Unmasking the Type Inference Capabilities of LLMs for Java Code Snippets

YIWEN DONG, University of Waterloo, Canada
ZHENYANG XU, University of Waterloo, Canada
YONGQIANG TIAN, Monash University, Australia
CHENGNIAN SUN, University of Waterloo, Canada

Type inference is a crucial task for reusing online code snippets. Although code snippets are prevalently shared on platforms like StackOverflow, they frequently lack essential type information, such as fully qualified names (FQNs) and required libraries. Recent studies have leveraged Large Language Models (LLMs) to perform type inference for such code snippets, demonstrating promising performance. However, these evaluations may suffer from data leakage, as the benchmark suite, StatType-SO, used for evaluation has been publicly available on GitHub since 2017. Consequently, it remains uncertain whether the strong performance of LLMs reflects genuine semantic understanding of code or is due to the ground truth being included in the training set.

This paper strives to comprehensively evaluate the genuine type inference capabilities of LLMs on Java code snippets and identify potential limitations of LLMs. First, we created ThaliaType, a new, previously unreleased benchmark suite designed for type inference evaluation. Second, using the StarCoder2 LLM as a baseline, we uncovered data leakage from StatType-SO in StarCoder2's open-source training set and observed that other state-of-the-art LLMs exhibit similar performance drops when evaluated on ThaliaType, with precision decreasing by up to 59% and recall by up to 72%. Finally, we developed semantic-preserving code transformations to further investigate the capabilities of LLMs in understanding the execution semantics of code snippets. Our results showed that the performance of LLMs on StatType-SO is far less robust to these transformations than on ThaliaType, suggesting that the performance on StatType-SO may be biased by data leakage and have limited generalizability.

These findings highlight the importance of carefully designed, leakage-free benchmarks for evaluating LLMs on type inference tasks. We recommend future studies adopt ThaliaType to ensure rigorous and reliable assessments of LLMs' genuine type inference capabilities.

CCS Concepts: • **Software and its engineering → Automated static analysis**.

Additional Key Words and Phrases: type inference, Java, code snippets, LLM, empirical

1 Introduction

Java code snippets found online, such as those from StackOverflow, are often not compilable or directly reusable because they lack crucial type information, including fully qualified names (FQNs, *i.e.*, import statements) and required libraries. Developers must manually infer and recover this missing information in order to reuse these code snippets [69, 70]. This process requires understanding the libraries and identifying the correct FQNs based on the names and functions used in the code snippets. For instance, if a code snippet references a type with the simple

Authors' Contact Information: Yiwen Dong, yiwen.dong@uwaterloo.ca, University of Waterloo, Waterloo, ON, Canada; Zhenyang Xu, zhenyang.xu@uwaterloo.ca, University of Waterloo, Waterloo, ON, Canada; Yongqiang Tian, Monash University, Australia, yongqiang.tian@monash.edu; Chengnian Sun, cnsun@uwaterloo.ca, University of Waterloo, Waterloo, ON, Canada.

ACM 1557-7392/2026/1-ART
https://doi.org/10.1145/3790099

Table 1. Overview of the benchmark suites and language models used to evaluate LLM performance. The early publication of StatType-SO, combined with the late knowledge cutoff dates of the models, increases the likelihood of data leakage.

(a) The benchmark suites used for evaluation.

Benchmark Suite	First Available	#Code Snippets
StatType-SO	2017 (see Table 2)	267
ThaliaType	*Always New*	300

(b) Evaluated models with their respective version identifiers, knowledge cutoff dates, and whether StatType-SO was present in their training data. The question mark denotes that StatType-SO's presence cannot be confirmed.

Model	Version	Knowledge Cutoff	Data Leakage
StarCoder2:15b	instruct-v0.1-q4_0	September 2023 [45]	✓
GPT-4o	2024-08-06	October 2023 [7]	?
GPT-4o-mini	2024-07-18	October 2023 [7]	?
Llama3.1:8b	instruct_q4_0	December 2023 [48]	?
Llama3.1:70b	instruct_q4_0	December 2023 [48]	?

name `Logger`, the developer must determine whether the appropriate import is `java.util.logging.Logger` from Java's utilities library or `org.slf4j.Logger` from another logging library, depending on the methods invoked on the objects of `Logger` in the code snippet.

Type inference for code snippets can assist developers in recovering missing type information needed to reuse these code snippets [19, 32, 36, 57, 60, 68]. For example, current approaches to type inference for Java code snippets generally fall into two categories: *constraint-based* and *machine learning (ML)-based* techniques. Constraint-based methods build a knowledge base of libraries and the types contained within [19, 68]. The state-of-the-art technique SnR [19] leverages this knowledge base and uses constraints extracted from the code snippet to identify the precise types being used.

ML-based approaches [31, 36, 57, 60] learn from prior examples of type usage in existing code to perform type inference. They often produce incorrect inferences due to their limited understanding of code structure and the rules governing Java's type system. Recent advancements in large language models (LLMs), such as OpenAI's proprietary GPT family and Meta's open-weight Llama models, offer new possibilities for overcoming these limitations. By leveraging training on diverse and extensive datasets, LLMs have shown potential in performing various software engineering tasks [29]. Prior studies [17, 31, 36] suggest that LLMs achieved performance comparable to that of the state-of-the-art SnR.

However, the evaluation of these prior studies [17, 31, 36] is potentially affected by *data leakage* [33] where the training dataset includes the benchmark dataset together with the ground truth, and data leakage allows LLMs to regurgitate the training data rather than conducting type inference. The benchmark suite StatType-SO, used for evaluating type inference techniques has been fully, publicly available on GitHub since 2017 [2]. As shown in Table 1, recent state-of-the-art LLMs have knowledge cutoff dates in late 2023, creating the potential for leakage. Thus, it remains uncertain whether the LLMs' strong performance stems from their ability to understand the semantics of the code snippets or merely from retrieving the ground truth from their training data. While recent work has called attention to this data leakage problem for software engineering research [55, 63], *no study has thoroughly, empirically evaluated LLMs' performance on type inference on code snippets.* Because prior work relied on GPT models where the exact training data is kept confidential, detecting such leakage is difficult [14, 51].

To address this limitation, we first investigated The Stack v2 [11] (simply referred to as StackV2), an open-source dataset used to train the StarCoder2 model. Our exploration revealed that all code snippets from StatType-SO with *ground truth* were present in StackV2. This means that StarCoder2 was trained using code snippets along with

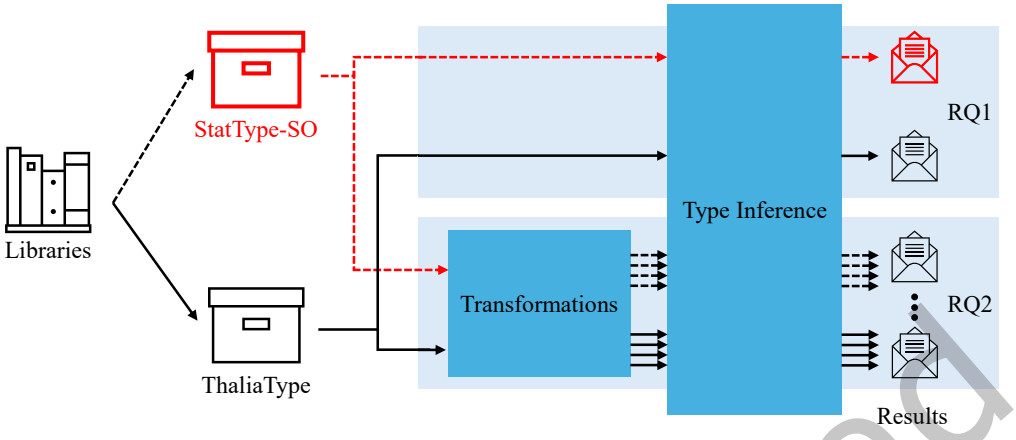


Fig. 1. The general workflow. The libraries used to collect code snippets in StatType-SO were also used to generate code snippets in ThaliaType. These code snippets are used in RQ1, and their transformed versions are used in RQ2 to evaluate various type inference techniques. Their results are compared to evaluate the genuine type inference capabilities of LLMs for Java code snippets. Red indicates potential data leakage.

their expected import statements from StatType-SO. Thus, StarCoder2's performance is affected by data leakage, and can potentially recall the code snippets and their expected import statements from training rather than performing type inference by analyzing the semantics of the code snippets. To better understand the behavior of LLMs, we compare StarCoder2's performance against Llama and GPT models. We organized our inquiry into the following two research questions as illustrated in Figure 1.

In this paper, we thoroughly evaluate LLMs' type inference performance on Java code snippets, focusing on their ability to leverage type information, the impact of data leakage, and their generalizability to unseen code.

RQ1: How well do LLMs perform type inference on unseen code snippets? To address data leakage, and investigate LLMs' genuine ability to perform type inference using the semantic information present in code snippets, we generated a new benchmark suite named ThaliaType using Thalia, a program synthesis technique originally designed for testing static type systems [67]. ThaliaType comprises 300 code snippets that are guaranteed to be unseen by the evaluated LLMs, thereby avoiding data leakage. Unlike StatType-SO, which was published in 2017 (Table 1a), new code snippets can always be generated using our replication package, enabling future evaluations without data leakage. To generate ThaliaType, we used the same set of libraries used in StatType-SO to (1) enable a direct comparison of type inference performance between the two benchmark suites and (2) ensure the evaluated LLMs were consistently and sufficiently exposed to the types used in both benchmark suites during training. By evaluating LLMs on ThaliaType and contrasting the results with those from StatType-SO, we can gain a better understanding of the impact of data leakage on performance.

RQ2: To what extent do LLMs understand the execution semantics of code snippets? To investigate whether LLMs truly understand the execution semantics of code snippets during type inference, we designed *semantic-preserving* code transformations to create syntactically different versions of the input code (§ 6). These transformations were intended to hinder LLMs' reliance on memorized training data by making direct recall more challenging. By observing how LLM' performance changes on these semantically equivalent but syntactically distinct versions, we can gain insights into their ability to grasp the underlying semantic meaning of the code.

Following recommendations from prior software engineering research [63], we evaluate LLM's performance using three open-weight models StarCoder2, Llama3.1:8b, and Llama3.1:70b, along with state-of-the-art closed-source models GPT-4o and GPT-4o-mini (see Table 1b). Among these, StarCoder2 serves as a baseline model with confirmed data leakage (see § 4). For additional comparison, we include SnR [19], a state-of-the-art constraint-based type inference technique, as a non-LLM baseline. This evaluation aims to provide deeper insights into the potential limitations of using LLMs and to guide future research evaluating LLMs' performance on other software engineering tasks.

Findings of RQ1. In RQ1, we found that LLMs are prone to making incorrect inferences on code snippets from ThaliaType, likely because these snippets are truly unseen and not affected by data leakage. All tested LLMs exhibited a similar decline in performance as StarCoder2:15b which experienced a 66.8% decrease in F1-score. For example, Llama3.1:8b and GPT-4o showed comparable declines of 67.2% and 48.5% respectively. On both StatType-SO and ThaliaType, StarCoder2:15b outperformed Llama3.1:8b but was outperformed by Llama3.1:70b, GPT-4o-mini, and GPT-4o. In contrast, SnR, which is not affected by data leakage, obtained a precision of 84.15% and a recall of 84.43% on ThaliaType. The consistent drop in LLMs' performance between the baseline StarCoder2:15b and other models suggests that all tested LLMs are potentially affected by data leakage. In addition, although StarCoder2:15b was trained on StatType-SO snippets, it only achieved an F1-score of 81.67%, indicating that even in the presence of data leakage, models do not perfectly recall the training data.

Findings of RQ2. In RQ2, LLMs showed robustness to individual transformations on both benchmark suites, and also to the combined transformation on ThaliaType code snippets. StarCoder2:15b showed only a 3.0% decrease in F1-score in the worst case, while Llama3.1:8b experienced a 7.9% drop. *However*, combined transformations on StatType-SO consistently led to performance degradation across all models, including a 16.8% reduction for StarCoder2 and up to 30.0% for Llama3.1:8b. Notably, these same combined transformations did not consistently degrade performance on ThaliaType. In fact, StarCoder2:15b experienced a slight improvement in F1-score. One possible explanation is the presence of data leakage, which may allow models to recall answers from training. When transformations introduce enough variation to prevent direct recall, the models are forced to rely on semantic reasoning. Under these conditions, their performance on StatType-SO aligned more closely with that on truly unseen data.

Our results suggest that future evaluations of LLM-based techniques should also assess generalizability, defined as the ability for LLM-based techniques to maintain consistent performance across unseen code snippets. This can be achieved by incorporating both transformed code snippets and ThaliaType, rather than relying solely on StatType-SO, which may suffer from data leakage.

Contribution. This paper makes the following contributions.

- We introduce ThaliaType, a new benchmark suite constructed through a novel adaptation of Thalia for rigorously evaluating the type inference performance of LLMs while mitigating potential bias from data leakage. Using ThaliaType, we demonstrate that prior evaluations based on StatType-SO are likely affected by data leakage, leading to inflated and potentially misleading performance results.
- We evaluate whether LLMs understand the execution semantics of code snippets by applying transformations that preserve the semantics and comparing their effects. Our results on StatType-SO and ThaliaType show that LLMs are generally able to maintain performance across individual transformations. However, complex transformations on StatType-SO result in disproportionately large performance drops compared to both their effects on ThaliaType and the individual transformations. This lack of generalizability can be caused by data leakage where models learned the correct answer from training.
- To enable generation of new code snippets for future research on LLM type inference for code snippets, we have made our benchmark suite and code publicly available at https://github.com/uw-pluverse/thalia-type.

```
1       String s = "Example";
2       Logger.getLogger(s);
```

(a) An example code snippet using a type with the simple name `Logger` and a method called `getLogger` that takes a `String`.

Simple Name	Inferred FQN
String	java.lang.String
Logger	java.util.logger.Logger

(b) The inferred FQNs for the simple names used in the code snippet.

Simple Name	FQN	Method Name	Return Type	Argument Types
Logger	java.util.logging.Logger	getLogger	java.util.logging.Logger	java.lang.String
Logger	java.util.logging.Logger	info	void	java.lang.String
Logger	org.slf4j.Logger	info	void	java.lang.String
Logger	org.slf4j.Logger	debug	void	java.lang.String
String	java.lang.String	length	int	∅
...	...	...	...	...

(c) A simplified knowledge base containing mappings of simple names, methods, return types, and argument types. The symbol ∅ represents that the method takes in no arguments. Additional types and methods are omitted with ...for brevity.

Fig. 2. An example code snippet with example output and example knowledge base used by constraint-based type inference techniques for resolving ambiguities in types.

2 Type Inference on Java Code Snippets

Type inference on code snippets aims to automatically determine the precise types within the code, specifically their FQNs [19, 31, 36, 57, 60, 68]. This process is essential for effectively reusing code snippets from sources like StackOverflow, where type information is often incomplete or missing entirely.

A key challenge in type inference lies in resolving ambiguities with simple names. For instance, as shown in Figure 2a, the simple name `Logger` could refer to types defined in various libraries, such as the `JDK` [1], `SLF4J` [3], `Log4j` [5], or `Logback` [6]. Disambiguating these references necessitates a deep semantic analysis of the snippet, including method signatures (*e.g.*, `getLogger`), and their argument types. The complexity of this task is significantly increased by the vast search space of possible type combinations across numerous libraries, making the inference process computationally demanding.

This section provides an overview of state-of-the-art machine learning-based approaches to type inference, along with our prompting strategy (§ 2.1). Additionally, we briefly introduce SnR, a constraint-based type inference technique, as a baseline for comparison against LLMs (§ 2.2).

2.1 Machine Learning-Based Type Inference

Machine learning-based methods use statistical models and learned representations of code to conduct type inference and resolve ambiguous simple names in code snippets [31, 36, 57, 60]. Of these, LLMs have recently achieved state-of-the-art performance on type inference on the StatType-SO [17, 31, 36] benchmark suite. Existing approaches typically employ a simple but effective prompt to guide LLMs in generating FQNs for types in the code snippet [17, 31, 36]. Notably, Kabir et al. [36] extended the single-prompt approach by incorporating a secondary prompt to address compilation errors in the code snippet generated by the first prompt. This iterative process achieved 99.5% precision on StatType-SO. In contrast, our work focuses on the issue of *data leakage* in type inference rather than repairing code snippets. Data leakage is a fundamental problem that compromises the validity of results, regardless of whether a single-prompt or iterative approach is used. To ensure consistency and

System	You are a helpful programming assistant.
User	Add import statements to the following Java code. Do not use wildcard imports. Include only the necessary import statements. Do not import nonexistent types. Please note that you need to pay close attention and your response should be specific and accurate. Avoid repetition. Reply with the import statements only. `{input_code}`

(a) Example prompt used to infer types on code snippets with LLMs. The `{input_code}` is replaced by the code snippet.

```java
import java.util.List;
```

(b) Example response from LLMs using the example prompt.

Fig. 3. Example prompt used to infer types on code snippets with LLMs, along with an example response. The placeholder `{input_code}` is substituted with the given code snippet.

comparability with prior research, we adopted a single-prompt approach for type inference. Following established practices [31, 36], we developed a straightforward but highly effective prompt for LLM-based type inference.

Prompt Template. Our prompt, shown in Figure 3, is designed to guide the LLM in inferring the type information in Java code snippets. It includes two key components: (1) a system message that sets the LLM's role as a helpful assistant, and (2) a user instruction explicitly requesting the addition of import statements for Java code. The placeholder `{input_code}` is replaced with the actual code snippet during inference. Figure 3a presents the exact wording of the prompt, while Figure 3b demonstrates the corresponding output generated by the LLM for an example code snippet. Despite its simplicity, our prompt achieves high accuracy and aligns with best practices for LLM prompt engineering [4]. Notably, our results using GPT-4o on the StatType-SO benchmark suite represent a 7% improvement over the prior results [36], demonstrating the robustness of our prompt while maintaining a similar level of complexity.

2.2 Constraint-Based Type Inference with SnR

SnR [19] is the state-of-the-art technique that uses constraint-based type inference [19, 68]. It has proven highly effective for inferring types in code snippets from StatType-SO. To illustrate the information SnR leverages, consider the example code in Figure 2a. SnR processes the snippet by extracting constraints, such as identifying a type with the simple name `Logger`, which has a `getLogger` method that accepts a `java.lang.String` argument. Using the knowledge base shown in Figure 2c, SnR assigns `java.util.logging.Logger` to `Logger`, satisfying the constraints. This approach precisely eliminates incorrect candidates, such as `org.slf4j.Logger`, which lacks the required `getLogger` method.

SnR demonstrates how semantic relationships in the code snippet alone can solve type inference problems. By evaluating LLMs alongside SnR, we aim to analyze the extent to which LLMs leverage such semantic information for type inference. These findings will provide deeper insights into LLMs' type inference capabilities and highlight opportunities for future research.

3 Benchmark Suites for Evaluating Type Inference

We utilize two benchmark suites StatType-SO and ThaliaType for evaluating LLMs' type inference performance on code snippets.

3.1 StatType-SO

The StatType-SO benchmark suite is prevalent for testing type inference tools [19, 32, 57, 60]. It is constructed from real-world, manually repaired Java code snippets from Stack Overflow. StatType-SO contains 267 code

```java
package jodatime;

import org.joda.time.Chronology;
import org.joda.time.DateTime;
import org.joda.time.DateTimeZone;
import org.joda.time.chrono.GJChronology;

//ID = 2182921
public class JodaTime05 {
 public static void main(String[] args) {
    DateTimeZone zone = DateTimeZone.forID("Europe/London");
    Chronology coptic = GJChronology.getInstance(zone);

    DateTime dt = new DateTime(coptic);
    DateTime minusOneDay = dt.minusDays(1);

    System.out.println(minusOneDay );
 }
}
```

Fig. 4. Example code snippet from StatType-SO using the JodaTime and JDK libraries. Excessive newlines have been removed for clarity of presentation. These code snippets are generally short. The import statements serve as the ground truth and are removed before the code snippet is used for evaluating type inference.

snippets from six popular Java libraries (Android, GWT, Hibernate, JDK, JodaTime, XStream). The snippets were extracted from 50 randomly selected Stack Overflow posts for each library. To prepare the benchmark suite, all parsing errors in the code snippets were first corrected. Next, the missing import statements were manually inferred and added to the code snippet, serving as the ground truth for evaluation. Figure 4 shows an example from StatType-SO. The code snippets from StatType-SO are generally short and make a small number of calls to the libraries and assign the intermediate results to variables.

While StatType-SO is a widely adopted benchmark suite for evaluating type inference techniques, it is important to acknowledge the potential for data leakage. Given its long-standing availability since 2017 at GitHub, StatType-SO may have been inadvertently incorporated into the training data of some LLMs, potentially leading to artificially inflated performance scores.

3.2 ThaliaType: A New Benchmark Suite for Type Inference

We introduce ThaliaType, a new benchmark suite specifically designed to *rigorously evaluate* the type inference capabilities of LLMs while *mitigating potential biases* from data leakage. Code snippets in ThaliaType require reasoning over type-related semantic information to accurately infer the types used. Through this design, ThaliaType evaluates how effectively LLMs can leverage type information in code snippets, ensuring that the measured performance is generalizable to previously unseen code. If an LLM understands the type information provided, then it can correctly infer the types for any given code snippet.

ThaliaType is generated using Thalia [15, 67], a tool originally developed for testing the type system of compilers. Thalia has uncovered 117 confirmed or fixed type-related bugs across four major compilers. This large number of bugs demonstrates that compiler developers value those found bugs though triggered by automatically generated programs, which indirectly demonstrates the typing relations in Thalia-generated code snippets might resemble

Algorithm 1: Generating ThaliaType from the given set of libraries. Similar to StatType-SO, each ThaliaType code snippet primarily uses a single API.

```
def generate(libraries, dependencies):
    Input: libraries. A set of jar libraries, i.e., Android, GWT, Hibernate, JDK, JodaTime, and XStream.
    Input: dependencies: Additional libraries needed to compile the generated code snippets.
    Output: code_snippets: The list of generated code snippets.
    code_snippets ← [ ]
    for library in libraries:
        apis ← ExtractAPIs(library)          # Extracts public classes, fields, and methods from the given library.
        libs ← {JDK, library, ...dependencies}          # Libraries required for compilation.
        snippets ← Thalia(apis, libs, 50)          # Generates 50 code snippets for the given apis using Thalia.
        code_snippets.append(snippets)
    return code_snippets
```

the typing relations in real-world code snippets. Recognizing its value for LLM evaluation, we repurposed Thalia to create a diverse set of syntactically valid, well-typed Java programs. Given a set of types, fields, and methods, Thalia systematically generates programs that utilize a subset of these components. By reasoning with the type information within the code snippets, LLMs can infer the types used in ThaliaType.

The general process for generating ThaliaType code snippets is summarized in Algorithm 1. Six libraries that were used in StatType-SO were given to Thalia as input. By selecting the same libraries, we ensured that the types in the generated code snippets were familiar to the LLMs, eliminating performance differences caused by varying familiarity with the libraries. For each library, 50 code snippets are generated. Each snippet is based on a single library, following the convention in StatType-SO. In total, 300 code snippets were generated.

Figure 5 shows an example code snippet generated by ThaliaType. Each code snippet consists of a `Main` class and a `test` method. In this code snippet, the code creates a new `LocalTime` from a given time represented by a `long` value of -58. The chronology from this `LocalTime`, which in JodaTime is `ISOChronology`, is then used to construct the current `DateMidnight`. Finally, the `getDayOfYear()` method is called to retrieve the ordinal day number of the year.

Similarity between ThaliaType and StatType-SO. The code snippets in ThaliaType are similar to the code snippets from StatType-SO. Figure 6 compares the statistics of the two benchmark suites. The code snippets in both suites are similar in terms of lines of code and the number of assignments. While Thalia-generated programs are type-intensive, ThaliaType includes fewer method calls but utilizes more types (via import statements) in each snippet. More importantly, our experiments show that SnR achieved comparable performance on both ThaliaType and StatType-SO (§ 5.2), demonstrating that ThaliaType provides as much semantic information as StatType-SO for the type inference task. If LLMs genuinely understand the semantics of code snippets, LLMs should be able to perform similarly on both suites.

To the best of our knowledge, ThaliaType is the first benchmark suite designed for evaluating LLMs on code snippet type inference while addressing data leakage. Future research can leverage our replication package to generate additional code snippets using Thalia using their desired libraries to evaluate newer state-of-the-art LLMs.

4 Data Leakage in LLM-Based Type Inference

In this section, we aim to investigate whether the evaluation of LLM-based type inference techniques may be affected by the data leakage issue [33]. Data leakage occurs when the training datasets overlap with the benchmark datasets used for evaluation, leading models to achieve artificially high performance by memorizing

```java
package src.toady;

import org.joda.time.chrono.ZonedChronology;
import org.joda.time.Chronology;
import org.joda.time.LocalTime;
import org.joda.time.DateMidnight;

class Main {
  static public final <K, I extends ZonedChronology, X> void test()
      throws Exception {
    long elf = (long)-58;
    Chronology pantsuits = new LocalTime(elf).getChronology();
    DateMidnight fitness = DateMidnight.now(pantsuits);
    int liner = fitness.getDayOfYear();
  }
}

interface Function0<R> { public R apply(); }
interface Function1<A1, R> { public R apply(A1 a1); }
interface Function2<A1, A2, R> { public R apply(A1 a1, A2 a2); }
interface Function3<A1, A2, A3, R> { public R apply(A1 a1, A2 a2, A3 a3); }
```

Fig. 5. Formatted code snippet generated by Thalia using the JodaTime and JDK libraries. The variable names are randomly generated. The function interfaces were generated by Thalia but are unused in ThaliaType code snippets.

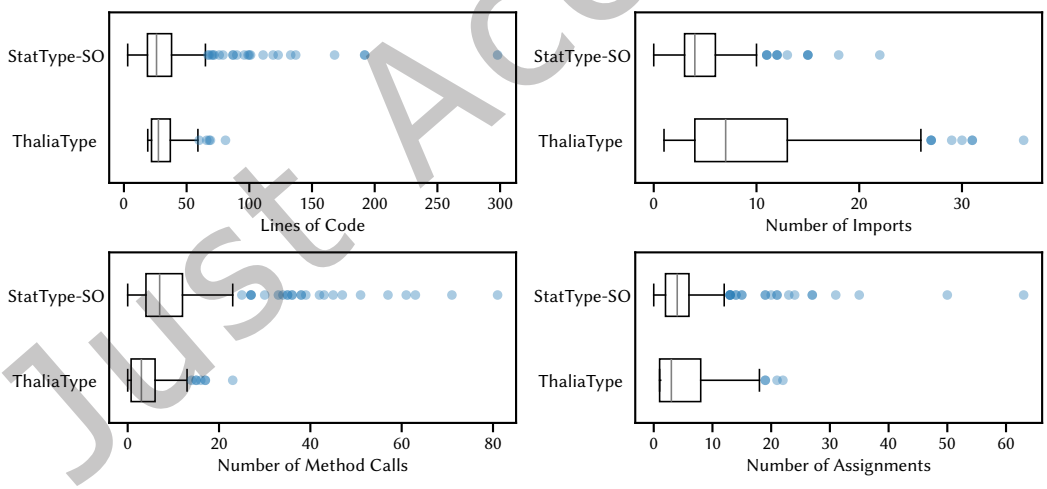


Fig. 6. Box plots comparing the features found in StatType-SO and ThaliaType.

examples rather than demonstrating genuine generalization. The evaluation of the prior studies that either compared against [17, 31] or incorporated LLMs [36] largely relied on the StatType-SO dataset as the benchmark suite, which has been publicly available on GitHub since 2017 [2]. Because the GPT-based models used in these

Algorithm 2: Finding StatType-SO code snippets in the StackV2 dataset based on file names.

```
def getReposInStackV2(filenames):
    Input  : filenames. The list of file names for code snippets in the StatType-SO benchmark suite.
    Output : repo_counts and author_counts. Repositories and authors that are potentially sources of StatType-SO code snippets in
             the dataset are returned.
    repo_counts ← dict()                                   # Initialize repository frequency dictionary.
    author_counts ← dict()                                 # Initialize author frequency dictionary.
    for data in StackV2Dataset():                          # Iterate over the data in the StackV2 dataset.
        if basename(data.path()) in filenames:             # Match the file in the dataset using the file name.
            if data.repo() not in repo_counts:
                repo_counts [data.repo()] = 0
            repo_counts [data.repo()] += 1                 # Increment count for the repo.
            if data.author() not in author_counts:
                author_counts [data.author()] = 0
            author_counts [data.author()] += 1             # Increment count for author.
    return repo_counts, author_counts
```

Table 2. Repositories found in the StackV2, along with the number of code snippets and the repository's last commit date.

Repository Owner	Repository Name	# of code snippets	Last Commit Date
miketran238	AndroidOracle	50	2017-02-14
miketran238	JodatimeOracle	50	2017-02-12
mrthlinh	Oracle-GWT	50	2017-02-13
mrthlinh	Oracle-Hibernate	50	2017-02-13
mrthlinh	TypeResolution_Oracle	219	2017-02-12
pdhung3012	TypeResolution_Oracle	267	2017-06-30

evaluations were likely trained on datasets containing StatType-SO, their reported performance may be affected by data leakage.

However, directly detecting data leakage from LLMs' training is challenging, as most datasets are proprietary and undisclosed, with only a few released to the broader community. Two widely used LLM families, GPT [52] and Llama [48], are trained on proprietary datasets with only partial disclosure of their training data. GPT's creator, OpenAI, has stated that the models are trained on a mixture of publicly available data (such as internet text) and data licensed from third-party providers. However, the exact composition of the dataset remains undisclosed, and access to the model is limited to APIs. Similarly, while Llama models are open-weight and thus freely available for download and use, the data used to train them is not publicly released. Meta discloses that Llama is trained on publicly available data, primarily sourced from the internet, but the exact sources are not specified. This lack of transparency makes it difficult to assess the risk of data leakage in either model.

In contrast, StarCoder2 [45] is trained entirely on the openly available dataset StackV2 [11], which comprises 67.5TB of source code across 658 programming languages. Because this dataset is openly accessible, it is possible to directly examine the training corpus and assess the extent of any data leakage.

To investigate the content of the StackV2, the metadata of the files are examined. Using the matching procedure described in Algorithm 2, file names in StackV2 were compared against those in the StatType-SO benchmark suite, which identified repositories containing overlapping code snippets. This process produced 26 repositories and three repository owners with five or more matching file names, forming the basis for manual inspection.

Manual analysis identified six repositories (shown in Table 2) containing code snippets alongside their ground truth from the StatType-SO benchmark suite. Notably, the `TypeResolution_Oracle` repository by `mrthlinh` includes 219 code snippets with ground truth, excluding only those dependent on the `Android` library. In addition, `pdhung3012`'s `TypeResolution_Oracle` repository contains the complete StatType-SO benchmark suite. All identified repositories were committed in 2017, well before the knowledge cutoffs of GPT and Llama, making it highly likely that these models used for evaluation were trained on the full StatType-SO benchmark suite.

Finding 1: Multiple public GitHub repositories contain code snippets with the ground truth from the StatType-SO benchmark suite. Critically, these repositories were discovered within the StackV2 training set. This inclusion strongly suggests that LLMs trained on internet data have likely been exposed to StatType-SO, raising concerns regarding data leakage.

5 RQ1: How well do LLMs perform type inference on unseen code snippets?

Given the strong possibility of data leakage, we sought to rigorously assess the genuine type inference capabilities of LLMs. To do this, RQ1 investigates their performance on two benchmark suites: the original StatType-SO (which may have been seen during training), and ThaliaType (which contains only unseen code snippets). This comparison allows us to isolate the impact of potential training overlap and better understand how well these models generalize to genuinely novel code.

5.1 Method

Code snippets from StatType-SO and ThaliaType are given to LLMs using the prompt pattern in Figure 3 and also our baseline SnR, one at a time, *without* import statements. To check the output, import statements in LLMs' response are extracted and compared against the original import statements in the code snippets. To ensure reproducibility, all the inferences are performed with a fixed seed (set to one) and a temperature of zero. GPT-4o-mini and GPT-4o are accessed via the OpenAI API [9], while StarCoder2:15b, Llama3.1:8b, and Llama3.1:70b are accessed through a self-hosted Ollama API [8] running on an A100 GPU. The total cost of using the GPT models for RQ1 was $4.47, based on OpenAI's pricing as of early 2025.

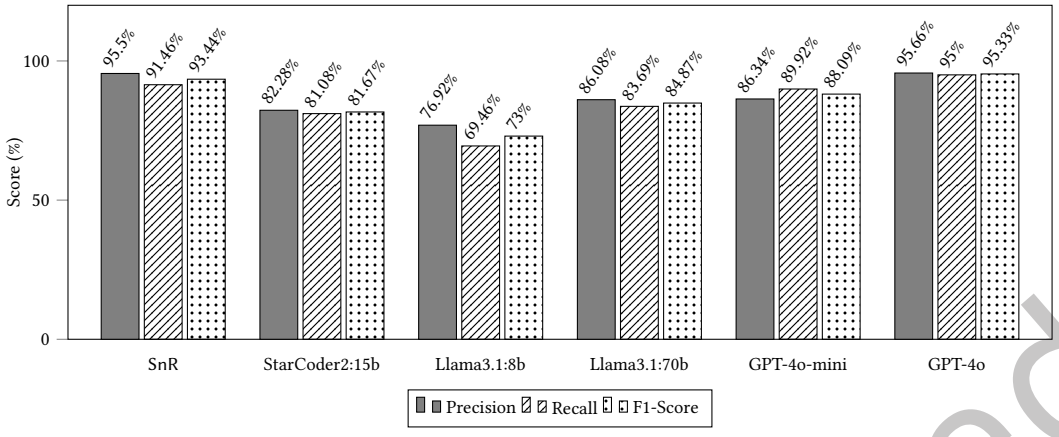
Precision, recall, and F1-scores are computed to evaluate the performance of type inference. Precision measures the proportion of correctly inferred FQNs among all inferred FQNs, while recall measures the proportion of expected FQNs that are correctly inferred.

$$\text{Precision} = \frac{\text{Correctly Inferred FQNs}}{\text{All Inferred FQNs}} \qquad \text{Recall} = \frac{\text{Correctly Inferred FQNs}}{\text{All Expected FQNs}} \qquad \text{F1} = \frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$$

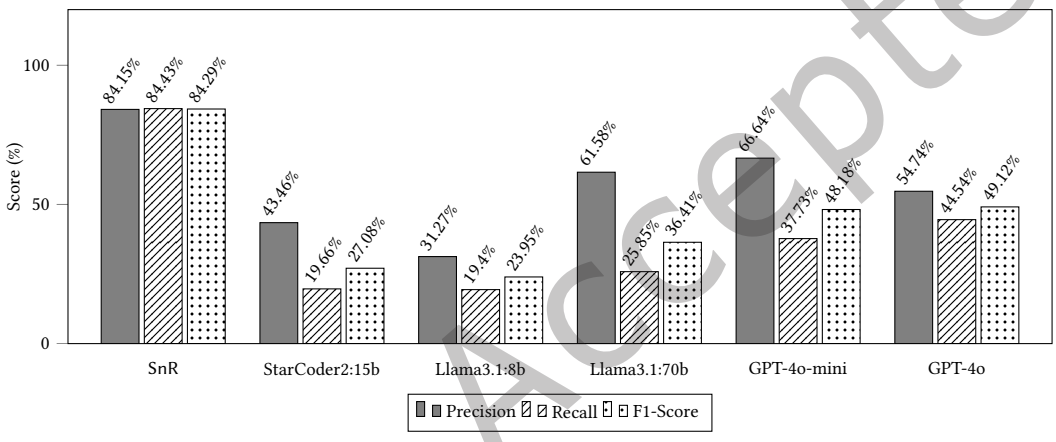
5.2 Results

Despite achieving strong results in StatType-SO (Figure 7), all evaluated LLMs demonstrated lower performance on unseen code snippets in ThaliaType. StarCoder2:15b consistently ranked between Llama3.1:8b and Llama3.1:70b across both benchmark suites. Specifically, as shown in Figure 7a, StarCoder2:15b achieved an F1 score of 81.67% on StatType-SO, higher than Llama3.1:8b, but lower than Llama3.1:70b, GPT-4o, and GPT-4o-mini. However, its performance dropped sharply to 27.08% on ThaliaType, representing a 66.8% decrease. Despite this decline, it still outperformed Llama3.1:8b but remained below the other larger models. Notably, even the best performing model, GPT-4o, experienced a 48.5% decrease in F1 score when evaluated on ThaliaType. The consistent performance drop across all models, including StarCoder2:15b with confirmed data leakage, suggests that LLMs are likely affected by data leakage, which potentially inflated LLMs' type inference performance on StatType-SO.

Unlike LLMs, SnR relies solely on the names of types, method names, and relationships between types in code snippets. SnR only experienced a 9.8% decrease in F1 score when evaluated on ThaliaType, due to the



(a) Type inference performance on StatType-SO.



(b) Type inference performance on ThaliaType.

Fig. 7. Type inference performance (precision, recall, and F1-score) of SnR, StarCoder2:15b, Llama3.1:8b, Llama3.1:70b, GPT-4o-mini, and GPT-4o, on the StatType-SO and ThaliaType benchmark suites. StarCoder2:15b performed between Llama3.1:8b and Llama3.1:70b for both suites. All LLMs experienced a large drop in type inference performance on ThaliaType compared to StatType-SO.

difference in available information in the code snippets. The fact that LLMs outperformed SnR on StatType-SO while performing poorly on ThaliaType suggests that LLMs leverage additional information in StatType-SO to perform type inference. These performance discrepancies motivated further investigations into the specific factors contributing to LLMs' type inference capabilities. In particular, in RQ2 (§ 6), we investigate what types of information LLMs might leverage for type inference, and whether their performance depends on matching exact syntax of the code snippets seen during training.

Table 3. Recall grouped by document frequency of FQNs on GitHub for types in StatType-SO and ThaliaType. The best result for each frequency group is colored using ▢. The TP column lists the number of correctly inferred FQNs by each tool for each frequency group.

	Document Frequency	[0,1e2)		[1e2,1e3)		[1e3,1e4)		[1e4,1e5)		>=1e5	
StatType-SO	Total FQNs	27		69		312		429		463	
		TP	Recall	TP	Recall	TP	Recall	TP	Recall	TP	Recall
	SnR	27	100.00%	60	86.96%	297	95.19%	372	86.71%	433	93.52%
	StarCoder2:15b	7	25.93%	36	52.17%	235	75.32%	372	86.71%	404	87.26%
	Llama3.1:8b	2	7.41%	28	40.58%	173	55.45%	322	75.06%	378	81.64%
	Llama3.1:70b	7	25.93%	31	44.93%	228	73.08%	383	89.28%	439	94.82%
	GPT-4o-mini	4	14.81%	43	62.32%	272	87.18%	399	93.01%	451	97.41%
	GPT-4o	6	22.22%	53	76.81%	296	94.87%	421	98.14%	459	99.14%
ThaliaType	Total FQNs	994		597		439		324		331	
		TP	Recall	TP	Recall	TP	Recall	TP	Recall	TP	Recall
	SnR	735	73.94%	533	89.28%	407	92.71%	300	92.59%	292	88.22%
	StarCoder2:15b	22	2.21%	58	9.72%	94	21.41%	120	37.04%	234	70.69%
	Llama3.1:8b	19	1.91%	42	7.04%	64	14.58%	138	42.59%	258	77.95%
	Llama3.1:70b	18	1.81%	80	13.40%	144	32.80%	182	56.17%	270	81.57%
	GPT-4o-mini	56	5.63%	161	26.97%	227	51.71%	269	83.02%	300	90.63%
	GPT-4o	103	10.36%	233	39.03%	256	58.31%	288	88.89%	316	95.47%

Finding 2: LLM type inference performance declined dramatically when applied to generated, unseen code snippets, potentially as a consequence of data leakage. In these cases, LLMs performed substantially worse than the constraint-based method, highlighting a promising area for future improvement in LLM-based type inference.

5.2.1 Impact of Document Frequency on LLM Type Inference. To better understand the large performance disparity between StatType-SO and ThaliaType, we investigated how the document frequency of a type impacts the performance of LLMs. Document frequency measures the number of source files that reference a specific type. Since LLM outputs are biased by training data [30, 42], their performance may vary depending on the prevalence of a type in the training corpus. Understanding this bias helps ensure that LLMs deliver similar performance across all potential types used in the real world. To quantify a type's document frequency, we analyzed source files from the Boa [20, 21] GitHub dataset. Specifically, the document frequency $F(T)$ is calculated as follows for a type T.

$$F(T) = \sum_{f \in \mathcal{F}} \begin{cases} 1 & \text{if } T \in f, \\ 0 & \text{otherwise.} \end{cases} \qquad (\mathcal{F} \text{ is the set of all source files in the GitHub dataset.})$$

Based on the document frequency, we categorized the FQNs in the StatType-SO and ThaliaType benchmark suites into distinct frequency ranges. For each group, we calculated the recall to investigate how a type's real-world frequency impacts LLM performance. This categorization and analysis provide insights into the extent to which LLMs are biased toward more frequently encountered types.

Results. LLMs performed poorly on less frequently used FQNs and worse on the unseen benchmark ThaliaType than on StatType-SO, even for frequently used FQNs. To analyze how performance varies across FQNs with differing usage frequencies, we grouped FQNs into frequency ranges by orders of magnitude (0-100, 100-1,000, 1,000-10,000, 10,000-100,000, and over 100,000 GitHub files). The results, shown in Table 3, indicate that LLM

performance degrades as the document frequency of FQNs decreases. Specifically, for the least frequently used FQNs, GPT-4o only achieved a recall of 22.22% on StatType-SO and 10.36% on ThaliaType. In contrast, SnR demonstrated consistent performance regardless of a type's popularity, achieving recalls of 100.00% and 73.94% on StatType-SO and ThaliaType respectively. On ThaliaType, LLMs only outperformed SnR on the most frequently used FQNs, specifically those appearing in over 100,000 GitHub source files. This disparity may arise from LLMs preferring more frequent FQNs for a given simple name regardless of the context in the code snippet. For example, for a type with the simple name `View`, if `android.view.View` is the most frequent type in the >=1e5 category and `javax.swing.text.View` belongs to a less frequent category, then always recommending `android.view.View` will result in a high recall in the >=1e5 category but a lower recall in other categories.

Finding 3: LLM type inference performance diminishes for less frequently used FQNs, highlighting a key limitation in applying LLMs to real-world applications. On unseen code snippets in ThaliaType, SnR outperformed all LLMs except for GPT-4o and GPT-4o-mini, but only for the most frequently used FQNs. This shortcoming may hinder the effectiveness of LLMs for developers seeking assistance with these less common FQNs.

Interestingly, despite StarCoder2:15b being trained on code snippets from StatType-SO, it achieved a maximum recall of only 87.26%. As in the previous analysis, its performance generally fell between Llama3.1:8b and Llama3.1:70b on both StatType-SO and ThaliaType. Although training data may inflate performance, StarCoder2:15b does not reproduce code snippets verbatim, as models are designed to generalize beyond specific examples. This generalization, while desirable, makes detecting instances of data leakage difficult.

6 RQ2: To what extent do LLMs understand the execution semantics of code snippets?

The performance decline on unseen data observed in RQ1 highlights the need to investigate how LLMs perform type inference. To explore whether this decline stems from a lack of semantic understanding in LLMs, we conducted an experiment using semantic-preserving transformations that alter syntax without changing the semantics. Our hypothesis is that if LLMs possess a deep understanding of code snippet semantics, their performance should remain stable despite such transformations. Conversely, significant performance degradation would suggest that LLMs rely heavily on superficial syntactic patterns rather than a genuine semantic comprehension.

6.1 Method

The transformations, detailed in Algorithm 3, explore the extent to which syntactic changes affect LLMs' semantic understanding of code. Three specific transformations were employed to assess the impact of modifications to identifier names (§ 6.1.2), code structures (§ 6.1.3), and Java keywords in comments (§ 6.1.4). These were selected to highlight potential limitations in LLMs' semantic understanding, though future research could explore additional transformations. Their impact was then evaluated using the methodology outlined in § 5. The *Wilcoxon signed-rank test* [72] was used to determine whether the transformations caused significant changes in precision, recall, and F1 scores. The average running cost per transformation for GPT models was $6.30 in RQ2, slightly higher than in RQ1 because of longer code snippets.

6.1.1 Background: Java Grammar. The Java grammar depicted in Figure 8 is used to illustrate the key transformations on the Java code snippet. Each code snippet contains one or more classes along with a set of import statements. Classes may contain fields, methods, and blocks, with methods comprising blocks of statements and expressions. Highly repetitive Java statements and expressions were omitted as they follow the same transformation principles.

Algorithm 3: Procedure for the three code transformations applied in RQ2.

```
def RenameIdentifier(p):
    Input  : p, representing the parsed code snippet.
    Output : Transformed code snippet with renamed identifiers.
    for variable_declaration in FindVariableDeclaration(p):          # Rename all variables.
        random_name ← GetRandomName()
        ReplaceAllVariableName(p, random_name, variable_declaration.getName())
    skip_list ← {}
    for expression_method in FindExpressionMethodCalls(p):           # Skip method calls with expression.
        if expression_method.hasExpression():
            skip_list.add(expression_method.getName())
    for method_declaration in FindMethodDeclaration(p):              # Rename all method names.
        if method_declaration.getName() in skip_list or "@Override" in method_declaration.getAnnotations():
            continue                                                 # Skip methods that potentially override parent methods.
        random_name ← GetRandomName()
        ReplaceAllMethodName(p, random_name, method_declaration)
    RenameClassNames(p)                                              # Rename class names.
    RenamePackageNames(p)                                            # Rename package names.
    return p
def LowerCode(p):
    Input  : p, representing the parsed code snippet.
    Output : Transformed code snippet with lowered expressions.
    for expression in findExpressions(p):                            # Lower expressions.
        if IsExprStatement(expression.parent()) or IsLoopCondition(expression) or IsLambdaExpression(expression):
            continue                                                 # Skip expression statements, loop conditions, or lambda expressions.
        random_name ← GetRandomName()
        InsertBefore("var {random_name} = {expression};", expression)
        Replace(p, random_name, expression)
    for field in findFields(p):                                      # Lower field initializations.
        if field.hasInitialization():
            field_name ← field.getName()
            initializer ← field.getInitializer()
            field.removeInitializer()
            InsertBefore("{\n {field_name} = {initializer}; \n }", field)
    return p
def AddKeyword(snippet):
    Input  : snippet, representing the original code snippet.
    Output : Code snippet with added keyword comments.
    new_lines ← []
    for line in snippet.split("\n"):                                 # Append a comment to every line.
        new_lines.add("{line} //{GetRandomKeyword()}")
    return new_lines
```

6.1.2 Identifier Renaming. This transformation investigates whether LLMs rely on specific identifier names for type inference. Although identifier names can provide useful signals, overreliance on unique identifiers or specific identifier sequences seen during training may limit a model's ability to generalize. In real-world code snippets, identifiers are often lowercase variants of type names or abbreviated forms, providing no additional semantic information.

```
Class          {Modifier} class SimpleName [extends Name] [implements {Name}]
               `{'{ClassMember}`}'
Modifier       Annotation | public | protected | private | static | abstract | final
ClassMember    Field | Method | Block
Field          {Modifier} Type SimpleName [= Expr] ;
Method         {Modifier} Type SimpleName ({Type SimpleName}) Block
Annotation     @ Name
Block          `{'{Statement}`}'
Statement      Expr ; | Expr = Expr ; | Block | return [Expr] ;
               | Type SimpleName [= Expr] ;
               | if ( Expr ) Statement else Statement
               | while ( Expr ) Statement
               | for ( Expr ; Expr ; Expr ) Statement
               | for ({Modifier} Type SimpleName [= Expr] ; Expr ; Expr ) Statement
Expr           Name | Literal | this | super | Expr Op Expr
               | ( Type ) Expr | Expr instanceof Name
               | [Expr .] SimpleName | [Expr .] SimpleName ({Expr})
               | new Name ({Expr}) | ! Expr | - Expr
               | ({[Type] SimpleName}) -> Block | SimpleName -> Block
               | ({[Type] SimpleName}) -> Expr | SimpleName -> Expr
Literal        null | NumberLiteral | StringLiteral | BooleanLiteral
Op             + | - | * | / | % | > | == | >= | !=
Name           FQN | SimpleName
FQN            Name . SimpleName
Type           PrimitiveType | Name | var | void
PrimitiveType  byte | short | int | long | float | double | boolean | char
```

Fig. 8. Simplified Java grammar to illustrate our transformations. {*} denotes that the enclosed term occurs zero or more times. [*] denotes that the enclosed term occurs zero or one time.

The `RenameIdentifier` function in Algorithm 3 details the process, which systematically renames identifiers such as variables, methods, classes, and packages in the code snippet. First, all variable declarations are identified using the `FindVariableDeclaration` function (Algorithm 3), which extracts statements matching the pattern *Type SimpleName* [= *Expr*] ;. The algorithm then traverses each declaration, generating a unique, random three-word name for each variable using the `GetRandomName` function (Algorithm 3), from a predefined word list from prior work [67] to ensure uniqueness. The algorithm replaces every occurrence of the variable name in the code (extracted using the `getName` function in Algorithm 3).

Next, the algorithm processes method declarations. To avoid renaming references that could alter semantics, all the method call expressions (*Expr*s with the form [*Expr* .] *SimpleName* ({*Expr*})) that have the first optional part ([*Expr* .)]) are potentially external, thus added to a `skip_list` (lines 7-8) to be filtered out (Algorithm 3). Additionally, overridden methods annotated with `@Override` are skipped, as their names must match those in

```java
package p;

class C extends A {
  A a;
  void m() {
    a.m();
  }
  int n() {
    int n = 1;
    return 1;
  }
}
```

(a) Before renaming.

Before	After
p	GrouseScabsShelley
C	TashaMonroviaTimbers
a	RowsGarlickyThump
n()	ListerineStupefiesFetlock()
n	CrackerSherbertsPlod

(b) Renamed identifiers.

```java
package GrouseScabsShelley;

class TashaMonroviaTimbers extends A {
  A RowsGarlickyThump;
  void m() {
    RowsGarlickyThump.m();
  }
  int ListerineStupefiesFetlock() {
    int CrackerSherbertsPlod = 1;
    return 1;
  }
}
```

(c) After applying identifier renaming.

Fig. 9. Example identifier renaming on a simplified code snippet.

the super class (Algorithm 3). Lastly, the algorithm renames classes and packages (lines 14-15), following the same procedure used for variables. The details are omitted here for brevity.

Figure 9 provides an example of this transformation, showing how variables, methods, classes, and packages are renamed. The method `m` declared at § 6.1.2 likely overrides a method on the super class, `A.m()`. Out of an abundance of caution to preserve semantic consistency, `m` is not renamed. Furthermore, it should be noted that different types of names are renamed separately. For example, the method name `n` is renamed differently from the variable name `n`.

6.1.3 Code Lowering. The code lowering transformation examines the impact of structural changes on LLMs' type inference. A key aspect of type inference is understanding the types present in a program and how they relate to one another. Code lowering does not alter these type relationships but instead disrupts the exact sequence of tokens presented to the model. If a model relies heavily on specific token sequences seen during training, its generalizability may be impaired, leading to failures in type inference when the same program is presented in a different structure.

The algorithm first extracts and traverses all the expressions (Algorithm 3) in the code snippet. For each expression, if its parent is not an expression statement (*Statement*s with the form *Expr* `;`) and it is not a loop condition (lines 19-20), the algorithm creates a new variable with a random name, assigns the expression to the variable, and inserts this assignment statement before the expression (lines 21-22). Next, the original expression is replaced with the newly created variable assigned with the value of the original expression (Algorithm 3). Such transformations modify the code structure while preserving the semantics of the code snippet. The expressions in expression statements must be skipped because transforming them may result in invalid code. For example, an expression in an expression statement can be a method call with no return value (*i.e.*, with a `void` return type). In such a case, the created assignment statement would be invalid. Conditions in loops cannot be transformed either since such transformations could change the semantics of the code.

In addition to expressions, code lowering also applies to fields with an initializer. For each field in the code snippet (Algorithm 3), if it has an initializer (Algorithm 3), the algorithm splits it into a declaration and a block initializing the field (lines 26-29).

Figure 10 demonstrates this transformation. In this example, the field `C c = null` is split into a declaration and an initializer block with `c = null`. Additionally, the expression in the `if` statement is replaced with a newly created variable, and an assignment statement is inserted before its use to assign the original value to that variable.

6.1.4 Adding Keyword Comments. The keyword comment transformation evaluates whether comments containing Java keywords distract LLMs. If a model relies on specific sequences of keywords seen during training,

```java
class C {
  C c = null;
  void m() {
    int a = 0;
    if (a == 0) {
      return;
    }
  }
}
```

(a) Before code lowering.

```java
class C {
  {
    c = null;
  }
  C c;
  void m() {
    int a = 0;
    var GrouseScabsShelley = a == 0;
    if (GrouseScabsShelley) {
      return;
    }
  }
}
```

(b) After applying code lowering.

Fig. 10. Example code lowering on a simplified code snippet.

```java
class C {
  void m() {
    return;
  }
}
```

(a) Before adding keyword comments.

```java
class C { // continue
  void m() { // void
    return; // int
  } // opens
} // case
```

(b) After adding keyword comments.

Fig. 11. Example of adding keyword comments on a simplified code snippet.

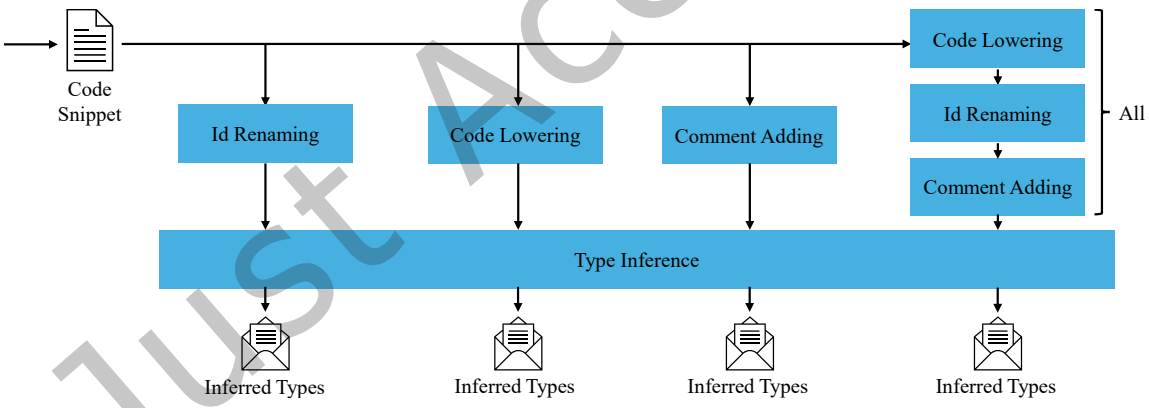


Fig. 12. RQ2 workflow overview. Three code transformations were designed and applied individually to assess their isolated impact. A fourth transformation combines all three sequentially to evaluate potential non-linear interactions.

rather than on the actual content of the code, then this transformation may lead to reduced performance and an inability to generalize to unseen code snippets. To perform this transformation, the `AddKeyword` function in Algorithm 3 appends a single Java keyword [26] as a line comment at the end of each line in the code snippet. An example of the transformed snippet is presented in Figure 11.

Table 4. The precision, recall, and F1-scores of tools after transformations identifier renaming (Id Renaming), code lowering, and adding keyword comments (comment adding) were applied. Note that ▫ represents $p < 0.05$, ▫ represents $p < 0.01$, and ▫ represents $p < 0.001$ when performance decreases, and ▫ represents $p < 0.05$, and ▫ represents $p < 0.001$ when performance improves.

(a) The precision, recall, and F1-scores of tools on StatType-SO and transformed code snippets.

	SnR			StarCoder2:15b			Llama3.1:8b			Llama3.1:70b			GPT-4o-mini			GPT-4o		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
StatType-SO	95.50%	91.46%	93.44%	82.28%	81.08%	81.67%	76.92%	69.46%	73.00%	86.08%	83.69%	84.87%	86.34%	89.92%	88.09%	95.66%	95.00%	95.33%
Id Renaming	95.50%	91.46%	93.44%	82.09%	71.23%	76.28%	65.28%	56.69%	60.68%	85.34%	76.15%	80.49%	89.63%	88.46%	89.04%	97.03%	95.62%	96.32%
Code Lowering	95.43%	91.62%	93.49%	83.72%	79.92%	81.78%	69.43%	65.15%	67.22%	83.81%	80.46%	82.10%	86.14%	88.92%	87.51%	94.82%	95.69%	95.25%
Comment Adding	95.50%	91.46%	93.44%	83.98%	75.00%	79.24%	78.65%	67.46%	72.63%	85.39%	80.92%	83.10%	87.90%	89.38%	88.63%	96.06%	95.77%	95.92%
All	95.43%	91.62%	93.49%	79.34%	59.38%	67.93%	55.83%	47.15%	51.13%	77.93%	70.08%	73.80%	84.60%	82.85%	83.72%	93.29%	94.15%	93.72%

(b) The precision, recall, and F1-scores of tools on ThaliaType and transformed code snippets.

	SnR			StarCoder2:15b			Llama3.1:8b			Llama3.1:70b			GPT-4o-mini			GPT-4o		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
ThaliaType	84.15%	84.43%	84.29%	43.46%	19.66%	27.08%	31.27%	19.40%	23.95%	61.58%	25.85%	36.41%	66.64%	37.73%	48.18%	54.74%	44.54%	49.12%
Id Renaming	84.15%	84.43%	84.29%	46.24%	22.42%	30.20%	31.36%	19.11%	23.74%	57.08%	24.47%	34.25%	52.47%	38.44%	44.37%	43.88%	45.77%	44.80%
Code Lowering	84.14%	84.39%	84.27%	48.09%	23.87%	31.91%	27.06%	18.21%	21.77%	60.16%	25.25%	35.57%	58.04%	36.01%	44.45%	43.45%	44.43%	43.93%
Comment Adding	84.15%	84.43%	84.29%	42.96%	19.66%	26.98%	25.00%	19.22%	21.73%	59.56%	25.07%	35.28%	64.27%	36.57%	46.62%	32.26%	44.80%	37.51%
All	84.14%	84.39%	84.27%	40.81%	20.74%	27.51%	27.57%	17.77%	21.61%	45.92%	23.87%	31.41%	47.50%	36.87%	41.52%	56.30%	45.29%	50.20%

6.1.5 Putting Everything Together. Figure 12 illustrates the complete set of transformations used in RQ2. Each of the first three transformations was applied independently to measure its individual impact. The final transformation combines all three sequentially to more strongly perturb the input, thereby making it more difficult for models to recall StatType-SO code snippets, if they were seen during training. These transformed code snippets are then passed to LLMs. The inferred types are analyzed to see if transformations significantly impacted LLMs understanding of the types used in the code snippet, affecting performance.

6.2 Results

The effects of applying transformations to the StatType-SO and ThaliaType benchmark suites are shown in Tables 4a and 4b, respectively. These transformations significantly affected the precision, recall, and F1-scores of all evaluated LLMs. However, the results are complex and merit detailed examination.

Focusing on each transformation in isolation (identifier renaming, code lowering, comment adding), the results show that LLMs generally exhibit resilience, meaning that simple transformations often do not significantly change a model's performance. This was particularly true for more advanced models such as GPT-4o but also held for smaller models like StarCoder2:15b. Notably, StarCoder2:15b, known to have been trained on StatType-SO code snippets, exhibited significant performance drops only under identifier renaming (recall) and comment adding (recall and F1-score) when evaluated on StatType-SO. The isolated transformation results on StatType-SO and ThaliaType indicate a degree of generalizability in LLMs' understanding of code.

Finding 4: LLMs, especially more advanced models, demonstrate generalizability to simple transformations that preserve execution semantics, maintaining overall performance despite syntactic changes such as identifier renaming, code lowering, and comment adding. This shows that the models tested do not rely on any one syntactic element for type inference.

However, when all three transformations were applied simultaneously, performance changes were amplified, clearly, significantly, and negatively affecting all LLMs' precision, recall, and F1-scores on StatType-SO. For example, F1-scores dropped by 1.7% on GPT-4o, 16.8% on StarCoder2:15b, and 30.0% on Llama3.1:8b. Similar trends were observed for precision and recall. In contrast, applying the same three transformations on ThaliaType code snippets produced less consistent and generally smaller effects. In some cases, performance even improved. For instance, StarCoder2:15b achieved a higher F1-score despite a lowered precision, and GPT-4o demonstrated higher precision, with no significant changes in other metrics on ThaliaType code snippets.

Focusing on GPT-4o on StatType-SO, despite identifier renaming and comment adding causing insignificant performance increases, and code lowering causing a small but significant decrease in precision without impacting F1, when all combined, significantly decreased performance in precision, recall, and F1-scores. These effects were not observed for any model using ThaliaType code snippets.

The discrepancies in performance indicate that the results in StatType-SO may be influenced by data leakage and have limited generalizability compared to the results in ThaliaType. A potential explanation is that transformations reduced the ability of LLMs to recognize code patterns from StatType-SO snippets from training, leading to degraded performance. In contrast, ThaliaType consists of newly generated code snippets that were not in the training data. As a result, models must rely more on reasoning over execution semantics rather than recall from training. The finding that performance on ThaliaType does not consistently decline, and sometimes even improves, suggests that current LLMs can withstand simple, execution-semantic preserving transformations. LLMs generalize more effectively on ThaliaType than their StatType-SO performance suggests.

In contrast, SnR, which analyzes the semantic meaning behind the code snippets, was unaffected by these transformations, experiencing only minor, statistically insignificant variations under the code lowering transformation. These variations occurred when multiple types satisfied all constraints, with the order of type variables serving as a tie-breaker.

Finding 5: In contrast to SnR, which is designed to extract semantic meaning from code snippets and remains unaffected by semantic-preserving transformations, LLMs' type inference generalizability is negatively impacted by combined transformations on StatType-SO, a pattern not observed in ThaliaType. This contrast suggests that while LLMs are capable of generalizing in ThaliaType and its transformed variants, their generalization ability diminishes in StatType-SO, potentially due in part to data leakage.

7 Discussion

In this section, we discuss the generalizability of ThaliaType to other programming languages and transformations, as well as threats to validity.

7.1 Generalizability

Although ThaliaType consists of Java code, our technique for constructing and evaluating leakage-free benchmarks is broadly applicable to other programming languages [28, 56, 59]. Many languages, whether statically or dynamically typed, have mechanisms for importing externally defined types, functions, or libraries. For instance, Python uses `import` statements to incorporate external modules, while JavaScript and TypeScript use `require`

or `import` syntax for similar purposes. Adapting ThaliaType to these languages would involve extending the program generator to produce syntactically valid code in the target language and modeling its type system and import semantics to ensure that the generated programs are well-typed.

Future work may also explore leveraging more expressive program generators [16, 43, 44, 76] to create code snippets that exercise more advanced type-system features. While Thalia currently supports parameterized types, overloading, and higher-order functions, a more expressive generator could model additional constructs such as currying or type aliasing, as found in languages like Scala or Haskell, to further diversify the benchmark.

In addition, the semantic-preserving transformations applied to the code snippets can be generalized beyond Java. These transformations, which modify naming schemes, control-flow constructs, and comments, operate on structural patterns common across many programming languages. By tailoring them to each language's syntax and semantics, ThaliaType's transformations can support assessments of LLM robustness and semantic understanding across languages. Moreover, prior work on program transformation and mutation [18, 39, 40, 79] could be incorporated to further diversify the benchmark. Recent advancements in LLM-based code transformation techniques [53] also present promising avenues for expanding these transformations in future work.

7.2 Threats to Validity

There are four main threats to validity. First, the *representativeness of ThaliaType* may not fully capture certain qualities of real-world Java code snippets, such as variable naming. However, this limitation is mitigated by the primary objective of our evaluation, which is to assess the type inference capabilities of LLMs. Specifically, ThaliaType is designed to evaluate whether LLMs can infer correct type information by reasoning about the execution semantics of the code snippets, rather than merely recalling examples from training. We acknowledge that LLMs may leverage additional semantic cues present in StatType-SO but not found in ThaliaType for type inference. To account for these potential differences, RQ2 applies transformations to both benchmark suites. The identifier renaming transformation on StatType-SO even slightly increased type inference performance of GPT-4o, which means that factors such as identifier names do not significantly affect GPT-4o's performance on StatType-SO and cannot explain LLMs' diminished performance on ThaliaType. Furthermore, the strong performance of SnR on both StatType-SO and ThaliaType indicates that, (1) ThaliaType shares key features that are necessary for type inference with StatType-SO, (2) it presents a meaningful challenge for type inference techniques, and (3) it exposes areas where LLMs require further improvement.

Second, variations in the *document frequency of FQNs* between ThaliaType and StatType-SO could influence LLM type inference performance, since LLMs' outputs are biased by the training data [30, 42]. LLM's performance may vary depending on the prevalence of a type within LLM's training data. To address this, we analyzed the performance of LLMs across different document frequency levels. Despite these efforts, LLMs consistently exhibited lower performance on unseen code snippets from ThaliaType compared to StatType-SO.

Third, our findings may *not generalize to other LLMs*. We mitigated this by following best practices [63] and evaluated a diverse set of models, including both open-weight (StarCoder2:15b, Llama3.1:8b, Llama3.1:70b) and state-of-the-art closed models (GPT-4o, GPT-4o-mini) of varying sizes, providing a representative sample of current LLM capabilities.

Fourth, *prompt engineering* may enable LLM to achieve better performance for type inference on code snippets. To mitigate this threat, we followed best practices to design a simple but effective prompt [4], which achieved high performance on the StatType-SO suite. Regardless, data leakage remains a fundamental issue that can influence LLM evaluations regardless of prompt quality. Our evaluation highlights how data leakage may have impacted prior assessments of LLM type inference performance.

8 Related Work

In this section, we briefly discuss different techniques for type inference on code snippets and LLM data leakage in other fields including automated program repair, code generation, refactoring, and more.

8.1 Type Inference for Code Snippets

Existing type inference approaches for code snippets mainly fall into two categories, constraint-based approaches and machine-learning-based approaches.

Constraint-Based Approaches. The general idea of constraint-based approaches is to first analyze the code snippet and derive a collection of constraints on the types that need to be inferred. With the derived constraints, a set of APIs that satisfies all the constraints can be obtained by performing a constraint-solving algorithm. Baker is the first work that applied constraint-based type inference for code snippets [68]. A subsequent study, SnR, outperformed Baker by improving the handling of parameterized types and leveraged Datalog to efficiently solve the derived constraints against a knowledge base of known types [19].

Machine-Learning-Based Approaches. Machine-learning-based approaches typically leverage models trained on a large set of programs from open-source projects. Phan et al. [57] proposed StatType, which learns the FQNs that often co-occur from a large corpus. With such knowledge, StatType can derive the FQN for an API based on the neighboring API names. A subsequent work by Saifullah et al. [60] made improvements by leveraging both local and global contexts. Huang et al. [31] employed the pre-trained CodeBert [24], a transformer-based masked language model to predict FQNs in the code snippet. Compared to LLMs evaluated in this study, CodeBert is much smaller with 125 million parameters, whereas even Llama3.1:8b's contains 8 billion parameters. Chen et al. [17] proposed a hybrid approach that iteratively combines constraint-based techniques, such as SnR, with machine learning methods to refine the results further. Kabir et al. [36] introduced ZS4C, an LLM-based approach that employs multiple prompts and incorporated compiler error messages to infer and correct compilation errors in the code snippet using LLMs. However, since our work focuses on mitigating data leakage, which fundamentally undermines inference reliability regardless of the prompting strategy used, we used a single prompt. More broadly, data leakage can also diminish the contribution of ZS4C's use of compiler error messages, as it enables LLMs to achieve artificially high scores on StatType-SO, thereby masking the substantial improvements gained when compiler error messages are incorporated.

8.2 Type Inference for Dynamically Typed Languages

Type inference for dynamically typed languages faces extra challenges as there can be variables with insufficient static type constraints in programs written in these languages, which static type inference cannot soundly handle. Many existing studies resort to deep learning-based approaches to overcome this challenge. For example, Hellendoorn et al. [28] propose DeepTyper, a deep learning model that is trained to provide type suggestions given certain contexts and relations. Pradel et al. [59] propose TypeWriter, which uses a deep learning-based approach to predict the types and then utilizes a search-based approach to validate the predicted types. Peng et al. [56] propose HiTyper, a deep learning-based approach that combines with static type inference. It conducts static inference and DL-based prediction iteratively to construct a type dependency graph, which records type dependency information among variables. This paper focuses on the performance of LLMs in type inference for Java code snippets, leaving the investigation of type inference for dynamically typed languages to future work.

8.3 LLM Data Leakage

Data leakage is not limited to type inference but also affects other software engineering tasks. Currently, LLMs have been integrated into a wide variety of software engineering tools such as automated code repair [12, 37,

54, 74, 75, 77, 78, 80], code generation [23, 35, 41, 50], code refactoring [58, 66], and code completion [27, 64, 73], which can all be potentially affected by data leakage either currently or in the future.

Xia et al. [74] discovered that 15% of the bug-fixing patches for automated program repair generated by earlier LLMs (*i.e.*, CodeT5, GPT-Neo, GPT-J, and GPT-NeoX) were already present in the training data. Sainz et al. [61, 62] examined several academic datasets and reported that most of them were either used as training data for ChatGPT or likely exposed to it. A subsequent work by Golchin and Surdeanu [25] proposes a more advanced detection technique and detects the presence of test data of several datasets in the training data of LLMs. Marone and Van Durme [47] propose data portraits to efficiently check whether a given string is present in the training data, assuming the training data is available. Recently, Kong et al. [38], detected memorization in LLMs using low-probability events such as exact code repair matches. However, for type inference on code snippets, there is generally one ground truth of types that are expected. Thus, exact matches do not precisely indicate data leakage. As LLM training datasets grow larger and increasingly opaque, detecting instances of data leakage during evaluation becomes progressively more challenging.

Nevertheless, data leakage remains an open challenge for evaluating LLMs. Some studies have been conducted in other fields [10, 13, 22, 46, 49, 71] regarding data leakage and LLM evaluation. Mirzadeh et al. [49] showed that LLMs have significant limitations when conducting genuine mathematical reasoning, which suggests that LLMs are conducting sophisticated pattern matching rather than true logical reasoning. A recent study by [13] demonstrated that the performance of LLMs is far behind undergraduate students on the proposed project-level Java benchmark that exercises object-oriented programming features.

Our evaluation on type inference rather than math questions or code generation also showed a significant drop in performance with generated code snippets which suggests that understanding the type system in code is challenging for LLMs. So far, there is no conclusive evidence showing that LLMs are conducting genuine type inference on code snippets. Small changes to the inputs can drastically alter model outputs [34, 49, 65]. While some guidelines have been raised for software engineering research using LLMs [55, 63], protecting against data leakage remains an open challenge. We hope our work will shed further light on the data leakage issue in software engineering.

9 Conclusion

This paper conducted a comprehensive assessment of LLMs' type inference capabilities on Java code snippets. First, to address potential data leakage we introduced a new benchmark suite, ThaliaType, and compared it against the widely used StatType-SO. Second, using StarCoder2 as a baseline (which we confirmed to suffer from data leakage), our evaluation revealed that all tested LLMs exhibited similar performance degradations on unseen code snippets. Specifically, we observed up to a 59% decrease in precision and a 72% decrease in recall. Moreover, LLM performance diminished when inferring less commonly used FQNs, suggesting potential bias and presenting opportunities for future research. Finally, we designed and applied three semantic-preserving code transformations to both benchmark suites, to investigate LLMs' understanding of the execution semantics. While LLMs maintained consistent performance under simple transformations, their performance significantly declined when transformations are combined on StatType-SO. These results on StatType-SO, not observed on ThaliaType, indicate that LLM performance on StatType-SO may be inflated by data leakage and may not be generalizable. Overall, our findings underscore the need for future evaluations to incorporate unseen benchmarks, such as newly generated ThaliaType code snippets and transformed variants, while discontinuing StatType-SO. This approach provides a more reliable assessment of LLMs' performance in type inference tasks. All code and datasets used in this paper are publicly available at https://github.com/uw-pluverse/thalia-type.

Acknowledgments

We thank all the anonymous reviewers of TOSEM for their constructive feedback and insightful comments. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, and by CFI-JELF Project #40736.

References

[1] 2014. Logger (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/logging/Logger.html
[2] 2017. Github. https://github.com/pdhung3012/TypeResolution_Oracle
[3] 2021. Logger (SLF4J javadoc). https://www.slf4j.org/api/org/slf4j/Logger.html
[4] 2024. LLM prompting guide. https://huggingface.co/docs/transformers/en/tasks/prompting
[5] 2024. Logger (Apache Log4j API 2.24.3 API). https://logging.apache.org/log4j/2.x/javadoc/log4j-api/org/apache/logging/log4j/Logger.html
[6] 2024. Logger (Logback-Parent 1.5.15 API). https://logback.qos.ch/apidocs/ch.qos.logback.classic/ch/qos/logback/classic/Logger.html
[7] 2024. Models - OpenAI API. https://platform.openai.com/docs/models
[8] 2024. Ollama. https://ollama.com/
[9] 2024. Overview - OpenAI API. https://platform.openai.com/docs/overview
[10] Simone Balloccu, Patrícia Schmidtová, Mateusz Lango, and Ondřej Dušek. 2024. Leak, Cheat, Repeat: Data Contamination and Evaluation Malpractices in Closed-Source LLMs. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics.
[11] BigCode. 2024. The Stack v2 dedup. https://huggingface.co/datasets/bigcode/the-stack-v2-dedup.
[12] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 2188–2200. doi:10.1109/ICSE55347.2025.00157
[13] Jialun Cao, Zhiyong Chen, Jiarong Wu, Shing-Chi Cheung, and Chang Xu. 2024. JavaBench: A Benchmark of Object-Oriented Code Generation for Evaluating Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 870–882.
[14] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting Training Data from Large Language Models. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2633–2650. https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting
[15] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 183–198. doi:10.1145/3519939.3523427
[16] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (Feb. 2020), 36 pages. doi:10.1145/3363562
[17] Zhixiang Chen, Anji Li, Neng Zhang, Jianguo Chen, Yuan Huang, and Zibin Zheng. 2024. iJTyper: An Iterative Type Inference Framework for Java by Integrating Constraint- and Statistically-based Methods. arXiv:2402.09995 [cs.SE] https://arxiv.org/abs/2402.09995
[18] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated testing of refactoring engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 185–194. doi:10.1145/1287624.1287651
[19] 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
[20] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 35th International Conference on Software Engineering* (San Francisco, CA) *(ICSE'13)*. 422–431.
[21] Robert Dyer, Hridesh Rajan, and Tien N. Nguyen. 2013. Declarative Visitors to Ease Fine-grained Source Code Mining with Full History on Billions of AST Nodes. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences* (Indianapolis, IN) *(GPCE)*. 23–32.
[22] Aparna Elangovan, Jiayuan He, and Karin Verspoor. 2021. Memorization vs. Generalization : Quantifying Data Leakage in NLP Performance Evaluation. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics:*

Main Volume, Paola Merlo, Jorg Tiedemann, and Reut Tsarfaty (Eds.). Association for Computational Linguistics, Online, 1325–1335. doi:10.18653/v1/2021.eacl-main.113

[23] Lishui Fan, Zhongxin Liu, Haoye Wang, Lingfeng Bao, Xin Xia, and Shanping Li. 2025. FAIT: Fault-Aware Fine-Tuning for Better Code Generation. arXiv:2503.16913 [cs.SE] https://arxiv.org/abs/2503.16913

[24] 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

[25] Shahriar Golchin and Mihai Surdeanu. 2023. Time travel in llms: Tracing data contamination in large language models. *arXiv preprint arXiv:2308.08493* (2023).

[26] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2023. *The Java® Language Specification, Java SE 21 Edition*. Oracle. https://docs.oracle.com/javase/specs/jls/se21/html/jls-3.html#jls-3.9

[27] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. LongCoder: a long-range pre-trained language model for code completion. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 486, 10 pages.

[28] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 152–162.

[29] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* (Sept. 2024). doi:10.1145/3695988 Just Accepted.

[30] Dong Huang, Qingwen Bu, Jie Zhang, Xiaofei Xie, Junjie Chen, and Heming Cui. 2024. Bias Testing and Mitigation in LLM-based Code Generation. arXiv:2309.14345 [cs.SE] https://arxiv.org/abs/2309.14345

[31] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xin Peng, Xiwei Xu, and Qinghua Lu. 2023. FQN Inference in Partial Code by Prompt-tuned Language Model of Code. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 31 (dec 2023), 32 pages. doi:10.1145/3617174

[32] 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

[33] Huseyin Atahan Inan, Osman Ramadan, Lukas Wutschitz, Daniel Jones, Victor Rühle, James Withers, and Robert Sim. 2021. Training Data Leakage Analysis in Language Models. (February 2021). https://www.microsoft.com/en-us/research/publication/training-data-leakage-analysis-in-language-models/

[34] Bowen Jiang, Yangxinyu Xie, Zhuoqun Hao, Xiaomeng Wang, Tanwi Mallick, Weijie J Su, Camillo J Taylor, and Dan Roth. 2024. A Peek into Token Bias: Large Language Models Are Not Yet Genuine Reasoners. *arXiv preprint arXiv:2406.11050* (2024).

[35] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-Planning Code Generation with Large Language Models. *ACM Trans. Softw. Eng. Methodol.* 33, 7, Article 182 (Sept. 2024), 30 pages. doi:10.1145/3672456

[36] Azmain Kabir, Shaowei Wang, Yuan Tian, Tse-Hsun (Peter) Chen, Muhammad Asaduzzaman, and Wenbin Zhang. 2025. ZS4C: Zero-Shot Synthesis of Compilable Code for Incomplete Code Snippets Using LLMs. *ACM Trans. Softw. Eng. Methodol.* 34, 4, Article 90 (April 2025), 30 pages. doi:10.1145/3702979

[37] Jiaolong Kong, Xiaofei Xie, Mingfei Cheng, Shangqing Liu, Xiaoning Du, and Qi Guo. 2025. ContrastRepair: Enhancing Conversation-Based Automated Program Repair via Contrastive Test Case Pairs. *ACM Trans. Softw. Eng. Methodol.* (March 2025). doi:10.1145/3719345 Just Accepted.

[38] Jiaolong Kong, Xiaofei Xie, and Shangqing Liu. 2025. Demystifying Memorization in LLM-Based Program Repair via a General Hypothesis Testing Framework. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE120 (June 2025), 23 pages. doi:10.1145/3729390

[39] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. doi:10.1145/2594291.2594334

[40] Thanh Le-Cong, Dat Nguyen, Bach Le, and Toby Murray. 2025. Towards Reliable Evaluation of Neural Program Repair with Natural Robustness Testing. *ACM Trans. Softw. Eng. Methodol.* 34, 7, Article 213 (Aug. 2025), 44 pages. doi:10.1145/3716167

[41] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured Chain-of-Thought Prompting for Code Generation. *ACM Trans. Softw. Eng. Methodol.* 34, 2, Article 37 (Jan. 2025), 23 pages. doi:10.1145/3690635

[42] Yan Liu, Xiaokang Chen, Yan Gao, Zhe Su, Fengji Zhang, Daoguang Zan, Jian-Guang Lou, Pin-Yu Chen, and Tsung-Yi Ho. 2024. Uncovering and quantifying social biases in code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 110, 13 pages.

[43] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. doi:10.1145/3428264

[44] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI, Article 181 (June 2023), 22 pages. doi:10.1145/3591295

[45] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE] https://arxiv.org/abs/2402.19173

[46] Inbal Magar and Roy Schwartz. 2022. Data Contamination: From Memorization to Exploitation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Dublin, Ireland, 157–165. doi:10.18653/v1/2022.acl-short.18

[47] Marc Marone and Benjamin Van Durme. 2023. Data Portraits: Recording Foundation Model Training Data. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 15121–15135. https://proceedings.neurips.cc/paper_files/paper/2023/file/3112ee706d21d734c15532c1239773e1-Paper-Datasets_and_Benchmarks.pdf

[48] Meta. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[49] Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. GSM-Symbolic: Understanding the Limitations of Mathematical Reasoning in Large Language Models. arXiv:2410.05229 [cs.LG] https://arxiv.org/abs/2410.05229

[50] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proc. ACM Softw. Eng.* 1, FSE, Article 103 (July 2024), 23 pages. doi:10.1145/3660810

[51] Milad Nasr, Nicholas Carlini, Jonathan Hayase, Matthew Jagielski, A. Feder Cooper, Daphne Ippolito, Christopher A. Choquette-Choo, Eric Wallace, Florian Tramèr, and Katherine Lee. 2023. Scalable Extraction of Training Data from (Production) Language Models. arXiv:2311.17035 [cs.LG] https://arxiv.org/abs/2311.17035

[52] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[53] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2025. The Mutators Reloaded: Fuzzing Compilers with Large Language Model Generated Mutation Operators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Hilton La Jolla Torrey Pines, La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 298–312. doi:10.1145/3622781.3674171

[54] Yicheng Ouyang, Jun Yang, and Lingming Zhang. 2024. Benchmarking Automated Program Repair: An Extensive Study on Both Real-World and Artificial Bugs. 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, 440–452. doi:10.1145/3650212.3652140

[55] Ipek Ozkaya. 2023. Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications. *IEEE Software* 40, 3 (2023), 4–8. doi:10.1109/MS.2023.3248401

[56] 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*. 2019–2030.

[57] 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* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 632–642. doi:10.1145/3180155.3180230

[58] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. 2024. Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 275–287. doi:10.1109/ICSME58944.2024.00034

[59] 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*. 209–220.

[60] C M Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal K. Roy. 2020. Learning from Examples to Find Fully Qualified Names of API Elements in Code Snippets. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 243–254. doi:10.1109/ASE.2019.00032

[61] Oscar Sainz, Jon Campos, Iker García-Ferrero, Julen Etxaniz, Oier Lopez de Lacalle, and Eneko Agirre. 2023. NLP Evaluation in trouble: On the Need to Measure LLM Data Contamination for each Benchmark. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 10776–10787. doi:10.18653/v1/2023.findings-emnlp.722

[62] Oscar Sainz, Jon Ander Campos, Iker García-Ferrero, Julen Etxaniz, and Eneko Agirre. 2023. Did ChatGPT cheat on your test? https://hitz-zentroa.github.io/lm-contamination/blog

[63] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the Silence: the Threats of Using LLMs in Software Engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (Lisbon, Portugal) *(ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 102–106. doi:10.1145/3639476.3639764

[64] Anton Semenkin, Vitaliy Bibaev, Yaroslav Sokolov, Kirill Krylov, Alexey Kalina, Anna Khannanova, Danila Savenkov, Darya Rovdo, Igor Davidenko, Kirill Karnaukhov, Maxim Vakhrushev, Mikhail Kostyukov, Mikhail Podvitskii, Petr Surkov, Yaroslav Golubev, Nikita Povarov, and Timofey Bryksin. 2025. Full Line Code Completion: Bringing AI to Desktop. arXiv:2405.08704 [cs.SE] https://arxiv.org/abs/2405.08704

[65] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. 2023. Large Language Models Can Be Easily Distracted by Irrelevant Context. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 31210–31227. https://proceedings.mlr.press/v202/shi23a.html

[66] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring Programs Using Large Language Models with Few-Shot Examples . In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE Computer Society, Los Alamitos, CA, USA, 151–160. doi:10.1109/APSEC60848.2023.00025

[67] Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. 2024. API-Driven Program Synthesis for Testing Static Typing Implementations. *Proc. ACM Program. Lang.* 8, POPL, Article 62 (Jan. 2024), 32 pages. doi:10.1145/3632904

[68] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API Documentation. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–652. doi:10.1145/2568225.2568313

[69] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. 2016. CSNIPPEX: Automated Synthesis of Compilable Code Snippets from Q&A Sites. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 118–129. doi:10.1145/2931037.2931058

[70] Valerio Terragni and Pasquale Salza. 2022. APIzation: generating reusable APIs from StackOverflow code snippets. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 542–554. doi:10.1109/ASE51524.2021.9678576

[71] Md Nayem Uddin, Amir Saeidi, Divij Handa, Agastya Seth, Tran Cao Son, Eduardo Blanco, Steven Corman, and Chitta Baral. 2025. UnSeenTimeQA: Time-Sensitive Question-Answering Beyond LLMs' Memorization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 1873–1913. doi:10.18653/v1/2025.acl-long.94

[72] Frank Wilcoxon. 1992. *Individual Comparisons by Ranking Methods*. Springer New York, New York, NY, 196–202. doi:10.1007/978-1-4612-4380-9_16

[73] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. REPOFORMER: selective retrieval for repository-level code completion *(ICML'24)*. JMLR.org, Article 2183, 21 pages.

[74] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1482–1494. doi:10.1109/ICSE48619.2023.00129

[75] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. 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, 819–831. doi:10.1145/3650212.3680323

[76] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532

[77] He Ye, Aidan Z.H. Yang, Chang Hu, Yanlin Wang, Tao Zhang, and Claire Le Goues. 2025. AdverIntent-Agent: Adversarial Reasoning for Repair Based on Inferred Program Intent. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA062 (June 2025), 23 pages. doi:10.1145/3728939

[78] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. CIRCLE: continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 678–690. doi:10.1145/3533767.3534219

[79] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. 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, 237–249. doi:10.1145/3611643.3616272

[80] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2024. Gamma: Revisiting Template-based Automated Program Repair via Mask Prediction. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 535–547. doi:10.1109/ASE56229.2023.00063