Demystifying and Detecting Misuses of Deep Learning APIs

Moshi Wei
York University
Toronto, Canada
moshiwei@yorku.ca

Nima Shiri Harzevili
York University
Toronto, Canada
nshiri@yorku.ca

YueKai Huang
Chinese Academy of Sciences
Beijing, China
huangyuekai18@mails.ucas.ac.cn

Jinqiu Yang
Concordia University
Montreal, Canada
jinqiu.yang@concordia.ca

Junjie Wang
Chinese Academy of Sciences
Beijing, China
junjie@iscas.ac.cn

Song Wang
York University
Toronto, Canada
wangsong@yorku.ca

ABSTRACT

Deep Learning (DL) libraries have significantly impacted various domains in computer science over the last decade. However, developers often face challenges when using the DL APIs, as the development paradigm of DL applications differs greatly from traditional software development. Existing studies on API misuse mainly focus on traditional software, leaving a gap in understanding API misuse within DL APIs. To address this gap, we present the first comprehensive study of DL API misuse in TensorFlow and PyTorch. Specifically, we first collected a dataset of 4,224 commits from the top 200 most-starred projects using these two libraries and manually identified 891 API misuses. We then investigated the characteristics of these misuses from three perspectives, i.e., types, root causes, and symptoms. We have also conducted an evaluation to assess the effectiveness of the current state-of-the-art API misuse detector on our 891 confirmed API misuses. Our results confirmed that the state-of-the-art API misuse detector is ineffective in detecting DL API misuses. To address the limitations of existing API misuse detection for DL APIs, we propose *LLMAPIDet*, which leverages Large Language Models (LLMs) for DL API misuse detection and repair. We build *LLMAPIDet* by prompt-tuning a chain of ChatGPT prompts on 600 out of 891 confirmed API misuses and reserve the rest 291 API misuses as the testing dataset. Our evaluation shows that *LLMAPIDet* can detect 48 out of the 291 DL API misuses while none of them can be detected by the existing API misuse detector. We further evaluate *LLMAPIDet* on the latest versions of 10 GitHub projects. The evaluation shows that *LLMAPIDet* can identify 119 previously unknown API misuses and successfully fix 46 of them.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Software libraries and repositories**; • **Computing methodologies → Machine learning**.

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510159

KEYWORDS

API misuse, deep learning APIs, empirical study, detection

ACM Reference Format:
Moshi Wei, Nima Shiri Harzevili, YueKai Huang, Jinqiu Yang, Junjie Wang, and Song Wang. 2024. Demystifying and Detecting Misuses of Deep Learning APIs. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510159

1 INTRODUCTION

Over the last decade, Deep Learning (DL) libraries have played an important role in various domains, such as image processing [39], autonomous driving [47], face recognition [17], drug discovery [8], and natural language processing [31]. These libraries facilitate the implementation of complex deep-learning algorithms and have revolutionized the field of computer science. However, developers often face challenges when using the DL APIs within these libraries. Due to the fact that the DL development paradigm differs greatly from traditional software development [24], knowledge in traditional software development may not be directly applicable to DL APIs. Developers who lack familiarity with DL APIs and relevant concepts may frequently encounter API misuses.

Existing studies on API misuse have primarily focused on Java APIs of traditional software [5, 6, 44, 55]. However, there has been limited investigation specifically into API misuse within Python DL APIs [7, 19, 49]. Islam et al. [19] conducted a comprehensive analysis of DL bugs by manually examining over 2,500 StackOverflow posts and GitHub commits related to five popular DL frameworks. Although they categorized API bugs, they did not delve into the specific details of these instances. Baker et al. [7] performed a manual analysis of 15 TensorFlow bugs, identifying 7 API misuse patterns based on a dataset from a pre-existing DL defect study [18, 57]. However, their studies' limited scale raises concerns about their representativeness. Wan et al. [49] conducted an API misuse study on cloud computing service APIs for four major cloud computing service providers across three application directions. Their focus was on performance problems and user experience of commercial APIs, leaving the characteristics of API misuse in lower-level building-block APIs like PyTorch and TensorFlow unexplored. While prior studies have provided basic insights into DL API misuses, a comprehensive study in this area remains absent, which would be highly valuable as it can provide developers with guidance regarding potential pitfalls when using APIs in the development of new applications.

In this study, we present the first comprehensive DL API misuse study of two major DL libraries, i.e., TensorFlow and PyTorch. Specifically, we first collected 18,794 bug-fix commits from the top 200 most-starred projects using either PyTorch or TensorFlow. We then employ a set of heuristic filtering to eliminate irrelevant commits (e.g., changes that do not involve API names, API conditions, or API parameters), resulting in a refined dataset of 4,224 commits for manual analysis. The details of the data collection are described in Section 3. After that, we manually investigated and analyzed the 4,224 commits and successfully identified 891 API misuses. To better understand the characteristics of DL API misuse, we performed the investigation from three perspectives, i.e., API misuse categories, root causes of API misuse, and symptoms of API misuse. In addition, we investigated the feasibility of utilizing a state-of-the-art API misuse detector [7] to identify DL API misuses. Our analysis reveals a fundamental distinction between identifying DL API misuses and traditional software API misuses. Traditional API misuse detectors [26, 27, 37, 51] mainly rely on static program analysis, whereas DL API misuse primarily involves issues related to data and device usages such as returning incorrect float types or accidentally running GPU tasks on CPU. These issues often manifest as performance differences or incorrect results, rather than triggering errors that can be easily detected through static analysis. Also, identifying these types of DL API misuses heavily relies on human experience and a comprehensive understanding of the source code semantics. As a result, existing detectors may not be suitable for effectively detecting DL API misuses, given the substantial contextual shift and the unique characteristics compared to traditional software API misuses.

In response to the limitations of existing API misuse detection for DL API misuses, we propose *LLMAPIDet* that leverages Large Language Models (LLMs) to detect and fix DL API misuses. We base our study on ChatGPT [32] as it has shown great potential in software engineering, e.g., code generation [9], bug repair [43], and program understanding [52], among comparable LLMs [50].

The primary idea of *LLMAPIDet* is to identify API misuse by applying a set of predefined rules to API usage instances. The proposed approach initiates the process by constructing a corpus of natural language API misuse rules by querying the ChatGPT with our collection of API misuse code examples. For each API usage code snippet, the approach first utilizes the ChatGPT to generate its code explanation. Subsequently, it employs a retrieval-based few-shot learning technique to provide the most relevant API misuse rules to ChatGPT, enabling it to determine whether the code snippet involves an API misuse. Then we instruct ChatGPT to generate the patch.

We build *LLMAPIDet* by prompt-tuning a chain of ChatGPT prompts on 600 out of 891 confirmed API misuses and reserve the remaining 291 API misuses as the testing dataset. Our evaluation shows that *LLMAPIDet* can successfully detect 48 API misuses. Note that as ChatGPT's training data include source code from publicly available open-source projects, to avoid data leaking issue [3, 21], we further collected 4,359 DL API usage instances from the latest versions of 10 GitHub projects built with Pytorch and TensorFlow other than the 200 projects used in our empirical analysis. All the versions are released after July 1st, 2023. Our evaluation on the 4,359 DL API usage instances shows that *LLMAPIDet* can identify 119 previous unknown API misuses and successfully fix 46 of them, whereas the state-of-the-art tool could only detect 5.

The contribution of this paper is as the following:

- We present the first large-scale analysis to demystify and detect DL API misuses in PyTorch and TensorFlow.
- We create a benchmark DL API misuse dataset including 891 instances of DL API misuse. We provide detailed taxonomies regarding the types, root causes, and symptoms of DL API misuses.
- Motivated by the ineffectiveness of state-of-the-art API misuse detection on DL API misuses, we further present a novel LLM-based API misuse detector, i.e., *LLMAPIDet*, to detect and repair DL API misuses. Our evaluation shows that our tool can outperform the state-of-the-art API misuse detection.
- We provide a set of practical guidelines to help machine learning development teams develop reliable programs by avoiding and detecting DL API misuses.
- We release the dataset and source code of our experiments to help other researchers replicate and extend our study[1].

The structure of the rest of the paper is as follows: Section 2 introduces the background and related work of this study. Section 3 describes the approach used for the empirical study. Section 4 presents the empirical study and result analysis. Section 5 presents the evaluation of state-of-the-art API misuse detection on DL API misuses. Section 6 details our proposed LLM-based DL API misuse detector. Section 7 discusses our findings. Section 8 discusses threats to validity. Lastly, Section 9 summarizes the paper.

2 BACKGROUND AND RELATED WORK

2.1 API Misuse

An API misuse refers to the incorrect usage of an Application Programming Interface (API), where there are violations of the API's usage constraints, such as call order or preconditions [44]. These misuses can lead to various issues, including software crashes, bugs, data loss, and vulnerabilities.

Most of the existing API misuse studies mainly focus on Java projects. Amann et al. [5] presented the API misuse dataset, i.e., MUBENCH, which contains manually identified 89 API misuses from 33 general Java projects. Later, they further presented a systematic evaluation of existing API misuse detectors for Java [6].

Recently, Baker et al. [7] conducted an API misuse study on TensorFlow API misuses and also proposed an API misuse detector for TensorFlow APIs. They performed a manual investigation on a limited scale of 15 Tensorflow bugs and identified 7 API misuse patterns based on existing datasets from two empirical studies on DL program bugs [18, 57]. We manually investigated over 4,000 API misuse instances and confirmed 891 API misuses on Pytorch and TensorFlow APIs. As far as we know, this is the first large-scale API misuse study focused on DL libraries exclusively. Li et al. [25] performed a large-scale study on API misuse for over 500,000 bug fixes and classified them into 9 categories for general Java Projects. Compared to our work, they focus on general API misuse in open-source Java projects, while we focus on Python Deep Learning

[1] https://anonymous.4open.science/r/LLMAPIDet_replication-C157

APIs. Also, our findings reveal several significant differences in misuse categories and patterns. Wan et al. [49] conducted an API misuse study on machine learning cloud service APIs for 4 major service providers on 3 primary application types. Compared to our work, they focus on the cloud service APIs from commercial service providers such as Microsoft Azure and 3 application types such as vision, language, and speech, while we focus on open-source backbone deep learning libraries APIs in PyTorch and TensorFlow.

2.2 LLM in Software Engineering

Recently, the field of software engineering research has witnessed a significant impact with the emergence of Large Language Models (LLMs) [50]. Chen et al. [9] introduced Codex, the first GPT-based LLM for source code, which subsequently served as the foundational model for code-related research. Codex has found applications in multiple software engineering domains, including automated program repair (Prenner et al. [36] and Sobania et al. [42]), automated test case generation (Xie et al. [53] and Yuan et al. [54]), and vulnerability detection (Cheshkov et al. [12] and Nair et al. [30]).

Despite the impressive code generation abilities exhibited by LLMs, researchers have begun to evaluate the quality of the generated code from multiple perspectives, including correctness (Liu et al. [28]), performance (Feng et al. [15]), and security (Khoury et al. [23]). Prompt engineering is the primary approach for building LLM-based applications. Previous work has studied the methodology of prompt tuning (Zhang et al. [56] and Shrivastava et al. [41]) specifically in the context of software engineering. In our work, we design and refine prompts through iterative testing and evaluation on a small dataset.

3 METHODOLOGY

3.1 Data Collection

3.1.1 Experiment Library and Project Selection. To collect experiment data, we choose PyTorch and TensorFlow as the target DL libraries due to their wide acceptance in both industry and academia. PyTorch [33], officially released in 2016 by the FAIR research lab, has gained immense popularity in the field of deep learning research due to its flexibility and ease of use. On the other hand, TensorFlow [1], developed by the Google Brain team in 2016, has found extensive utilization in industrial settings owing to its comprehensive ecosystem and broad coverage of use cases. Note that, we have excluded other DL libraries, such as Caffe [20], MXNet [11], Theano [4], and CNTK [40], due to their comparatively lower popularity and maintenance activity. We have also excluded machine learning libraries like scikit-learn [34], XGBoost [10], and LightGBM [22] due to the significant differences in the paradigm and workflow between deep learning libraries and traditional statistical-based machine learning libraries.

To conduct our study, we have collected the top 200 most-starred projects for the two studied DL libraries (i.e., PyTorch and TensorFlow) from GitHub, and we also pulled the complete commit history of each project. The selected projects span from the years 2015 to 2022, with an average project creation date of 2018. The number of forks ranges from 248 to 21,624, with an average of 2,708 forks per project. The size of the repositories varies significantly, ranging from 7 MB to 19 GB, with an average size of 198 MB. The complete list of project names can be found in the replication package.

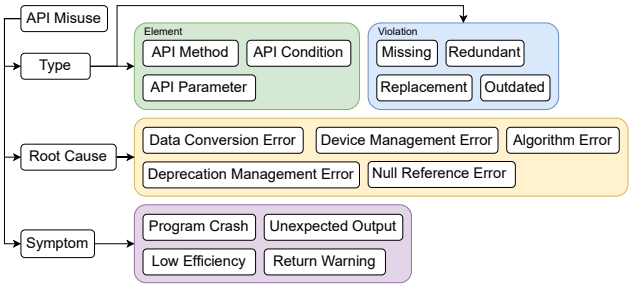

Figure 1: API Misuse Categorization

3.1.2 Data Filtering. For each experimental project, to identify its commit changes that involve DL APIs, we first used the API lists provided in the API documents of TensorFlow and PyTorch to filter out irrelevant commits. Similar to previous research [6], we observed that API misuse often involves a small number of line edits. Consequently, we restricted the number of lines in a change to less than 10, considering both additions and deletions. Please note that certain API-related commits, such as custom API document updates, string updates, typos, and logic changes, were excluded from our study since these categories are not related to API misuses but rather normal development updates. After the data collection process, we followed an existing study [25] and employed an AST diff checker called GumTree [14] to collect commit diffs at the AST tree structure level. We also observed the presence of code clones during the investigation, where the exact same code change occurred in multiple commits. Since our focus was on identifying API misuse patterns rather than code clones, we retained only one instance of each code change and eliminated the duplicates. Furthermore, we removed code changes that did not contain any methods. Following these steps, we ultimately obtained a filtered set of 4,224 commits.

3.2 Manual Analysis

For our manual analysis, three of the authors, with an average of 6 years of experience in DL development, were involved. To better understand the dataset, we first performed an exploratory data analysis (EDA) using a variation of Grounded Theory [16]. Similar to prior work [6], we manually labeled 200 randomly sampled commits and took notes on whether the instance is an API misuse, the API misuse types, involved API elements, the violation, symptoms, and root causes. We determined the labels during the process until each API misuse was tagged with one label. Each sample took an average of 6.3 minutes to label, including the process of reading commit messages, code changes, API documentation, and contextual information within the code commits.

We then summarized the collected information and made decisions on the dimensions of classification and the names of each class based on the notes taken during the EDA. Based on the above practice, we identified 3 dimensions for categorization: API misuse types, API misuse symptoms, and API misuse root causes. Figure 1

shows the dimensions and classes of our categorization. After determining the categories and classification rules, we classified all 4,224 commits and categorized them according to the above three dimensions if the instance is confirmed to be an API misuse instance. We finally confirmed 891 API misuse instances. The identified API misuses contain 311 unique API methods, covering about 10% of the APIs in PyTorch and TensorFlow libraries (3,622 APIs).

API Misuse Types: In our analysis, API misuse types are categorized based on two sub-dimensions: API Violations and API Elements. Li et al. [25] categorized the type of violation into three types, i.e., *missing*, *redundant*, and *replacement*. In this work, we expand the replacement class to *Replacement* and '*Outdated* to reflect the significant difference in the characteristics of these two types in DL API misuse. The difference is that the *Replacement* API element represents the complete replacement from one API method to another, while the *Outdated* API element represents an outdated class or API method that requires a version update. Amann et al. [6] proposed a classification scheme that included four elements of Java API misuse, i.e., *method call*, *condition*, *iteration*, and *exception handling*, while we identified 3 types of API elements for Python DL API misuse based on the observation, i.e., *API method*, *API parameter*, and *API condition*. The iteration category was omitted from our classification since it was not observed. Also, we moved the exception handling category to the root cause classification as it aligns more appropriately with that aspect.

API Misuse Root Cause: To better understand the technical characteristics of API misuse, we categorized the existing work based on root causes. Existing research on the misuse of general Python libraries APIs [25] did not specify the root cause. However, they did identify a category that overlaps with our classification, which is the null reference check category versus condition check. This category includes an if statement that checks whether a variable is null. Based on the unique feature of deep learning, we identified 4 new categories, which are *Data conversion*, *Device Management*, *Algorithm*, and *Deprecation*.

API Misuse Symptom: To understand the impact of API misuses, we classified the severity of API misuses based on their symptoms. We identified four symptom categories, i.e., *Low Efficiency*, *Program Crash*, *Unexpected Output*, and *Return Warning*. Note that *Low Efficiency* was also reported in an existing study [7].

4 CHARACTERISTICS OF API MISUSE IN DL LIBRARIES

4.1 Taxonomy of API Misuse Type

We categorize API misuse into two sub-dimensions, which are API violations (Section 4.1.1) and API elements (Section 4.1.2). Specifically, there are four types of API violations and three types of API elements. In total, 12 API misuse types are derived from considering different API violations and API elements. Table 2 shows the detailed distribution of API misuse in these two dimensions.

4.1.1 ***API Violation Type.*** We categorized API violations into four types, i.e., *Missing*, *Redundant*, *Replacement*, and *Outdated*. In general, the distribution of each violation is fairly even. The *Missing* violation indicates the absence of an API element. It contains 244 instances and is the most common violation type. Conversely, the *Redundant* violation entails a redundant API element. The redundant API method, with 138 instances, is the most common API misuse type among all 12 types. The *Replacement* violation involves replacing an existing API element, including replacing an existing API with a new one. API condition Replacement, with only 17 instances, is the rarest category among the 12 types. The *Outdated* violation represents a class change or renaming of the API elements. The difference between an *Outdated* and a *Replacement* lies in their semantics. An *Outdated* refers to a minor alteration that only affects the class or name while keeping the semantics almost the same, while a *Replacement* implies a significant modification to the API method.

4.1.2 ***API Element Type.*** We categorized API elements into three types, i.e., *API Method*, *API Parameter*, and *API Condition*. An *API Method* refers to a deep learning API method involved in the API misuse with or without parameters. *API Parameter* refers to the parameters used in an API method. The value of the parameters can be defined in the API context or in the API method argument assignment. *API Condition* refers to the internal state of the program at runtime or condition before the API method call. To better understand the meaning of each category, we present one example for each category in Table 1. API method is the most common misuse element in the Missing, Redundant, and Replacement categories. **Missing API Method** represents an API misuse because of missing the required API call. One common example in PyTorch is changing *tensor_a* to *tensor_a.to(device)* to move a tensor to a specific device. **Redundant API Method** denotes an API misuse caused by an unneeded API call. For example, in earlier versions of PyTorch (versions 0.4 and earlier), *cpu()* was used to convert a tensor into a CPU tensor. However, in a later version of PyTorch, *cpu()* is no longer required. **API Method Replacement** involves replacing one API call with another. For instance, in the PyTorch library, a mask tensor requires a boolean type. Thus, changing '.int()' to '.bool()' in 'tensor_a.int()' ensures that the tensor has a strictly boolean type.

> **Finding 1:** The most common API element of API misuse is the *API Method* (455, 51.06%). The most common violation of API misuse is *Missing* (244,27.38%). The most common combination of violation and element is the *Redundant API Method* (138,61%) and the least common combination is *API Condition Replacement* (17,8%).

4.2 Taxonomy of API Misuse Root Cause

We conducted an investigation into the underlying root cause behind API misuses, categorizing them into five distinct types: *Algorithm Error*, *Deprecation Management Error*, *Data Conversion Error*, *Device Management Error*, and *Null Reference Error*. Table 3 shows the API misuse distribution of each root cause category. In general, *Device Management Error* with 337 instances is the most frequent root cause. On the other hand, *State Handling Error* and *Argument Error* are extremely rare in root causes.

Algorithm Error represents the math-related API misuses, typically involving errors such as division by zero or incorrect calculations. this type of error rarely exists in traditional libraries but

Table 1: DL API misuse examples

Type		Examples	
Element	**Violation**	**Misuse**	**Fix**
Method	Missing	tensor_a	tensor_a.to(device)
	Redundant	embeds.detach().cpu().numpy()	embeds.detach().numpy()
	Replacement	tensor_a.int()	tensor_a.bool()
	Outdated	tf.scalar_summary('learning_rate', lr)	tf.summary.scalar('learning_rate', lr)
Parameter	Missing	torch.tensor(data_arg)	torch.tensor(data_arg, device=device_arg)
	Redundant	torch.zeros(arg_a, requires_grad=True)	torch.zeros(arg_a)
	Replacement	DenseLayer(arg_a,arg_b, act=tf.identity)	DenseLayer(arg_a,arg_b, act=None)
	Outdated	torch.chunk(inputs, arg_a, dim=1)	torch.chunk(self.inputs, arg_a, dim=1)
Condition	Missing	dtype = dtype_a	if isinstance(dtype, object): dtype = dtype_a
	Redundant	if self.flatten: x = x.flatten(1)	x = self.flatten(x)
	Replacement	if version < version_a:	if version < version_b:
	Outdated	if type == type_a:	if type == type_a or dtype == type_a

Table 2: Distribution of DL API misuse types

	Missing	**Redundant**	**Replacement**	**Outdated**
API Method	113 (46%)	138 (61%)	130 (62%)	74 (34%)
API Parameter	88 (36%)	56 (25%)	60 (29%)	115 (52%)
API Condition	43 (17%)	29 (17%)	17 (8%)	28 (13%)
Total	244	223	207	217

Table 3: Distribution of DL API misuse root causes

Root Cause Category	**TensorFlow**	**PyTorch**	**Total**
Data Conversion Error	72	174	246
Device Management Error	68	269	**337**
Algorithm Error	20	68	88
Deprecation Management Error	76	101	177
Null Reference Error	13	20	33
Other	2	8	10

commonly exists in deep learning libraries due to the computation-intensive nature of deep learning libraries. For example, an API call missing the *eps* value as a parameter can result in a division-by-zero problem. By including the *eps* value (e.g., *eps = config.eps*) in the *nn.LayerNorm(dim)* API call, a small non-zero number is introduced to prevent division by zero. Identifying and addressing API misuse can be challenging, as it requires knowledgeable and experienced deep learning developers, which is often rare in the market.

Deprecation Management Error involves API misuses due to deprecated APIs or refactoring. API calls must be updated to accommodate the refactoring and version change, ensuring their continued usability. For instance, if one Script uses the *torch.nn.functional.softmax()* API and the other Script uses the *nn.functional-.softmax()* API, the API in another Script should be updated to match the one in the first Script. This standardizes API references and prevents potential program defects. Identifying API deprecation is one of the simplest forms of API misuse detection, as developers adhere to best practices by programming warning messages when deprecated APIs are used in the latest versions. Deprecation-related API misuse is also relatively easy to address, as modern integrated development environments (IDEs) support automated reference updates after refactoring. In fact, we observed that a considerable number of API misuse fixes are generated by code linting software. Although deprecation management may appear straightforward, several specific cases, such as condition checks, status updates, and parameter updates, can still be challenging.

Device Management Error pertains to the API misuse related to hardware and resource utilization. This category is particularly relevant to deep learning APIs due to their distinct hardware utilization characteristics. Unlike traditional software libraries that assume CPU-only hardware and local execution, machine learning tasks are computationally intensive and primarily utilize GPUs or GPU clusters instead of CPUs. Therefore, machine learning libraries commonly offer GPU support and distributed computing capabilities. The assumption that the CPU is the sole hardware resource is no longer valid for machine learning libraries, leading to API misuses caused by incorrect hardware or resource configuration, as well as flawed assumptions about the hardware environment. An example of a common API misuse is the omission of the *device* parameter in an API call. In machine learning tasks, data objects must be stored in the appropriate hardware cache for accurate execution. Failing to provide the *device* parameter may result in incorrect hardware assignment of the data object, leading to program crashes.

Data Conversion Error relates to the incorrect shape or type of API input or output. Type problems are well-known issues in Python and other dynamic programming languages. In Python, variables lack specific types upon creation and may change types during runtime. While this design simplifies syntax to a great extent, it also introduces significant challenges. API calls often assume specific types for their variables, leading to failures when the parameter value provided to the API call does not match the assumptions. Developers already consider this problem when designing machine learning APIs and provide parameters that explicitly specify the type. For example, including the parameter $dtype = dtype$ in $tf.ones\_like()$ explicitly sets the return type of $tf.ones\_like()$ to $dtype$. On the other hand, Shape mismatch occurs often when passing variables to APIs. Executing deep learning models requires

Table 4: Distribution of DL API misuse symptoms

Symptom Category	TensorFlow	PyTorch	Total
Program Crash	97	226	**323**
Unexpected Output	57	153	210
Low Efficiency	63	218	281
Return Warning	30	34	64
Others	4	9	13

strict shape match when passing a tensor between layers. Each layer has its own constraint on the input tensor. Developers need to make necessary transformations to the tensor to make it compatible with the receiver. For instance, certain models require tensors to be flattened before passing them to the next layer. This problem can be resolved by adding the $.flatten(Tensor)$ API call. Shape mismatch is a unique type of API misuse in deep learning libraries because these libraries heavily rely on tensor computations.

Identifying data conversion API misuse can be challenging since the program might not immediately raise an error but rather produce incorrect results due to the incorrect calculation introduced by the API misuse. Detecting such API misuses requires a lot of experience in both deep learning libraries and algorithms. Given the absence of immediate errors, developers must manually review the code line by line to identify API misuses. While existing work has addressed tensor shape-related bug repair using static analysis approaches, there is still room for improvement in terms of efficiency, coverage, and ease of use.

Null Reference Error represents the null pointer exceptions-related API misuse. Null pointer exceptions are classic errors in computer programming. Thanks to the linting feature of modern IDEs, most API misuses are identified and rectified before merging the code into the project's codebase. Therefore, instances of such API misuse are rarely observed.

Others includes cases that may not belong to any of the predefined categories. One such example is argument error, which involves API misuses where necessary API arguments are missing. Instances of this type are infrequent since APIs typically require all necessary parameters, and omitting them would result in error messages.

> **Finding 2:** The most common root cause of API misuse in PyTorch is *Device Management Error* (269, 30.19%), while in TensorFlow, it is *Deprecation Management Error* (76, 8.52%). On the other hand, *Null Reference Error* shows the lowest number of API misuses in both libraries (13 in TensorFlow and 20 in PyTorch). Notably, *Device Management Error* (337), *Data Conversion Error* (246), and *Deprecation Management Error* (177) are the top three most common types of API misuse in both TensorFlow and PyTorch.

4.3 Taxonomy of API Misuse Symptom

In the sections below, we classified DL API misuse instances based on the symptoms they exhibited. The symptoms of API misuse were classified into four distinct categories, i.e., *Program Crash*, *Unexpected Output*, *Low Efficiency*, *Return Warning*, and others. Table 4 displays the instance count of each symptom. In the subsequent sections, we will explain each category in detail.

Program Crash represents a category of API misuse that is relatively easy to identify. It is the most common symptom of API misuse. Instances falling under this category result in immediate program failures or crashes. For example, a previously functional program may fail to work properly after a version update or code refactoring due to inappropriate handling of the refactored API. The program crashes as a direct consequence of the failure to update the changes introduced in the API. Timely detection and rectification of such issues are vital to maintaining the stability and reliability of software systems that rely on the API. This category is the easiest to identify because it would crash the program immediately and throw an error with the line number.

Unexpected Output includes instances where developers deviated from the prescribed usage of the API, thereby employing an unexpected way of using API. Consequently, the output obtained differs from the expected result. This category has the most complex root causes because it does not closely relate to any specific root cause. Any root causes may result in an unexpected output as long as it does not throw an error. This type of symptom is hard to identify and debug since such a program does not throw an error but instead produces an incorrect result.

Low Efficiency refers to the slow execution of an API. It is closely related to device management errors because deep learning APIs can be accelerated by GPU devices. Failing to configure the device properly may very likely result in slow execution speed. Identifying errors related to low efficiency in API usage can be particularly challenging. Unlike other categories, low efficiency does not manifest as a warning or incorrect error. Instead, it produces a negative impact on the performance of the program, resulting in decreased efficiency or slower execution. Inexperienced developers may not notice the error because they may not have enough experience to have a proper expectation of the performance[2]. Such issues often require careful analysis of performance bottlenecks to pinpoint and resolve the underlying problems. In the deep learning industry, identifying these inefficiencies and optimizing the code is crucial to ensure the overall performance and responsiveness of the application or system utilizing the API.

Return Warning refers to instances where the API usage returns warnings instead of errors. Return warnings typically occur when developers use deprecated APIs in a newer version of the software. The impact of such warnings on the program's functionality depends on the actions taken by the library developers. Some developers provide backward compatibility APIs, allowing the continued use of deprecated functionalities. However, in some cases, library developers discontinue support for deprecated APIs in newer versions. As a result, developers need to heed these warnings seriously and take appropriate actions, such as updating their code to utilize alternative APIs or adopting compatible replacements, to ensure the smooth operation and maintainability of their software.

Others refers to cases where the information we have collected may not be sufficient to draw a definitive conclusion regarding a specific symptom. In such cases, we categorize these instances under the label of 'Others'.

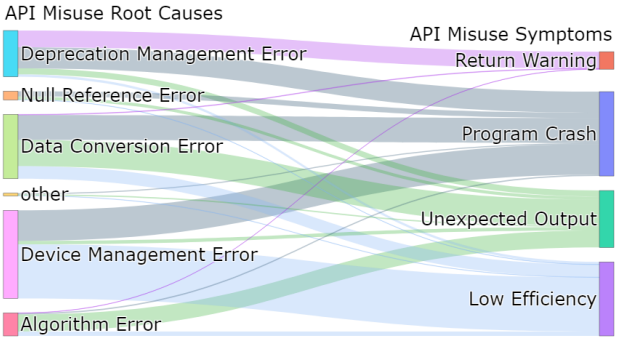

Figure 2: Mapping between root causes and symptoms

We also created a Sankey graph illustrating the relationship between root causes and symptoms to interpret the outcomes of API misuses, as shown in Figure 2. The graph clearly demonstrates a correlation between the root cause and the resulting symptom. For example, the majority of low-efficiency symptoms stem from device management errors, while nearly all return warning symptoms are caused by deprecation management errors.

Finding 3: The most common symptom of DL API misuse in PyTorch and TensorFlow is *Program Crash* (226 for PyTorch and 97 for TensorFlow). On the other hand, *Return Warning* shows the lowest number of API misuses in both libraries (30 in TensorFlow and 34 in PyTorch).

5 PERFORMANCE OF SOTA ON DL API MISUSE DETECTION

To assess the effectiveness of the existing state-of-the-art API misuse detector, i.e., *TADAF* [7], in detecting DL API misuses, we run the tool on our dataset that contains 891 DL API misuses.

Please note that the replication package of *TADAF* is not publicly available. Despite our efforts to contact the author for the latest version of the replication package, we did not receive a response. Consequently, we proceeded with our replication of *TADAF* to the best of our effort. We rigorously followed the process of *TADAF* as described in the paper to replicate the tool based on their description. Specifically, *TADAF* utilizes a keyword-matching mechanism based on the 11 API misuse patterns to detect API misuses.

To ensure the accuracy of our replication, we run our replication on the same dataset used in the experiments for evaluating *TADAF*. Our replication reports the same set of bugs as *TADAF* in its experimental projects, which confirms the correctness of our replication.

For the evaluation of *TADAF*, we applied our replication to our DL API misuse dataset, which contains 891 confirmed API misuse instances. The result shows that *TADAF* only detected 3 out of 891 API misuse instances. *TADAF*'s low performance can be attributed to its limited API misuse patterns used to find API misuses, while mining patterns demand non-trivial manual effort from expert developers to identify, summarize, and validate each misuse pattern.

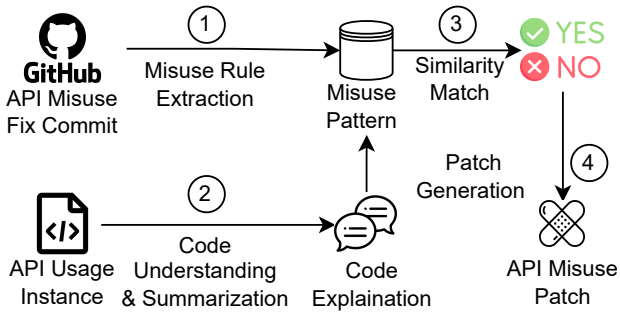

Figure 3: Overview of our LLM-based DL API misuse detector

6 *LLMAPIDET*: AN LLM-BASED DL API MISUSE DETECTOR

As we show in Section 4, DL API misuse primarily involves issues related to data and device usage, such as returning incorrect float types or accidentally running GPU tasks on the CPU. These issues are beyond the detection scope of most of the existing API misuse detection tools, which rely on the matching of errors or warnings to trigger the detection process. Detecting such misuse often relies on manually created API rules by experienced experts, which can be both costly and impractical in terms of scalability.

Inspired by the recent great progress of LLMs (Large Language Models) in code understanding and various source code-related tasks [9, 36, 53], we propose *LLMAPIDet*, the first LLM-based DL API misuse detector for automated misuse rule extraction, API misuse detection, and patch generation. Note that we base our study on ChatGPT [32] as it has shown great potential in software engineering, e.g., code generation [9], bug repair [43], and program understanding [52], among comparable LLMs.

6.1 Approach

The overview of our proposed tool is illustrated in Figure 3, comprising four steps.

In step ①, we construct a knowledge base of misuse rules based on the 891 confirmed API misuses identified during the empirical analysis. We extract API misuse rules using ChatGPT to comprehend source code and summarize the rules. Table 5 shows the Prompt template for misuse rules extraction. In the prompt, we describe the task and provide two manually created examples created manually, including the code before the fix (i.e., code removed), code after the fix (i.e., code added), and the manually defined misuse rule. We then follow the same pattern and extract the 'code removed' and 'code added' information from one of the code changes to extract the rules. For each of the 891 confirmed API misuses, we apply ChatGPT using the same examples to generate the corresponding output for all the 891 API misuses. For simplicity of display, we omit the actual example text and only show the template in Table 5.

In step ②, given a new API usage instance, we utilize ChatGPT to generate a code explanation. Table 6 shows the template for generating code explanations. The prompt is composed of an instruction that prompts ChatGPT to describe a given piece of code snippet, along with a 'code_snippet' variable. For each ChatGPT

Table 5: Prompt template for extracting misuse rules.

"**prompt**": Please identify the rules for fixing the API method problem in the following code change.
Example One: {code_removed_one} {code_added_one} {misuse_rule_one}
Example Two: {code_removed_two} {code_added_two} {misuse_rule_two}
Question: {code_removed}{code_added}

"**output**": {misuse_rule}

Table 6: Prompt template for generating code explanation

"**prompt**": Please describe what the following code snippet does in two sentences:
Code snippet: {code_snippet}

"**output**": {Answer}

API call, we fill the 'code_snippet' variable with the API usage instance along with its context. We set the context length similar to existing GitHub historical commit studies [29, 38, 46], which is 4 lines both above and below the API usage instance.

In step ③, we let ChatGPT determine the applicability of any of the misuse rules to the given API usage instance. If an API usage instance satisfies the conditions defined by the misuse rules, we consider it as a detected API misuse. The prompt for ChatGPT in this step is constructed by concatenating the API usage instance and the list of misuse rules filtered by the API usage method name. To identify potential misuse rules that best match the given instance, we re-rank the misuse rule list by cosine similarity between the code explanation obtained in step ② and each misuse rule in our knowledge base. The reason is that we observed suboptimal cosine similarity between source code embeddings and misuse rule embeddings due to the representation gap between source code and plain English text. To address this issue and achieve better alignment, we replaced the source code with its corresponding English description, resulting in an enhanced cosine similarity calculation. After the re-ranking, the top 4 misuse rules are then selected as candidates. We set the number of rules to 4 to ensure that the total prompt length remains within the token limit of ChatGPT. We feed this composed prompt into the ChatGPT and query whether any of the identified misuse rules can be applied to the provided example. This step results in a response of either 'YES' or 'NO', indicating the presence or absence of API misuse in the code. Table 7 shows the template for detecting API misuses in this step.

In step ④, for each instance identified as an API misuse from ③, we guide ChatGPT through a step-by-step reasoning process. In the prompt template, we instruct ChatGPT to generate the thinking steps and then create a potential patch based on the corresponding misuse rule. The thinking steps and patch are generated following a predefined output format. The prompt template for generating the patch is shown in Table 8.

Table 7: Prompt template for API misuse detection

"**prompt**": Please read the following code snippet and API misuse rules. Then, answer whether the misuse rules can be applied to the code snippet. If the pattern can be applied, answer "Yes"; if not, answer "No".
Code snippet: {code_snippet}
Misuse rules: {misuse_rules}

"**output**": {Decision}

Table 8: Prompt template for patch generation

"**prompt**":
Please read the following code snippet and misuse rules. Then, think step by step and generate a patch for the code snippet. Please ignore any indentation problems in the code snippet. Fixing indentation is not the goal of this task. If the pattern can be applied, generate the patch.
Code snippet: {code_snippet}
Misuse rules: {misuse_rules}

"**output**": {Think steps}{Patch}

Table 9: Performance of *LLMAPIDet* in detecting API misuse

	Exp. One		Exp. Two		
	Detected	Recall	Detected	Verified	Precision
LLMAPIDet	48	16.49%	368	119	32.33%
TADAF	0	-	41	5	12.2%

Table 10: Performance of *LLMAPIDet* in patch generation

		Generated	Verified	Accuracy
Exp.One	*LLMAPIDet*	48	10	22.83%
	TADAF	0	-	-
Exp.Two	*LLMAPIDet*	119	46	38.65%
	TADAF	5	5	100%

6.2 Experiment Setting

We designed two experiments to evaluate the performance of *LLMAPIDet* and the benchmark approach TADAF. Due to the quote limitation of ChatGPT 4.0 at the time we conducted the experiments, we used GPT 3.5 for the experiments. **In the first experiment**, we built *LLMAPIDet* using 600 randomly selected instances from the 891 confirmed API misuses and reserved the remaining 291 API misuses as the testing dataset (i.e., Experiment One). We tested both tools on Experiment One to detect API misuses and compare their recall rates.

Additionally, we designed the second experiment to address concerns about memorization in LLM-based approaches [3, 21], where correct predictions are based on publicly available online answers. To mitigate the memorization issue, **in the second experiment**, we collected 4,359 API usage instances from the latest version of 10 open-source GitHub projects (i.e., Experiment Two) other than the 200 projects used in the empirical study (see Section 4). All of these projects are actively maintained, as their latest updates range from July 25, 2023, to July 29, 2023, which is after the timestamp of

the live version of ChatGPT. The number of API usage instances in these projects ranges from 78 to 2,131, with an average count of 436 instances. The dataset covers a total of 204 unique APIs. We demonstrate how much API misuse can be detected in Experiment Two and manually verify its validity using both tools.

6.3 Performance of *LLMAPIDet*

The results on the detection of DL API misuses are presented in Table 9. As we can see, in experiment one, *LLMAPIDet* successfully detected 48 API misuses, whereas TADAF could not detect any instances. The recall values are 16.49% and 0% for *LLMAPIDet* and TADAF, respectively. In the experiment two dataset, *LLMAPIDet* identified 368 instances of API misuse, while TADAF only found 41 misuse instances. We manually examined each instance and confirmed that 119 API misuses reported by *LLMAPIDet* and 5 misuses detected by TADAF are true bugs. The precision of *LLMAPIDet* is 32.33%, while the precision of TADAF is 12.2%.

We show the results of patch generation in Table 10. In the first experiment, *LLMAPIDet* generated patches for 48 API misuses, while TADAF did not generate any patches. The accuracy of *LLMAPIDet* is 22.83%. In the second experiment, *LLMAPIDet* generated patches for 119 API misuses, while TADAF generated patches for 5 instances. We manually investigated each patch and confirmed that 46 patches for *LLMAPIDet* and 5 patches for TADAF are correct. The accuracy of *LLMAPIDet* is 38.65% while the precision of TADAF is 100%. To validate the findings, we reported the API misuse instances to the developers of these projects by creating GitHub issues.

The following example demonstrates a successfully detected API misuse. In line 3, the input tensor should be assigned to a device to avoid potential device management errors when the args.mode is not equal to 'gpu' in line 5. *LLMAPIDet* detects this API misuse and suggests a fix (i.e., assigning the tensor to a device) in line 9, which has been verified as correct.

```
1# LLMAPIDet detects API Misuse in the code below:
2if args.bbox_init == 'two':
3    input = transform(img_step2).unsqueeze(0)
4    with torch.no_grad():
5        if args.mode == 'gpu':
6            input = input.cuda()
7# Patch:
8if args.bbox_init == 'two':
9    input = transform(img_step2).unsqueeze(0).to(device)
10   with torch.no_grad():
11       if args.mode == 'gpu':
12           input = input.cuda()
```

The following example illustrates a case in which *LLMAPIDet* fails to detect API misuse. The problem arises from the presence of incomplete code in the last line of the context, leading ChatGPT to generate code to complete the line instead of primarily focusing on examining and editing the existing code for misuse detection.

```
# LLMAPIDet fails to detect API Misuse in the code below:
with tf.name_scope('distribution_errors'):
    ...other code...
    tf.compat.v2.summary.scalar(
# Patch:
with tf.name_scope('distribution_errors'):
    ...other code...
    tf.compat.v2.summary.scalar('mean',
         tf.reduce_mean(distribution_errors),
         step=self.train_step_counter)
```

One way to address the problem above is to gather more context for the code snippet. However, this can be challenging as it is difficult to determine the appropriate amount of context to provide. If too much context is given, irrelevant information may be included, resulting in noise. Although existing methods like code slicing can help alleviate the issue, achieving high-quality code slicing in Python is not a straightforward task.

7 DISCUSSION

7.1 Difference between DL API misuses and API misuses of traditional software

Existing studies on API misuse primarily focus on general software with programming-related categories, such as synchronization, control flow, and state handling [5, 6, 44]. DL API misuses exhibit distinct characteristics such as tensor-related API misuse and resource-related API misuse. These differences arise from the unique design, problem-solving paradigm, knowledge representation, and computationally intensive nature of DL APIs. The following sections delve into a detailed description of these contrasting characteristics.

Differences in data type-related API misuse: Our study confirms that DL libraries also experience type-related issues. However, we identified a few differences compared to traditional libraries. For instance, when comparing the return value of a torch tensor to a number, developers may need to cast the number as a tensor using *torch.tensor()* to match the API's return type; otherwise, the comparison may fail. However, value comparison in traditional Python programs often allows comparison between different data types without specifically casting one type to another. Additionally, types are sometimes intentionally adjusted to accommodate business logic. For example, developers may cast float64 numbers to float32 to optimize computation resources and expedite calculations at the cost of slight precision loss.

Shape and algorithm-related API misuse: API misuses can arise from incorrect shape assumptions or erroneous algorithms. These types of API misuse are relatively uncommon in traditional API misuse as they typically do not involve tensor computations. Identifying and locating these types of errors can be challenging since they often do not cause immediate program crashes. For example, if an API expects a 3x2 shape input tensor (e.g., [1,2],[3,4],[5,6]), but a 2x3 shape tensor is provided (e.g., [1,2,3],[4,5,6]), the appropriate API misuse fix would be to invoke *Tensor.transpose()* to transpose the shape. However, developers without enough mathematical knowledge might instead use *Tensor.reshape()* to reshape the tensor to an incorrect shape (e.g., [1,4],[2,5],[3,6]), resulting in incorrect data. While this API misuse may not trigger an immediate error, it eventually leads to incorrect results and difficulties in tracing the root cause. Given their lack of transparency and intricacy, we

consider these bugs the most challenging to handle. Prior research has also highlighted similar issues in machine learning defects[35].

Resource and hardware-related API misuse: We observed a significant number of API misuses related to resource management (37.8%). Given the computational intensity of machine learning tasks, GPU acceleration, and distributed computing are commonly employed. As a result, API execution shows inconsistencies depending on resource availability and environment configuration. For example, tensors involved in the same calculation process must reside on the same hardware device; otherwise, the program crashes when it fails to locate the tensor on the assumed device. Other resource-related misuses include setting the GPU as the default option on CPU-only devices and encountering missing GPU support libraries when switching between different GPU models.

Versioning-related API misuses: Machine learning libraries experience frequent refactoring and version updates compared to other libraries. While traditional software design builds on top of established computer science theories and software development best practices, the rapidly evolving nature of machine learning theory introduces new concepts regularly. The rapidly evolving nature of machine learning libraries introduces many refactoring-related API misuses. Our observations revealed three levels of refactoring-related API misuse fixes in PyTorch and TensorFlow, ranging from easy to challenging. The first level involves class-level refactoring, where the class of the API changes while leaving the API method unchanged. This is a relatively straightforward update that can be automated. The second level involves refactoring both class and method names, requiring manual effort to create a mapping between the old and new APIs. Despite the additional complexity, this type of update remains feasible due to the guaranteed one-to-one mapping. The third and most challenging level involves library redesign, where numerous APIs are removed, functionalities are split into multiple new APIs, or certain functionalities are discontinued. In such cases, developers may need to learn the new library and rewrite their code as an alternative API after refactoring is not guaranteed.

7.2 Guidelines for avoiding DL API misuse

Variable type enforcement: Based on our analysis, we suggest incorporating type annotations and type assertions into APIs and API parameters to mitigate type errors. By explicitly specifying the expected types, developers can catch type-related issues early in the development process. Type annotations also facilitate more transparent type management in Python, as they serve as documentation for both humans and automated tools. Additionally, we recommend integrating code lint checkers like PEP8[48] into the development pipeline to enforce consistent coding conventions and improve code quality.

Resource availability checking: To ensure efficient resource and environmental management in machine learning applications, we suggest implementing a global configuration file that is consulted before executing any business logic. Currently, machine learning library APIs often require explicit device specifications, which are handled individually by each API. By centralizing device management in a dedicated layer before the business logic layer, we can ensure consistent resource and environment assumptions for all resource-sensitive machine learning APIs. This approach simplifies the process of managing resources such as GPUs and facilitates seamless execution across different hardware configurations. Applying techniques like environment detection and configuration generation further enhances the reliability and reproducibility of resource-sensitive machine learning APIs and workflows.

API deprecation handling: Although developers already make efforts to handle API deprecation by issuing warnings and maintaining backward compatibility, unforeseeable risks can still arise. As deep learning theory and practices evolve, popular architectures may become suboptimal, requiring a redesign of the library structure to meet new technical requirements. Such redesigns and refactorings are often unavoidable. In some cases, the redesigned library may not provide alternative APIs, leaving developers to face the challenge of rewriting their code using the new version. This task can be frustrating, and developers may struggle to find comprehensive guidelines and solutions for version migration. To enhance the robustness of deprecation support, it is crucial to provide not only thorough documentation but also automated API mapping mechanisms. These mechanisms can aid in transitioning existing codebases to new versions by suggesting equivalent replacement APIs and minimizing the effort required for code migration. By improving archive code support, developers can more effectively adapt to evolving libraries while maintaining code compatibility and minimizing disruptions to their projects.

8 THREAT TO VALIDITY

Construct Validity: In the empirical study, we focused exclusively on Python deep learning libraries, excluding deep learning libraries in R or Java due to differences in programming language characteristics. To determine the number and names of categories, we conducted a preliminary analysis based on 200 randomly selected examples to determine the appropriate granularity and category names. We chose *TADAF* as our benchmark tool due to its high relevance to our study. Unlike other tools, such as *Ariadne* [13], which serve as general bug detectors for DL libraries, *TADAF* specifically addresses DL API misuse detection. This specialization makes *TADAF* the most suitable choice for our research. Although *TADAF* is designed for TensorFlow, some of its rules are still suitable for PyTorch.

Internal Validity: The manual labeling process required significant effort. We ensured accuracy by having multiple rounds of discussion to establish a consensus on the labeling standards for 200 randomly selected examples. The rest of the labeling work was then assigned to several experienced Ph.D. students. However, it is important to recognize that this approach may impact the overall accuracy of the classification.

External Validity: During the evaluation of the detector, the non-deterministic nature of LLMs introduced variability in the output of the LLM model. Consequently, these differences could propagate through subsequent interactions with the LLM in the detector workflow, leading to different results. Employing a fully open-sourced model such as LLaMA [45] with a frozen seed may enhance reproducibility.

9 CONCLUSION

This paper presented the first extensive study on API misuse within two major DL libraries, i.e., TensorFlow and PyTorch. We identified 891 API misuses from the 200 most starred projects on GitHub built with the two libraries and provided insights into their categories, root causes, and symptoms. To address the challenges in detecting DL API misuses, we introduced *LLMAPIDet*, a novel LLM-based tool for DL API misuse detection and patch generation. Our evaluation demonstrated the effectiveness of *LLMAPIDet*. The contribution of this work is, to the best of our knowledge, this is the first large-scale analysis to demystify and detect DL API misuses in PyTorch and TensorFlow. Additionally, we created a benchmark with 891 instances of DL API misuse and released the dataset and source code for replication. In future work, we aim to expand *LLMAPIDet*'s capabilities and explore more automation strategies for DL API misuse detection.

10 ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback which helped improve this paper. This work is partially supported by Natural Sciences and Engineering Research Council of Canada (NSERC), the Basic Research Program of ISCAS Grant No. ISCAS-JCZD-202304, and Youth Innovation Promotion Association Chinese Academy of Sciences.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. 2021. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 802–814.

[3] Rachith Aiyappa, Jisun An, Haewoon Kwak, and Yong-Yeol Ahn. 2023. Can we trust the evaluation on ChatGPT? *arXiv preprint arXiv:2303.12767* (2023).

[4] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* (2016), arXiv–1605.

[5] Sven Amann, Sarah Nadi, Hoan A Nguyen, Tien N Nguyen, and Mira Mezini. 2016. MUBench: A benchmark for API-misuse detectors. In *Proceedings of the 13th international conference on mining software repositories*. 464–467.

[6] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1170–1188.

[7] Wilson Baker, Michael O'Connor, Seyed Reza Shahamiri, and Valerio Terragni. 2022. Detect, fix, and verify TensorFlow API misuses. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 925–929.

[8] Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. 2018. The rise of deep learning in drug discovery. *Drug discovery today* 23, 6 (2018), 1241–1250.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[10] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, Rory Mitchell, Ignacio Cano, Tianyi Zhou, et al. 2015. Xgboost: extreme gradient boosting. *R package version 0.4-2* 1, 4 (2015), 1–4.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[12] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of ChatGPT Model for Vulnerability Detection. *arXiv preprint arXiv:2304.07232* (2023).

[13] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: analysis for machine learning programs. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 1–10.

[14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324.

[15] Yunhe Feng, Sreecharan Vanam, Manasa Cherukupally, Weijian Zheng, Meikang Qiu, and Haihua Chen. 2023. Investigating Code Generation Performance of ChatGPT with Crowdsourcing Social Data. In *Proceedings of the 47th IEEE Computer Software and Applications Conference*. 1–10.

[16] Barney Glaser and Anselm Strauss. 2017. *Discovery of grounded theory: Strategies for qualitative research*. Routledge.

[17] Guosheng Hu, Yongxin Yang, Dong Yi, Josef Kittler, William Christmas, Stan Z Li, and Timothy Hospedales. 2015. When face recognition meets with deep learning: an evaluation of convolutional neural networks for face recognition. In *Proceedings of the IEEE international conference on computer vision workshops*. 142–150.

[18] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1110–1121.

[19] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. 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*. 510–520.

[20] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. 675–678.

[21] Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences* 103 (2023), 102274.

[22] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).

[23] Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How Secure is Code Generated by ChatGPT? *arXiv preprint arXiv:2304.09655* (2023).

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[25] Xia Li, Jiajun Jiang, Samuel Benton, Yingfei Xiong, and Lingming Zhang. 2021. A Large-scale Study on API Misuses in the Wild. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 241–252.

[26] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 306–315.

[27] Christian Lindig. 2015. Mining patterns and violations using concept analysis. In *The Art and Science of Analyzing Software Data*. Elsevier, 17–38.

[28] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210* (2023).

[29] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 70–79.

[30] Madhav Nair, Rajat Sadhukhan, and Debdeep Mukhopadhyay. 2023. Generating secure hardware using chatgpt resistant to cwes. *Cryptology ePrint Archive* (2023).

[31] Daniel W Otter, Julian R Medina, and Jugal K Kalita. 2020. A survey of the usages of deep learning for natural language processing. *IEEE transactions on neural networks and learning systems* 32, 2 (2020), 604–624.

[32] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[34] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[35] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1027–1038.

[36] Julian Aron Prenner and Romain Robbes. 2021. Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs. *arXiv preprint arXiv:2111.03922* (2021).

[37] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Path-sensitive inference of function precedence protocols. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 240–250.

[38] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 155–165.

[39] Muhammad Imran Razzak, Saeeda Naz, and Ahmad Zaib. 2018. Deep learning for medical image processing: Overview, challenges and the future. *Classification in BioApps: Automation of Decision Making* (2018), 323–350.

[40] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2135–2135.

[41] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*. PMLR, 31693–31715.

[42] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653* (2023).

[43] Nigar M Shafiq Surameery and Mohammed Y Shakor. 2023. Use chat gpt to solve programming bugs. *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290* 3, 01 (2023), 17–22.

[44] Amann Sven, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2019. Investigating next steps in static API-misuse detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 265–275.

[45] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[46] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering*. 483–494.

[47] Simon Ulbrich, Till Menzel, Andreas Reschka, Fabian Schuldt, and Markus Maurer. 2015. Defining and substantiating the terms scene, situation, and scenario for automated driving. In *2015 IEEE 18th international conference on intelligent transportation systems*. IEEE, 982–988.

[48] Guido van Rossum, Barry Warsaw, and Nick Coghlan. 2001. *Style Guide for Python Code*. PEP 8. https://www.python.org/dev/peps/pep-0008/

[49] Chengcheng Wan, Shicheng Liu, Henry Hoffmann, Michael Maire, and Shan Lu. 2021. Are machine learning cloud apis used correctly?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 125–137.

[50] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software Testing with Large Language Model: Survey, Landscape, and Vision. arXiv:2307.07221 [cs.SE]

[51] Andrzej Wasylkowski and Andreas Zeller. 2011. Mining temporal specifications from object usage. *Automated Software Engineering* 18 (2011), 263–292.

[52] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839* (2023).

[53] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764* (2023).

[54] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *arXiv preprint arXiv:2305.04207* (2023).

[55] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples on an online q&a forum reliable? a study of api misuse on stack overflow. In *Proceedings of the 40th international conference on software engineering*. 886–896.

[56] Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. 2023. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*. PMLR, 41832–41846.

[57] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 129–140.