Identifying and Mitigating API Misuse in Large Language Models

Terry Yue Zhuo, Junda He, Jiamou Sun, Zhenchang Xing, David Lo, John Grundy, and Xiaoning Du

Abstract—API misuse in code generated by large language models (LLMs) presents a serious and growing challenge in software development. While LLMs demonstrate impressive code generation capabilities, their interactions with complex library APIs are often error-prone, potentially leading to software failures and vulnerabilities. In this paper, we conduct a large-scale study of API misuse patterns in LLM-generated code, analyzing both method selection and parameter usage across Python and Java, using three representative LLMs (StarCoder-7B, Qwen2.5-Coder-7B, and GitHub Copilot). Based on extensive manual annotation of 3,209 method-level and 3,492 parameter-level misuses, we identify and categorize four recurring misuse types by building on and refining prior API misuse taxonomies. Our evaluation of three widely used LLMs, StarCoder-7B, Qwen2.5-Coder-7B, and GitHub Copilot, reveals persistent challenges in API usage, particularly hallucination and intent misalignment. To address these issues, we propose Dr.Fix, an LLM-based automatic repair approach guided by our taxonomy. Dr.Fix improves repair accuracy compared to baseline prompting and existing repair methods, with gains of up to 38.4 BLEU and 40% in exact match on benchmark datasets. This work offers important insights into the current limitations of LLMs in API usage and provides insights into current limitations and points to directions for improving automated misuse repair in code generation systems.

Index Terms—Code Generation, Large Language Model, API Misuse, Automatic Program Repair, Empirical Software Engineering

1 Introduction

Recent advances in large language models (LLMs) have significantly enhanced the sophistication of automatic code generation, as these models are trained on vast datasets of source code [1]–[6]. However, according to previous work, LLMs may still generate suboptimal or incorrect API usage in code generation despite being exposed to extensive code during training [7]. Library APIs have been pivotal in software development, enabling complex functionalities across various domains, including network communication, data visualization, and scientific computation [8]. Moreover, as libraries evolve by introducing new features, deprecating old ones, or changing usage patterns, LLMs may struggle to adapt to these changes [9], [10]. With over 590,000 external libraries[1] in languages like Python, using the right API correctly can be a considerable challenge, even for experienced developers.

Despite these advancements, existing studies on API misuse in LLM-generated code have focused primarily on controlled, synthetic scenarios. For example, [11] investigated the quality of code generated by several LLMs using StackOverflow-style programming questions, finding that GPT-4 generated code with API misuses in 62% of cases when tested on 18 widely used Java APIs. Similarly, [12] studied API misuse in LLMs by examining security-related programming questions in Java, revealing that LLMs misused APIs in approximately half of the tasks. A more recent error analysis of 1,000 samples from existing benchmarks identified three primary error types, "AttributeError", "TypeError", and "ValueError", commonly associated with API misuse in LLM-generated code. While these studies have demonstrated that specific LLMs can generate code with API misuse on limited libraries, APIs, and programming languages, their findings may not generalize to real-world programming contexts due to their reliance on human-curated questions and scenarios.

A more common scenario in real-world software development involves LLMs integrated within IDE environments, which assist developers in continuing or completing existing code [13]. LLMs are often prompted with an incomplete code snippet and asked to suggest an appropriate API call. In this context, LLMs face distinct challenges: They may erroneously hallucinate functions from other libraries, suggest irrelevant or incorrect APIs, or specify invalid parameters, even when familiar with the target library. These context-dependent completion tasks present challenges different from those of generating standalone code snippets in response to specific queries. While previous research has studied API misuse of LLMs on task-oriented synthetic datasets [11], [12], there remains a critical gap in understanding whether LLMs exhibit similar patterns of API misuse in real-world code completion. Specifically, it remains unclear whether LLMs make the same types of mistakes in API usage as human developers do. Moreover, investigating API misuse patterns across both open-source and closed-source models is essential, as open-source models often lag behind their closed-source counterparts in terms of performance. Current approaches to mitigating API misuse primarily focus on automatic detection using prede-

Corresponding author: Xiaoning Du
T.Y.Zhuo and J.Grundy and X.Du are with Monash University, Australia. E-mail: terry.zhuo@monash.edu, john.grundy@monash.edu, xiaoning.du@monash.edu
T.Y.Zhuo, J.Sun, and Z.Xing are with CSIRO's Data61, Australia. E-mail: frank.sun@data61.csiro.au, zhenchang.xing@data61.csiro.au
J.He and D.Lo are with Singapore Management University, Singapore. E-mail: jundahe@smu.edu.sg, davidlo@smu.edu.sg
Z.Xing is also with Australian National University, Australia.

1. https://pypi.org/

fined misuse types and API documentation. However, these methods face significant limitations when applied to LLM-based code generation, which involves millions of APIs and varying contexts. Even when misuse is detected correctly, rule- and program-analysis-based automatic program repair (APR) techniques often fail to fix misused APIs due to their limited semantic understanding [14], [15]. This suggests that LLM-based APR approaches, which can leverage the semantic understanding of code, may be more suitable for addressing such issues.

In this work, we investigate how LLMs misuse APIs and examine their behavior in real-world code-generation scenarios. Specifically, we evaluate three representative decoder-only LLMs used in IDE settings: StarCoder-7B [3] and Qwen2.5-Coder-7B, two open-source models for code completion, and Copilot, a closed-source model powered by GPT-4 [2]. Both models are widely used in IDE environments such as VSCode[2] . While previous human-centric API misuse research often studies patches from fixed commits [14], [15], it is not feasible to treat LLM-generated code in the same way, as LLMs generate outputs in various formats. We sample code snippets from GitHub in Python and Java to systematically assess how these models misuse APIs. We conduct program analysis to localize the API positions and provide the code context before these APIs as input to the models. Unlike existing studies, we assess the correctness of API invocation in two key areas: method selection and parameter usage. The first aspect evaluates whether LLMs can suggest an appropriate API name based on the code context, while the second examines whether LLMs can predict reasonable API parameters. These aspects correspond to typical use cases in IDE environments' code filling [16] and left-to-right code completion [17].

Building on existing taxonomies of API misuse [18], [19], we adapt and refine them to the context of LLM-generated code. Specifically, we retain violation types that can be reliably identified from snippet-level context, such as Missing and Redundant, and introduce two additional categories that capture LLM-specific failure modes, namely Intent misuse and Hallucination misuse, while de-emphasizing categories that depend on broader project or version information, such as Replacement and Outdated. Through extensive manual annotation of 3,209 method-level and 3,492 parameter-level cases, we identify four recurring misuse patterns in LLM-generated API code: (1) Intent misuse, where a model selects an API that is syntactically valid but semantically inappropriate for the intended task; (2) Hallucination misuse, where the model produces non-existent or fabricated API methods or parameters; (3) Missing item misuse, where required API methods or parameters are omitted; and (4) Redundancy misuse, where unnecessary API calls or arguments are introduced, often leading to inefficiency or potential errors.

Our systematic study reveals that LLMs struggle significantly with API misuse, particularly hallucination and intent misalignment, with distinct patterns observed between Python and Java. While Copilot and Qwen2.5-Coder generally outperform StarCoder, their shared error patterns indicate common challenges in API usage across mod-

```python
import torch
import torch.nn as nn

class GRUCell(nn.Module):
    """
    A simple implementation of a GRUCell in PyTorch.
    """
    def __init__(self, input_size, hidden_size, bias=True)
        :
        super(GRUCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.bias = bias
        # LLM is expected to complete the next line with
            an API call, such as nn.Linear
        self.ih =
```

Fig. 1. An example of LLM code completion suggesting a PyTorch API call for the last line.

els. To address these issues, we propose Detect-reason-Fix (`Dr.Fix`), a taxonomy-guided LLM-based APR method. Compared with baseline prompting and prior repair methods, `Dr.Fix` achieves substantial improvements, with BLEU scores increasing by up to 38.4 points and exact match rates improving by up to 40 percentage points across different models and languages.

The key contributions of this research include:

- An empirical study of API misuse in LLM-generated code completion, conducted on Python and Java using three representative decoder-only LLMs (StarCoder-7B, Qwen2.5-Coder-7B, and Copilot).
- An adaptation of existing API misuse taxonomies to the LLM setting, refined through the manual annotation of 6,452 misuse cases across method and parameter levels.
- A publicly available dataset and replication package to support transparency and reproducibility.
- A taxonomy-guided LLM-based APR approach, `Dr.Fix`, evaluated against baseline prompting and prior work, showing significant improvements in repair accuracy.

The rest of the paper is organized as follows. We outline the background in Section 2. We elaborate on the details of the experimental setting for the case study and introduce the misuse taxonomy in Section 3. Based on the annotations on generating two LLMs on Python and Java, we conduct quantitative analysis and share several insights. Then, we propose the `Dr.Fix` for in-the-wild API misuse repair and evaluate its performance against LLM-based APR on our benchmark in Section 5. To facilitate the future research of API misuse, we provide a replication package [20] and plan to release the code and dataset publicly on GitHub upon acceptance.

2 Background

2.1 Motivation

Consider the example of LLM code generation shown in Figure 1. When prompted with an incomplete code snippet, an LLM may suggest an inappropriate API call due to

2. https://code.visualstudio.com/

misunderstanding the usage context. Typical errors include: (1) incomplete method calls or those with erroneous parameters, (2) calls to similar but unrelated APIs, (3) extraneous method calls not needed in the current context, (4) incorrect sequencing of method calls for library setup, and (5) incorrect integration of APIs from multiple libraries.

With the rise of support for both code completion and code infilling in modern IDEs such as Cursor[3] , it is important to examine how reliably LLMs can generate correct API usage under these two scenarios. Since API calls are the central building blocks of modern software development, even small deviations from correct usage can propagate into functional errors, security issues, or inefficiencies. Therefore, before presenting our evaluation setup, we first scope what we consider to be API misuse in the context of LLM-generated code and how it differs from general programming errors.

2.2 API Misuse

APIs are essential components of modern software systems, enabling developers to interact with third-party libraries and services. Correct usage of APIs ensures that functionality is accessed as intended, while misuse can result in unexpected behavior, degraded performance, or even security vulnerabilities [21]. API misuse typically occurs when developers violate usage constraints, such as required input types, parameter ordering, or call sequences. Unlike general syntax or language-level type errors (e.g., null dereferencing [22]), API misuses may not always lead to immediate failure but can still cause subtle, incorrect behavior [23]. The causes of API misuse are multifaceted, including incomplete documentation, lack of domain knowledge, and evolving API designs [24]. Consequences range from performance degradation to system failure, particularly in high-stakes domains like healthcare or finance [12]. Prior studies have extensively investigated human-written API misuses through bug-fixing commits and empirical analysis [18], [25]–[27].

In this work, we adopt the usage constraint framework from Schlichtig et al. [25] and apply the taxonomy proposed by Wei et al. [18] and He et al. [19], which classifies API misuses based on violation types (e.g., Missing, Redundant) and API elements (e.g., Method, Parameter, Condition). Although originally developed for deep learning libraries, we generalize this taxonomy to broader third-party APIs. We follow a closed coding approach to categorize LLM-generated misuses under this schema. Following [28], we define an API misuse as an incorrect use of an API that violates its documented contract or commonly expected usage constraints at the level of a specific API element. For instance, passing a string where a list is required is classified as an API misuse, even if the code is syntactically valid in Python, since it semantically violates the API's intended usage. In contrast, general language-level errors such as undefined variables, misspellings, or unrelated type mismatches are excluded unless they directly affect API invocation [18], [19], [28]. We adapt this taxonomy for the LLM setting and introduce two descriptive refinements, *intent misuse* and *hallucination*, as surface-level subtypes of existing misuse categories. Intent misuse reflects cases where the model selects a valid API element that is semantically incorrect given the task (e.g., using `vs.abs` instead of a vector magnitude function). Hallucination captures cases where the model generates nonexistent methods or parameters. While both map to known categories like *Replacement* or *Missing–Method*, they are particularly prevalent in LLM outputs and not emphasized in prior human-centric taxonomies. We include them to better reflect distinctive patterns in LLM-generated code without altering the underlying taxonomy.

3. https://docs.cursor.com/en/tab/overview

2.3 Mitigation of API Misuse

API usage can be categorized into two primary types: directly calling API methods or instantiating objects from API classes [29]. API misuses are defined as violations of the implicit or explicit usage constraints of APIs [21]. Recently, numerous API-misuse detectors have been developed [30]–[32]. Generally, there are three types of API misuse detection approaches: static detectors, which identify API misuses through static analysis of source or binary code [33]; dynamic detectors, which detect API misuses through dynamic analysis [34]; and hybrid approaches that combine mining techniques with static analysis [35]. Regardless of the detection technique, existing methods (1) require API specifications, considering the violation of these specifications as API misuse. However, these specifications are often incomplete and difficult to obtain [32]; or (2) do not require explicit API specifications, instead relying on the assumption that the majority usage pattern in a large-scale code corpus represents valid usage. An API is considered misused if it deviates from this majority pattern. The limitation is that this assumption may not always hold, especially for rarely used APIs where the majority pattern does not necessarily reflect valid usage [31]. More recently, there has been promising work exploring the use of LLMs to repair API misuse by fine-tuning paired commit data [15]. Other approaches, such as [14], [18], provide domain-specific or rule-based repair mechanisms. In this work, we compare our proposed method not only against prompting baselines, but also contextualize it with these prior repair techniques, highlighting the advantages and limitations of LLM-based approaches for API misuse repair.

3 Qualitative Studies

This section presents our qualitative analysis of API misuse in LLM-generated code. We do not propose a new taxonomy; rather, we apply a well-established classification framework from Schlichtig et al. [25], supplemented with categories from Wei et al. [18] and He et al. [19], to annotate API misuse cases. These taxonomies define violations in terms of location (method, parameter, condition) and type (missing, redundant, replacement, outdated). Our goal is to assess whether LLMs exhibit similar misuse patterns to human developers, and where they differ.

We use this framework to analyze outputs from LLMs in realistic code completion tasks, focusing on API correctness in two common generation scenarios.

3.1 Research Questions

We wanted to answer the following key research questions about API misuse by current LLMs:

RQ1: How well do LLMs handle API invocation in code generation? We first investigate how reliably LLMs can generate correct API usages in two modern IDE scenarios: code completion, where models predict the next line or token sequence, and code infilling, where models fill in masked elements such as method names or parameters. These objectives reflect real-world developer interactions, where models are often used to suggest API methods or complete parameter lists within partially written code. Our design builds on traditional studies of code completion [36], [37] as well as recent work on infilling techniques [3], [16], [38], ensuring that our evaluation setup aligns with both established and emerging IDE practices.

RQ2: How do LLMs misuse APIs in the wild? As automatic evaluation metrics can only reflect the overall correctness, we then wanted to understand the specific misuse patterns of LLMs from the human perspective.

3.2 Evaluation Elements

To invoke an API correctly, LLMs are expected to predict both the correct API method name and pass valid arguments for the API parameters, where the objectives are similar to the human-written code [25]. We design the following two scenarios to evaluate the correctness of the generated API elements:

3.2.0.1 **API Method Infilling**: In this evaluation setup, we mask out the API method name in a code snippet and ask the models to infer the appropriate method, similar to type inference [39], [40]. This evaluation assesses whether the models can comprehend the intent of the code context and recommend a suitable method name based on the imported packages and the parameters' context.

```
import requests

def fetch_data(url):
    # model will fill in the mask by predicting an API
        name
    response = requests.[MASK](url)
    return response.json()
```

Fig. 2. Example for API Method Infilling: The model should predict 'get' as the appropriate method name.

As shown in Figure 2, the model is expected to predict 'get' as the appropriate method name.

3.2.0.2 **API Parameter Completion**: In this case, rather than focusing on predicting the method name, we evaluate the models' ability to generate valid and contextually appropriate parameter values for a given API method. Here, the model is provided with a code snippet where the API method name is already specified, and it is tasked with generating the appropriate argument(s) within the method call. All surrounding context after the method name is masked, requiring the model to infer correct usage based on its knowledge of the API. The models are expected to complete the arguments correctly by recalling the parameter names and types associated with the API method and identifying potential variables as valid arguments.

```
import requests

def fetch_data():
    url = "http://example.com/api/data"
    response = requests.get([MASK])
    return response.json()
```

Fig. 3. Example for API Parameter Completion: The model should predict 'url' as the appropriate parameter.

As shown in Figure 3, the model is expected to predict 'url' as the appropriate parameter.

For evaluation, Exact Match (EM) is computed at the level of the masked code elements (API method name or parameters), rather than the entire snippet, to directly capture API correctness. In addition to BLEU and CodeBERTScore, we also discuss precision and recall at the API-element level to better reflect the accuracy of API invocations.

3.3 Models

LLMs differ in architecture, size, and pretraining tasks. In practice, decoder-only models are preferred in IDE environments, where developers rely on them for code completion and comment generation [41]. They are also used to summarize code for comprehension or generate test cases for target snippets. Current IDEs such as VSCode and Cursor[4] primarily deploy decoder-only base models that directly predict the next tokens without instruction tuning. This design supports fast inference and seamless integration into interactive editing workflows [42]–[44]. Open-source LLMs are typically executed locally to maximize inference speed and customization flexibility (e.g., domain-specific fine-tuning) [45], while closed-source LLMs offer higher performance but can only be accessed via hosted APIs. To capture representative behaviors of LLMs commonly used in IDEs, we began our experiments in early 2024 and selected three widely used models for code generation [46]–[49]: StarCoder-7B, an open-source decoder-only model from Hugging Face; Qwen2.5-Coder-7B, an open-source model from Alibaba's Qwen2.5 family; and GitHub Copilot, a closed-source model powered by OpenAI's GPT-4. These models collectively represent both open and closed ecosystems, balancing between transparency, accessibility, and real-world deployment in IDE workflows. Although newer versions such as StarCoder2 have since been released, our selected models remain representative of IDE-integrated LLMs at the time of study.

StarCoder is the first fully permissive LLM family trained on code by Hugging Face, outperforming popular LLMs and matching or surpassing closed-source models like 'code-cushman-001' from OpenAI (the original Codex model that powered early versions of GitHub Copilot). With a context length of over 8,000 tokens, the StarCoder models can process more input than any other open-source LLM, enabling a wide range of interesting applications. In addition, the model supports both left-to-right code completion and fill-in-the-middle code prediction. Unlike previous LLMs that are only trained on common languages,

4. https://www.cursor.com/

StarCoder supports 86 programming languages from high resources (e.g., Java) to low resources (e.g., Dart). Furthermore, StarCoder provides three different model sizes, from 1B to 15B. The 1B model maximizes the inference speed, and the 15B model optimizes the generation quality. In our study, we chose StarCoder-7B, which is the best model balancing between accuracy and efficiency.

Qwen2.5-Coder-7B is a recent open-source code generation model from Alibaba's Qwen2.5 series. Like StarCoder, it adopts a decoder-only transformer architecture optimized for next-token and fill-in-the-middle prediction tasks. It is pretrained on a large corpus of multilingual source code and natural language, supporting over 90 programming languages. Compared with StarCoder, Qwen2.5-Coder incorporates more recent repositories and improved tokenizer coverage for Python and Java, making it well-suited for IDE-style code completion without requiring instruction tuning.

Copilot is a closed-source small code suggestion model distilled from OpenAI's GPT-4, initially launched in 2022. While the model is further fine-tuned for coding tasks, it has a much better understanding of natural language context than open-source LLMs like StarCoder. More than 46% of developers' code files, on average, are reportedly generated by GitHub Copilot. Similar to StarCoder, Copilot offers two generation paradigms: code completion and code infilling. Besides, Copilot utilizes a client-side model to reduce the frequency of unwanted suggestions when they might prove disruptive to a developer's workflow, helping it better respond to each developer using it. In this work, we use the Copilot version, which started in 2024, and manually paste code snippets into VSCode for model inference.

3.4 Hyperparameters

To align real-world software development in the IDE with LLMs, we use the default hyperparameters specified in each VSCode extension. Regarding StarCoder, we follow the configurations of Hugging Face's `llm-ls`[5], a Language Server Protocol server leveraging LLMs for code completion. Specifically, we use the temperature of 0.2, the maximum new token number of 150, and the top_p value of 0.95. For Qwen2.5-Coder-7B, we use the recommended hyperparameters, with the temperature of 0.3 and the top_p value of 0.95. For GitHub's Copilot model, we use the official v1.252.0 extension in VSCode 1.95.3 and prompt the model with the code snippet in a single file. Specifically, we directly query the models for autocompletion without any customized instructions.

3.5 Dataset

We describe how we construct the dataset for our qualitative study. We leverage The Stack [50] data, the largest and most up-to-date permissively licensed GitHub corpus containing source code files in 358 programming languages from GitHub repositories. Due to the high data quality resulting from the comprehensive filtering mechanisms, The Stack dataset has been partially utilized by a series of LLMs for pretraining. Specifically, we use v1.2, the latest version that opted out of the requested and malicious data, and exclude the subset used to train StarCoder to avoid data contamination issues [51].

To conduct a thorough investigation, we selected two popular programming languages, Python and Java. As each code file can contain multiple library APIs, we localize all the positions of APIs (methods and parameters) via static analysis. It is important to note that there could be alternative signatures for the same APIs, particularly where methods share the same name but differ in their parameters (e.g., overloading or different argument types). Additionally, we consider APIs that provide the same functionality but reside in different paths or namespaces due to version differences. To account for this, we ensure that all distinct signatures are included in our dataset. For example, an API could appear in various forms depending on the number and types of parameters used. To ensure comprehensive coverage of different programming scenarios, we include only one sample for each fully qualified-named API signature in our dataset, which helps us capture the diversity of API usage across various contexts. Furthermore, there are many duplicated APIs invoked in various code snippets. To ensure our examination covers diverse programming scenarios in the wild, we restrict the dataset to at most one instance of each unique, fully-qualified API signature **per file**. This prevents highly frequent APIs (e.g., `numpy.array`, `requests.get`) from dominating the dataset while still allowing us to capture multiple usages of the same API across different projects and contexts. In total, we randomly sample 3,000 unique APIs for both Python and Java, respectively, based on their distinct fully qualified names. The APIs are collected from 631 Python and 486 Java repositories, covering 4,641 Python packages and 6,258 Java packages identified through import statements. This broad collection ensures coverage of diverse libraries and API usage contexts, reflecting a wide range of real-world development practices. For each API usage sample, we treat the code before the API method as a prefix and the parenthesized parameters as a suffix for *API Method Infilling*, and the code before the API parameter as a prefix for *API Parameter Completion*, resulting in 36,000 generated samples.

Although The Stack applies rigorous filtering, we acknowledge that human-written code may still contain errors; therefore, we treat it as an approximate ground truth, assuming that the majority of examples represent correct API usage while recognizing that occasional misuses or stylistic inconsistencies may introduce noise. Since our dataset emphasizes frequently used APIs, the findings may not fully generalize to rarely used or newly introduced APIs, which we discuss further in threats to validity in subsection 6.3.

3.6 Evaluation Metrics

As we randomly sample our dataset from a raw GitHub code corpus without environment configurations (e.g., dependencies) and oracles, it is hard to assess the code quality via compilation and test cases. Following prior works [52]–[54], we use BLEU [55], CodeBERTScore [56], and Exact Match (EM) as evaluation metrics to determine the (un)likelihood that the models can invoke the same APIs as human developers. Since automatic metrics have known

5. https://github.com/huggingface/llm-ls

limitations, we further complement them with element-level precision and recall, as well as manual analysis of misuse categories.

BLEU BLEU calculates the percentage of 4-gram overlap between the reference fixed code and the candidate code generated by the models. For simplicity, we refer to it as BLEU in our work. The candidate fixed code represents the code generated by the models, while the reference fixed code represents the ground-truth code written by developers. BLEU is defined as follows:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{4} w_n \log p_n\right)$$

where p_n is the precision of n-grams, w_n is the weight for n-grams (usually equal), and BP is the brevity penalty to account for shorter candidate sequences. BLEU captures overall fluency but may be diluted by tokens outside the masked API elements.

CodeBERTScore CodeBERTScore, inspired by BERTScore [57], measures semantic consistency between generated and reference code by computing the cosine similarity of their token representations. Specifically, CodeBERTScore computes the F scores by combining the precision and recall. Precision (CodeBERTScore$_P$) is calculated as the average maximum similarity between each token in the generated code and tokens in the reference code, while recall (CodeBERTScore$_R$) is computed as the average maximum similarity between each token in the reference code and tokens in the generated code. These are combined using a weighted harmonic mean to produce the final score. In this work, we adopt CodeBERTScore$_{F_3}$, which prioritizes recall over precision, as it better aligns with functional correctness in code generation tasks. The formula is defined as:

$$\text{CodeBERTScore}_{F_3} = \frac{10 \cdot \text{CodeBERTScore}_P \cdot \text{CodeBERTScore}_R}{9 \cdot \text{CodeBERTScore}_P + \text{CodeBERTScore}_R}$$

While CodeBERTScore is more flexible than EM in capturing semantic similarity, it has limitations. In particular, it may underestimate correctness when two different but functionally equivalent APIs are used, since token-level embeddings may not capture equivalence unless such relationships were observed during pretraining. Thus, CodeBERTScore should be interpreted as an approximation of semantic similarity.

Exact Match (EM) EM is defined as the count of generated code instances $\hat{Y}$ whose abstract syntax trees (ASTs) exactly match the ASTs of the corresponding reference/ground-truth code Y^*. It is a strict metric that evaluates syntactic equivalence and may exclude reasonable outputs that differ in syntax but are semantically correct. In our study, EM is computed at the level of the masked API element (method name or parameter), rather than the entire snippet, to directly assess correctness of API usage. EM serves as a lower bound, as different programs can achieve the same functionality but be written differently. The formula is defined as:

$$\texttt{EM}(\hat{Y}, Y^*) \triangleq \sum_{\hat{y} \in \hat{Y}} \texttt{matches}(\texttt{AST}(\hat{y}), \texttt{AST}(y^*))$$

A higher EM score consistently indicates stronger performance. However, since human-written code is used as ground truth, EM may penalize models that generate different but still valid API usages. We therefore report EM alongside BLEU, CodeBERTScore, and precision/recall to provide a more complete evaluation of API invocation correctness.

3.7 EM Failure Qualitative Example Analysis

We examined samples that failed on the EM score, indicating that they are highly likely to be misused by LLMs. We employed an iterative qualitative analysis approach inspired by grounded theory and qualitative content analysis. We randomly sampled 300 pairs of failed cases from each model, categorized by element type (method or parameter) and programming language (Python or Java), yielding a total of 1,200 failure samples. This sample size ensures statistical significance, achieving a 99% confidence level with a 5% margin of error.[6]

Two authors, each with over five years of programming experience in Python and Java, reviewed and categorized the API misuse samples. We adopt a closed coding strategy by mapping LLM-generated API misuses to Wei et al.'s taxonomy [18], which classifies misuses along two dimensions: violation types (Missing, Redundant, Replacement, Outdated) and API elements (Method, Parameter, Condition). However, because our evaluation is based on isolated code snippets, without access to full projects, temporal version histories, or documentation, we focus only on the violation types that are reliably annotatable in this setting: *Missing* and *Redundant*.

We exclude *Replacement* and *Outdated* categories, as accurately identifying whether a method has been superseded (Outdated) or semantically replaced (Replacement) often requires knowledge of API evolution, project-specific context, or historical intent, which are unavailable in our static snippet-based setting. For example, detecting an outdated method would require knowing the release timeline or deprecation status of the corresponding library, while identifying a replacement misuse would require understanding the intended functionality and its modern equivalent. Since these contextual signals are absent from isolated code samples, including them could lead to annotation ambiguity and unreliable labeling. Consequently, we focus on misuses that can be confidently recognized from self-contained code evidence while maintaining conceptual consistency with Wei et al.'s taxonomy [18].

To analyze LLM-specific characteristics, we annotate two recurring surface patterns, *intent misuse* and *hallucination*, as tags within the broader Missing/Redundant framework. For example, hallucinated methods typically reflect a non-existent method generated in place of a missing or misused one, and can be understood as a form of *Missing Method*. Similarly, intent misuses, while superficially aligned with

6. https://www.surveysystem.com/sscalc.htm

Replacement–Method, are treated as semantic misfires under *Redundant or Incorrect Method*, when replacement judgment is unreliable. This constrained coding preserves alignment with prior taxonomies while recognizing the limits of static code-only analysis in LLM-generated contexts.

First, the two authors independently familiarized themselves with the dataset by reviewing a subset of API misuse cases. Each annotator labelled 50 randomly selected samples, took detailed notes on their observations, During this process, they were allowed to search online for API information when necessary to validate the correctness of the method and parameter usage. Following this initial review, the annotators conducted axial coding, systematically grouping similar misuses into broader conceptual categories based on shared characteristics. They then met to compare their annotations, resolve discrepancies, and iteratively refine their categorization framework. Through multiple review cycles, they refined their labels, clarified ambiguous definitions, and introduced new categories where necessary. In the end, they progressively consolidated similar misuse patterns into four distinct categories: *Intent Misuse*, *Hallucination*, *Missing Parameter*, *Redundant API Call*. With the finalized codebook, the two authors independently labelled the remaining API misuse cases. To assess inter-annotator agreement, we computed Cohen's Kappa score [58], achieving a final value of 0.97, indicating almost perfect agreement. The annotation process spanned 300 hours, encompassing multiple iterations of refinement to ensure accuracy and consistency.

It is important to note that while we focus on four main categories of API misuse, there are other potential misuse types, such as security-related or deprecation issues. However, these types are more knowledge-intensive and cannot be easily identified without sufficient contextual information or detailed code snippets. For instance, detecting deprecated methods or security vulnerabilities requires understanding the specific API versions and security considerations, which cannot be reliably determined from code snippets alone. Therefore, our analysis focuses on categories that are more directly identifiable through the provided code and are less dependent on external, domain-specific knowledge. We present each category, which is introduced below, with illustrative examples.

3.8 API Misuse Classification

3.8.0.1 LLM-Specific – Intent Misuse: ***Intent misuse occurs when a model selects an API or method that is syntactically correct but does not align with the intended functionality or task.*** This results in incorrect behavior due to the model's misunderstanding of the method's purpose. In Figure 4, the method `farthest()` is intended to compute the farthest point from a reference point, which requires calculating vector magnitudes. The model incorrectly uses `np.abs`, a `numpy` function that returns element-wise absolute values, rather than `vx.magnitude`, which directly computes vector magnitudes. Although syntactically valid, this misuse reflects a semantic mismatch between the selected API and the intended operation. While such cases may be superficially mapped to the Replacement–Method category in Wei et al.'s taxonomy [18], prior studies have not systematically

```
import numpy as np
from blmath.numerics import vx

def farthest(from_point, to_points):
    '''
    Find the farthest point among the inputs, to the
        given point.
    Return a tuple: farthest_point,
        index_of_farthest_point.
    '''
-   absolute_distances = vs.abs(to_points - from_point)
+   absolute_distances = vx.magnitude (to_points -
    from_point)
    index_of_farthest_point = np.argmax(
        absolute_distances)
    farthest_point = to_points[index_of_farthest_point]
    return farthest_point, index_of_farthest_point
```

Fig. 4. A diff-formatted example of **intent misuse**: Incorrect use of **`abs`** instead of **`magnitude`** in Python.

examined intent misuses in the context of LLM-generated code. We adapt and extend the taxonomy to explicitly capture this recurring pattern, which emerges frequently in LLM-assisted code completion. Importantly, we distinguish intent misuses from cases where multiple APIs could plausibly achieve the same functionality (e.g., different logging or plotting methods). In such cases, annotators did not label the output as misuse, since the generated API remained consistent with the overall intent of the code.

```
import java.util.Collections;

public static List<String> getMenuTitles(final Resources
    res) {
    final List<String> menuList = new ArrayList<String>();
-   strArray = res.getStringArray(R.array.
     navigation_main_menu_titles);
+   strArray = res.getStringArray(R.array.
     navigation_main_titles);
    Collections.addAll(menuList,  strArray)
    if (!FeatureConfig.USE_OPENDATA) {
        menuList.remove(MENU_STATISTICS_INDEX);
    }
    return menuList;
}
```

```
import us.ihmc.commons.MathTools;

@Override
public void initialize() {
    currentTime.set(0.0);
-   MathTools.setCubic(trajectoryTime.getDoubleValue(),
     0.0, Double.POSITIVE_INFINITY);
+   MathTools.checkIntervalContains(trajectoryTime.
     getDoubleValue(), 0.0, Double.POSITIVE_INFINITY);
    parameterPolynomial.setQuintic(0.0, trajectoryTime.
         getDoubleValue(), 0.0, 0.0, 0.0, 1.0, 0.0, 0.0);

    currentOrientation.set(initialOrientation);
    currentAngularVelocity.setToZero();
    currentAngularAcceleration.setToZero();
}
```

Fig. 5. Diff-formatted Examples of **hallucination misuse**: non-existent **`setCubic`** instead of **`checkIntervalContains`** in Java.

3.8.0.2 LLM-Specific – Hallucination Misuse: ***Hallucination occurs when a model introduces an entirely***

incorrect API method or parameter that does not exist or is not required for the task. The model may fabricate an API call that seems plausible but is invalid in context, leading to erroneous behavior or compilation/runtime errors. Although prior work has discussed hallucination broadly in LLMs [59]–[61], this work presents the first systematic categorization of hallucination as an API misuse type, where models generate non-existent API methods or parameters. We adapt existing taxonomies by explicitly introducing this category to capture a recurring and distinctive error pattern in LLM-generated code. As shown in Figure 5, the model incorrectly predicts `res.getStringArray(R.array.navigation _main_menu_titles)` instead of the correct `res.getStringArray(R.array.navigation _main_titles)`. The fabricated parameter name represents a case of *parameter hallucination*. In the same figure, the model generates a non-existent method call `parameterPolynomial.setCubic(...)` in place of the valid `MathTools.checkIntervalContains()`, illustrating *method hallucination*. Both cases highlight how LLMs may generate plausible but invalid APIs, a failure mode that is not emphasized in prior human-centric misuse studies but emerges prominently in code completion tasks.

```
import org.chromium.base.PathUtils;

public static void loadLibrary() {
    Context appContext = ContextUtils.
         getApplicationContext();
-   PathUtils.setPrivateDataDirectorySuffix(
     PRIVATE_DATA_DIRECTORY_SUFFIX);
+   PathUtils.setPrivateDataDirectorySuffix(
     PRIVATE_DATA_DIRECTORY_SUFFIX, appContext);

    try {
        LibraryLoader libraryLoader = LibraryLoader.get(
             LibraryProcessType.PROCESS_WEBVIEW);
        libraryLoader.loadNow(appContext);
        libraryLoader.switchCommandLineForWebView();
    } catch (ProcessInitException e) {
        throw new RuntimeException("Cannot load WebView",
             e);
    }
}
```

Fig. 6. A diff-formatted example of **missing item misuse**: Omission of the required **`appContext`** parameter in **`setPrivateDataDirectorySuffix`** in Java.

3.8.0.3 Generic – Missing Item Misuse [18]: ***Missing item misuse occurs when the model omits a required API method or parameter, which results in incomplete or incorrect behavior. The model may fail to include a necessary parameter or method, leading to errors or malfunctioning of the system. This type of misuse typically occurs when the model does not fully understand the API's requirements or ignores important details that are essential for correct usage.*** In Figure 6, the model incorrectly predicts the API usage by omitting a necessary parameter from the method `PathUtils.setPrivateDataDirectorySuffix()`. The expected method call should include both the `PRIVATE_DATA_DIRECTORY_SUFFIX` constant and the `appContext` parameter. The model fails to include `appContext`, which is required for the method to properly set the private data directory suffix. This omission leads to parameter omission misuse, where the model overlooks a crucial parameter in the API call, causing potential runtime errors or unexpected behavior.

```
import org.chromium.base.PathUtils;

public static void loadLibrary() {
    Context appContext = ContextUtils.
         getApplicationContext();
    LibraryLoader libraryLoader = LibraryLoader.get(
         LibraryProcessType.PROCESS_WEBVIEW);

    try {
        // Redundant method chaining
-       libraryLoader.loadNow(appContext).initialize();
     // initialize() is unnecessary
+       libraryLoader.loadNow(appContext);  // Correct:
     loadNow() is sufficient

        libraryLoader.switchCommandLineForWebView();
    } catch (ProcessInitException e) {
        throw new RuntimeException("Cannot load WebView",
             e);
    }
}
```

Fig. 7. A diff-formatted example of **redundancy misuse**: Unnecessary chaining of **`initialize()`** after **`loadNow()`** in Java. According to the official API documentation, `loadNow()` already performs the necessary initialization, so an additional call to `initialize()` has no effect and is therefore redundant.

3.8.0.4 Generic – Redundancy [18]: ***Redundancy misuse occurs when the model includes unnecessary method chaining or repeated API calls, which do not contribute to the task and lead to inefficiency or potential errors.*** In the example provided in Figure 7, the model incorrectly chains the `initialize()` method after `loadNow()`, even though `loadNow()` already performs the necessary initialization. This redundant method of chaining can lead to confusion and potential side effects. The correct behavior would involve using only the necessary method call, `loadNow()`, which is sufficient for the task.

4 Results

4.1 RQ1: How well do LLMs handle API invocation in code generation?

TABLE 1
Performance comparison of StarCoder, Copilot, and Qwen2.5-Coder in Python and Java for method and parameter prediction tasks. EM is reported as percentage over 3,000 samples.

	Model	**Python**			**Java**		
		BLEU	CodeBERTScore	EM (%)	BLEU	CodeBERTScore	EM (%)
Method	StarCoder	84.6	92.3	75.6	81.2	86.4	44.3
	Copilot	88.3	92.6	63.7	86.1	87.7	53.1
	Qwen2.5-Coder	92.5	95.2	85.4	91.4	92.3	61.2
Parameters	StarCoder	85.0	92.8	77.1	82.0	87.0	44.3
	Copilot	89.0	93.0	65.2	87.0	88.2	53.9
	Qwen2.5-Coder	93.4	91.1	80.2	93.6	94.7	63.5

We first wanted to investigate how well our studied LLMs can generate code to correctly invoke APIs in code generation. As described above, we sample 3,000 distinct code snippets in Python and Java, respectively, from The Stack dataset. After prompting StarCoder, Copilot, and Qwen2.5-Coder to complete *Method Prediction* and *Parameter*

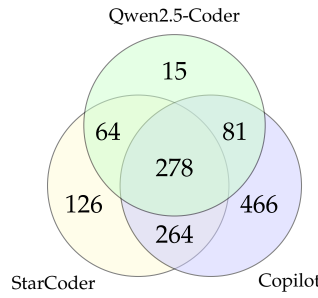
Generation independently, we collect 36,000 samples in total. We measure the correctness of the API invocation using three metrics introduced in subsection 3.6: BLEU, CodeBERTScore, and EM. Table 5 summarizes the performance of the three models on in-the-wild API usage tasks for both Python and Java.

For Python, the results reveal distinct trends across the models. Copilot and Qwen2.5-Coder achieve higher BLEU (88.3 and 92.5) and CodeBERTScore (92.6 and 95.2) than StarCoder (84.6 and 92.3), indicating that they produce more fluent and semantically consistent completions. However, StarCoder maintains competitive exact match performance (75.6%), while Qwen2.5-Coder achieves the highest EM (85.4%), suggesting that it not only generates semantically correct invocations but also matches human-written references more precisely. This improvement likely results from Qwen2.5-Coder's stronger multilingual code representation and its extensive code-centric training corpus. The divergence between EM and the other metrics highlights how LLMs may generate plausible but syntactically different API invocations, a challenge that remains especially noticeable in Copilot's outputs.

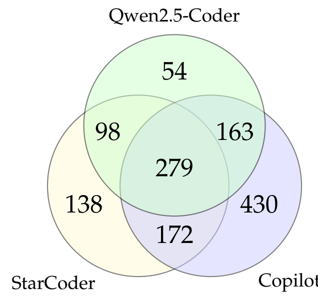
For Java, both Copilot and Qwen2.5-Coder outperform StarCoder across all metrics. Copilot achieves an EM of 53.1%, while Qwen2.5-Coder achieves 61.2%, compared to StarCoder's 44.3%. Similar trends are observed for BLEU and CodeBERTScore, where Qwen2.5-Coder leads (BLEU 91.4, CodeBERTScore 92.3), followed by Copilot (86.1 and 87.7) and StarCoder (81.2 and 86.4). This indicates that Qwen2.5-Coder, although not instruction-tuned, generalizes better to Java APIs, possibly due to broader coverage and better pretraining on strongly typed languages.

The comparison among these three decoder-based models highlights how pretraining strategies and dataset diversity shape model behavior. StarCoder shows strong memorization and syntactic alignment in Python due to its Python-focused fine-tuning. Copilot achieves consistent semantic fluency across languages through large-scale, multi-language exposure, while Qwen2.5-Coder balances both, delivering strong exact matches and high fluency simultaneously. This suggests that newer open-source models are closing the performance gap with proprietary systems in real-world API invocation tasks.

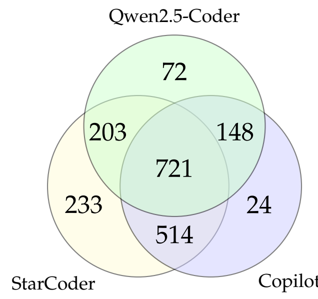
Figure 8 shows the overlap and discrepancies in API misuse among the three models. In Python Method Infilling (8a), StarCoder has 126 unique errors, Copilot 466, and Qwen2.5-Coder 15, with substantial overlaps where 278 errors are shared by all three models. Similar patterns hold for Java Method Infilling (8c), where StarCoder has 233 unique errors, Copilot 24, and Qwen2.5-Coder 72, again with a large shared subset (721). For parameter generation, the models exhibit analogous overlap patterns where shared errors dominate, although Qwen2.5-Coder consistently produces fewer unique misuses. These results indicate that while Qwen2.5-Coder shows fewer unique errors, all three models converge on many of the same failure cases, reflecting persistent limitations in LLM reasoning about API semantics. The observed differences can be traced back to each model's training focus. StarCoder's specialization in Python contributes to high syntactic fidelity but limited cross-language transfer, Copilot benefits from multi-language fine-tuning but sacrifices exactness, and Qwen2.5-Coder attains strong generalization through diverse code-centric pretraining without instruction tuning.



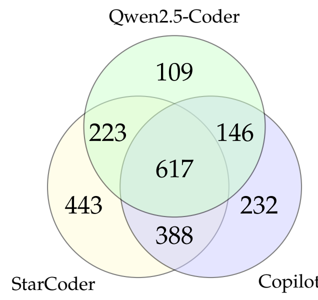
(a) Wrong Method Infilling in Python



(b) Wrong Parameter Generation in Python



(c) Wrong Method Infilling in Java



(d) Wrong Parameter Generation in Java

Fig. 8. Overlap of incorrect API usages among StarCoder, Copilot, and Qwen2.5-Coder for method infilling and parameter generation in Python and Java.

Finding 1: Qwen2.5-Coder achieves the highest overall accuracy across both languages, combining strong fluency with precise matching. Copilot follows closely, while StarCoder remains competitive in Python due to domain specialization. Despite these differences, all three models share a substantial number of common API misuses, which shows that correct API invocation remains a fundamental challenge for current LLMs.

4.2 RQ2: How do LLMs misuse APIs in the wild?

TABLE 2
Comparison of API Misuse by StarCoder in Java and Python. "None" indicates that the LLM-based API usage was considered acceptable based on human inspection.

Misuse Type	Java		Python	
	Params (%)	Methods (%)	Params (%)	Methods (%)
Intent	5.5	27.0	7.4	30.4
Hallucination	43.3	34.6	31.3	40.9
Redundancy	8.2	10.2	14.8	5.3
Missing Item	15.0	11.3	26.5	3.0
None	28.0	16.9	20.0	20.4

TABLE 3
Comparison of API Misuse by Copilot in Java and Python. "None" indicates that the LLM-based API usage was considered acceptable based on human inspection.

Misuse Type	Java		Python	
	Params (%)	Methods (%)	Params (%)	Methods (%)
Intent	4.0	22.0	6.0	25.0
Hallucination	39.0	29.0	25.0	33.0
Redundancy	7.0	12.0	12.0	7.0
Missing Item	15.0	12.5	29.0	7.7
None	35.0	24.5	28.0	27.3

TABLE 4
Comparison of API Misuse by Qwen2.5-Coder in Java and Python. "None" indicates that the LLM-based API usage was considered acceptable based on human inspection.

Misuse Type	Java		Python	
	Params (%)	Methods (%)	Params (%)	Methods (%)
Intent	4.2	20.5	6.5	23.6
Hallucination	33.5	26.8	23.0	29.4
Redundancy	6.3	10.5	10.7	6.4
Missing Item	13.0	11.7	25.4	8.1
None	43.0	30.5	34.4	32.5

We further analyze the patterns and distributions of API misuse across StarCoder, Copilot, and Qwen2.5-Coder. Specifically, we study the annotated outputs introduced in subsection 3.7, which correspond to samples that failed on EM score. Table 2, Table 3, and Table 4 show the distribution of API misuse types across Java and Python, categorized by methods and parameters.

Hallucination remains the most frequent misuse across all three models, although its prevalence varies. For StarCoder, hallucinations dominate all configurations, accounting for up to 43.3% of parameter misuses in Java. Copilot shows slightly reduced rates, while Qwen2.5-Coder exhibits further decreases, with hallucination rates below 30% in most cases. This indicates that Qwen2.5-Coder is better grounded in valid API namespaces, reducing the tendency to generate non-existent or version-inconsistent APIs.

Missing Item misuses are prominent for Python parameters, particularly for Copilot and Qwen2.5-Coder. Copilot omits required parameters in 29.0% of Python parameter cases, making it the leading error type for that configuration. Qwen2.5-Coder also shows relatively high omission rates (25.4%), although slightly lower than Copilot's. This pattern suggests that the challenge of inferring correct argument lists persists even for newer models, especially in Python where argument flexibility and version heterogeneity increase complexity.

Intent misuses continue to appear frequently but are less severe in Qwen2.5-Coder. In Python, StarCoder records 30.4% intent errors for methods, while Copilot and Qwen2.5-Coder reduce these to 25.0% and 23.6% respectively. Similar trends appear in Java. These improvements suggest that newer models capture functional intent more reliably, although confusion between similar API methods still occurs.

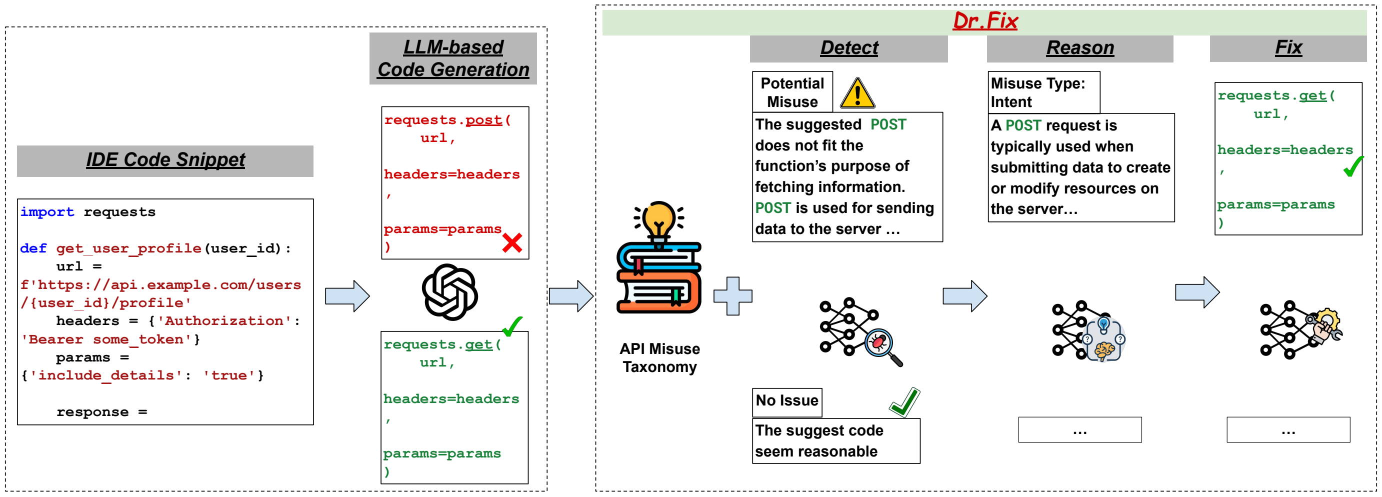
Redundancy misuses are less frequent but remain consistent across models. Qwen2.5-Coder reduces redundant arguments and superfluous method calls compared to StarCoder and Copilot, showing the lowest rates overall (around 6–10%). This aligns with its improved parameter consistency and contextual inference capabilities.

Overall, Python parameter completion remains the most error-prone scenario. The results across all three models indicate that Python's dynamic typing and overlapping library interfaces still pose major challenges. Frequent hallucination and omission misuses demonstrate that LLMs struggle to fully align API invocations with real-world library constraints, especially for parameters that require contextual understanding of object states or external imports.

Finding 2: Hallucination remains the most common misuse overall, although Qwen2.5-Coder reduces its occurrence compared with StarCoder and Copilot. Missing Item errors remain prominent in Python parameter prediction for all models. Intent misuses are frequent in method prediction, but Qwen2.5-Coder demonstrates improved grounding and fewer redundant errors, suggesting gradual progress toward more reliable API usage.

5 MITIGATION

Mitigating LLM-based API misuse in real-world applications presents several challenges. Existing automatic misuse detection methods often require substantial manual effort to apply fixes, underscoring the need for approaches that can both detect and repair misuses. However, program repair methods tailored specifically for API misuse remain largely unexplored. The only relevant work is [15], which evaluated pre-trained models fine-tuned on large-scale git commit patches, but focused only on Java. More recently, Wei et al.



```
import requests

def get_user_profile(user_id):
    url =
f'https://api.example.com/users
/{user_id}/profile'
    headers = {'Authorization':
'Bearer some_token'}
    params =
{'include_details': 'true'}

    response =
```

```
requests.post(
    url,

headers=headers
,

params=params
)
```

```
requests.get(
    url,

headers=headers
,

params=params
)
```

The suggested POST does not fit the function's purpose of fetching information. POST is used for sending data to the server ...

The suggest code seem reasonable

A POST request is typically used when submitting data to create or modify resources on the server...

```
requests.get(
    url,

headers=headers
,

params=params
)
```

Fig. 9. Workflow of Dr.Fix to repair the API misuse.

proposed LLMAPIDet [18], a detector and repair tool for deep learning libraries based on ChatGPT. While effective in that domain, it is limited by its reliance on closed-source models and its focus on TensorFlow and PyTorch APIs. By contrast, Dr.Fix is designed to be backbone-agnostic and taxonomy-driven, making it applicable to general-purpose third-party APIs across different programming languages.

Repairing LLM-generated API misuses is particularly important because such misuses often arise from the way LLMs generate code, rather than from human misunderstanding of documentation. Unlike traditional APR methods, which rely on static analysis or heuristics, LLM-based repair can leverage contextual knowledge of APIs and natural language reasoning to propose fixes. This enables handling of misuse types such as hallucinations or incorrect intent selections, which are less common in human-written code but prevalent in LLM outputs. At the same time, it is valuable to investigate whether LLMs reproduce the same kinds of misuses as humans, or if they introduce distinctive patterns that call for new repair strategies.

We explore a promising direction for API misuse repair: prompt-based LLM program repair. Unlike conventional APR methods trained on task-specific datasets, prompt-based approaches offer greater flexibility and generalization. Prior work has shown that LLMs are capable of self-repair and debugging through structured prompting [62], motivating our design of Dr.Fix.

5.1 Dr.Fix: Repairing API Misuse via Detection and Reasoning

Repairing LLM-based API misuses requires nuanced semantic understanding, particularly of the implicit intents within the code context. While LLMs can generate repairs when directly prompted, prior work shows that naive prompting often produces syntactically valid but semantically incomplete or inconsistent fixes [62]–[65]. Moreover, recent studies on domain-specific APIs highlight that LLM-based repair is sensitive to how misuses are detected and framed, and simple prompts may overlook subtle but critical constraints [18], [66]. These observations motivate the need for structured prompting strategies that incorporate detection, classification, and reasoning, rather than relying on single-pass repair. At the same time, traditional APR methods, reliant on static analysis or heuristic rules, remain limited when applied across diverse programming environments.

Figure 9 overviews our Dr.Fix approach, which integrates detection and reasoning capabilities through multiple stages of LLM prompting. The process begins with the detection stage, where the model is prompted to recognize potential API misuse within the code. Once potential misuses are identified, a taxonomy classification stage follows, mapping them into predefined categories such as intent or hallucination misuse. In the reasoning stage, the model validates the detected misuse with a detailed analysis that considers the broader context and intended functionality. This stage is motivated by self-verification reasoning [67], which has been effective in natural language processing tasks. Finally, the repair suggestion stage produces candidate fixes that aim to be both syntactically correct and semantically meaningful, aligning with the developer's intent.

Incorporating Dr.Fix into the development workflow reduces the manual effort required to address API misuses. By automating both detection and repair, Dr.Fix improves the reliability and efficiency of software development. Our design demonstrates the potential of combining LLM capabilities with structured reasoning techniques to address complex challenges in automated program repair.

5.2 Backbone Instruction-tuned Models

LLMs serve distinct purposes in software engineering workflows. Models designed for *IDE-based code completion* prioritize efficiency and responsiveness, enabling seamless interaction within development environments. In contrast, models aimed at *code repair* focus on semantic correctness and reliability rather than generation speed, as repair involves complex reasoning over code context and intent. Table 5 summarizes the speed and cost characteristics of representative models in both categories. While the IDE-based code completion results are derived from our previous experiments, in this work we concentrate on LLMs for API misuse repair when evaluating the performance of Dr.Fix. To ensure the highest possible repair quality, we adopt frontier LLMs which better represent the current technological upper bound in reasoning and code understanding. This distinction allows us to balance efficiency and precision while analyzing the contribution of repair-oriented LLMs to Dr.Fix's overall effectiveness.

TABLE 5
Model Comparison: Speed and Cost. Speed and cost for all models except StarCoder and Copilot are obtained from OpenRouter [68]. For StarCoder and Copilot, we measure inference speed locally; both are free to use in this setup. We note that the cost is measured per 1M tokens.

IDE			
Metric	**StarCoder**	**Copilot**	**Qwen2.5-Coder**
Speed (tokens/s)	217.4	135.0	229.0
Cost ($ in/out)	0.0 / 0.0	0.0 / 0.0	0.03 / 0.09
Repair			
Metric	**Llama 3.1**	**Qwen2.5-Coder-Instruct**	**GPT-4o**
Speed (tokens/s)	22.65	38.17	59.3
Cost ($ in/out)	0.4 / 0.4	0.04 / 0.16	2.5 / 10

5.2.0.1 Llama-3.1: Llama 3.1 [69] is a collection of multilingual LLMs, a collection of pre-trained and instruction-tuned generative models in 8B, 70B, and 405B sizes released by Meta AI. The Llama 3.1 instruction-tuned text-only models (8B, 70B, 405B) are optimized for multilingual dialogue use cases and outperform many available open-source and closed-chat models on common industry benchmarks. Compared to the original Llama 3 series, Llama-3.1 is significantly improved in terms of math and reasoning capabilities. In our evaluation, we use Llama-3.1 70B, a variant achieving state-of-the-art (SOTA) performance on various benchmarks.

5.2.0.2 Qwen2.5-Coder-Instruct: Qwen2.5-Coder-Instruct [47] is a series of code-specific generative models designed to excel in a wide range of programming tasks such as code generation, completion, reasoning, and repair. Built upon the Qwen2.5 architecture, these models come in sizes ranging from 0.5B to 32B parameters and have been pre-trained on a vast corpus of over 5.5 trillion tokens. Through a combination of extensive data cleaning, scalable synthetic data generation, and balanced data mixing, Qwen2.5-Coder-Instruct achieves SOTA performance across more than 10 code-related benchmarks. It consistently outperforms other models of similar size, showcasing impressive capabilities in code intelligence while retaining strong general and math skills. With its permissive licensing, Qwen2.5-Coder-Instruct is positioned to drive innovation in code-related AI research and be widely adopted by developers in real-world applications. We chose Qwen2.5-Coder-Instruct 32B, the most capable one among the series.

5.2.0.3 GPT-4o: GPT-4o[7] is a highly advanced variant of OpenAI's GPT-4 model, fine-tuned specifically for enhanced efficiency and specialized performance across various natural language processing and multimodal tasks. Available in different configurations, GPT-4o delivers superior reasoning, problem-solving, and contextual understanding capabilities compared to its predecessors. With a focus on optimizing performance while reducing computational overhead, GPT-4o excels in both complex language tasks and more specialized use cases, such as coding, translation, and content generation. Leveraging a blend of advanced training techniques, including reinforcement learning from human feedback, GPT-4o sets a new standard for AI models in terms of accuracy, responsiveness, and versatility, making it ideal for real-world applications across industries.

7. https://openai.com/index/hello-gpt-4o/

5.3 Hyperparameters

In this study, we set a maximum token length to 2048, a temperature of 0, and a top-p of 0.95. The few-shot learning approach [70] is used to prompt the models across four stages: detection, classification, reasoning, and repair suggestion. Specifically, we use two shots per stage as a demonstration to ensure the output format. For each type inside the API misuse taxonomy, we sample two failure cases to teach the models.

The prompts for each stage are designed as follows: (1) Detect - "Identify any incorrect usages in the following code snippet based on one of the following categories: Intent, Hallucination, Redundancy, Missing." The definition of each category is provided after the instructions. To prevent LLMs from hallucinating API misuse, we append "Please answer with "No Issue" if there is no obvious API misuse." at the end of the prompt. (2) Reason - "Think step by step and explain why this API usage is incorrect and how it deviates from the intended usage." (3) Fix - "Propose a correction that aligns with the intended functionality of the API."

The models' performance is evaluated based on BLEU, CodeBERTScore, EM accuracy, and refusal rate to assess their ability to detect and repair API misuses across different categories and programming languages.

The refusal rate is introduced as an additional evaluation metric inspired by [71]–[73]. It measures the model's ability to avoid hallucinating an API misuse when none exists in the code snippet. This score is defined as the percentage of instances where the model correctly refuses to detect a misuse when no misuse is present in the code. The formula for the refusal rate is:

$$\text{Refusal Rate} = \frac{\text{Number of Correct Rejections}}{\text{Total Number of No Misuse Samples}} \times 100$$

5.4 Previous Approaches

We use [15] as a representative baseline of learning-aided automated program repair (APR), which outperforms previous APR methods [74], [75]. Zhang et al. [15] systematically evaluated pre-trained models, including CodeBERT and CodeT5, on repairing API- and semantic-related bugs. Their approach fine-tunes PLMs on a large-scale dataset of human-written `git commit` patches and evaluates repair performance primarily on Java projects.

This work is relevant as it demonstrates how supervised PLMs can be adapted for code repair. However, it has two key limitations: (1) it focuses exclusively on Java and deep learning library APIs, which restricts its applicability to other programming languages such as Python, and (2) it depends on fine-tuning closed or specialized models, limiting generalization to diverse libraries and cross-language settings. In contrast, our work investigates prompt-based, instruction-tuned LLMs (Dr.Fix) that require no fine-tuning, operate on a general API misuse taxonomy, and extend beyond a single domain.

For comparison, we replicate their best-performing baselines, CodeBERT and CodeT5, for the Java Method and Parameter tasks, and report them alongside Dr.Fix in our evaluation.

TABLE 6
10% Sample of Misuse Types for Java and Python

Misuse Type	**Java**		**Python**	
	Params #	**Methods #**	**Params #**	**Methods #**
Intent	11	55	10	39
Hallucination	89	71	40	53
Redundancy	17	21	19	7
Missing	31	23	34	4
Total	149	133	105	98

5.5 Dataset

We sampled 10% of code snippets from each annotated misuse type of StarCoder and Copilot in Section 3. As shown in Table 6, the dataset includes a representative sample of API misuse types across Java and Python. Specifically, the dataset captures the following misuse types: Intent, Hallucination, Redundancy, Position, and Missing. For each type, a proportional number of samples was extracted based on their original distribution. To further enhance the evaluation of repair approaches, including Dr.Fix and other baselines, we introduced an additional 100 snippets that exactly match the reference source code. This addition allows us to test the ability of repair tools to avoid unnecessary hallucinations or misinterpretations of correct API usage. By including these reference-aligned snippets, we can measure the precision of tools in distinguishing between actual misuse and correct implementations.

5.6 Results Analysis

The performance of the models across different methods is summarized in Table 7. The results highlight the significant improvement in API misuse detection and repair capabilities achieved by incorporating the Dr.Fix methodology, with and without the taxonomy. This section provides an in-depth analysis of the findings for both Java and Python across the evaluated metrics.

Java Performance: For Java, **both Dr.Fix and Dr.Fix (w/o taxonomy) outperform the baseline method across all metrics**. In the *Method* misuse category, GPT-4o with Dr.Fix achieves the highest BLEU score, CodeBERTScore, and EM, demonstrating its ability to generate syntactically and semantically appropriate repairs. The intermediate version, Dr.Fix (w/o taxonomy), also shows substantial improvements, with GPT-4o achieving a BLEU score of 76.1

TABLE 7
Performance of Models across Different Methods. **Dr.Fix (w/o taxonomy)** means that the models are not prompted with the defined API misuse taxonomy and demonstrated examples, and perform the detection based on their own understanding. **Baseline** denotes the LLM-based APR method where the models are prompted to generate the fixed code snippet if they detect the API misuse inside the given snippet.

Model	Method	BLEU	CodeBERTScore	EM
	Java – Method			
CodeBERT	[15]	12.4	15.8	3
CodeT5	[15]	20.1	24.7	5
Llama-3.1	Baseline	30.5	25.7	35
	Dr.Fix (w/o taxonomy)	65.2	66.3	75
	Dr.Fix	68.9	69.2	80
Qwen2.5-Coder-Instruct	Baseline	36.3	28.4	24
	Dr.Fix (w/o taxonomy)	55.1	60.2	92
	Dr.Fix	58.3	62.6	96
GPT-4o	Baseline	57.8	46.1	52
	Dr.Fix (w/o taxonomy)	76.1	65.3	92
	Dr.Fix	79.4	68.2	96
	Java – Parameter			
CodeBERT	[15]	13.5	17.2	2
CodeT5	[15]	22.8	26.4	6
Llama-3.1	Baseline	53.6	64.6	34
	Dr.Fix (w/o taxonomy)	72.3	85.1	72
	Dr.Fix	75.8	88.3	78
Qwen2.5-Coder-Instruct	Baseline	55.7	73.3	21
	Dr.Fix (w/o taxonomy)	73.2	89.5	80
	Dr.Fix	76.4	92.3	85
GPT-4o	Baseline	62.7	82.6	26
	Dr.Fix (w/o taxonomy)	80.1	89.2	84
	Dr.Fix	82.5	91.5	88
	Python – Method			
Llama-3.1	Baseline	34.6	58.4	25
	Dr.Fix (w/o taxonomy)	54.2	87.6	55
	Dr.Fix	57.8	90.3	60
Qwen2.5-Coder-Instruct	Baseline	15.1	43.8	14
	Dr.Fix (w/o taxonomy)	49.3	81.2	48
	Dr.Fix	52.6	84.6	52
GPT-4o	Baseline	47.0	69.5	31
	Dr.Fix (w/o taxonomy)	60.5	90.1	66
	Dr.Fix	63.7	92.7	71
	Python – Parameter			
Llama-3.1	Baseline	78.7	75.7	12
	Dr.Fix (w/o taxonomy)	82.1	84.3	40
	Dr.Fix	84.5	86.8	45
Qwen2.5-Coder-Instruct	Baseline	81.2	62.3	10
	Dr.Fix (w/o taxonomy)	85.1	87.2	52
	Dr.Fix	87.4	89.4	56
GPT-4o	Baseline	84.8	67.2	11
	Dr.Fix (w/o taxonomy)	86.9	90.8	34
	Dr.Fix	88.6	92.4	38

and an EM of 92, which further enhance to 79.4 and 96, respectively, with the full taxonomy. Similarly, Qwen2.5-Coder-Instruct and Llama-3.1 show marked improvements with both versions of Dr.Fix, with BLEU scores increasing from 36.3 to 58.3 and from 30.5 to 68.9, respectively. In the *Parameter* misuse category, the results follow a similar trend. GPT-4o achieves the best performance with BLEU, CodeBERTScore, and EM. Dr.Fix (w/o taxonomy) notably enhances the performance of all models, with improvements in both syntactic and semantic alignment, as reflected in the BLEU and CodeBERTScore metrics, which are further boosted by the full taxonomy.

In addition, we include Zhang et al. [15] as representative learning-based APR baselines. Their performance on both Java *Method* and *Parameter* repair is substantially lower than instruction-tuned LLMs. For instance, CodeBERT and CodeT5 achieve BLEU scores of 12.4 and 20.1 on the *Method* task, and 13.5 and 22.8 on the *Parameter* task, respectively, which are far below GPT-4o with Dr.Fix (79.4 and 82.5). This highlights the advantage of instruction-tuned LLMs, especially when guided by structured misuse taxonomies. This gap highlights that fine-tuned PLMs trained on commit-level patches are far less effective for API misuse repair than prompt-based LLMs. Moreover, these approaches lack awareness of the distinctive misuse patterns emerging in LLM-generated code, such as hallucination and intent misuses, limiting their applicability in this setting. These results further confirm that Dr.Fix offers not only relative gains over naive prompting baselines, but also a clear advantage over prior supervised APR approaches.

Python Performance: For Python, **both Dr.Fix and Dr.Fix (w/o taxonomy) demonstrate significant improvements over the baseline method**. In the *Method* misuse category, GPT-4o achieves the highest BLEU score and CodeBERTScore, indicating superior generalization and repair accuracy. The EM score also improves from 31 to 71 with the full taxonomy, while Dr.Fix (w/o taxonomy) achieves an EM of 66. Similarly, Qwen2.5-Coder-Instruct and Llama-3.1 achieve notable gains in BLEU and CodeBERTScore metrics with both versions of Dr.Fix, showcasing the effectiveness of the methodology in improving code repair suggestions. In the *Parameter* misuse category, all models exhibit enhanced performance with Dr.Fix. GPT-4o achieves a BLEU score of 88.6 and a CodeBERTScore of 92.4 with the full taxonomy, surpassing other models in both accuracy and precision. While the baseline method performs reasonably well in this category, both versions of Dr.Fix provide a clear advantage, ensuring that repair suggestions align more closely with developer intentions, with the full taxonomy offering the most significant improvements.

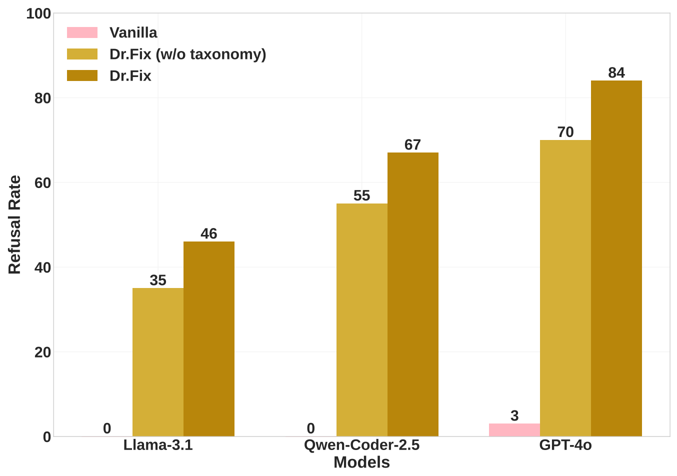


Fig. 10. Refusal rate comparison of Llama-3.1, Qwen2.5-Coder-Instruct, and GPT-4o with and without Dr.Fix.

Refusal Rate: The refusal rate metric evaluates the models' ability to avoid hallucinating API misuses when no misuse exists. As shown in Figure 10, **Dr.Fix significantly enhances the refusal rates across all tested models**. GPT-4o achieved the highest refusal rate of 84% with Dr.Fix, demonstrating its exceptional ability to differentiate between correct and incorrect API usage. Similarly, Qwen2.5-Coder-Instruct and Llama-3.1 showed substantial improvements, achieving refusal rates of 67% and 46%, respectively, compared to their baseline configurations, which had refusal rates of 0%, 55%, and 35%, respectively. Dr.Fix (w/o taxonomy) also showed promising results, with refusal rates of 70%, 55%, and 35% for GPT-4o, Qwen2.5-Coder-Instruct, and Llama-3.1, respectively. Notably, Dr.Fix slightly decreases the refusal rate when no API misuse taxonomy is given for the model, suggesting that it effectively reduces false positives by avoiding unnecessary refusals for correct API usage. These results highlight Dr.Fix's effectiveness in reducing hallucinated API misuses across diverse LLM architectures, with the full taxonomy version providing the most significant improvements.

Our experimental results demonstrate that Dr.Fix significantly enhances API misuse repair across all models, achieving up to 130% improvement in exact match rates compared to baseline approaches while maintaining high code quality. The framework reduces refusal errors by 14-20% absolute percentage points across different architectures, showcasing its effectiveness as a reliable program repair solution for diverse programming languages.

6 Threats to Validity

6.1 Internal Validity

One potential threat to internal validity is the reliance on subjective human annotations for the initial classification of API misuse. Although two annotators with over five years of programming experience independently reviewed the samples, biases in understanding or interpretation of the API misuse may have influenced the categorization. We mitigated this by ensuring high inter-annotator agreement, as indicated by the high Cohen's Kappa score, and resolving disagreements through detailed discussions. Another threat arises from the inherent variability in LLM outputs due to stochastic sampling. To address this, we kept the model parameters fixed across experiments, varying only the prompts as necessary. However, given resource constraints, experiments were conducted once, and results may vary under repeated evaluations. Future studies should consider multiple iterations to assess consistency and variability in repair suggestions. Finally, our closed-coding process to adopt taxonomy categories may have introduced subjectivity. While grounded in prior work, different coding decisions could lead to slightly different categorization. We chose closed coding because prior studies [18], [19] on API misuse provide taxonomies that can be readily adapted with minor modifications, making them suitable for our context. This approach allows us to capture patterns relevant to LLM-generated code without reinventing categories from scratch, though it carries a risk of overlap with existing classifications. Another potential threat concerns the breadth of our API completion tasks. We mitigated this by sampling 3,000 distinct API signatures each for Python and Java from The Stack, covering overloads and version-specific variants through static analysis. While this approach ensures diversity across frequently used APIs, the topical coverage of the underlying packages is difficult to determine, since open-source repositories often mix multiple domains and lack consistent metadata. As a result, our dataset may not fully capture the distribution of APIs across specific application areas, and we leave a more fine-grained analysis of domain coverage for future work.

6.2 External Validity

A significant threat to external validity is the generalizability of the findings to other programming languages, APIs, and LLMs. Our dataset is limited to Python and Java, focusing on specific API misuse categories. While these languages and categories are widely used and relevant, the results may not directly extend to other contexts, such as low-level languages or highly specialized APIs. Similarly, the LLMs evaluated for Dr.Fix in this study, including GPT-4o, Qwen2.5-Coder-Instruct, and Llama-3.1, represent a subset of available models. The effectiveness of Dr.Fix with other LLMs or on alternative programming datasets remains uncertain. Furthermore, our comparison with prior APR baselines (e.g., CodeBERT and CodeT5) is limited to Java and deep learning APIs, following the scope of that work. Hence, relative improvements may differ in other ecosystems. Moreover, because our dataset emphasizes frequently used APIs, the results may not fully generalize to rarely used or deprecated APIs, which could exhibit distinct misuse patterns or outdated usage behaviors. Finally, while our analysis is grounded in large-scale manual annotation, this process is extremely resource-consuming and limits the breadth of models and datasets we could feasibly include. As such, our findings may not generalize to other LLM families not studied here, and we leave broader cross-model validation to future work.

6.3 Construct Validity

Construct validity may be threatened by the choice of evaluation metrics and dataset design. Metrics such as BLEU and CodeBERTScore, while widely used, primarily measure syntactic and semantic similarity and may not fully capture the functional correctness of the repairs. LLMs-as-Judges [76], [77] is hence a more promising direction to explore for code quality evaluation. Moreover, the inclusion of 100 snippets without API misuse to evaluate refusal rates introduces a potential bias in the dataset composition. Although this addition aims to measure the models' ability to avoid hallucinating misuse, its proportion relative to actual misuse cases may influence the results. Finally, our use of human-written code from The Stack as the ground truth may embed biases or errors, and multiple valid API usages could be penalized under our evaluation setup. We also rely on the open-source community's sustained effort to ensure the quality and relevance of The Stack, which may introduce variability beyond our control. In addition, the EM metric, while capturing structural matches at the AST level, may conflate syntactic similarity with semantic correctness, limiting its interpretability. Future work could explore alternative metrics or evaluation methods that better assess functional correctness and developer usability.

7 Related Work

7.1 Large Language Models for Code

LLMs have emerged as a transformative approach for code generation tasks, broadly categorized into standard language models and instruction-tuned models. Standard language models are pre-trained on raw corpora using next-token prediction, exemplified by models such as Codex [78], CodeGen [79], CodeGeeX [80], CodeT5 [81] and StarCoder [3]. Codex, with 12B parameters, and StarCoder, featuring 15.5B parameters, demonstrated significant advancements in program synthesis through training on GitHub code and additional sources. Instruction-tuned models represent the next evolution, with examples like ChatGPT employing Reinforcement Learning with Human Feedback [82]. Open-source alternatives include WizardCoder [83], which fine-tunes StarCoder using techniques like Evol-Instruct, and InstructCodeT5+ [84], which extends CodeT5 through instruction-tuned processes. These developments have enabled zero-shot capabilities in code generation tasks, demonstrating emergent behaviors that were previously unattainable while maintaining the essential citations and academic format.

7.2 Bug Study of Large-Language-Model-Generated Code

Understanding bugs in LLM-generated code is crucial for improving code generation capabilities. While prior studies have analyzed bugs in deep learning systems broadly [85]–[87], recent research has focused specifically on defects in LLM-generated code [88]. [89] investigated bugs in non-standalone code generated by LLMs, though their study was limited by potential data leakage issues and the use of now-outdated models. [90] addressed these limitations by evaluating LLM-generated code with a robust benchmark designed to minimize data leakage, providing a more accurate representation of model performance in practical applications. Our work builds upon these insights by focusing specifically on API misuse patterns in LLM-generated code, introducing a taxonomy of error types and proposing the Dr.Fix repair mechanism.

7.3 Program Repair by Large Language Models

The success of LLMs in Natural Language Processing has led to their application in Automated Program Repair (APR) [91], [92]. Key advancements include integrating LLMs with additional contextual information, such as TFix [93], which builds on the T5 model [94] by incorporating error messages from diagnostic tools to improve repair accuracy. Specialized models for code review and repair have also emerged [95], [96], along with zero-shot learning approaches that frame repair as a cloze-style task. Notable examples include Xia and Zhang's work [92] using masked language modeling and CIRCLE [97]'s prompt-based approach for code completion. Our work, Dr.Fix, builds upon these foundations by specifically addressing API misuse in generated code, combining detection, reasoning, and repair capabilities across Python and Java codebases.

8 Conclusion and Future Work

In this work, we systematically studied API misuse in LLM-generated code, analyzing thousands of code snippets across Python and Java. Through rigorous manual annotation, we developed a comprehensive taxonomy of API misuse types and quantified their prevalence in popular code generation models. Our findings reveal that LLMs struggle significantly with hallucination and intent misalignment, with distinct patterns emerging between programming languages. The shared error patterns between Copilot and StarCoder highlight fundamental challenges in API usage that persist across different model architectures. We introduced Dr.Fix, a novel LLM-based APR approach that combines detection and reasoning capabilities to address these challenges. Our evaluation demonstrates that Dr.Fix significantly outperforms existing learning-based APR methods, substantially improving repair accuracy and maintaining high refusal rates for correct API usage. The results underscore the effectiveness of our staged approach in understanding and fixing API misuse.

Future work should focus on extending Dr.Fix's capabilities to handle more complex API misuse scenarios and investigating its applicability to other programming languages. Additionally, exploring the integration of Dr.Fix with development environments and investigating ways to improve its reasoning capabilities through enhanced prompting strategies could further advance automated API repair. We also plan to expand our taxonomy and dataset as new patterns of API misuse emerge with the evolution of code generation models.

Acknowledgements

TY.Zhuo is supported by CSIRO's Data61 PhD Scholarships and IBM PhD Fellowship Awards. J.Grundy is supported by ARC Laureate Fellowship FL190100035. This research was supported by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant (Project ID: 23-SOL-SMU-004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

References

[1] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[3] R. Li, L. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: May the source be with you!" *Transactions on machine learning research*, 2023.

[4] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[5] N. Muennighoff, Q. Liu, A. R. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. V. Werra, and S. Longpre, "Octopack: Instruction tuning code large language models," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=mw1PWNSWZP

[6] T. Y. Zhuo, Q. Liu, Z. Wang, W. U. Ahmad, B. Hui, and L. B. Allal, "NLP+Code: Code intelligence in language models," in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*, V. Pyatkin and A. Vlachos, Eds. Suzhou, China: Association for Computational Linguistics, Nov. 2025, pp. 9–11. [Online]. Available: https://aclanthology.org/2025.emnlp-tutorials.4/

[7] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul *et al.*, "Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions," *arXiv preprint arXiv:2406.15877*, 2024.

[8] H. Zhong and H. Mei, "An empirical study on api usages," *IEEE Transactions on Software Engineering*, vol. 45, no. 4, pp. 319–334, 2017.

[9] N. Islah, J. Gehring, D. Misra, E. Muller, I. Rish, T. Y. Zhuo, and M. Caccia, "Gitchameleon: Unmasking the version-switching capabilities of code generation models," *arXiv preprint arXiv:2411.05830*, 2024.

[10] S. Kuhar, W. U. Ahmad, Z. Wang, N. Jain, H. Qian, B. Ray, M. K. Ramanathan, X. Ma, and A. Deoras, "Libevolutioneval: A benchmark and study for version-specific code generation," *arXiv preprint arXiv:2412.04478*, 2024.

[11] L. Zhong and Z. Wang, "Can llm replace stack overflow? a study on robustness and reliability of large language model code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 19, 2024, pp. 21 841–21 849.

[12] Z. Mousavi, C. Islam, K. Moore, A. Abuadbba, and M. A. Babar, "An investigation into misuse of java security apis by large language models," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1299–1315.

[13] F. F. Xu, B. Vasilescu, and G. Neubig, "In-ide code generation from natural language: Promise and challenges," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–47, 2022.

[14] M. Kechagia, S. Mechtaev, F. Sarro, and M. Harman, "Evaluating automatic program repair capabilities to repair api misuses," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2658–2679, 2021.

[15] T. Zhang, I. C. Irsan, F. Thung, D. Lo, A. Sharma, and L. Jiang, "Evaluating pre-trained language models for repairing api misuses," *arXiv preprint arXiv:2310.16390*, 2023.

[16] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, S. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," in *The Eleventh International Conference on Learning Representations*.

[17] A. Radford, "Improving language understanding by generative pre-training," 2018.

[18] M. Wei, N. S. Harzevili, Y. Huang, J. Yang, J. Wang, and S. Wang, "Demystifying and detecting misuses of deep learning apis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[19] X. He, X. Liu, L. Xu, and B. Xu, "How dynamic features affect api usages? an empirical study of api misuses in python programs," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 522–533.

[20] "Replication package," https://github.com/terryyz/llm_api_misuse, 2025, replication package submitted for artifact evaluation.

[21] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2018.

[22] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software: Practice and Experience*, vol. 30, no. 7, pp. 775–802, 2000.

[23] R. Khatchadourian, Y. Tang, M. Bagherzadeh, and B. Ray, "An empirical study on the use and misuse of java 8 streams." in *FASE*, 2020, pp. 97–118.

[24] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, "A systematic review of api evolution literature," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–36, 2021.

[25] M. Schlichtig, S. Sassalla, K. Narasimhan, and E. Bodden, "Fum-a framework for api usage constraint and misuse classification," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 673–684.

[26] X. Ren, J. Sun, Z. Xing, X. Xia, and J. Sun, "Demystify official api usage directives with crowdsourced api misuse scenarios, erro-

neous code examples and patches," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 925–936.

[27] A. Galappaththi, S. Nadi, and C. Treude, "An empirical study of api misuses of data-centric libraries," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024, pp. 245–256.

[28] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: A benchmark for api-misuse detectors," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 464–467.

[29] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 302–321, 2010.

[30] M. Kechagia, X. Devroey, A. Panichella, G. Gousios, and A. Van Deursen, "Effective and efficient api misuse detection via exception propagation and search-based testing," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 192–203.

[31] H. J. Kang and D. Lo, "Active learning of discriminative subgraph patterns for api misuse detection," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2761–2783, 2021.

[32] Z. Li, A. Machiry, B. Chen, M. Naik, K. Wang, and L. Song, "Arbitrar: User-guided api misuse detection," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1400–1415.

[33] M. Monperrus and M. Mezini, "Detecting missing method calls as violations of the majority rule," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, pp. 1–25, 2013.

[34] M. Pradel and T. R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 288–298.

[35] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking api protocol conformance with mined multi-object specifications," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 925–935.

[36] S. Proksch, S. Amann, S. Nadi, and M. Mezini, "Evaluating the evaluations of code recommender systems: a reality check," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 111–121.

[37] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, 2014, pp. 419–428.

[38] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[39] F. Cassano, M.-H. Yee, N. Shinn, A. Guha, and S. Holtzen, "Type prediction with program decomposition and fill-in-the-type training," *arXiv preprint arXiv:2305.17145*, 2023.

[40] Y. Peng, C. Wang, W. Wang, C. Gao, and M. R. Lyu, "Generative type inference for python," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 988–999.

[41] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1433–1443.

[42] M. Izadi, J. Katzy, T. Van Dam, M. Otten, R. M. Popescu, and A. Van Deursen, "Language models for code completion: A practical evaluation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[43] W. Takerngsaksiri, C. Warusavitarne, C. Yaacoub, M. H. K. Hou, and C. Tantithamthavorn, "Students' perspectives on ai code completion: Benefits and challenges," in *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2024, pp. 1606–1611.

[44] A. Semenkin, V. Bibaev, Y. Sokolov, K. Krylov, A. Kalina, A. Khannanova, D. Savenkov, D. Rovdo, I. Davidenko, K. Karnaukhov *et al.*, "Full line code completion: Bringing ai to desktop," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2025, pp. 563–574.

[45] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "Serverlessllm: Low-latency serverless inference for large language models," in *18th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2024, pp. 135–153.

[46] K. Wiggers, "Hugging face and servicenow release a free code-generating model," *TechCrunch*. [Online]. Available: https://techcrunch.com/2023/05/04/hugging-face-and-servicenow-release-a-free-code-generating-model/

[47] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[48] IBM, "Ibm unveils watsonx generative ai capabilities to accelerate mainframe application modernization," *IBM Newsroom*, August 2023.

[49] T. Dohmke, "Universe 2023: Copilot transforms github into the ai-powered developer platform," *GitHub Blog*, November 2023.

[50] D. Kocetkov, R. Li, L. Jia, C. Mou, Y. Jernite, M. Mitchell, C. M. Ferrandis, S. Hughes, T. Wolf, D. Bahdanau *et al.*, "The stack: 3 tb of permissively licensed source code," *Transactions on Machine Learning Research*.

[51] O. Sainz, J. A. Campos, I. García-Ferrero, J. Etxaniz, O. L. de Lacalle, and E. Agirre, "Nlp evaluation in trouble: On the need to measure llm data contamination for each benchmark," in *The 2023 Conference on Empirical Methods in Natural Language Processing*.

[52] Q. Hu, Y. Guo, X. Xie, M. Cordy, L. Ma, M. Papadakis, and Y. Le Traon, "Active code learning: Benchmarking sample-efficient training of code models," *IEEE Transactions on Software Engineering*, 2024.

[53] M. Liu, T. Yang, Y. Lou, X. Du, Y. Wang, and X. Peng, "Codegen4libs: A two-stage approach for library-oriented code generation," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 434–445.

[54] Z. Ma, S. An, B. Xie, and Z. Lin, "Compositional api recommendation for library-oriented code generation," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 87–98.

[55] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[56] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, "Codebertscore: Evaluating code generation with pretrained models of code," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 13 921–13 937.

[57] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with bert," in *International Conference on Learning Representations*.

[58] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[59] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[60] Z. Zhang, Y. Wang, C. Wang, J. Chen, and Z. Zheng, "Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation," *arXiv preprint arXiv:2409.20550*, 2024.

[61] Y. Tian, W. Yan, Q. Yang, Q. Chen, W. Wang, Z. Luo, and L. Ma, "Codehalu: Code hallucinations in llms driven by execution-based verification," *arXiv preprint arXiv:2405.00253*, 2024.

[62] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," in *The Twelfth International Conference on Learning Representations*.

[63] M. Sclar, Y. Choi, Y. Tsvetkov, and A. Suhr, "Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting," in *The Twelfth International Conference on Learning Representations*.

[64] N. Miao, Y. W. Teh, and T. Rainforth, "Selfcheck: Using llms to zero-shot check their own step-by-step reasoning," in *The Twelfth International Conference on Learning Representations*.

[65] M. Mizrahi, G. Kaplan, D. Malkin, R. Dror, D. Shahaf, and G. Stanovsky, "State of what art? a call for multi-prompt llm evaluation," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 933–949, 2024.

[66] S. Kwon, S. Lee, T. Kim, D. Ryu, and J. Baik, "Exploring llm-based automated repairing of ansible script in edge-cloud infrastructures," *Journal of web engineering*, vol. 22, no. 6, pp. 889–912, 2023.

[67] Y. Weng, M. Zhu, F. Xia, B. Li, S. He, S. Liu, B. Sun, K. Liu, and J. Zhao, "Large language models are better reasoners with self-

verification," in *The 2023 Conference on Empirical Methods in Natural Language Processing*.

[68] OpenRouter. (2024) Openrouter: Unified access to large language models. Accessed: 2025-10-13. [Online]. Available: https://openrouter.ai/

[69] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[70] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[71] X.-Y. Zhang, G.-S. Xie, X. Li, T. Mei, and C.-L. Liu, "A survey on learning to reject," *Proceedings of the IEEE*, vol. 111, no. 2, pp. 185–215, 2023.

[72] H. Xu, Z. Zhu, S. Zhang, D. Ma, S. Fan, L. Chen, and K. Yu, "Rejection improves reliability: Training LLMs to refuse unknown questions using RL from knowledge feedback," in *First Conference on Language Modeling*, 2024. [Online]. Available: https://openreview.net/forum?id=lJMioZBoR8

[73] H. Zhang, S. Diao, Y. Lin, Y. Fung, Q. Lian, X. Wang, Y. Chen, H. Ji, and T. Zhang, "R-tuning: Instructing large language models to say 'i don't know'," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2024, pp. 7106–7132.

[74] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[75] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 341–353.

[76] T. Y. Zhuo, "Ice-score: Instructing large language models to evaluate code," in *Findings of the Association for Computational Linguistics: EACL 2024*, 2024, pp. 2232–2242.

[77] J. He, J. Shi, T. Y. Zhuo, C. Treude, J. Sun, Z. Xing, X. Du, and D. Lo, "From code to courtroom: Llms as the new software judges," *arXiv preprint arXiv:2503.02246*, 2025.

[78] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[79] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*.

[80] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5673–5684.

[81] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2021.

[82] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[83] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," in *The Twelfth International Conference on Learning Representations*.

[84] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, "Codet5+: Open code large language models for code understanding and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 1069–1088.

[85] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 510–520.

[86] G. Wang, Z. Wang, J. Chen, X. Chen, and M. Yan, "An empirical study on numerical bugs in deep learning programs," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[87] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 129–140.

[88] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–26, 2024.

[89] F. Tambon, A. M. Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol, "Bugs in large language models generated code," *arXiv preprint arXiv:2403.08937*, 2024.

[90] S. Dou, H. Jia, S. Wu, H. Zheng, W. Zhou, M. Wu, M. Chai, J. Fan, C. Huang, Y. Tao *et al.*, "What's wrong with your code generated by large language models? an extensive study," *arXiv preprint arXiv:2407.06153*, 2024.

[91] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[92] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[93] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *International Conference on Machine Learning*. PMLR, 2021, pp. 780–791.

[94] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.

[95] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1035–1047.

[96] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "Coditt5: Pretraining for source code and natural language editing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[97] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin, "Circle: continual repair across programming languages," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 678–690.

Terry Yue Zhuo a PhD student at Monash University and CSIRO's Data61, Australia. He previously worked at CSIRO's Data61, Singapore Management University, Sea AI Lab, TikTok AI Innovation Center, and Amazon AWS AI. He has authored more than 40 papers in refereed journals or conferences, such as The Web Conference (WWW), ACM Transactions on Software Engineering and Methodology (TOSEM), International Conference on Learning Representations (ICLR), Network and Distributed System Security (NDSS), Annual Meeting of the Association for Computational Linguistics (ACL) and Conference on Empirical Methods in Natural Language Processing (EMNLP). He received the best paper award in the Deep Learning for Code (DL4C) workshop at ICRL 2023, 2024-2025 IBM PhD Fellowship Awards, and 2025 DAAD AInet Fellowship. He serves as a Senior Area Chair of EMNLP 2025 and ACL 2026. His research interests include empirical software engineering, code intelligence, and security intelligence. More information about him can be found at https://terryyz.github.io/.

Junda He is a PhD candidate at the Singapore Management University, supervised by Professor David Lo. He earned his Master of Science degree in Software System Engineering and his Bachelor of Science degree in Computer Science, both from University College London. Junda's current research focuses on testing and enhancing the trustworthiness of AI agents. More information about him can be found at https://jundahe.github.io/.

Jiamou Sun is a Research Scientist in the SE4AI team at CSIRO's Data61, specialising in responsible AI, software supply chain security, and AI-driven automation. His research bridges artificial intelligence, cybersecurity, and software engineering, with a particular focus on knowledge graph–based vulnerability detection and risk assessment. He has led and contributed to numerous high-impact projects, including the CSIRO–Google Software Supply Chain Security initiative, the AISI Joint AI Safety Testing, VAISS co-editing, and Responsible AI for AI4M. His work has been recognised through publications in top-tier venues such as ICSE, ASE, WWW and UIST, as well as the distinguished paper awards in the ICSME'2018 and 2019. More information about him can be found at https://scholar.google.com/citations?user=I2UCgDYAAAAJ.

John Grundy is Australian Laureate Fellow, Senior Deputy Dean and Professor of Software Engineering at Monash University. He leads the HumaniSE lab in the Faculty of Information Technology, investigating "human-centric" issues in software engineering. These include impact of personality on software engineers and users; emotion-oriented requirements engineering; impact of different languages, cultures and belief sets on using and engineering software; usability and accessibility of software, particularly for ageing people and people with physical and mental challenges; issues of gender, age, socio-economic status and personal values on software, software requirements, and software engineering teams. He is IEEE Fellow, Fellow of Engineers Australia and Fellow of Automated Software Engineering. More information about him can be found at https://sites.google.com/site/johncgrundy.

Zhenchang Xing is the Senior Principal Research Scientist at CSIRO's Data61. He is the Science Lead of Software System group. His research interests are Software Engineering, Responsible AI, and Human-Computer Interaction, with specific focus on the software engineering infrastructure for AI systems. His research received several ACM SIGSOFT Distinguished Paper Awards and IEEE TCSE Disginguished Paper Awards, as well as ACM SIGSOFT Most Influential Paper Award at ASE 2021. More information about him can be found at https://people.csiro.au/X/Z/Zhenchang-Xing/.

David Lo is the OUB Chair Professor of Computer Science and Founding Director of the Center for Research in Intelligent Software Engineering (RISE) at Singapore Management University. Championing the area of AI for Software Engineering (AI4SE) since the mid-2000s, he has demonstrated how AI — encompassing data mining, machine learning, information retrieval, natural language processing, and search-based algorithms — can transform software engineering data into actionable insights and automation. Through empirical studies, he has also identified practitioners' pain points, characterized the limitations of AI4SE solutions, and explored practitioners' acceptance thresholds for AI-powered tools. His contributions have led to over 20 awards, including four Test-of-Time awards and thirteen ACM SIGSOFT/IEEE TCSE Distinguished Paper awards, and his work has garnered over 40,000 citations. An ACM Fellow, IEEE Fellow, ASE Fellow, and National Research Foundation Investigator (Senior Fellow), Lo has also served as a PC Co-Chair for ASE'20, FSE'24, and ICSE'25. More information about him can be found at http://www.mysmu.edu/faculty/davidlo/.

Xiaoning Du is an ARC DECRA Fellow and Senior Lecturer (equivalent to U.S. Associate Professor) at the Faculty of Information Technology, Monash University. She received her Ph.D. from Nanyang Technological University in 2020 and her Bachelor's degree from Fudan University in 2014. Her research primarily focuses on the security and quality assurance of intelligent software systems, with a particular emphasis on intelligent software engineering tools. She has published over 40 papers in top-tier conferences and journals. Her work has won or been nominated for the ACM SIGSOFT Distinguished Paper Award multiple times. She also received the 2024 FIT Dean's Early Career Researcher of the Year Award and the 2024 Google Research Scholar Award. Check out more about her at http:https://xiaoningdu.github.io/.