

# Detecting and Fixing API Misuses of Data Science Libraries Using Large Language Models

Akalanka Galappaththi  
University of Alberta  
Edmonton, Canada  
akalanka@ualberta.ca

Francisco Ribeiro  
New York University Abu Dhabi  
Abu Dhabi, United Arab Emirates  
francisco.ribeiro@nyu.edu

Sarah Nadi  
New York University Abu Dhabi  
Abu Dhabi, United Arab Emirates  
sarah.nadi@nyu.edu

**Abstract**—Data science libraries, such as scikit-learn and pandas, specialize in processing and manipulating data. The data-centric nature of these libraries makes the detection of API misuse in them more challenging. This paper introduces DSCHECKER, an LLM-based approach designed for detecting and fixing API misuses of data science libraries. We identify two key pieces of information, API directives and data information, that may be beneficial for API misuse detection and fixing. Using three LLMs and misuses from five data science libraries, we experiment with various prompts. We find that incorporating API directives and data-specific details enhances DSCHECKER’s ability to detect and fix API misuses, with the best-performing model achieving a detection  $F_1$ -score of 61.18% and fixing 51.28% of the misuses. Building on these results, we implement DSCHECKER<sub>agent</sub> which includes an adaptive function calling mechanism to access information on demand, simulating a real-world setting where information about the misuse is unknown in advance. We find that DSCHECKER<sub>agent</sub> achieves 48.65% detection  $F_1$ -score and fixes 39.47% of the misuses, demonstrating the promise of LLM-based API misuse detection and fixing in real-world scenarios.

**Index Terms**—API misuse, data science libraries, detection, repair, large language models

## I. INTRODUCTION

Most software systems rely on third-party libraries through Application Programming Interfaces (APIs). APIs streamline development, but failing to adhere to API guidelines or assumptions also known as *API misuse* can introduce bugs [1]–[3]. These misuses can lead to run-time errors, performance issues, or incorrect output [1], [2], [4], [5].

Most API misuse research has focused on statically typed languages like Java or C [6]–[9]. However, the flexibility of dynamic typing in languages like Python introduces unique challenges for correct API usage [5]. Additionally, the increasing use of Python for data analysis and machine learning introduces new misuse types. For example, Wei et al. [4] identified data conversion errors in Deep Learning (DL) libraries which is a new misuse characteristic.

Building on these findings, Galappaththi et al. [10] argued that these characteristics extend beyond DL libraries and apply to any library with extensive data processing needs, which they refer to as *data-centric libraries*. In essence, libraries with these characteristics are commonly used in data science applications where extensive data processing, data manipulation, machine learning, or visualization is done. Accordingly,

```
1 from sklearn.impute import SimpleImputer
2 import pandas as pd; import numpy as np
3
4 df = pd.read_csv("data.csv")
5
6 - impt = SimpleImputer(missing_values=np.nan, strategy
7   = "mean")
8 + impt = SimpleImputer(missing_values=np.nan, strategy
9   = "constant", fill_value=0.0001)
10 imp_array = impt.fit_transform(df)
11 print(imp_array[:, 1])
```

df content	A	B
1	1	NaN
2	2	NaN
...	...	...
NaN	NaN	NaN

Fig. 1: SimpleImputer misuse [10]: strategy="mean" drops column B as it contains only NaN values, causing an indexing error when accessed (from Stack Overflow 60527883).

Galappaththi et al. [10] study API misuses in five additional non-DL data science libraries (Numpy, pandas, Matplotlib, scikit-learn, and seaborn) and concluded that they suffer from most of the same misuse types found in DL libraries [4].

Figure 1 shows a misuse from scikit-learn in a diff format, highlighting the original code containing the API misuse and the corrected version [10]. The behavior of SimpleImputer [11] depends on both the input data and the selected imputation strategy, as constrained by the following *API directive* [12]: “Columns which only contained missing values at fit are discarded upon transform if strategy is not ‘constant’.” [11]. Accordingly, given the original API call on Line 6, the missing values in the first column of `imp_array` get replaced by its mean; however, the second column which contains only missing values (i.e., NaN) will be discarded. Subsequently, the print statement on Line 9 raises an `IndexError` because `imp_array` has only one column, making access to index 1 invalid. One potential fix is to change the `strategy` parameter to “constant” and introduce a `fill_value` parameter with a constant argument, as shown on Line 7. Galappaththi et al. [10] defined this type of misuses as *data-dependent misuses*, because the original code works if column B contained at least one numerical value.

Previous research showed that API misuses are a prevalent problem and the majority of them cause program crashes [6], [13]. Additionally, detecting API misuses in libraries that involve a lot of data processing is challenging, because we have to account for the internal composition of complex data structures (e.g., DataFrames or arrays) that these APIs process. With the known limitations of static analysis in capturing such dynamic information [4] and the known shortcomings

of traditional pattern-based API misuse detectors [8], [9], [13]–[15], we need new ways for detecting misuses of data science APIs. With recent developments in large language models (LLMs), Wei et al. [4] experiment with LLMs to detect and fix DL API misuses. Motivated by their work, in this paper, we investigate whether LLMs can detect and fix API misuses in data science libraries. Our key expectation is that determining the correct API usage in this context would likely require knowledge of the API directives as well as dynamic information about the data being processed. Accordingly, we design an LLM-based API misuse detection and fixing approach, DSCHECKER, that employs a simple prompting strategy, leveraging both static information retrieved from the API documentation and dynamic information obtained by running the code. To evaluate our approach, we conduct an empirical study on five data science libraries, answering the following questions:

- RQ1: Which information is most effective in guiding LLMs to detect and fix API misuses of data science libraries?** We conduct an ablation study using zero-shot prompts to assess the role of API directives and data context.
- RQ2: Does few-shot prompting improve LLMs’ API misuse detection and fixing?** After finding the best zero-shot prompt, we test if few-shot prompts lead to improvement.
- RQ3: Can DSCHECKER detect/fix misuses of other data science libraries?** We go beyond the five initial libraries by using 15 DL misuses from Wei et al. [4]’s dataset.
- RQ4: How does DSCHECKER compare to other LLM-based API misuse detection/fixing tools?** We compare DSCHECKER against LLMAPIDET [4], an existing LLM-based method for detecting and fixing DL API misuses.
- RQ5: Can DSCHECKER<sub>agent</sub> detect and fix API misuses?** We explore if LLMs equipped with function calling (DSCHECKER<sub>agent</sub>) can autonomously retrieve API documentation and data information in real-world scenarios, where this is not readily available.

Our results show that incorporating API directives and variable information significantly improves LLMs’ performance with an average 7% increase in  $F_1$ -score for detecting and a 5% increase in fix rate for fixing data science API misuses. DSCHECKER also outperformed a prior misuse detection tool for DL libraries [4] (46.15% vs. 16.49%). Additionally, while DSCHECKER<sub>agent</sub>’s performance drops in comparison to DSCHECKER, it shows potential of real-world application where precise information may not be readily available.

In summary, our contributions include:

- 1) A new LLM-based approach, DSCHECKER, that combines API directives and dynamic data information for enhanced API misuse detection and fixing.
- 2) Evaluation of our approach on an existing data science API misuse dataset [10], including an ablation study to understand the effect of the different prompt information and experimenting with prompting techniques.
- 3) A comparison of DSCHECKER to a different LLM-based misuse detection/fixing approach previously developed for

DL libraries as well as the evaluation of DSCHECKER on additional DL misuses.

- 4) Implementation and evaluation of an agentic version of DSCHECKER, DSCHECKER<sub>agent</sub>, for the practical application of LLM-based detection and fixing where no prior information about the misuse is known.

Our replication package contains our results as well as the source code and data to run all our experiments: <https://doi.org/10.6084/m9.figshare.28327148>.

## II. BACKGROUND AND RELATED WORK

*a) API Misuse:* An *API misuse* occurs when there is any deviation from the APIs intended usage, such as failing to invoke a required method [1], [2], [4], [5], using unsupported parameter values [5], or passing incorrect data types [5]. API misuses can cause run-time errors, performance issues, or incorrect outputs [1], [2], [4], [5]. In this paper, we focus on data science libraries that process, analyze, and derive insights from data. These libraries typically accept diverse data structures such as pandas DataFrames or NumPy arrays and have complex processing workflows [10].

*b) API Misuse Detection:* Most of the early work on detecting API misuses is based on mining frequent usage patterns from source code to identify correct usages with deviations from those mined patterns considered as misuses [6], [8], [9], [13]–[15]. However, these approaches typically suffer from low precision and recall [1]. To address this, researchers proposed automatically learning correct usages from API documentation [3], [16]. However, this typically relies on static parameters and return types to define usage constraints, making it effective for statically typed languages like Java but not suitable for Python. For example, Matplotlib’s `plot` accepts various parameter types for `x` and `y`. Accordingly, this creates runtime-dependent constraints for subsequent calls to additional APIs such as `axvspan`. To the best of our knowledge, static fine-grained API usage graphs [3], [16] cannot capture such downstream dynamic API typing dependencies, making them inapplicable to our work. Another misuse detection direction uses domain specific languages that encode usage constraints, which are either automatically extracted from the documentation or manually specified by an expert [7], [17]–[19].

Given the above limitations, researchers started leveraging LLMs for API misuse detection. For example, Baek et al. [20] demonstrated the effectiveness of LLMs for detecting Java cryptographic API misuses, showing that their fine-tuned model outperformed state-of-the-art rule-based detectors in this domain [21]–[23]. In contrast, our work uses additional contextual information for off-the-shelf LLMs, focusing on a different domain (data science) and language (Python).

Wei et al. [4] study API misuses in PyTorch and TensorFlow. They construct a dataset of 891 misuses from version-control history (DLMISUSES), and categorize them by type (e.g., missing method), cause (e.g., device mismanagement), and symptom (e.g., runtime error). They propose LLMAPIDET, an LLM-based detection and fixing approach with three steps:

(1) Few-shot prompting is used to extract fixing rules from 600 before-and-after misuse examples; the remaining 291 are reserved for evaluation; (2) To detect misuses, the LLM generates a natural language summary of a code snippet, which is compared to existing rules to retrieve the top four matches; a match constitutes a misuse detection; (3) If a misuse is found, the LLM is prompted with the code and retrieved rules to produce an explanation and a fix. While LLMAPIDET outperforms a rule-based detector [24], it still achieves low recall (16.49%) and has a 23% fix success rate. Our approach uses API documentation and dynamic variable information instead of rule mining, simplifying the pipeline. As DL libraries fall under data-centric, we evaluate generalizability in RQ3 (applying DSCHECKER to DLMISUSES) and relative performance in RQ4 (applying both tools to a shared subset).

#### c) Code generation and program repair with LLMs:

LLMs have demonstrated effectiveness in understanding code and generating natural language explanations for code snippets [25]. Leveraging these capabilities, LLMs have been applied to generate code snippets for detecting bugs in DL applications [26]. Guan et al. [27] used contextual information to improve LLM’s performance in generating test cases to detect model optimization bugs in DL applications. Similarly, we also provide LLMs with contextual information (e.g., API directives) but for a different problem and beyond only DL applications.

In the context of automatic program repair, Xia et al. [28] evaluated different prompt settings for generating fixes. They find that generating fixes for buggy lines is more effective than reconstructing the entire function. We also generate fixes to replace buggy lines using a unified diff format. However, unlike Xia et al. [28], we do not provide prefix or suffix indicators to highlight bug locations, as part of our goal is to assess the LLMs’ ability to identify problematic usage.

d) *LLM agents*: LLM agents are autonomous applications interacting with external tools without direct human intervention [29]. *Function calling* is a mechanism that empowers these agents to communicate with code or external services, enabling them to retrieve information or perform specific tasks [30]. For example, *WebGPT* [31] enabled web browsing for generating comprehensive answers, while *ReAct* [32] explored reasoning and action-taking, such as planning and interacting with external resources. While we do not focus on complex reasoning, we leverage a form of agency by allowing LLMs to call functions, enabling direct interaction with code.

### III. DSCHECKER: AN LLM-BASED MISUSE DETECTION AND FIXING APPROACH

DSCHECKER is based on the simple idea of prompting an LLM with the right information. It takes a piece of code that uses third-party data science APIs that we want to check. The output indicates whether there is a misuse in the code and if so, how to fix it. Figure 2 shows an example of the prompt DSCHECKER uses to detect and fix the misuse in Figure 1. LLMs have demonstrated natural language understanding capabilities [33] as well as code understanding capabilities [34]. Thus, our first key insight is that if an API has a directive

```

1 Libraries have certain usage constraints that ensure their correct usage.
2 For example, in Java, developers need to check if 'it.hasNext() == True' before
3 calling 'it.next()' on iterator 'it'.
4
5 The following code uses the library scikit-learn.
6 Check this piece of code and decide if it correctly uses the library scikit-learn.
7
8 Here is information about the variable df at line 5 used in this code snippet.
9
10 pandas.core.frame.DataFrame
11
12      A      B
13 0  1.0  NaN
14 1  2.0  NaN
15
16 #      Column  Non-Null Count  Dtype
17
18 0      A      3 non-null      float64
19 1      B      0 non-null      float64
20 dtypes: float64(2)
21
22
23
24 scikit-learn documentation states this guideline "Columns which only contained
25 missing values at fit are discarded upon transform if strategy is not "constant"
26 for SimpleImputer API, which could be useful to decide if the code correctly
27 uses the library."
28
29 Respond in the following JSON format. If the code is correct, leave the remaining
30 fields empty after setting field "correct" to yes:
31 {
32   "correct": "yes/no",
33   "patch": "Provide unified diff format output that can be used to patch the buggy
34             code(e.g., 19c19,20<n<foo())\n--\n>n>foo(bass='bazz')\n>bar())"
35   "explanation": "explanation of why the code is incorrect"
36 }
37
38 In the following code, the numbers on the left-hand side represent line numbers (as
39 shown in an IDE or text editor) and are not part of the actual code.
40
41 1  from sklearn.impute import SimpleImputer
42
43
44 6  impt = SimpleImputer(missing_values=np.nan, strategy="mean")
45
46 7  imp_array = impt.fit_transform(df)
47
48 8  print(imp_array[:, 1])

```

Fig. 2: Example of prompt provided to an LLM to detect and fix the misuse in Figure 1

describing its expected usage, an LLM can potentially check if the directive is being followed in a given piece of code. In Figure 2, lines 24-27 show this API directive in the prompt. The second insight is that since these APIs process a lot of data and the nature of this input data can influence whether the API behaves correctly [10], providing the LLM with information about the passed data, may help it determine the correct API usage. Lines 8-22 shows the data information for this example.

The prompt asks the model to use a JSON object as the response format. If the LLM detects a misuse, the output will follow the format: {"correct": "no", "patch": "...", "explanation": "...". Also, we instruct the LLM to generate a patch in unified diff format, and a free text explanation. If no misuse is found, the model simply responds with {"correct": "yes"}.

## IV. DATASET PREPARATION AND EVALUATION METRICS

### A. Dataset Preparation: DSMISUSES

We use the API misuse dataset by Galappaththi et al. [10]. We now describe the original dataset and our extensions of it.

*Original Data*: Galappaththi et al. [10]’s dataset contains 49 API misuses collected from Stack Overflow posts and GitHub commits related to five data science libraries: NumPy, pandas, Matplotlib, scikit-learn, and seaborn. Each misuse comes with a minimal reproducible example to illustrate the misuse, and indication of whether it is data dependent. If the misused API has an explicit directive, typically found in API references or occasionally in user guides and examples from the online library documentation, it is also included.

*Vet and Reproduce:* We first carefully review the provided data to make sure we can verify the misuse and reproduce the error or incorrect behavior through the provided code example. Based on this review, we remove 8 misuses that we could not reproduce, leaving us with 41 misuses. We also remove 3 duplicate misuses (e.g., two instances of seaborn’s `set_palette()` API misuse extracted from two different Stack Overflow posts). This leaves us with 38 misuses, out of which 18 are misuses of an API with an explicit usage directive while 20 are data dependent.

*Replace Hard-coded Data:* We replace hard-coded data, such as DataFrames created from dictionaries, with data loaded from a file to assess the effect of providing data information.

*Add data information:* DSCHECKER prompts require information about the data processed by the API, which the original dataset does not provide. To extract this information, we manually inspect the code to identify relevant variables, and then instrument it with `print(type(var))` statements to identify variable types. We find three main data structures that appear in the code snippets: DataFrames, NumPy arrays, and lists. For DataFrames, we additionally use `print(var.info())` and `print(var.head(3))` to record column names, types, and sample rows respectively. For NumPy arrays, we use `print(var.shape, var.dtype)` to capture the shape and data type of the array. For lists, we use `print(len(var))` to record list length. Running the instrumented code yields outputs such as type, and associated content. For instance, for the variable `df` at line 4 in Figure 1, the output includes its type `DataFrame`, sample values, and column metadata.

*Create correct usages:* To evaluate effectiveness in identifying both correct and incorrect API usages, we augment the dataset with a fixed code snippet for each of the 38 misuses. We refer to the final updated dataset as DSMISUSES which contains 76 code snippets.

## B. Evaluation Metrics

In all research questions, our evaluation focuses on two primary aspects: misuse detection and fixing. Since the metrics are common across all RQs, we discuss them here first.

*1) Misuse detection metrics:* When evaluating DSCHECKER’s ability to detect API misuses, we consider that it correctly detects a misuse if it flags the code snippet as not correct (i.e., outputs `"correct": "no"`) and provides an accurate explanation about the misuse. We first automatically compare the `correct` field of the model’s response to the ground truth in DSMISUSES. We then manually review the explanations of misuses correctly marked as `no`. If the explanation accurately describes the misuse, we confirm this as a correct detection; otherwise, we mark it as not detected. We follow this two-step process as we found that relying solely on the yes/no classification is insufficient; LLMs occasionally flag a code snippet as having a misuse but then describe non-existent issues or do not provide adequate explanations for the detected problems.

At the end of this process, we calculate precision ( $P$ ), recall ( $R$ ), and  $F_1$ -score for misuse detection. We calculate *precision*

as the number of correctly detected misuses divided by the total number of code snippets flagged as misuse by the tool and *recall* as the total number of correctly detected misuses divided by the total number of misuses in the dataset. We calculate  $F_1$ -score as the harmonic mean of the precision and recall:  $F_1 = 2 * P * R / (P + R)$ .

*2) Misuse fixing metrics:* To evaluate fixing, we automatically apply the LLM-provided patch to the corresponding code snippet and execute the patched code. We then manually evaluate whether the patched code resolves the original error or produces the correct output. For misuses that result in errors, we consider a fix to be correct *only* if it resolves the original error without introducing additional errors. For misuses that result in incorrect output, we ensure that the problem has been fixed based on the misuse description in the dataset. We calculate *Fix Rate* as the proportion of correct patches relative to the total number of misuses (38 for DSMISUSES).

*3) Bootstrap sampling:* To evaluate whether the observed performance differences between LLMs and prompt variations are statistically significant, we apply bootstrap sampling [35]. For each setting, we repeatedly sample 20 code snippets 50 times and compute  $F_1$ -scores and fix rates for each sample, yielding performance distributions. After confirming non-normality with the ShapiroWilk test [36], we apply Dunns test with Bonferroni adjustments [37] at a 95% confidence level ( $\alpha = 0.05$ ) to identify statistically significant differences between distribution pairs.

## V. RQ1: WHICH INFORMATION IS MOST EFFECTIVE IN GUIDING LLMs TO DETECT AND FIX API MISUSE?

To answer RQ1, we conduct an ablation study to compare the effect of the different prompt components from Figure 2.

### A. RQ1 Setup

Figure 2 shows the full prompt with an API directive and data information. We generated this prompt using a prompt template that contains fixed text and placeholders that we substitute values for according to the target code snippet. Figure 2 shows these placeholders in gray rectangles (e.g., `$lib` is a placeholder for library name), while showing the substituted text for the example in bold. For easier visualization, the dotted lines separate the sections that contain data information (Lines 8-22) versus the API directive (Lines 24-27). We now discuss the four variations of the prompt used in RQ1. Note that these are all zero-shot prompts where no examples are provided.

*1) Prompt with API directives and data information ( $P_{full}$ ):* This is the full prompt template with all available information in the sections labeled “data information” and “API directive”. To generate a concrete prompt for a given code snippet, we replace the placeholders `$lib` and `$code` with the corresponding library name and code snippet, respectively. The “data information” section has three placeholders: `$variable` and `$linenum`, which are replaced with the corresponding variable name and line number respectively, and `$data`, which is replaced with the type of the variable, sample values, and any additional information about the variable, as recorded in the dataset (See Section IV-A).

TABLE I: DSCHECKER’s misuse detection (RQ1-2)

Metric	Model	Zero-shot (RQ1)				Few-shot (RQ2)
		$P_{base}$	$P_{data}$	$P_{dir}$	$P_{full}$	$P_{fewshot}$
All data (76)						
P	4o-mini	<b>45.65%</b>	37.04%	38.46%	33.33%	41.67%
	4o	48.78%	51.16%	52.38%	<b>55.00%</b>	<b>57.14%</b>
	llama	48.89%	50.00%	52.17%	<b>56.52%</b>	<b>56.82%</b>
R	4o-mini	<b>53.85%</b>	51.28%	51.28%	46.15%	<b>64.10%</b>
	4o	51.28%	<b>56.41%</b>	<b>56.41%</b>	<b>56.41%</b>	<b>61.54%</b>
	llama	56.41%	48.72%	61.54%	<b>66.67%</b>	64.10%
F1	4o-mini	<b>49.41%</b>	43.01%	43.96%	38.71%	<b>50.51%</b>
	4o	50.00%	53.66%	54.32%	<b>55.70%</b>	<b>59.26%</b>
	llama	52.38%	49.35%	56.47%	<b>61.18%</b>	60.24%
Potentially misused API has a directive (36)						
P	4o-mini	<b>56.52%</b>	48.28%	41.94%	40.00%	45.45%
	4o	54.17%	54.55%	59.09%	<b>65.00%</b>	<b>66.67%</b>
	llama	50.00%	50.00%	56.52%	<b>60.00%</b>	59.09%
R	4o-mini	68.42%	<b>73.68%</b>	68.42%	63.16%	<b>78.95%</b>
	4o	<b>68.42%</b>	63.16%	<b>68.42%</b>	<b>68.42%</b>	<b>73.68%</b>
	llama	57.89%	42.11%	68.42%	<b>78.95%</b>	68.42%
F1	4o-mini	<b>61.90%</b>	58.33%	52.00%	48.98%	57.69%
	4o	60.47%	58.54%	63.41%	<b>66.67%</b>	<b>70.00%</b>
	llama	53.66%	45.71%	61.90%	<b>68.18%</b>	63.41%

In Figure 2, “API directive” section has three placeholders: `$lib`, `$directive`, and `$api`. We replace them with the library name, the API directive, and the corresponding API name, respectively. The presentation of this section varies according to the directive’s context. For example, if a directive is specific to a parameter, we mention the parameter name.

Note that a code snippet may contain multiple variables and APIs, but in RQ1 (and later RQ2), we focus only on those relevant to the potentially misused API to isolate the effect of providing the LLM with different information.

2) *Prompt with only API directives ( $P_{dir}$ )*: This variant disregards data information and provides only API directives.

3) *Prompt with only data information ( $P_{data}$ )*: This variant ignores API directives and provides only data information.

4) *Base prompt ( $P_{base}$ )*: The base prompt is the control prompt of RQ1 which establishes a baseline for evaluating the effect of additional information on detecting and fixing misuses. It strips out all additional information on Lines 8-27 and provides only the code snippet and the library being used.

*Selected LLMs*: We select three LLMs for our experiments based on their performance in recent studies [27], [38]: two proprietary models—OpenAI’s gpt-4o-mini-2024-07-18 and gpt-4o-2024-05-13—and one comparably sized open-source model, Meta’s llama-3.1-405b-Instruct. As a short-hand, we refer to these models as 4o-mini, 4o, and llama respectively.

## B. RQ1 Results

1) *Misuse detection*: Table I shows RQ1 detection results.

a) *Full dataset*: We first focus on the top left part of Table I, which presents the zero-shot results over all 76 code snippets. Looking at the  $F_1$ -score, we find that for both 4o and llama,  $P_{full}$  is most effective at detecting misuses. However, 4o-mini performed best with  $P_{base}$  and had its worst performance when provided  $P_{full}$ .

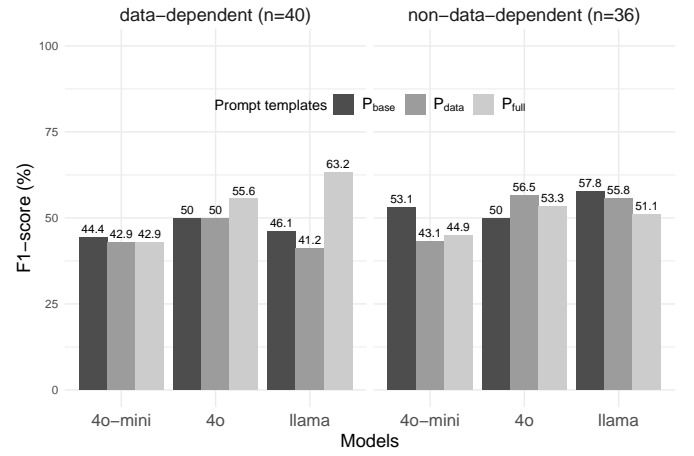


Fig. 3: Effect of adding data to data-dependent misuses vs. non-data-dependent misuses.

We find that while 4o-mini correctly stated the presence of misuses for 35 misuses, it failed to provide adequate explanations for most of them when provided any additional information, resulting in a lower number of true positives and thus lower recall and precision. For example, for the misuse in Figure 1, 4o-mini with  $P_{full}$  claimed that passing the entire DataFrame is not allowed, despite SimpleImputer accepting pandas DataFrames. On the other hand, 4o and llama flagged this misuse and provided the right explanation. These findings confirm that some models are more effective than others in interpreting provided information [39], [40].

We find that the performances of 4o and llama is significantly higher than 4o-mini ( $p < 0.05$ ). However, the performance between 4o and llama is not significant ( $p = 0.468$ ).

b) *Code snippets with API directives*: If a code snippet does not have an API directive to include in the prompts,  $P_{dir}$  and  $P_{full}$  will be equivalent to  $P_{base}$  and  $P_{data}$ , respectively. This may blur our understanding of the effect of API directives. Accordingly, we next focus *only* on the subset of 36 code snippets in DSMISUSES where the potentially misused API has an explicit directive such that the generated prompts are different. This is shown in the bottom left part of Table I. When compared to  $P_{base}$ ,  $P_{dir}$  increased (or kept) the recall of all models and the precision and  $F_1$ -score of two of the three models. For both 4o and llama, adding directives significantly improves their ability to detect API misuses ( $p < 0.05$ ). These results further highlights the positive effect of providing the API directive to the LLM.

c) *Data-dependent misuses*: Recall that not all misuses are data dependent. It could be the case that providing data information helps only when the potential misuse is data dependent but confuses the model otherwise. Thus, we further compare the performance on subsets of data-dependent and non-data-dependent code snippets.

Figure 3 presents the  $F_1$ -scores for the three models across data-dependent (40) and non-data-dependent (36) code snippets using the prompt templates  $P_{base}$ ,  $P_{data}$ , and  $P_{full}$ . Given the negative impact of adding information on 4o-mini,

TABLE II: DSCHECKER’s fix rate (RQ1-2)

Model	Zero-shot (RQ1)				Few-shot (RQ2)
	$P_{base}$	$P_{data}$	$P_{dir}$	$P_{full}$	$P_{fewshot}$
All misuses (38)					
4o-mini	<b>43.59%</b>	38.46%	38.46%	38.46%	41.03%
4o	41.03%	35.90%	<b>46.15%</b>	43.59%	<b>53.85%</b>
llama	46.15%	35.90%	46.15%	<b>51.28%</b>	<b>51.28%</b>
Misuses where misused API has a directive (18)					
4o-mini	<b>52.63%</b>	<b>52.63%</b>	<b>52.63%</b>	47.37%	<b>52.63%</b>
4o	47.37%	42.11%	57.89%	<b>63.16%</b>	<b>63.16%</b>
llama	47.37%	31.58%	52.63%	<b>57.89%</b>	52.63%

we focus our analysis on 4o and llama, for which Table I indicates  $P_{full}$  is generally most effective. In the data-dependent facet, adding only data information ( $P_{data}$ ) did not improve, or even decreased,  $F_1$ -scores for these two models. However,  $P_{full}$  always increased the  $F_1$ -score on these data-dependent code snippets, with a particularly large increase for llama. This increase is statistically significant for both 4o and llama ( $p < 0.05$ ). On the other hand, for non-data-dependent code snippets, we find that  $P_{full}$  resulted in lower  $F_1$ -scores for llama compared to  $P_{base}$ , and for 4o compared to  $P_{data}$ . This does suggest that the data and directive combination sometimes hurts when the API usage is not data dependent. However, since the nature of the API usage is unknown beforehand and the benefit on data-dependent misuses is much higher than the slight performance loss on the other code snippets, we conclude that  $P_{full}$  is still worthwhile.

2) *Misuse fixing*: The left side of Table II shows DSCHECKER’s fix rate for each model and prompt. Similar to detection results, for all misuses (top left part), DSCHECKER still achieves the highest fix rate using the full prompt ( $P_{full}$ ) with llama (51.28%). We found that llama’s fix rate is significantly higher than those of 4o-mini and 4o ( $p < 0.05$ ). We observe similar trends to detection where 4o-mini still achieves its best performance with  $P_{base}$  (43.59%). For fixing, we find that only providing directives ( $P_{dir}$ ) worked best for 4o, achieving 46.15% fix rate across all misuses (top part of Table II). However, for the subset of code snippets where the misused API had a directive, the full prompt performed better, reaching 63.16% (bottom part of Table II).

**RQ1 findings:** Our results show that the majority of the models have higher detection  $F_1$ -score with  $P_{full}$ . Among the three models, llama with  $P_{full}$  has both the best detection  $F_1$ -score (61.18%) and the fix rate (51.28%).

## VI. RQ2: DOES FEW-SHOT PROMPTING IMPROVE LLMs’ API MISUSE DETECTION AND FIXING?

### A. RQ2 Setup

To answer RQ2, we experiment with few-shot prompting where we provide solved examples to the models. Based on the zero-shot prompting results, we select the  $P_{full}$  template to proceed with few-shot prompting. To include in the few-shot prompt, we create two new examples (available on our artifact), one correct and one incorrect, that use APIs from `scikit-learn` and `pandas`, respectively. These two examples are inspired by the misuses in the dataset, but neither direct replicas nor reuse any of the misused APIs

in DSMISUSES. We include these examples along with the corresponding responses at Line 4 in Figure 2. We refer to this prompt as  $P_{fewshot}$ .

### B. RQ2 Results

1) *Misuse detection*: The last column of Table I shows the results of  $P_{fewshot}$ . If  $P_{fewshot}$  achieves a higher score than the highest performing zero-shot prompt in that row, we show it in bold. Overall, we find that misuse detection using few-shot prompting produces varied results. It improved  $F_1$ -score detection for both 4o-mini and 4o (both statistically significant), although the increase for 4o is much higher. On the other hand, llama’s few-shot prompt resulted in a slight drop ( $< 1\%$ ). Overall, llama’s zero-shot  $P_{full}$  still achieved the highest detection  $F_1$ -score across all variations.

2) *Misuse fixing*: Looking at all misuses in Table II, we find that few-shot examples significantly improved 4o’s fix rate and significantly hindered 4o-mini’s performance ( $p < 0.05$ ). In contrast, the fix rate for llama remained effectively unchanged, with no statistically significant difference ( $p = 0.714$ ). Compared to llama, 4o’s fixing ability is also not significantly different ( $p = 0.790$ ).

**RQ2 findings:** Few-shot prompting leads to varied results across models. It significantly improves both detection and fixing performance for 4o, but negatively affects detection for llama and has no impact on its fixing performance.

## VII. RQ3: CAN DSCHECKER DETECT/FIX MISUSES OF OTHER DATA SCIENCE LIBRARIES?

### A. RQ3 Setup

Since our approach (i.e., adding directives and data) based on the conclusions of Galappaththi et al. [10], running DSCHECKER on additional libraries gives us insights into generalizability. Accordingly, we consider DLMISUSES [4] dataset described in Section II. To apply DSCHECKER, we require minimal reproducible examples, API directives when available, and information about variables processed by the API. Since the code in DLMISUSES comes from larger repositories and is not readily runnable, we select a sample of misuses to reduce the manual effort of creating minimal examples.

a) *Sampling DL misuses to reproduce*: The 891 misuses in DLMISUSES span various misuse types and root causes, the combination of which can be divided into stratas. Accordingly, we follow a stratified sampling approach to diversify the selected DL misuses to cover all stratas. We first discard misuses of the violation type “outdated” and the root cause “deprecation management errors” to be consistent with DSMISUSES as considering deprecation as a misuse is controversial [10]. This leaves 395 misuses. Based on a 95% confidence interval and 10% error margin, we need a sample size of 78 misuses.

Since reproducing misuses is time-consuming, we set a cap of 30 hours on our reproduction task. During this time period, we managed to review 73 misuses covering all strata. We disregard 52 misuses that were logical bugs. For example, Wei et al. [4] considered this change—`return relu(Y+X)` to `return relu(Y)` [41]—as a parameter replacement. But we observe that the line before the return statement is `Y +=`

X. Thus, we conclude the change is a logical bug fix instead of a misuse fix. We also discarded 6 misuses whose reproduction step took more than twenty minutes, which is the maximum time we spent on a single misuse. Overall, we reproduced 15 misuses and confirmed their type and root cause. Twelve of these misuses are data dependent and 7 have API directives.

Similar to DSMISUSES, we add a correct version of each misuse, totaling 30 code snippets. In this RQ, we use `llama` with the  $P_{full}$  prompt based on its highest  $F_1$ -score in RQ1.

### B. RQ3 Results

Given the 30 DL code snippets, we find that the detection  $F_1$ -score of DSCHECKER is 48%, it also fixed 4 misuses resulting in a fix rate of 26.67%. This suggests that DSCHECKER works for other data-centric libraries. However, given that DSCHECKER’s detection  $F_1$ -score on DSMISUSES was 61.18% (Table I), it seems that detecting DL API misuses is more challenging. This is not surprising as Wei et al. [4] report a recall of 16.49% and a precision of 32.33% on their complete data set. It is interesting to note that if we focus only on the 14 code snippets where the used API had a directive, DSCHECKER’s detection  $F_1$ -score increases to 66.67%, closely matching its  $F_1$ -score of 68.18% on DSMISUSES, as shown at the bottom of Table I. Three of the four misuses DSCHECKER fixed were misuses with API directives, further highlighting the importance of directives in misuse detection.

To further contextualize our results, we look at LLMAPIDET’s results to see how it performed on the 15 misuses we reproduced. We find that only four of our 15 reproduced misuses overlapped with those in Wei et al. [4]’s test set. Two of these four common misuses were not detected by either tool, one was detected and fixed only by DSCHECKER, while one was detected and fixed by only LLMAPIDET. One of the undetected misuses involved a missing `NaN` check on the output of the neural network’s loss function, caused by an algorithmic error. The other undetected misuse was the absence of a `to()` call to set a tensor’s device to match that of another interacting tensor. The misuse that only DSCHECKER detected and fixed involved passing a list to tensors when calculating the mean along the rows. Since `mean()` requires a tensor as input, the tensors in the list needed to be stacked first using `stack()`. The misuse that only LLMAPIDET detected and fixed involved a project specific type check.

*RQ3 findings:* DSCHECKER’s detection  $F_1$ -score on the DL sample dataset is 48%, with a 26.67% fix rate. It detected and fixed a DL misuse that LLMAPIDET did not detect.

## VIII. RQ4: HOW DOES DSCHECKER COMPARE TO OTHER LLM-BASED API MISUSE DETECTION/FIXING TOOLS?

### A. RQ4 Setup

We also compare how LLMAPIDET [4], described in Section II, performs on DSMISUSES. As a reference point, recall that as per their reported results on deep learning misuses, LLMAPIDET detected 48 misuses (16.49% recall rate) and fixed 10/48 (22.83%) of them [4]. As a proportion of the total number of evaluated misuses, its fix rate is  $\sim 3\%$  (10/291).

To use LLMAPIDET on DSMISUSES, we need to use some misuses for the rule generation phase that are different from the misuses we use for evaluation. Since Wei et al. had multiple instances of the same/similar misuses in DLMISUSES, they created a single split between rule generation data and evaluation data. However, DSMISUSES mainly contains *unique* misuses. To still enable the comparison, we come up with two setups. In the first setup, we apply LLMAPIDET to our data using the rules that Wei et al. [4] created from PyTorch and TensorFlow misuses. Such an experiment can reveal if the misuse rules they collected are generalizable to other data science libraries. In the second setup, we create API misuse rules from the DSMISUSES dataset using a 5-fold cross-validation and report average recall for misuse detection and average fix rate for fixing misuses.

We use `4o-mini` instead of the now-deprecated `gpt-3.5` used by Wei et al. [4], as it is the closest available alternative. Reproducing LLMAPIDET with `4o-mini` yields a recall of 17.81%, closely matching the originally reported 16.49%. This result validates our reproduction setup. Similar to Wei et al. [4], we only report recall for detection by applying LLMAPIDET to the 38 misuses in the DSMISUSES.

### B. RQ4 Results

In the first setup, we find that LLMAPIDET could not detect any of the 38 misuses. Even though the DSMISUSES contains misuses caused by similar root causes, such as data conversion errors due to a missing `dtype` argument in NumPy APIs (which is also a common misuse pattern in DL APIs), LLMAPIDET failed to detect them. We observed that the rules generated by LLMAPIDET contain specific DL API names and when the tool tries to find semantically matching rules to apply for a given code snippet, it first applies a filtering step where it looks for overlapping API names in the code snippet and rules. As a result, LLMAPIDET fails to find matching rules for code snippets in DSMISUSES, since none use DL APIs.

In the second setup with 5-fold cross validation, LLMAPIDET detects only four unique misuses across all folds, achieving 11.19% average recall. We note that all these four misuses were of Matplotlib APIs. As shown in the results of RQ1 in Table I,  $P_{full}$  with `4o-mini` achieves a 46.15% recall, representing a 34.96% increase over LLMAPIDET.

LLMAPIDET did not fix any of the four detected misuses. Upon further inspection of the candidate fix rules selected by the LLM, we find that while the rules were related to Matplotlib, none were applicable—possibly due to the unique nature of each misuse in DSMISUSES. This highlights a key advantage of DSCHECKER: it does not rely on prior examples of API misuse fixes.

*RQ4 findings:* DSCHECKER detects  $\sim 35\%$  more misuses than LLMAPIDET on DSMISUSES. Additionally, LLMAPIDET was not able to fix any of the misuses it detected.

## IX. RQ5: CAN DSCHECKER<sub>agent</sub> DETECT AND FIX API MISUSES?

### A. RQ5 Setup

All the above results suggest that DSCHECKER is effective at detecting and fixing misuses. However, our setup so far



has been idealistic where we know which API to provide information about, and we have this information readily available. To investigate a more realistic setup, we implement `DSCHECKERagent` which relies solely on function calling (without agentic frameworks such as ReAct [32]) to allow the LLM to request API documentation and variable information on demand. We use  $P_{base}$  that contains only the code snippet and the library used in it and modify the system message to inform the LLM that it can request information by calling functions. We implement two functions that allow the LLM to request additional information, without restricting the number of times they can be called.

1) *Function 1: `get_variable_info`*: This function takes a variable name and line number as input. Based on the input parameters, it then analyzes the abstract syntax tree (AST) of the current code snippet to determine where to insert instrumentation. The inserted code uses appropriate print statements similar to those manually added in Section IV-A to extract relevant information. The function then executes the instrumented code and returns the variables type, sample values, and any additional available details.

2) *Function 2: `get_api_documentation`*: This function takes an API name as input and returns its full documentation by querying a locally maintained index covering five libraries in `DSMISUSES`. Note that this function returns the full documentation of a given API, rather than a specific API directive. Since we do not know which directive is relevant and automatically extracting directives from documentation is challenging [12], [42], [43], we provide full API documentation and leave the LLM to decide how to use that information.

Since `4o` and `llama` had no statistically significant differences in RQ1, we use both models. In contrast to  $P_{full}$  in RQ1 where the prompt includes exact pre-determined information catered to each misuse, RQ5 setup lets the LLM decide which information it needs and for which exact variable or API.

## B. RQ5 Results

1) *Misuse detection*: We find that `DSCHECKERagent` achieves an  $F_1$ -score of 43.33% with `llama` and 48.65% with `4o` for misuse detection, with the latter being significantly higher ( $p < 0.05$ ). These results suggest that, overall, the LLM (`4o` in this case) can effectively request the information it needs to determine API misuse. This shows that our approach could be integrated into a practical setting without pre-defined knowledge of the misused API.

On the other hand, `DSCHECKERagent` achieves a  $\sim 7\%$  significantly lower  $F_1$ -score than `DSCHECKER` with `4o` on  $P_{full}$  ( $p < 0.05$ ). To understand this drop, we examine the function calls made by `DSCHECKERagent`. In total, `4o` made 136 function calls, averaging 1-2 calls per code snippet. Each code snippet in our dataset contains 1-23 APIs (median 6), belonging to the targeted library and other libraries. `4o` called for documentation 111 times, 26 of which were made for the corresponding potentially misused API. This shows that the LLM often requested the relevant API information. However, the decreased  $F_1$ -score suggests that providing the full API

documentation is less effective than directly offering the relevant directive. `4o` called `get_variable_info` 25 times, 14 of which were for the variable processed by potentially misused API. In comparison to `4o`, `llama` also made 111 function calls, all of which were for API documentation. Only 15 of 111 were for the potentially misused API which possibly explains its lower  $F_1$ -score relative to `4o`.

2) *Misuse fixing*: In misuse fixing, `4o` and `llama` achieved fix rates of 39.47% and 21.05%, respectively. Compared to the best zero-shot fix rate in Table II, the 12% drop for `4o` is not statistically significant ( $p = 0.941$ ). In contrast, `llama`'s 30% decrease relative to its best zero-shot performance is statistically significant ( $p < 0.05$ ).

*RQ5 findings:* `DSCHECKERagent` achieved a detection  $F_1$ -score of 48.65% and a fix rate of 39.47% with `4o`. Out of 136 function calls, 40 were for potentially useful information.

## X. DISCUSSION

In this section, we discuss the specific implications of our work for these developers and their support tools, as well as for researchers and library authors.

a) *Implications for tooling and developers*: Our findings demonstrate the strong potential of using LLMs for detecting and fixing API misuses with a data-centric nature. `DSCHECKER` eliminates the need for prior knowledge of fixing misuses, which are requirements that have traditionally constrained both conventional API misuse detection tools [6], [8], [9], [13]–[15] and even more recent machine-learning/LLM-based solutions [4], [20]. Our experiments show that, when provided with the right information, such as API directives and dynamic data information, LLMs can provide effective tooling off-the-shelf for this task. While the performance of `DSCHECKERagent` suggests practical promise for developer use, the decreased detection and fixing scores indicate that further improvements by researchers may be needed, which we discuss next.

b) *Implications for researchers and library authors*: To the best of our knowledge, this paper is the first to explore LLMs' agentic behavior in the context of API misuse. Our findings raise several challenges for future research.

First, while the LLM often requested API documentation, we noticed hallucinated requests for APIs that either did not exist or were not used in the current code snippet. Researchers can explore potential validation steps before executing the called functions or feedback loops that inform the LLM of its meaningless requests.

Second, even in cases where the LLM requested documentation for the correct API, it often struggled to effectively utilize this information for accurate detection and repair. This suggests that current API documentation formats, primarily designed for human consumption, pose a challenge for LLMs. We also observed cases where the critical API directive was not in the retrieved API reference documentation page but in other tutorial or example pages in the library's documentation. While automatically identifying API directives remains an open and difficult problem [3], [12], [16], [42], [43], our



findings underscore the need for further work in this area. With developers' increased reliance on LLMs, library authors could also focus on augmenting API documentation with LLM-friendly structured formats to improve machine readability.

#### XI. THREATS TO VALIDITY

*Internal Validity.* We manually assess the correctness of the LLM's fixes and explanations. While manual processes are often subjective, the dataset contains detailed descriptions of the problem and the error messages or incorrect output reflecting the problem. Thus, we could precisely assess if the fix addressed the problem, especially with re-running the code.

*Construct Validity.* A model may label code as incorrect but suggest an unrelated fix, which would inflate detection rates without reflecting true understanding. To reduce this risk, we require that the explanation aligns with the misuse description in the dataset to consider it a successful detection. Additionally, since a misuse can have multiple valid fixes, we run the patched code to verify that the misuse is resolved, regardless of the implementation.

It is possible that our dataset might overlap with LLM's training data. To reduce this risk, we did not directly reuse code from Stack Overflow posts or GitHub commits when constructing the code snippets. This makes it less likely that models succeed merely by memorizing training data rather than demonstrating actual misuse detection ability.

*Conclusion Validity.* Our dataset size may limit the reliability of the comparisons. Additionally, using a single measurement to compare treatments, such as LLM variants and prompts, may provide a misleading picture. To reduce this risk, we compare distributions of measurements through bootstrapping, enabling a statistically grounded analysis.

*External Validity.* We evaluate only three LLMs in this work. Experimenting with additional LLMs may produce different results. However, we tried to select representative LLMs that have been shown to perform well in previous work [27], [38]. We mix between different size proprietary LLMs from OpenAI as well as an open-source model with a large size.

LLMs are inherently non-deterministic, meaning their outputs can vary across different runs. This variability can be particularly challenging in our evaluation process, which involved two manual steps: assessing free-text explanations and executing generated code. Due to the resource-intensive nature of these evaluations, instead of running multiple times for each data point, we set the temperature to 0.0, to produce more focused and deterministic responses as much as possible [44].

Our primary evaluation was conducted on the DSMISUSES, a dataset with a limited number of misuses but which has the crucial advantage of providing minimally reproducible examples for thorough validation. To explore broader applicability, we extended our evaluation to a sample of DL misuses.

#### XII. CONCLUSION

This study investigated LLM effectiveness in detecting and fixing API misuses in data science libraries, which are inherently harder to detect given their data-centric nature. We show that providing the LLM with API directives and dynamic variable information enhances performance: the best model,

llama, achieved a detection  $F_1$ -score of 61.18% and correctly fixed 51.28% of misuses with this added context. Our experiments further demonstrate that DSCHECKER is applicable beyond the initial data science libraries, successfully detecting API misuses in deep learning libraries, even identifying cases missed by an existing LLM-based tool specifically designed for DL misuse detection. Notably, that tool struggled when it lacked examples to learn misuse-fixing patterns. We also explored equipping the LLM with agentic behavior for real-world settings. While the detection  $F_1$ -score dropped to 48.65% and fix score to 39.47%, the results show promise for practical settings where no prior information about the potential misuse is known. We also identified future directions to further pursue and improve LLM agentic behavior.

#### XIII. ACKNOWLEDGEMENT

We used ChatGPT [45] for proofreading and content reduction, per IEEE's submission policy [46].

#### REFERENCES

- [1] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Trans. Soft. Eng.*, vol. 45, no. 12, pp. 1170–1188, 2019. [Online]. Available: <https://doi.org/10.1109/TSE.2018.2827384>
- [2] M. Schlichtig, S. Sassalla, K. Narasimhan, and E. Bodden, "FUM - A framework for API usage constraint and misuse classification," in *IEEE Int. Conf. Soft. Anal., Evol. and Reeng., SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 2022, pp. 673–684. [Online]. Available: <https://doi.org/10.1109/SANER53432.2022.00085>
- [3] H. Zeng, J. Chen, B. Shen, and H. Zhong, "Mining API constraints from library and client to detect API misuses," in *28th Asia-Pacific Soft. Eng. Conf., APSEC 2021, Taipei, Taiwan, December 6-9, 2021*. IEEE, 2021, pp. 161–170. [Online]. Available: <https://doi.org/10.1109/APSEC53868.2021.00024>
- [4] M. Wei, N. S. Harzevili, Y. Huang, J. Yang, J. Wang, and S. Wang, "Demystifying and detecting misuses of deep learning apis," in *Proc. 46th Int. Conf. Soft. Eng., ICSE 2024, Lisbon, Portugal, April 14 - 20, 2024*. IEEE, 2024, pp. 1–13.
- [5] 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 *IEEE Int. Conf. Soft. Anal., Evol. and Reeng., SANER 2023, Taipa, Macao, March 21-24, 2023*. IEEE, 2023, pp. 522–533. [Online]. Available: <https://doi.org/10.1109/SANER56733.2023.00055>
- [6] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating next steps in static api-misuse detection," in *Proc. 16th Int. Conf. Mining Soft. Repo., MSR 2019, 26-27 May 2019, Montreal, Canada*. IEEE / ACM, 2019, pp. 265–275. [Online]. Available: <https://doi.org/10.1109/MSR.2019.00053>
- [7] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath, "Cognicrypt: supporting developers in using cryptography," in *Proc. 32nd IEEE/ACM Int. Conf. Auto. Soft. Eng., ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 2017, pp. 931–936. [Online]. Available: <https://doi.org/10.1109/ASE.2017.8115707>
- [8] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large soft. code," in *Proc. 10th Europ. Soft. Eng. Conf. held jointly with 13th ACM SIGSOFT Int. Symp. Found. Soft. Eng., 2005, Lisbon, Portugal, September 5-9, 2005*. ACM, 2005, pp. 306–315. [Online]. Available: <https://doi.org/10.1145/1081706.1081755>
- [9] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proc. 6th joint meeting Euro. Soft. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Soft. Eng., 2007, Dubrovnik, Croatia, September 3-7, 2007*. ACM, 2007, pp. 35–44. [Online]. Available: <https://doi.org/10.1145/1287624.1287632>
- [10] A. Galappaththi, S. Nadi, and C. Treude, "An empirical study of API misuses of data-centric libraries," in *Proc. 18th ACM/IEEE Int. Symp. Emp. Soft. Eng. Meas., ESEM 2024, Barcelona, Spain, October 24-25, 2024*. ACM, 2024, pp. 245–256. [Online]. Available: <https://doi.org/10.1145/3674805.3686685>

- [11] scikit learn, “Simpleimputer,” 2023. [Online]. Available: <https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html>
- [12] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, “What should developers be aware of? an empirical study on the directives of API documentation,” *Empir. Softw. Eng.*, vol. 17, no. 6, pp. 703–737, 2012. [Online]. Available: <https://doi.org/10.1007/s10664-011-9186-4>
- [13] M. Monperrus and M. Mezini, “Detecting missing method calls as violations of the majority rule,” *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, pp. 7:1–7:25, 2013. [Online]. Available: <https://doi.org/10.1145/2430536.2430541>
- [14] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, “Graph-based mining of multiple object usage patterns,” in *Proc. 7th joint meeting of the European Soft. Eng. Conf. and the ACM SIGSOFT Int. Symp. on Found. Soft. Eng., 2009, Amsterdam, The Netherlands, August 24-28, 2009*. ACM, 2009, pp. 383–392. [Online]. Available: <https://doi.org/10.1145/1595696.1595767>
- [15] C. Lindig, “Mining patterns and violations using concept analysis,” in *The Art and Science of Analyzing Soft. Data*. Morgan Kaufmann / Elsevier, 2015, pp. 17–38. [Online]. Available: <https://doi.org/10.1016/b978-0-12-411519-4.00002-1>
- [16] X. Ren, X. Ye, Z. Xing, X. Xia, X. Xu, L. Zhu, and J. Sun, “Api-misuse detection driven by fine-grained api-constraint knowledge graph,” in *35th IEEE/ACM Int. Conf. Auto. Soft. Eng., ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 461–472. [Online]. Available: <https://doi.org/10.1145/3324884.3416551>
- [17] Z. Gu, J. Wu, C. Li, M. Zhou, Y. Jiang, M. Gu, and J. Sun, “Vetting API usages in C programs with imchecker,” in *Proc. 41st Int. Conf. Soft. Eng.: Comp. Proc., ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 91–94. [Online]. Available: <https://doi.org/10.1109/ICSE-Companion.2019.00046>
- [18] C. Li, Z. Gu, M. Zhou, J. Wu, J. Zhang, and M. Gu, “API misuse detection in C programs: Practice on SSL apis,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 29, no. 11&12, pp. 1761–1779, 2019. [Online]. Available: <https://doi.org/10.1142/S0218194019400205>
- [19] M. Gulami, “A human-in-the-loop approach to generate annotation usage rules,” Master’s thesis, University of Alberta, 2022.
- [20] H. Baek, M. Lee, and H. Kim, “Cryptollm: Harnessing the power of llms to detect cryptographic API misuse,” in *Comp. Sec. - ESORICS 2024 - 29th European Symp. Res. Comp. Sec., Bydgoszcz, Poland, September 16-20, 2024, Proc., Part I*, ser. Lecture Notes in Computer Science, vol. 14982. Springer, 2024, pp. 353–373. [Online]. Available: [https://doi.org/10.1007/978-3-031-70879-4\\_18](https://doi.org/10.1007/978-3-031-70879-4_18)
- [21] I. Muslukhov, Y. Boshmaf, and K. Beznosov, “Source attribution of cryptographic api misuse in android applications,” in *Proc. 2018 Asia Conf. Comp. Commun. Sec.*, ser. ASIACCS ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 133146. [Online]. Available: <https://doi.org/10.1145/3196494.3196538>
- [22] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, “An empirical study of cryptographic misuse in android applications,” in *Proc. 2013 ACM SIGSAC Conf. Comp. Commun. Sec.*, ser. CCS ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 7384. [Online]. Available: <https://doi.org/10.1145/2508859.2516693>
- [23] S. Krger, J. Spath, K. Ali, E. Bodden, and M. Mezini, “Crysl: An extensible approach to validating the correct usage of cryptographic apis,” *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2382–2400, 2021.
- [24] W. Baker, M. O’Connor, S. R. Shahamiri, and V. Terragni, “Detect, fix, and verify tensorflow API misuses,” in *IEEE Int. Conf. Soft. Anal., Evol. and Reeng., SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 2022, pp. 925–929. [Online]. Available: <https://doi.org/10.1109/SANER53432.2022.00110>
- [25] D. Nam, A. Macvean, V. J. Hellendoorn, B. Vasilescu, and B. A. Myers, “Using an LLM to help with code understanding,” in *Proc. 46th IEEE/ACM Int. Conf. Soft. Eng., ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 97:1–97:13. [Online]. Available: <https://doi.org/10.1145/3597503.3639187>
- [26] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models,” in *Proc. 32nd ACM SIGSOFT Int. Symp. on Soft. Test. Anal., ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*. ACM, 2023, pp. 423–435. [Online]. Available: <https://doi.org/10.1145/3597926.3598067>
- [27] H. Guan, G. Bai, and Y. Liu, “Large language models can connect the dots: Exploring model optimization bugs with domain knowledge-aware prompts,” in *Proc. 33rd ACM SIGSOFT Int. Symp. on Soft. Test. Anal., ISSTA 2024, Vienna, Austria, September 16-20, 2024*. ACM, 2024, pp. 1579–1591. [Online]. Available: <https://doi.org/10.1145/3650212.3680383>
- [28] C. S. Xia, Y. Wei, and L. Zhang, “Automated program repair in the era of large pre-trained language models,” in *45th IEEE/ACM Int. Conf. Soft. Eng., ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1482–1494. [Online]. Available: <https://doi.org/10.1109/ICSE48619.2023.00129>
- [29] A. Plaet, M. van Duijn, N. van Stein, M. Preuss, P. van der Putten, and K. J. Batenburg, “Agentic large language models, a survey,” 2025. [Online]. Available: <https://arxiv.org/abs/2503.23037>
- [30] OpenAI, “Function calling,” 2024. [Online]. Available: <https://platform.openai.com/docs/guides/function-calling>
- [31] R. Nakano, J. Hilton, S. Balaji, J. Wu, L. Ouyang, C. Kim, C. Hesse, S. Jain, V. Kosaraju, W. Saunders, X. Jiang, K. Cobbe, T. Eloundou, G. Krueger, K. Button, M. Knight, B. Chess, and J. Schulman, “Webgpt: Browser-assisted question-answering with human feedback,” *CoRR*, vol. abs/2112.09332, 2021. [Online]. Available: <https://arxiv.org/abs/2112.09332>
- [32] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafraan, K. R. Narasimhan, and Y. Cao, “React: Synergizing reasoning and acting in language models,” in *11th Int. Conf. Learn. Rep., ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.
- [33] N. Karanikolas, E. Manga, N. E. Samaridi, E. Tousidou, and M. Vassilakopoulos, “Large language models versus natural language understanding and generation,” in *Proc. 27th Pan-Hellenic Conf. Progress in Computing and Informatics, PCI 2023, Lamia, Greece, November 24-26, 2023*. ACM, 2023, pp. 278–290. [Online]. Available: <https://doi.org/10.1145/3635059.3635104>
- [34] D. Nam, A. Macvean, V. J. Hellendoorn, B. Vasilescu, and B. A. Myers, “Using an LLM to help with code understanding,” in *Proc. 46th IEEE/ACM Int. Conf. Soft. Eng., ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 97:1–97:13. [Online]. Available: <https://doi.org/10.1145/3597503.3639187>
- [35] T. Hesterberg, “Bootstrap,” *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 3, no. 6, pp. 497–526, 2011.
- [36] N. M. Razali, Y. B. Wah *et al.*, “Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests,” *Journal of statistical modeling and analytics*, vol. 2, no. 1, pp. 21–33, 2011.
- [37] S. García, D. Molina, M. Lozano, and F. Herrera, “A study on the use of non-parametric tests for analyzing the evolutionary algorithms behaviour: a case study on the cec2005 special session on real parameter optimization,” *Journal of Heuristics*, vol. 15, pp. 617–644, 2009.
- [38] H. Li, Y. Hao, Y. Zhai, and Z. Qian, “Enhancing static analysis for practical bug detection: An llm-integrated approach,” *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, pp. 474–499, 2024. [Online]. Available: <https://doi.org/10.1145/3649828>
- [39] P. Xu, W. Ping, X. Wu, L. McAfee, C. Zhu, Z. Liu, S. Subramanian, E. Bakhturina, M. Shoyebi, and B. Catanzaro, “Retrieval meets long context large language models,” in *12th Int. Conf. Learn. Rep.*, 2024. [Online]. Available: <https://openreview.net/forum?id=xw5nxFWMIo>
- [40] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, “Lost in the middle: How language models use long contexts,” 2023. [Online]. Available: <https://arxiv.org/abs/2307.03172>
- [41] d2l ai, “Interactive deep learning book with multi-framework code, math, and discussions,” 2025. [Online]. Available: <https://github.com/d2l-ai/d2l-en/commit/fa8337de966f77749a1fe80abd86f68f80146515>
- [42] M. P. Robillard and Y. B. Chhetri, “Recommending reference API documentation,” *Empir. Softw. Eng.*, vol. 20, no. 6, pp. 1558–1586, 2015. [Online]. Available: <https://doi.org/10.1007/s10664-014-9323-y>
- [43] W. Maalej and M. P. Robillard, “Patterns of knowledge in API reference documentation,” *IEEE Trans. Soft. Eng.*, vol. 39, no. 9, pp. 1264–1282, 2013. [Online]. Available: <https://doi.org/10.1109/TSE.2013.12>
- [44] OpenAI, “Making requests,” 2024. [Online]. Available: <https://platform.openai.com/docs/api-reference/making-requests>
- [45] —, “ChatGPT (June 25 version) [large language model],” 2024. [Online]. Available: <https://chatgpt.com/>
- [46] IEEE, “Submission policies,” 2025. [Online]. Available: <https://conferences.ieeeauthorcenter.ieee.org/author-ethics/guidelines-and-policies/submission-policies/>