a dummy

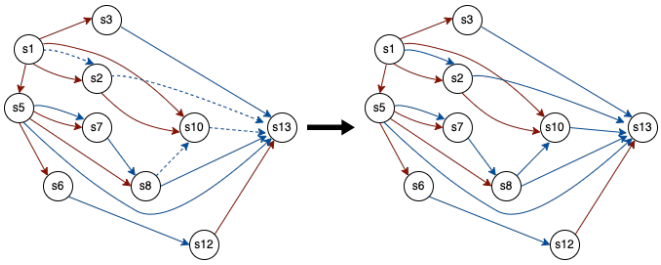


Figure 4: Augmenting partial code with relevant context helps retrieve missing dependencies (⇢ to →) with a dependence analyzer

method signature, creating a pseudo-syntactically valid program. We found that Joern captured a few data dependencies, while producing several error messages of the form *'Could not find type member. type=ABC, member=xyz'*. Here, *ABC* refers to a type name and *xyz* is the respective identifier (*e.g.*, the name of a variable/field).

To assess the precision of the detected data dependencies, we ran Joern on the complete code in the Hadoop project listed in Figure 3. Since our focus was solely on those within the original code snippet in Figure 2, we pruned the set of detected dependencies to retain the ones relevant to the incomplete code snippet. In Figure 4 (*left*), we present the respective PDG, marking the dependencies correctly predicted by Joern with a → and the missed ones with ⇢. Despite of *high precision*, we can see that edges corresponding to the variables `codec`, `filein`, `in` (missing variable declarations); and `LineReader` (unresolved APIs), were missed, resulting in *low recall*.

Control dependencies are also impacted by incomplete code. For instance, the control from a conditional statement may flow through either the `true` or `false` branch depending on the value of a variable that may be unassigned. In general, the missed dependencies illustrate the limitations of the traditional DA tools in effectively handling incomplete code. The ability to assess risks associated with integrating potentially vulnerable code is significantly hindered.

2.2 Key Ideas

In this paper, we propose, LλMDA, to provide higher correctness and completeness in the dependence analysis of partial programs.

LLMs can utilize typical usage patterns and identifier cues to generate contextually-appropriate completion for unsolved symbols in partial programs. By inferring structures, types, and dependencies, LLMs can approximate the intended semantics, producing an *approximately-complete* program. Such completions, however, can be highly project-specific with multiple valid implementations. For instance, the complete version in Hadoop (Figure 3) represents just one candidate for the partial program in Figure 2. While one can incorporate user intent in completion with LLMs, our focus is on *incorporating the information needed for dependence analysis*.

For illustration, LLMs can expand the code in Figure 2 in various ways, specifically by: (1) identifying the possible data types for `fileIn`, `codec`, `job`, `start`, `end`, *etc.*; (2) initializing these variables; (3) adding necessary `import` statements. Figure 5 shows an approximately-complete version generated by GPT-4o for the incomplete code in Figure 2. *This version may not perfectly match a human-completed version, but it provides sufficient structure for subsequent DA analysis.*

```
1  import java.io.IOException;
2  import java.io.InputStream;
3  import org.apache.hadoop.conf.Configuration;
4  import org.apache.hadoop.fs.FSDataInputStream;
5  import org.apache.hadoop.fs.Path;
6  import org.apache.hadoop.io.Text;
7  import org.apache.hadoop.io.compress.CompressionCodec;
8  import org.apache.hadoop...CompressionCodecFactory;...
9  public class FileProcessing implements ... {
10   public processFile (Configuration job, String fName, long start, long end,
           boolean skipFirstLine, CompressionCodec codec) throws IOException {
11     FSDataInputStream fileIn = new FSDataInputStream(fName);
12     LineReader in;
13     if (codec != null) {
14       in = new LineReader(codec.createInputStream(fileIn),job);
15       end = Long.MAX_VALUE;
16     } else {
17       if (start != 0) {
18         skipFirstLine = true;
19          --start;
20         fileIn.seek(start);
21       }
22       in = new LineReader(fileIn,job);
23     }
24     if (skipFirstLine) {
25       start += in.readLine(new Text(),0,(int) Math.min((long)
              Integer.MAX_VALUE,end-start));
26       }
27       this.pos = start;
28     }
```

Figure 5: Complete code predicted by the LLM in LλMDA

Despite their ability in generating approximately-complete programs, LLMs do not guarantee syntactic or semantic correctness. This can result in errors stemming from incorrect types or APIs from external libraries. To address this issue, prior research in code completion [9, 40] has proposed using semantic verifiers, such as compilers, to ensure the correctness of generated code. Similarly, we adopt a compiler for semantic verification, providing a feedback signal to the LLM to refine the approximately-complete programs (see Section 3.2). Alternative analyses may involve model checkers to verify properties, symbolic execution engines to explore multiple paths and identify potential runtime errors, runtime verification, or test case execution, among other methods.

Key Idea 1 (LLMs for Approximating Partial Programs). *Leverage programming patterns learned during the extensive pre-training of LLMs to create a syntactically and semantically-valid approximately-complete program.*

As seen in Figure 5, formal parameters are added to declare the variables `job`, `start`, `end`, `skipFirstLine`, and `codec`; and the associated `import` statements are also included. Interestingly, it declares `fileName` as a formal parameter and creates an instantiation of `FSDataInputStream` that takes `fileName` as an argument is initialized to `fileIn` at line 11. This statement is necessary to set up the API call `createInputStream` at line 14. In Figure 4 (*right*), we present the PDG for the approximately-complete program variant. Owing to the code populated by the LLM in LλMDA, for the lines 13-28 in Figure 5, when compared with the corresponding pruned PDG for the Hadoop code, we observed that Joern was able to produce correctly all the program dependencies. This includes dependencies that were missed in Figure 4 (*left*) for the partial code, thus helping capture more dependencies. That is, leveraging the approximately-complete program helps achieve a *high recall*, and using the DA tool to decide the dependencies helps achieve a *high precision, despite the differences with the correct intention in human-completed version.*

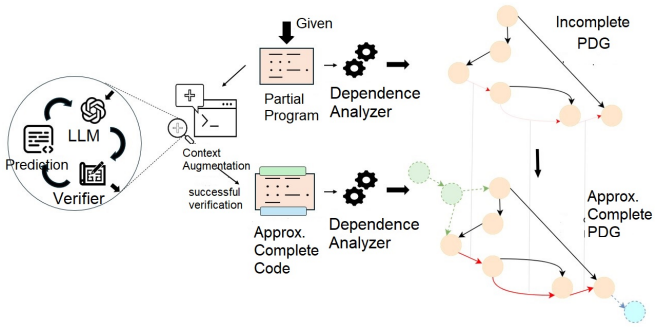


Figure 6: An overview of predictive dependence analysis framework with LλMDA

Key Idea 2 (DA ⊢ High Precision, LLM ⊢ High Recall). *Analyzing the syntactically and semantically valid, complete variant generated by an LLM for a partial program can help the* **precise** *retrieval of* **more** *(i.e., missed) dependencies.*

3 Partial Program Dependence Analysis

3.1 Important Concepts

Definition 1 (Partial Program). *A partial program P is a syntactically valid, non-empty subset of an otherwise complete program (i.e., $P \subset P_C$). The incompleteness of P arises from the presence of unknown symbols S within P (such as fields, methods, type expressions) that are originally defined in P_C (i.e., $S \in P_C$).*

The disambiguation of P involves *variable declarations, method signatures, class definitions,* `import` *statements,* `try-catch` *blocks, type casting, assignment of correct data types for typesafe (for local variables, return values, fields), code hardening, interface definitions, etc.*

Definition 2 (Context). *The context C for a partial program P comprises the additional program elements required to resolve all unknown symbols S within P and disambiguate it.*

Definition 3 (Approximately-Complete Program). *An approximately-complete program P_{AC} is the result of augmenting a partial program P with context C (generated by an LLM in LλMDA), such that P_{AC}=P+C. This integration resolves all unknown symbols S within P, ensuring that P_{AC} is both syntactically and semantically valid.*

P_{AC} (as obtained from LλMDA) provides sufficient context for the DA tool, e.g., Joern, to derive the missing dependencies.

3.2 Overview

Figure 6 illustrates LλMDA for partial dependence analysis. In the first phase of LλMDA, an LLM is tasked with disambiguating a given partial code by augmenting the necessary context required to retrieve syntactically and semantically valid, complete variant (referred to as *approximately-complete programs*). The augmented context can include: (1) variable declarations, (2) type information, (3) method signatures, (4) class definitions, (5) `import` statements, among other program constructs. To ensure its correctness, we guide and validate the LLM in iterative cycles of *self-correction*, providing *only the compiler's feedback helpful in disambiguating unresolved names.* This iterative process continues until the approximately-complete program is compilable. If a specified number of iterations is reached,

manual intervention may be necessary. Depending on the analysis requirements, the compiler could be replaced with other semantic verifiers, e.g., model or type checkers, specification verifiers, etc.

In the second phase, the DA tools are applied to the approximately-complete programs to retrieve program dependencies (including statements filled-in by the LLM as context). Finally, these dependencies are pruned to retain only those relevant to the program statements present in the original partial program. With the additional context, it is possible for DA tools to identify dependencies between program elements that were originally missed due to ambiguities associated with the unknown symbols in the partial program (*i.e.*, *high recall*). Furthermore, by design, DA tools provide soundness guarantees (*i.e.*, *high precision*). As a result, LλMDA improves over applying the DA tools directly to partial programs in recall; and over the rule-based [17] or learning-based approaches in precision.

3.3 Problem Formulation

For a partial program P, let P_H represent a manually completed variant by human developers (based on a specific use case) and let P_{AC} represent the approximately-complete variant generated by the LLM in LλMDA. Given a dependence analysis tool T, let the dependence graphs produced by T for all program variants P_i be $G_i = T(P_i)$. Let $\tilde{G}_H$ and $\tilde{G}_{AC}$ denote the pruned versions of G_H and G_{AC}, respectively, such that each contains only program elements present in P. We aim to show that our tool's design ensures that:

$$Precision(\tilde{G}_{AC}, \tilde{G}_H) \text{ is } \uparrow\uparrow \tag{1}$$

$$Recall(\tilde{G}_{AC}, \tilde{G}_H) > Recall(G, \tilde{G}_H) \tag{2}$$

4 Context Augmentation: $P \rightarrow P_{AC}$

For a given partial program P, the objective is to generate a compilable variant P_{AC} such that no program elements within P are modified. To facilitate this process, we leverage an LLM $\mathcal{M}$ and a compiler $\mathcal{V}$ (as a semantic verifier), which work in tandem iteratively to populate the necessary missing information in P.

4.1 Approximating Partial Programs with LLM

We first check if the partial program P is compilable, collecting all its errors $\mathcal{V}(P)$ if it is not. Next, in the first pass to the LLM (prompt shown in Figure 7 (*left*)), we provide the partial program P along with compiler errors, and a set of instructions describing the task.

In the interest of making P compilable, the two main objectives of the LLM in this phase are (a) code approximation, and (b) type analysis. The first task requires the LLM to fill-in essential program elements including necessary headers, `import` statements, method signatures, *etc.* which are crucial for the input program's functionality. For this process, we instruct the LLM to inspect the compiler's error message, diagnose, and subsequently attempt to rectify these errors. As seen in Figure 7 (*left*), an illustration of such unfiltered compiler outputs, this involves fixing syntax errors as well as disambiguating all unknown identifiers. To further facilitate this, we include the second task, which requires the LLM to accurately resolve types and enumerate all variables and their types, *e.g.*, (`fileIn`, `FSDataInputStream`). We expect this analysis help the LLM capture type dependencies intrinsically, which could facilitate a better identification of fully-qualified names for the unknown symbols. Finally, the LLM outputs a candidate approximately-complete program P^c_{AC} and the extracted type information $\mathcal{T}$. We adopt the few-shot setting with an exemplar, which serves as a reference for the LLM.

4.2 Refinement/Validation via Semantic Verifier

In this work, we leverage a compiler as a semantic verifier. We begin by checking the compilability of the candidate approximately-complete program P^c_{AC} generated by the LLM (Section 4.1). If P^c_{AC} does not compile, a refinement process is initiated. We collect all errors generated in a failed compilation attempt. These might cover a range of syntactic/semantic issues. Some errors, however, are tied to external dependencies, e.g., "*package does not exist*" and "*class name and file name do not match*". We discard these as the LLM fails to resolve them. If a "*symbol not found*" error does not correspond to an API, e.g., in the case of an undefined constant, we also discard it.

Next, we construct a feedback loop prompt (Figure 7 (right)) to guide the LLM to generate new candidate approximately-complete programs that address the errors identified by the compiler at the end of an iteration. The tasks and objectives of the LLM in this prompt are the same as in Section 4.1. We also include the following:

(1) original partial code snippet P
(2) LLM-generated candidate approximately-complete program at the end of the i-th iteration $P^{c(i)}_{AC}$
(3) filtered error messages from compiler

It is essential to incorporate both the original and the current modified versions, especially for longer code snippets, as that helps the LLM avoid getting stuck in a cycle of repeated errors and facilitates a more accurate understanding of the necessary corrections.

Finally, the approximately-complete program P_{AC} is $P^{c(i)}_{AC}$ if there are no more compilation errors. In contrast, if the threshold θ is reached with not all errors being resolved (i.e., the LLM failed to disambiguate the partial program P), manual intervention is needed.

5 Empirical Evaluation

To evaluate LλMDA, we seek to answer the following questions:
(RQ_1) Partial Program Dependence Analysis Effectiveness: Can LλMDA enhance dependence analysis for partial programs?
(RQ_2) Sensitivity Analysis: How many dependencies are correctly recovered after each iteration of compiler feedbacks?
(RQ_3) Program Constructs that LλMDA Completes the Code. What programming constructs does the LLM in LλMDA fill-in toward disambiguating unknown symbols?
(RQ_4) Adaptability to Exception Flow Analysis and Exception Handling Recommendations: Can LλMDA improve the analysis of exception-flows in partial Java programs? Is LλMDA useful for suggesting exceptions that need to be handled in partial programs?
(RQ_5) Ablation Study: How does LλMDA's feedback loop affect its effectiveness in capturing dependencies in partial code?
(RQ_6) Efficiency: How are LλMDA's efficiency and token costs in LLM usages compared to those of the baselines?

6 Partial Program Dependence Analysis (RQ_1)

We first evaluate how the code filled in by LλMDA helps the DA tool in better capturing the dependencies in the original, incomplete code.

6.1 Experimental Setup

6.1.1 Datasets. We selected two benchmark datasets from prior work, namely, StatType-SO [28] and COSTER-SO [32]. Both cover

Prompt for Approximating Partial Programs

Task. Given a partial Java code snippet, compiler output/errors, and the expected outcome, your task is to generate the necessary code to complete the snippet. The additional code should address the issues indicated by the compiler output and achieve the desired outcome. Focus on enhancing the code header to make the snippet a compilable Java unit, without modifying the original code body.

Partial Code Snippet (Do Not Modify):
<code>

Compiler Output/Errors:
<errors>

Expected Outcome:
Using the information provided by the compiler, the model should fill in the missing information of the partial code snippet, focusing solely on enhancing the code header. The goal is to make the partial code snippet a compilable Java unit without modifying any part of the existing code body. Your first task is to analyze the provided code, identify what are missing and complete the Java snippet. Do not modify the original code. Additionally, you do not need to write a method to solve missing API calls. After completing the code, your second task is to fill in type information
Here is an example procedure of how you should construct answer. Suppose you are given a partial code snippet
<sample code snippet>
Here is what your response is supposed to look like:
Code Approximation:
```java
... // LLM approximated code here.
```
Type Information:
... // LLM proposed type information here.
Disclaimer:
Do not include any explanation or comments.

Prompt for Incorporating Compiler Feedback for Self-Correction

Task. After incorporating the suggested code enhancements into the original code snippet, the code was recompiled, resulting in new compiler output/errors. Your task is to analyze these new errors, understand the context of both the original and modified code, and apply further modifications to correct the errors without altering the core functionality or logic of the original code.

Partial Code Snippet (Do Not Modify):
<original_code>

Modified Code with Errors:
<modified_code>

New Compiler Output/Errors:
<new_errors>
(*i.e.*, after filtering the compiler's outputs)

Expected Outcome:
Code Approximation:
```java
... // LLM approximated code here.
```
Type Information:
... // LLM proposed type information here.
Here is an example procedure of how you should construct answer.
Same as in Fig. 7 (*left*)
Disclaimer:
Do not include any explanation or comments.

Figure 7: Prompts to LLM in LλMDA for: (*left*) approximating the partial program; (*right*) providing feedback for self-correction

six Java libraries: `android`, `gwt`, `hibernate`, `joda-time`, `jdk`, and `xstream`. The authors of both benchmarks made these partial code snippets compilable by manually adding all required libraries in an iterative fashion and filling in missing details. Finally, they used Eclipse JDT compiler to validate the compilability of the code snippets. In the *Context Augmentation* phase in LλMDA, in simple terms, we aim to automate this process. Thus, we evaluated it on the incomplete S/O code snippets, using the manually filled-in code as the ground-truth. Overall, we collected 172 and 274 incomplete code snippets in two datasets, respectively, each containing 2–45 and 4–90 statements.

6.1.2 Procedure. We used the state-of-the-art Joern [4] as a DA tool. Our evaluation involved several baseline comparisons and approaches for generating PDGs from the incomplete code snippets ($D_{\text{incomplete}}$). First, we used a traditional baseline (⧫.A) by creating a pseudo-syntactically valid variant for each incomplete code snippet by wrapping around them a dummy method signature (D_{dummy}). We also applied two baselines JCoffee [17] (⧫.B) and Code2API [25] (⧫.C) to produce two complete versions for each code snippet. This modification is necessary as Joern requires syntactically valid inputs to construct PDGs. Second, we established learning-based baselines, NeuralPDA [39] (⊛.A) and PDBERT [23] (⊛.B), which were trained to "predict" the dependencies between program elements directly. Third, for LλMDA framework, we employed multiple LLMs including GPT-3.5, Claude 2, Claude 3.5 Sonnet, and GPT-4o (★.A–★.D) to obtain approximately-complete programs from the incomplete code snippets (D_{approx}). Finally, we used the manually-completed versions of the snippets (D_{complete}) to get the oracle dependencies.

For each dataset D_x ($x \in \{\text{dummy}, \text{approx}, \text{complete}\}$), we constructed the corresponding PDGs using Joern, *i.e.*, D_x + Joern $\rightarrow G_x$. Next we sliced all G_x to select the sub-PDGs (G'_x) corresponding to the statements in $D_{\text{incomplete}}$. By comparing such sub-PDGs from the partial and complete code, *i.e.*, G'_{dummy} with G'_{complete}, we assess the impact of code incompleteness on Joern's ability to build PDGs. Similarly, we quantify the impact of code approximation in LλMDA by comparing the sub-PDGs from the respective approximately-complete and complete code, *i.e.*, G'_{approx} with G'_{complete}. For the learning-based approaches, we aggregated the predicted dependencies to build the PDGs (G_{learning}) and compared with G'_{complete}.

6.1.3 Metrics. We use *Precision*, *Recall*, and *F1-Score* to measure the quality of the PDGs. A *True Positive* (TP) occurs when an edge in $G \in \{G'_{\text{dummy}}, G'_{\text{approx}}, G_{\text{learning}}\}$ along with the associated nodes matches exactly with those in G'_{complete}. *False Positives* (FP) occur when: an edge in G between two nodes is decided, but does not exist in G'_{complete}; an edge in G between two nodes is decided, but one of them does not exist in G'_{complete}. The latter is common in G'_{approx}, as the LLM sometimes tends to modify some of the original program statements to harden the code. *False Negatives* (FN) occur when an edge is absent in G, but is present in the corresponding G'_{complete}. Formally, the evaluation metrics are defined as follows: *Precision* = $\frac{\text{TP}}{\text{TP+FP}}$, *Recall* = $\frac{\text{TP}}{\text{TP+FN}}$, and *F1-Score* = $\frac{2*Precision*Recall}{Precision+Recall}$.

6.2 Empirical Results

In Table 1, we compare the PDGs constructed for partial Java code using LλMDA, with multiple LLMs (★.A–★.D), against traditional and learning-based approaches. In the case of the former, *i.e.*, when compared to the edges constructed for partial programs wrapped around in dummy method signatures (⧫.A), we saw that LλMDA *improves in recall* by 17.2%–33.5% and 27.6%–71.3% for the StatType-SO and COSTER-SO benchmarks, respectively. Interestingly, while there is a slight drop in precision by 3%–10.1% for the StatType-SO benchmark, in the case of COSTER-SO, it still goes up by 0.7%–3.1% (except for Claude 3.5 Sonnet). This difference in precision can possibly be attributed to the difference in lengths of the partial programs in StatType-SO and COSTER-SO, because with larger lengths, the LLMs contextualize the programs in COSTER-SO better

Table 1: Dependence analysis for partial code. Here, "–": NeuralPDA does not distinguish between dependence edges (RQ1)

Dataset (→) / Approach (↓)	StatType-SO									COSTER-SO								
	Data+Control			Data			Control			Data+Control			Data			Control		
	P	*R*	*F1*	*P*	*R*	*F1*	*P*	*R*	*F1*	*P*	*R*	*F1*	*P*	*R*	*F1*	*P*	*R*	*F1*
D_{dummy} + Joern (⧫.A)	99.0	69.8	81.9	98.8	68.0	80.6	100	82.6	90.4	96.2	48.5	64.5	94.9	40.6	56.9	100	71.1	83.1
D_{JCoffee} + Joern (⧫.B)	99.0	69.9	81.9	98.8	68.2	80.7	100	82.6	90.4	96.2	48.5	64.5	94.9	40.6	56.9	100	71.1	83.1
D_{Code2API} + Joern (⧫.C)	87.4	78.6	82.8	86.5	77.3	81.6	94.5	88.4	91.4	87.5	63.4	73.6	85.1	54.7	66.6	91.8	83.8	87.6
NeuralPDA (⊛.A)	14.5	92.4	25.1	–	–	–	–	–	–	11.9	89.5	21.0	–	–	–	–	–	–
PDBERT (⊛.B)	80.5	95.4	87.3	78.8	95.2	96.2	95.3	96.7	95.6	74.9	81.1	78.1	70 4	79.9	74.8	91.9	94.6	95.8
LλMDA *w/* GPT-3.5 (★.A)	94.9	81.8	87.8	94.5	79.7	86.4	98.4	96.7	97.5	96.9	71.2	82.1	96.5	63.6	76.6	98.1	93.0	95.5
LλMDA *w/* Claude-2 (★.B)	96.1	84.2	89.7	95.8	82.5	88.7	98.0	96.3	97.1	98.3	61.9	75.9	98.2	55.5	71.0	98.4	80.2	88.4
LλMDA *w/* Claude-3.5 (★.C)	89.9	93.2	91.6	88.7	93.0	90.8	99.1	95.0	97.1	89.0	81.5	85.1	86.7	77.8	81.9	96.5	92.4	94.3
LλMDA *w/* GPT-4o (★.D)	95.5	88.1	**91.7**	95.1	86.9	90.8	98.8	97.1	97.9	99.2	83.1	**90.5**	99.1	79.6	88.3	99.6	93.2	96.3

towards the correct identification of unknown symbols, eventually resulting in better approximation. Moreover, the *precision for both benchmarks is high while achieving an improvement in recall.* Overall, LλMDA improves over the traditional approaches in F1-score by 7.2%–12% and 17.7%–40.3% for both benchmarks.

Compared to program dependencies recovered by *Code2API* [25]+ Joern, LλMDA with GPT-4o relatively improves in recall by 12.1% and 31.1% for StatType-SO and COSTER-SO, respectively. Meanwhile, the improvement in F1-score is 10.7% and 23%. Upon further examination, we found that Code2API struggles with incomplete code snippets that *missed variable declarations* and *explicit data types*, leading Joern to discard the dependence edges involving those program elements. Code2API's main goal is "APIzation," *i.e.*, transforming a given semantically complete code snippet into a well-formed API method. To this end, it prompts LLMs to recover missing `import` statements, construct the method signature (including its name and parameters), and insert statements such as `return` and `throw`. However, in doing so, Code2API converts local variables into method parameters, which prevents Joern from constructing all dependence edges originating from those variables.

Compared to the PDGs produced by *JCoffee* [17]+Joern, LλMDA with GPT-4o relatively improves in recall by 26% and 71.3% for StatType-SO and COSTER-SO, respectively. The improvement numbers for F1-score are 11.9% and 40.3% for two datasets. Upon examining the results, we observed that JCoffee replaces any unresolved data type or unknown class name with the identifier `UNKNOWN`. This substitution causes confusion for Joern, which incorrectly interprets all instances of `UNKNOWN` as references to the same program element. As a result, Joern generates incorrect data flows involving these elements. Furthermore, this approach prevents Joern from identifying valid data dependencies between variables of unresolved types and others, as the necessary information was not recovered.

Next, among the learning-based approaches, NeuralPDA (⊛.A) shows a low performance in predicting program dependencies. This can be attributed to to its lack of pretraining on a broader code corpus, relying instead on specialized training with a limited dataset. As a result, it lacks generalization capabilities. The notably high recall and low precision suggests a bias toward over-predicting dependencies between program elements. In contrast, PDBERT (⊛.B) demonstrates the advantages of pretraining, achieving the highest recall among all approaches. However, its moderate precision suggests a slight tendency toward over-predicting dependencies, likely a trade-off to maintain high recall. While the precision-recall tradeoff factors in with the smaller programs in StatType-SO, with slightly more context, the LLM in LλMDA outperforms PDBERT in both precision and recall. Overall, LλMDA improves over the learning-based approaches by 5%–265.3% and 15.9%–331% in F1-score for StatType-SO and COSTER-SO, respectively.

Among the LLMs employed in LλMDA, GPT-3.5 and Claude-2 demonstrate similar performance, as do Claude-3.5 Sonnet and GPT-4o. We can see that Claude-2 excels at contextualizing smaller programs, yielding a high precision and recall on StatType-SO; while GPT-3.5 performs better at disambiguating unknown symbols better in larger programs, as seen in its higher recall on COSTER-SO. Similar trends are noted between GPT-4o and Claude-3.5 Sonnet.

Table 1 shows the results in recovering *control dependencies* for partial code. As seen, all approaches perform generally better for control than for data dependencies. This is expected, as the control flow between sequential statements is less sensitive to the incompleteness of code snippets. However, incompleteness still impacts control flow in cases involving conditional expressions or incomplete loop constructs, where the branching logic depends on variables with missing/undeclared types, or unresolved method calls.

For *data dependencies* (Table 1), we observed a similar trend to that reported earlier for PDGs. This is largely due to two factors: (1) the number of data dependency edges is approximately five times greater than that of control dependency edges, and (2) the models exhibit reasonably high performance on control dependencies. As a result, improvements in overall program dependency recovery are strongly influenced by the accuracy of data dependency recovery.

Our results align with the theoretical framework presented in Section 1, i.e., LλMDA effectively navigates Precision-Recall Conundrum *by exploiting the strength of each approach: DA and LLM.*

7 Sensitivity Analysis (RQ2)

We study how the approximately-complete code from each refinement between LLM and compiler, helps Joern in dependence analysis.

7.1 Experimental Setup

Here, we utilized the approximately-complete programs from all LLMs within the LλMDA framework for both StatType-SO and

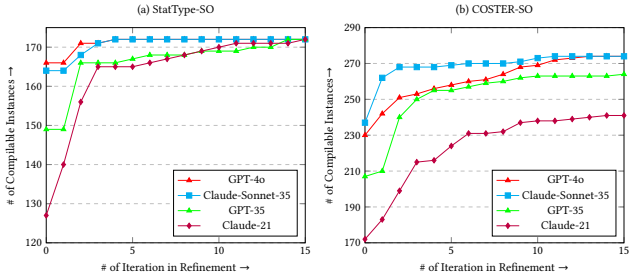


Figure 8: Successful compilation across refinement iterations

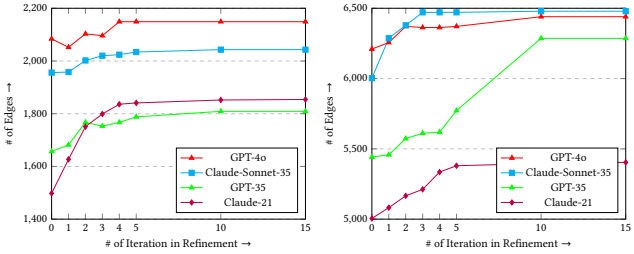


Figure 9: Edges recovered across refinement iterations

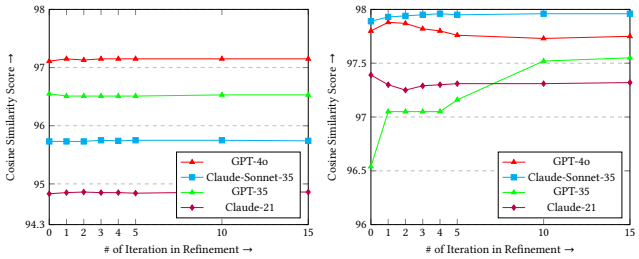


Figure 10: Semantic similarity of approx.-complete with manually completed programs across refinement iterations

COSTER-SO benchmarks. *First*, we quantified the interaction between the LLMs and the compiler by measuring *the number of iterations* required to transform the incomplete code into a valid, compilable version. *Second*, we evaluated the percentage of *correctly recovered data dependencies* from the ground truth after each refinement iteration in LλMDA. This process of recovering edges is illustrated in Figure 4. *Third*, we also measured the semantic correctness of the approximated code after each iteration. Specifically, we compared LλMDA's approximated code (D_{approx}) obtained via compiler feedback-augmented prompting with the manually completed versions of the code snippets in the two benchmarks.

To establish the semantic correctness of the approximated code in D_{approx}, we leveraged CodeBERT [14], a pre-trained code language model known for effectively capturing program semantics. As a measure for semantic similarity, we computed embeddings for both the approximated code and its manually-completed version and computed their cosine similarity scores. These range from 0 to 1, where 0 indicates no similarity and 1 represents an identical match.

7.2 Empirical Results

7.2.1 Sensitivity to Number of Refinement Iterations. In Figure 8, we present the total number of instances where the approximated code from each LLM becomes compilable by the end of each feedback

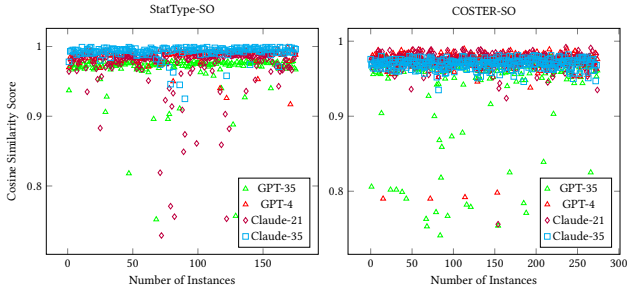


Figure 11: Scatter plots of cosine similarity of approx. complete code and semantically-correct, manually-completed code

cycle. In the first iteration—using only the initial feedback from the compiler—LλMDA enables the LLMs to generate compilable code snippets for 73.8% (127 instances with Claude 2.1) to 96.5% (166 instances with GPT-4o) on the StatType-SO dataset, and from 62.8% (172 instances with Claude 2.1) to 83.9% (230 instances with GPT-4o) on the COSTER-SO dataset. This shows that LλMDA can produce syntactically correct and compilable code using only the code approximation prompt and initial compiler feedback.

Furthermore, we can see that additional iterations of refinement via compiler feedback progressively reduces compilation errors. Almost *98% of the code snippets becoming compilable after only two iterations with GPT-4o* for StatType-SO, and 97% of them after only three iterations with Claude-Sonnet-3.5 on COSTER-SO.

Figure 9 shows the correlation between compilability and the number of data dependence edges recovered. As noted in Section 3.1, the approximately-complete programs generated by the LLMs in LλMDA may not exactly match that from developers. However, *the data dependence edge recovery is tied to syntactic and semantic completeness*, as is reflected by *the progressively increasing number of data dependence edges recovered as the number of compilable approximately-complete programs increase*. With just four iterations on the StatType-SO dataset and two iterations on the COSTER-SO dataset, GPT-4o successfully recovered 96.5% and 84% of the missing edges, respectively. These findings reinforce our Key Ideas 1 and 2–one can use LLMs to fill in missing statements such as undeclared variables or `import` statements, to disambiguate partial programs and improve the recovery of data dependencies.

7.2.2 Semantic Correctness of Approximately-Complete Code. In Figure 10, we plot the mean cosine similarity scores between the approximately-complete and manually-completed program variants in both StatType-SO and COSTER-SO benchmarks, across all refinement iterations. We can see that these are generally high, ranging between 0.94–0.98 for both datasets. In Figure 11, we present a scatter plot for these measures for more insights. We can see that across all instances, the approximately-complete programs from GPT-4o and Claude-3.5 Sonnet and their respective manually-completed are almost semantically similar (cosine similarity → 1).

There are a few outliers when using GPT-3.5 and Claude-2.1, with lower cosine similarity measures. This is consistent with the unsuccessful compilability observed using these LLMs, in which case, the semantic similarity with manually completed programs would be lower. However, *these low-matching programs still contribute to data dependence edge recovery, as indicated in Figure 9*. This

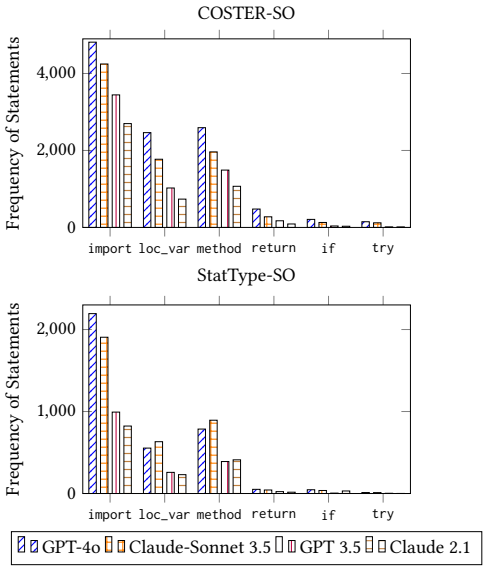


Figure 12: Different types of filled-in statements by LλMDA

shows that *even without exact matching with the human-written code, LλMDA still effectively augmented sufficient context to help recover dependencies in partial code.* Therefore, our findings align with Key Idea 1 (see Section 2.2) that context-augmented programs from LλMDA need not perfectly match manually-written ones for effective recovery of data dependencies.

Finally, vanilla LLM's performance (*i.e.*, without feedback loop) was sub-optimal: many completions failed to compile (Figure 8) and fewer edges were recovered (Figure 9). Thus, iterative refinement significantly improves performance over the base LLM, indicating that the gains stem from LλMDA's design, and not data leakage.

Our findings indicate that even when the approximately complete code from the LLM in LλMDA does not exactly match manually completed versions, the augmented context combined with semantic refinement aids the recovery of program dependencies.

8 Different Types of Filled-in Statements (RQ$_3$)

In this experiment, we aim to assess the quality of code approximation by the LLM in the Context Augmentation phase in LλMDA. To this end, we analyzed the types of program statements filled-in for approximation, across iterations. Figure 12 shows the frequencies of these statements. Note that our code approximation prompt does not explicitly refer to statement types when filling in the missing information. The LLM rather uses the feedback from the compiler in localizing and successfully fixing compilation errors. For example, missing `import` statements, local variable declarations typically trigger *"cannot find symbol"* errors at compile-time. These statements constitute 76.9% and 68.05% of the different types of statements populated by the LLM in the two datasets, respectively.

While filling in the `import` declarations, we observed that the LLM identified appropriate fully-qualified names for the unknown symbols. This aligns with research on *type recovery* on partial code [5, 19, 26, 27, 30, 36], which, however, do not focus on populating local or global variable type declarations due to requiring an understanding of type-specific dependencies. Overall, coupled with compiler feedback on the unknown symbols and their locations, *LLMs can gauge both external and local type-specific dependencies.*

Table 2: `import` statement recommendations (RQ$_3$)

Approach	Evaluation Metrics (in %)		
	Precision	*Recall*	*F1-Score*
SnR	98.2	82.1	89.4
Code2API	91.7	81.5	86.3
LλMDA	98.1	85.8	**91.6**

The LLM filled-in statements include *method declarations* in 20.7% and 22.5% of the cases in two datasets. Apart from the method signatures for the given code, these also include dummy methods that were added to resolve unresolved method calls. Having dummy methods within the same class does not affect the data dependencies within the method, but only helps in achieving syntactic validity.

We observed that the LLM also attempted to harden the code examples at various levels: *inserting a conditional that checks for corner cases* in 104 instances across both the datasets (*e.g.*, protecting code from invalid input data such as `null` or an out-of-bound index); and *handling any missing exceptions* in 21 cases.

GPT-4o and Claude 3.5 successfully generated approximately-complete code that passed the compiler for both datasets. In contrast, LλMDA with GPT-3.5 failed to approximate 10 programs in the COSTER-SO dataset, and Claude 2 failed 33 times. This difference may stem from the relatively worse coding abilities of GPT-3.5 and Claude 2, which likely limits their ability to resolve ambiguous or unresolved types–an essential factor in accurate approximation. This aligns with our earlier finding that, while GPT-3.5 and Claude-2 handle smaller programs effectively, they struggle with longer code snippets, which may require more complex type resolution.

We further evaluate the filled-in `import` statements in StatType-SO dataset. To establish the ground truth, we extracted the respective `import` statements from the manually-completed versions for unknown types and API elements. We skipped COSTER-SO as their manually-completed versions do not contain the `import` statements to the specific API elements. We compared our tool against the state-of-the-art `import` recommenders, LLM-based Code2API [25] and SnR [12]. SnR first extracts constraints that capture the relations between the types used in the snippet; then queries an internal knowledge base to resolve these constraints to output the types.

As shown in Table 2, LλMDA achieves a 6.2% relative improvement in F1-score over Code2API, with 7% and 5.3% increases in precision and recall. This shows the effectiveness of our feedback loop, which equips the LLM with explicit knowledge about unresolved API elements that must be addressed via `import` statements. Compared to SnR, LλMDA achieves comparable precision but improves recall by 4.5%. SnR relies on a predefined knowledge base of API libraries and therefore fails to generate `import` statements for libraries not in that database, resulting in lower recall and F1-score.

9 Exception-Flow-Related Dependence Analysis

9.1 Exception-Flow Analysis (RQ$_4$)

9.1.1 Experimental Setup. We aim to evaluate how the approximately-complete code from LλMDA helps Joern [4] in better capturing

Table 3: Performance on exception flow analysis (RQ4)

Dataset	D_x + Joern, x =	Evaluation Metrics (in %)		
		Precision	*Recall*	*F1-Score*
StatType-SO	*approx-naive*	75.6	37.1	49.8
	approx (LλMDA)	**98.4**	**72.0**	**83.2**
COSTER-SO	*approx-naive*	73.9	29.3	42.0
	approx (LλMDA)	**100**	**86.6**	**88.3**

exception-flow edges in an Exception-Flow Graph (EFG). An EFG is part of the control flow graph (CFG) that connects to the exception handling block(s). We used the same datasets as in Section 6, focusing on instances containing `try/catch` blocks. We exclude those with only general exceptions (e.g., `Throwable e`, `Exception e`), as these do not aid in identifying exception types. Overall, this yielded 23 and 38 code snippets from StatType-SO and COSTER-SO, respectively.

For our baseline, we prompted the LLM to make each code snippet compilable but without integrating the iterative feedback loop between the compiler and LLM as in LλMDA, a process we refer to as naive approximation. We input each snippet from the extracted datasets to LλMDA to fill in exception-handling code including the exception types and `catch` blocks. We then used Joern to build the EFGs by first building the CFGs, following which we statically slice from the beginning of the code to the exception handling blocks to retrieve the EFG edges. For the ground truth, we ran Joern on the manually-completed versions of the code snippets and generated corresponding EFGs through the same CFG and slicing process. We use the same evaluation procedure as in RQ_1, substituting data dependence edges with exception-flow edges.

9.1.2 Experimental Results. As seen in Table 3, LλMDA helps construct EFGs closest to the ones for the manually completed code, achieving an F1-score of 83.2% and 88.3% on both datasets. Similar trends were observed as in the dependence analysis in RQ1: with LLM and a compiler, LλMDA enhances both precision and recall.

9.2 Exception Handling Recommendation (RQ_4)

9.2.1 Experimental Setup. In this experiment, we assess whether LλMDA, as a framework, has utility beyond dependence analysis. We replace the dependence analysis tool with one that identifies potential exceptions in incomplete code snippets. By doing so, we test whether the approximately-complete code generated by LλMDA can enhance the accuracy of exception handling recommendations. Specifically, we replaced the DA tool with PA-NeurEx, the program analysis-based, exception handling recommendation tool developed as part of NeurEx [6]. It takes a code snippet and recommends a list of exception types to be handled. This can be empty if there is no such candidate. PA-NeurEx examines each API element and the `import` statements in the code snippet, aiming to resolve the type of the API element and map it to associated exception types based on a database of the libraries' documentation. We also used GPT-4o [7] with one-shot prompting as another baseline for exception type handling recommendation. In this case, we feed to GPT-4o the given code snippet (approximately-complete or incomplete), asking it to wrap the suspicious API calls that might throw exceptions in a `try-catch` block and extracted the corresponding exception types. We used the same metrics as in NeurEx [6] (Precision, Recall, and F1-score), and the same dataset as in Section 9.1.

Table 4: Exception handling recommendations (RQ4)

Approach	Evaluation Metrics (in %)		
	Precision	*Recall*	*F1-Score*
PA-Neurex + $D_{\text{incomplete}}$	10.8	31.0	16.1
PA-Neurex + D_{approx}	**15.7**	**36.7**	**22.0**
GPT-4o + $D_{\text{incomplete}}$	57.5	29.1	38.6
GPT-4o + D_{approx}	**63.1**	**78.0**	**69.7**

Table 5: Ablation on data dependence analysis (RQ5)

Task	Approach	Eval. Metrics (in %)		
		P	*R*	*F1*
Data Dependence Analysis	*w/o* refinement	95.8	79.8	87.1
	LλMDA	**96.8**	**92.0**	**94.4**
Exception Flow Analysis	*w/o* refinement	91.3	70.9	79.8
	LλMDA	**98.2**	**83.5**	**90.3**

9.2.2 Experimental Results. As seen in Table 4, as using approximated code instead of incomplete code, its recall improves by 168%. This increase stems from GPT-4o primarily using its existing knowledge, with LλMDA providing assistance in the form of type information, to help identify specific exception types. LλMDA's type resolution aids it in identifying appropriate `import` packages, enabling a better inference of which base Java packages to use. This significantly reduces incorrect exception predictions, improving the precision by 9.7%. Overall, as reflected by the F1-score, GPT-4o's performance in handling exceptions across both datasets improves by 80.6%.

In contrast, PA-NeurEx tends to be overly strict to avoid missing an exception. Thus, it has a relatively low precision in both datasets. With LλMDA's approximated code, the improvement of 45.4% in precision and 18.4% in recall is due to the filled-in fully-qualified names for certain APIs. This helps recover some associated exceptions, benefiting PA-NeurEx and increasing F1-score by 36.6%.

LλMDA is extensible to other dependency analysis, and useful in exception-flow analysis and exception handling recommendation.

10 Ablation Study (RQ_5)

In this experiment, we aim to assess the impact of *code refinement*. Accordingly, we compared the performance of LλMDA in all tasks, against that *without the feedback loop*, *i.e.*, where the LLM utilizes only the initial feedback from compiler.

As seen in Tables 5 and 6, the feedback loop is useful, helping mitigate compilation errors from the LLM's approximated code, thus improving the precision; and makes sure there are no more undeclared variables or unresolved APIs, thus improving the recall.

In the data dependence analysis task (rows 1–2 in Table 5), the feedback loop results in a relative increase in recall by 15.3%, resulting in a relative improvement in F1-score by 8.4%. The improvement in precision and recall for the exception-flow analysis task (rows 3–4 in Table 5) are 2.4% and 6.1%, respectively. The improvement can specifically be attributed to the accurate resolution of unknown APIs via code approximation (as opposed to both undeclared variables and unresolved APIs in data dependence analysis). Overall, introducing such a feedback loop between the LLM and compiler improves the performance by 4.4% in F1-score.

Table 6: Ablation on exception handling recomm. (RQ5)

Approach	Evaluation Metrics (in %)		
	Precision	*Recall*	*F1-Score*
PA-Neurex + LλMDA *w/o* refinement	9.8	26.7	14.3
PA-Neurex + LλMDA *w/* refinement	**15.7**	**36.7**	**22.0**
GPT-3.5 + LλMDA *w/o* refinement	28.5	22.78	25.32
GPT-3.5 + LλMDA *w/* refinement	**57.5**	**29.11**	**38.66**

Table 7: Time efficiency and token costs (RQ6)

Baseline	Code2API	GPT-3.5	Claude-2.1	Claude3.5	GPT-4o
Input Tokens	897	2,350	4,484	3,926	1,901
Output Tokens	369	731	1,649	1,793	819
Time (s)	4.56	14.91	17.34	7.35	8.76

PA-LλMDA's improvement for exception type recommendation (rows 1–2 in Table 6) are 60.2% in precision and 37.4% in recall. The feedback is valuable as it ensures that most snippets are syntactically correct. As seen in earlier, 20% of the code snippets are still not parseable by traditional PA tools with one round of refinement.

The baseline introduces an excessive statements: 29 `import` statements, 29 variable declarations, and 13 method definitions more than when using LλMDA as is. Not all of these program statements are relevant, which further affects the baseline's performance.

The feedback loop in LλMDA improves precision by mitigating compilation errors and improving recall by handling undeclared variables and APIs. This boosts compilation success rates, leading to better context for recovering more missing dependencies.

11 Time Efficiency and Token Costs (RQ_6)

The following baselines: ⧫.A, ⧫.B, ⊛.A, and ⊛.B, do not use LLMs, and thus, do not have total tokens-specific API costs. Meanwhile, the cost for the best variant of LλMDA with GPT-4o (including multiple iterations) is about 2.15× more than that of Code2API (Table 7). However, the former takes an average of 3-5 iterations (*i.e.*, fewer number of tokens per iteration), while Code2API has no refinement (*i.e.*, one iteration). The token counts for LλMDA with Claude-2.1/Claude3.5 are higher due to their tokenization schemes.

For one code snippet, LλMDA takes 8.76 seconds—about twice as long as Code2API—primarily due to its use of multiple reasoning iterations. Despite this, the latency remains within a range that is reasonably suitable for interactive use in IDEs or code analysis. GPT-3.5 and Claude-2.1 exhibit longer processing times, while the non-LLM solutions are faster. However, LλMDA+GPT-4o achieves highest performance with reasonable time efficiency and API costs.

12 Limitations and Threats to Validity

LλMDA faces challenges in code hardening for partial code. While it is adept at retrieving types and adding headers, it tends not to insert necessary API setup calls in the middle of the code due to our explicit instructions to preserve the original code (*e.g.*, `if file != null` then `file.close()`). This affects its capabilities in code hardening. The limitations of token input sizes for LLMs remain a barrier. *External Validity*: Our evaluation is limited to Java programs and a fixed set of benchmarks. While these may not be representative, they have been widely adopted in prior work. In addition, results may vary with other models or programming languages. *Internal Validity*: The results in running time might vary due to network traffic, latency issues or server loads when querying LLMs. The third-party tools, e.g., Joern or the compiler, may cause errors. *Construct Validity*: Variations in prompt can lead to differences in input distributions, impacting LLMs' outcomes. To mitigate this, we define prompts using a consistent and well-specified structure.

13 Related Work

Program Analysis for Partial Code. PPA [10] estimates missing bindings through type inference heuristics. GRAPA [41] enables static analysis of partial programs by resolving unknown identifiers with archive files. JCoffee [17] simulates all unresolved and missing code elements with the same dummy name, introducing significant ambiguity for DA. ReACC [24], a retrieval-augmented code completion model, uses sparse and dense retrieval techniques to enhance code completion with partial code as queries. Some IDEs [13, 20] build ASTs from partial code to help the compiler.

Machine learning for Partial Code Analysis. Code2API [25] transforms semantically complete SO code snippets into reusable API methods via Chain-of-Thought prompting. It does not incorporate any verification tool like compiler feedback or refinement. Huang *et al.* [18] used fully qualified name (FQN) masking techniques to train a masked language model for FQN retrieval from partial code. However, it does not complete other missing elements. SOChecker [8] uses vanilla prompting to CodeLlama (without iterative refinement) for code completion. It then uses symbolic execution for security vulnerability analysis in smart contracts. NeuralPDA [39] employs self-attention networks to directly learn program dependencies. Ding *et al.* [11] propose TRACED, an execution-aware pre-training strategy on UniXcoder [16] with a combination of source code, inputs, and execution traces. LExecutor [33] aims to execute code for any piece of code in an under-constrained manner.

Several ML approaches have been proposed for type inference and prediction. The solutions range from statistical learning [28], deep learning [5, 19, 22, 26, 27, 30, 36], to language models [8, 25]. While partial program dependence analysis needs type inference, it also faces other challenges: missing external libraries and `import` statements, missing variable and data type declarations, missing call/return flows and exception flows, unknown parameters, *etc.*

14 Conclusion

We introduce LλMDA, a framework that uses large language models (LLMs) to aid dependence analysis (DA) in partial programs. LλMDA mitigates the precision-recall tradeoff in traditional DA tools by leveraging LLMs to infer missing information in partial programs–improving the recovery of dependence edges. Our evaluation demonstrates improvements over both traditional and learning-based baselines, with gains of 5%-265% and 16%-331% in F1-score across StackOverflow benchmarks. Moreover, we showed its applicability in exception-flow analysis and exception-handling.

Data Availability. Our datasets and code are available [3].

Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) grant CNS-2120386.

References

[1] [n. d.]. . https://stackoverflow.com/questions/16180130/hadoop-filesplit-reading/16180910#16180910

[2] [n. d.]. . hadoop-1/hadoop-1.0.0/src/mapred/org/apache/hadoop/mapred/LineRecordReader.java

[3] [n. d.]. . https://anonymous.4open.science/r/PrePA-7157

[4] 2022. *Open-source code analysis platform for C/C++ based on code property graphs.* https://joern.io/

[5] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 91–105. doi:10.1145/3385412.3385997

[6] Yuchen Cai, Aashish Yadavally, Abhishek Mishra, Genesis Montejo, and Tien Nguyen. 2024. Programming Assistant for Exception Handling with CodeBERT. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 94, 13 pages. doi:10.1145/3597503.3639188

[7] ChatGPT [n. d.]. OpenAI. https://openai.com/.

[8] Jiachi Chen, Chong Chen, Jiang Hu, John Grundy, Yanlin Wang, Ting Chen, and Zibin Zheng. 2024. Identifying Smart Contract Security Issues in Code Snippets from Stack Overflow. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1198–1210. https://doi.org/10.1145/3650212.3680353

[9] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=KuPixIqPiq

[10] Barthélémy Dagenais and Laurie Hendren. 2008. Enabling Static Analysis for Partial Java Programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 313–328. doi:10.1145/1449764.1449790

[11] Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2024. TRACED: Execution-aware Pre-training for Source Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 36, 12 pages. doi:10.1145/3597503.3608140

[12] Yiwen Dong, Tianxiao Gu, Yongqiang Tian, and Chengnian Sun. 2022. SnR: constraint-based type inference for incomplete Java code snippets. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1982–1993. doi:10.1145/3510003.3510061

[13] Eclipse Foundation. [n. d.]. Eclipse Integrated Development Environment (IDE). https://www.eclipse.org/. Accessed: March 26, 2024.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. doi:10.1145/24039.24041

[16] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 7212–7225. doi:10.18653/v1/2022.acl-long.499

[17] Piyush Gupta, Nikita Mehrotra, and Rahul Purandare. 2020. JCoffee: Using Compiler Feedback to Make Partial Code Snippets Compilable. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 810–813. doi:10.1109/ICSME46990.2020.00099

[18] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2023. Prompt-tuned Code Language Model as a Neural Knowledge Base for Type Inference in Statically-Typed Partial Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 79, 13 pages. doi:10.1145/3551349.3556912

[19] Vladimir Ivanov., Vitaly Romanov., and Giancarlo Succi. 2021. Predicting Type Annotations for Python using Embeddings from Graph Neural Networks. In *Proceedings of the 23rd International Conference on Enterprise Information Systems - Volume 1: ICEIS,*. INSTICC, SciTePress, 548–556. doi:10.5220/0010500305480556

[20] JetBrains. [n. d.]. IntelliJ IDEA. https://www.jetbrains.com/idea/. Accessed: March 26, 2024.

[21] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Williamsburg, Virginia) *(POPL '81)*. Association for Computing Machinery, New York, NY, USA, 207–218. doi:10.1145/567532.567555

[22] Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N. Nguyen. 2023. DeMinify: Neural Variable Name Recovery and Type Inference. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 758–770. doi:10.1145/3611643.3616368

[23] Zhongxin Liu, Zhijie Tang, Junwei Zhang, Xin Xia, and Xiaohu Yang. 2024. Pre-training by Predicting Program Dependencies for Vulnerability Analysis Tasks. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 151, 13 pages. doi:10.1145/3597503.3639142

[24] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 6227–6240. doi:10.18653/v1/2022.acl-long.431

[25] Yubo Mai, Zhipeng Gao, Xing Hu, Lingfeng Bao, Yu Liu, and JianLing Sun. 2024. Are Human Rules Necessary? Generating Reusable APIs with CoT Reasoning and In-Context Learning. *Proc. ACM Softw. Eng.* 1, FSE, Article 104 (July 2024), 23 pages. doi:10.1145/3660811

[26] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2241–2252. doi:10.1145/3510003.3510124

[27] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2019–2030. doi:10.1145/3510003.3510038

[28] Hung Phan, Hoan Anh Nguyen, Ngoc M. Tran, Linh H. Truong, Anh Tuan Nguyen, and Tien N. Nguyen. 2018. Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 632–642. doi:10.1145/3180155.3180230

[29] Andy Podgurski and Lori A. Clarke. 1990. A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance. *IEEE Trans. Software Eng.* 16, 9 (1990), 965–979.

[30] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-Based Validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 209–220. doi:10.1145/3368089.3409715

[31] Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto. 2021. Toxic Code Snippets on Stack Overflow. *IEEE Transactions on Software Engineering* 47, 3 (2021), 560–581. doi:10.1109/TSE.2019.2900307

[32] C M Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal K. Roy. 2019. Learning from Examples to Find Fully Qualified Names of API Elements in Code Snippets. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 243–254. doi:10.1109/ASE.2019.00032

[33] Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1522–1534. doi:10.1145/3611643.3616254

[34] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh. 2022. An Empirical Study of C++ Vulnerabilities in Crowd-Sourced Code Examples. *IEEE Trans. Software Eng.* 48, 5 (2022), 1497–1514. doi:10.1109/TSE.2020.3023664

[35] W. Visser, K. Havelund, G. Brat, and Seungjoon Park. 2000. Model checking programs. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. 3–11. doi:10.1109/ASE.2000.873645

[36] Chong Wang, Jian Zhang, Yiling Lou, Mingwei Liu, Weisong Sun, Yang Liu, and Xin Peng. 2025. TIGER: A Generating-Then-Ranking Framework for Practical Python Type Inference . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 321–333. doi:10.1109/ICSE55347.2025.00019

[37] Mark D. Weiser. 1984. Program Slicing. *IEEE Trans. Software Eng.* 10, 4 (1984), 352–357.

[38] Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2024. A Learning-Based Approach to Static Program Slicing. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 97 (apr 2024), 27 pages. doi:10.1145/3649814

[39] Aashish Yadavally, Tien N. Nguyen, Wenbo Wang, and Shaohua Wang. 2023. (Partial) Program Dependence Learning. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*.

IEEE Press, 2501–2513. doi:10.1109/ICSE48619.2023.00209

[40] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 1001, 10 pages.

[41] Hao Zhong and Xiaoyin Wang. 2017. Boosting complete-code tool for partial program. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 671–681. doi:10.1109/ASE.2017.8115677

[42] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.* 29, 4 (1997), 366–427.