How and Why LLMs Use Deprecated APIs in Code Completion? An Empirical Study

Chong Wang*, Kaifeng Huang†, Jian Zhang*, Yebo Feng*, Lyuye Zhang*, Yang Liu*, and Xin Peng‡
*School of Computer Science and Engineering, Nanyang Technological University, Singapore
{chong.wang, jian_zhang, yebo.feng}@ntu.edu.sg, zh0004ye@e.ntu.edu.sg, yangliu@ntu.edu.sg
†School of Software Engineering, Tongji University, China
kaifengh@tongji.edu.cn
‡School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China
pengxin@fudan.edu.cn

***Abstract*—Large language models (LLMs), pre-trained or fine-tuned on large code corpora, have shown effectiveness in generating code completions. However, in LLM-based code completion, LLMs may struggle to use correct and up-to-date Application Programming Interfaces (APIs) due to the rapid and continuous evolution of libraries. While existing studies have highlighted issues with predicting incorrect APIs, the specific problem of deprecated API usage in LLM-based code completion has not been thoroughly investigated.**

To address this gap, we conducted the first evaluation study on deprecated API usage in LLM-based code completion. This study involved seven advanced LLMs, 145 API mappings from eight popular Python libraries, and 28,125 completion prompts. The study results reveal the *status quo* and *root causes* of deprecated API usage in LLM-based code completion from the perspectives of *model*, *prompt*, and *library*. Based on these findings, we propose two lightweight fixing approaches, REPLACEAPI and INSERTPROMPT, which can serve as baseline approaches for future research on mitigating deprecated API usage in LLM-based completion. Additionally, we provide implications for future research on integrating library evolution with LLM-driven software development.

I. INTRODUCTION

Large language models (LLMs) [1, 2, 3, 4, 5, 6] have significantly advanced various aspects of software engineering, including code completion [7, 8], code understanding [9], and program repair [10, 11]. These models, pre-trained or fine-tuned with extensive knowledge of code on large corpora, are effective for tailoring to different downstream tasks. In the realm of code completion, the state-of-the-art has evolved from statistics-based methods [12, 13] to LLM-based techniques [14, 15, 16]. Code completion is a sophisticated task that suggests variables, functions, classes, methods, and even entire code blocks, which depends both on tools' capability and developers' practical needs.

Motivation. To accelerate development, developers heavily rely on third-party libraries, interacting with them through Application Programming Interfaces (APIs). However, this reliance presents a challenge for code completion tools. Third-party libraries constantly evolve to undergo refactorings [17], fix bugs [18], apply security patches [19], or introduce new features. This rapid evolution leads to frequent API changes, with older APIs being deprecated and replaced by newer ones. Deprecated APIs are discouraged to use because they might not work well with newer features or data. These outdated APIs will eventually disappear in future library updates [20]. Taking PyTorch [21], a popular deep learning library for instance, the API *torch.gels()* was deprecated in version 1.2 (August 2019) in favor of *torch.lstsq()*. Then, *torch.lstsq()* was deprecated in version 1.9 (June 2021) in favor of *torch.linalg.lstsq()*. Consequently, newly developed code should avoid using the deprecated *torch.gels()* and *torch.lstsq()*. Therefore, it's crucial for code completion tools to suggest the correct, up-to-dated APIs to developers.

Literature. However, to the best of our knowledge, the capabilities of LLM-based code completion regarding API deprecation is understudied [22, 23]. Although there emerges a substantial number of evaluation on code completion, a body of research focused on assessing the overall accuracy across various benchmarks [9, 24, 25, 26, 27]. Interestingly, Ding *et al.* [28] identified undefined names and unused variables as the most common syntactic errors produced by LLMs in Python code completions. Izadi *et al.* [29] found that incorrect function name predictions were prevalent, accounting for 23% of all token-level errors. Furthermore, Liu *et al.* [30] highlighted the issue of hallucinations in LLM-generated code. Their findings indicate the prevalence and potential risks of using unexpected APIs. Nevertheless, while researchers have noted the prevalence of incorrect function name predictions, they have not investigated this issue in depth. Library APIs, which constitute an important part in predicting external function names, are worth attached importance to.

Study. To address this gap, we conducted a study to examine the issue of deprecated API usage in LLM-based code completion. The study aims to answer the primary research question:

> ***What are the status quo and root causes of deprecated and replacing API usage in LLM-based code completion?***

This question is explored through three detailed aspects:

Model Perspective (RQ1) investigates the status quo and root causes based on the performance of various LLMs;

Prompt Perspective (RQ2) examines the impact of different prompts on the status quo and root causes;
Library Perspective (RQ3) analyzes the status quo and root causes across different libraries.

To address these research questions, we conducted a series of experiments involving various libraries and LLMs. We collected 145 API mappings between deprecated APIs and their replacements from eight popular Python libraries. Based on these mappings, we retrieved 9,022 *outdated* functions and 19,103 *up-to-dated* functions using the deprecated APIs and replacing APIs, respectively. Each outdated or up-to-dated function was transformed into a line-level completion prompt by identifying the deprecated or replacing API and removing the containing and subsequent lines. The located API is referred to as the *reference* API. These prompts were then inputted into seven advanced code LLMs, including CodeLlama [5] and GPT-3.5, to generate completions and analyze the predicted API usages. If the predicted API usage corresponds to either the deprecated or replacement version of the reference API, it is annotated as *plausible*; otherwise, it is annotated as *irrelevant*.

The study results reveal the following findings: **Answer to RQ1:** All seven evaluated LLMs encounter challenges in predicting *plausible* API usages and face issues with deprecated API usages, due to the presence of deprecated API usages during model training and the absence of API deprecation knowledge during model inference. **Answer to RQ2:** For the two categories of prompts derived from outdated and up-to-dated functions, the LLMs' performance in predicting *plausible* and deprecated API usages differs significantly, influenced by the distinct code context characteristics of these prompts. **Answer to RQ3:** Across the eight libraries, the LLMs exhibit significant differences in their use of deprecated APIs, influenced by the characteristics of API deprecations during the evolution of these libraries.

Approach. Based on the study results and findings, we developed two lightweight fixing approaches to mitigate deprecated API usage in LLM-based code completion. Given a completion containing a deprecated API usage, the first approach, named **REPLACEAPI**, directly replaces the deprecated API usage with the replacement and regenerates the remaining parts (*e.g.,* argument list) during the decoding process. The second approach, named **INSERTPROMPT**, inserts an additional replacing prompt after the original prompt to guide the LLMs to use the replacement API and then regenerate the completions. We then evaluate the effectiveness of the proposed approaches in terms of fixing deprecated API usages and the accuracy in predicting line-level completions (**RQ4**). The evaluation results demonstrate that REPLACEAPI effectively addresses deprecated API usages for all evaluated open-source LLMs, achieving fix rates exceeding 85% with acceptable accuracy measured by Edit Similarity and Exact Match compared to ground-truth completions. While INSERTPROMPT does not currently achieve sufficient effectiveness and accuracy in fixing completions containing deprecated API usages, it shows potential for future exploration.

To summarize, this paper makes the following contributions:

- The first study that reveals the status quo and root causes of deprecated API usages from model perspective, prompt perspective, and library perspective. The study involves seven advanced LLMs, 145 API mappings from eight popular Python libraries, and 28,125 prompts derived from 9,022 outdated functions and 19,103 up-to-dated functions.
- Two lightweight fixing approaches, named REPLACEAPI and INSERTPROMPT, which can serve as baseline approaches for future research on mitigating deprecated API usage in LLM-based completion.
- The implications that provide potential research directions on the synergy of library evolution and LLM-driven software development.

II. RELATED WORK

We review the related work with respect to library evolution and LLM-based code completion.

A. Library Evolution

Library evolution involves refactorings [17], bug fixes [18], and new feature introductions. Typically, refactorings can deprecate old APIs and introduce new replacements. Several studies have examined the reasons that developers deprecate APIs and how the clients react to such deprecations [31, 32, 33, 34]. The reasons include improving readability, reducing redundancy, avoiding bad code practices and fixing functional bugs. Deprecated APIs can affect hundreds of clients [35], particularly when clients struggle to keep pace with rapidly evolving software [36]. McDonnell et al. [37] found that only 22% of outdated API usages are eventually upgraded to use replacement APIs. Similarly, Hora et al. [38] found that client developers consumes considerate time to discover and apply replacing APIs, with the majority of systems not reacting at all. When clients do not upgrade their APIs, they silently accumulate technical debt in the form of future API changes when they finally upgrade [39]. To locate the replacing API, existing works leverage change rules written by developers [40], developer recordings [41], similarity matching [42], mining API usage in libraries [43], and in client projects [44]. Henkel and Diwan [41] developed an IDE plugin that allows library developers to record API refactoring actions and client developers to replay them. Godfrey and Zou [45] proposed a semi-automated origin analysis using similarities in name, declaration, complexity metrics, and call dependencies. Wu et al. [46] introduced a hybrid approach combining call dependency and text similarity analysis to identify API change rules. Recently, **(author?)** [47] proposed RepFinder to find replacement APIs for deprecated APIs in library updates from multiple sources.

In this work, we aim to comprehend the statuses and causes of deprecated API usages in LLM-based code completion and provide implications for mitigating the deprecated API usages.

B. LLM-based Code Completion

Code completion is an important functionality in modern IDEs and editors. Historically, researchers have explored statistical models [12, 13]. With the advent in natural language

processing, researchers have embraced deep learning for code completion [7, 8] because they are similar in token-based prediction. To explore the capability of code completion tools driven by large language models (LLMs) [14, 15, 16], numerous evaluations of LLMs have been proposed. Ciniselli et al. [26, 27] conducted a large-scale study exploring the accuracies of state-of-the-art Transformer-based models in supporting code completion at various granularity levels, from single tokens to entire code segments. Zeng et al. [9] found that pre-trained models significantly outperform non-pre-trained state-of-the-art techniques in program understanding tasks. They also reveal that no single pre-trained model dominates across all tasks. Xu et al. [24] evaluated the performance of LLMs on the HumanEval dataset. Ding et al. [28] identified undefined names and unused variables as the most common errors produced by language models in Python code completions. Izadi et al. [29] evaluated the LLMs using real auto-completion usage data across 12 languages. They found that incorrect function name predictions, were prevalent, accounting for 23% of all token-level errors errors. Besides, Liu et al. [25] proposed EvaluPlus, which benchmarks the functional correctness of LLM-synthesized code using test cases. In addition to accuracy concerns, LLM-based approaches face issues such as security vulnerabilities and hallucinations. Sallou et al. [48] explored threats posed by LLMs, including unpredictability in model evolution, data leakage, and reproducibility. Liu et al. [30] categorized the hallucinations brought by LLM-generated code. More notably, Wu et al. [49] introduced VersiCode, the pioneering dataset aimed at evaluating the capability of large language models in generating verifiable code tailored to specific library versions.

The findings on incorrect function predictions partially motivate our study. However, our study focuses on the severity of predicting deprecated API usages in LLM-based code completion.

III. Study Setup

We chose Python, a popular programming language which ranks first among the most popular ones based in the recent year [50]. We targeted eight popular Python libraries. Five of these libraries were used in a previous study on Python API deprecation [19], including Numpy, Pandas, scikit-learn, SciPy, and seaborn. Additionally, we added three popular deep learning libraries, *i.e.,* TensorFlow[1], PyTorch[2], and Transformers[3]. The setup of our study is presented in Figure 1. It includes four steps. The *API Mapping Collection* step gathers mappings between deprecated APIs and their replacements from various libraries. The *Completion Prompt Construction* step involves creating completion prompts by identifying instances of deprecated and replacement API usage in open-source Python repositories. The *LLM-Based Code Completion* step uses various LLMs to generate code completions for these prompts. Finally, the *Completion Result Annotation* step automatically annotates the generated completions and calculates relevant metrics.

[1]https://www.tensorflow.org/
[2]https://pytorch.org/
[3]https://huggingface.co/docs/transformers/index

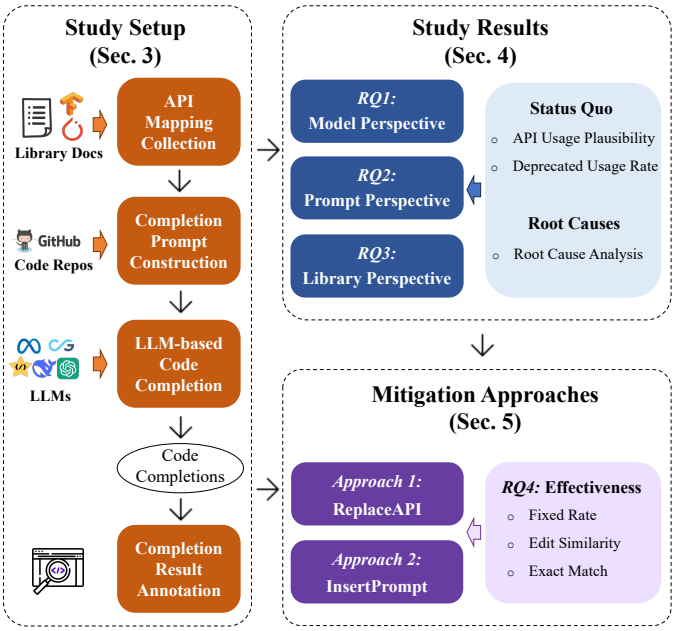

Fig. 1: Overview of Our Study

TABLE I: Statistics of our Collected API Mappings in Eight Python Libraries

Library	Version	# Mappings	# Functions	
			Outdated	Up-to-dated
Numpy	1.26.4	3	567	2,988
Pandas	2.2.2	10	69	69
scikit-learn	1.5.0	18	985	1,197
SciPy	1.13.0	4	245	1,458
seaborn	0.13.2	3	904	1,329
TensorFlow	2.16.1	57	1,491	4,830
PyTorch	2.3.0	21	4,726	6,406
Transformers	4.40.2	29	100	63
Total	–	145	9,022	19,103

A. API Mapping Collection

We identified API mappings (*i.e.,* deprecated APIs and the mapping replacements) from the documentation and change logs from each library following the previous study [19]. Specifically, we reviewed the documentation and change logs of each library and manually look for deprecated API occurrences which indicate the corresponding the mapping replacements. For instance, in the API documentation of PyTorch, version 1.9.0 [51], a deprecation message indicates that *"torch.lstsq() is deprecated in favor of torch.linalg.lstsq() and will be removed in a future PyTorch release."*, where the mapping of the deprecated API to the replacing API is *torch.lstsq* $\rightarrow$ *torch.linalg.lstsq*. For one-to-many mappings (*i.e.,* one deprecated API mapped to multiple replacing APIs), we split them into many one-to-one mappings. In total, the authors obtained 145 API mappings. The statistics of the API mappings are presented Table I.

B. Completion Prompt Construction

We constructed code completion prompts by searching the deprecated API and replacing API usages from open-source Python repositories.

1) Outdated and Up-to-dated Function Location.: We utilized Sourcegraph [52], a widely used code search service. It supports integration with GitHub where we can retrieve Python source files from millions of open-source code repositories. For each deprecated or replacing API, we constructed search queries using both its full qualified name (FQN) (*e.g., torch.lstsq*) and a logical disjunction of its constituent parts (*e.g.,* "torch AND linalg AND lstsq") to ensure comprehensive retrieval. For each retrieved Python source file, we parsed it into an Abstract Syntax Tree (AST) and extracted the containing functions that invoked the deprecated or replacing APIs. Specifically, we located function definition nodes in the AST and traversed its descendants. For each descendant, we checked if it is a function call node and matched the function call to the deprecated or replacing APIs. To correctly match the function call via API FQNs, we performed lightweight object type resolution and API alias resolution, similar to [19].

- **Object Type Resolution**: In the object-oriented programming (OOP) languages, the APIs can be encapsulated into a class as a method. Therefore, determining the FQN of the API invocation need to resolve the corresponding type of the invoking object. For example, the pandas library defines a core class *DataFrame* with a member method *loc()* and the client creates an object of class *DataFrame*, assigns to a variable *dt*, and invokes the method using *dt.loc()*. Typically, it requires resolving the type of *dt*. To that end, we analyzed the `assign` statements to track object definitions, enabling us to determine the class names for objects in function calls and infer the called APIs. For instance, if the object "dt" in the call *dt.loc()* was created in a preceding `assign` statement (*dt = pandas.DataFrame(...)*), we could infer that the corresponding API was *pandas.DataFrame.loc()*.
- **API Alias Resolution**: Developers can alias packages, classes, and functions in Python using the `import-as` feature [53]. This mechanism requires resolving API aliases by analyzing `import` statements. For example, the pandas package is often imported with the alias "pd" via the statement *import pandas as pd*. In this case, *pd.DataFrame.loc()* was resolved to *pandas.DataFrame.loc()*. Additionally, Python provides a `from-import` mechanism allowing developers to use APIs with short names instead of their FQNs. For example, through *from torch.linalg import lstsq*, the API in *torch.linalg* can be directly called via *lstsq()*. These short names were also resolved by analyzing the `import` statements.

After the lightweight object type resolution and API alias resolution, we obtained the corresponding FQN for each function call. We checked whether the corresponding FQNs matched the APIs in the collected API mappings, identifying the first matched API as the *reference* API.

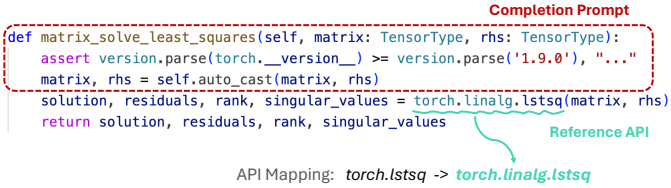

Fig. 2: Illustration of Completion Prompt Construction for An Up-to-dated Function.

We denote the containing function as an *outdated* function if a deprecated API was matched. Meanwhile, we denote it as an *up-to-dated* function if a replacing API was matched. In total, we collected Python 113,660 source files by querying SourceGraph. Among them, we extracted 9,022 outdated and 19,103 up-to-dated functions. The statistics of the outdated and up-to-dated functions are presented in Table I.

2) Incomplete Code Extraction.: In the task of code completion, developers usually have started with a few lines of code and pause in the middle, waiting for LLMs to return the suggested content based on the upward context. Therefore, to evaluate the performance of LLMs in the scenario, we constructed the *line-level code completion prompts*. For each outdated or up-to-dated function, we located the invocation line of the deprecated or replacing APIs, respectively. We collected the preceding lines before the invocation line into our *line-level code completion prompts* for an outdated or up-to-dated function, which is usually incomplete.

Figure 2 represents one of our collected up-to-dated functions. The function invokes a API of PyTorch, *i.e., torch.linalg.lstsq* in the fourth line. The line-level code completion prompt for this function is highlighted in the wine-red dotted rectangle.

After processing all outdated and up-to-dated functions, we obtained two corresponding datasets, denoted as $\mathcal{O}$ and $\mathcal{U}$, respectively. Each sample in $\mathcal{O}$ and $\mathcal{U}$ was formatted as ($pmpt$, $dep \rightarrow rep$), where $pmpt$ is a code completion prompt $pmpt$ and $dep \rightarrow rep$ denotes an API mapping from the deprecated API to the corresponding replacing API.

C. LLM-based Code Completion

We leverage multiple LLMs in the code completion task to observe their performance. The LLMs include both open-source and closed-source models, whose parameter sizes ranges from 350 million to 175 billion. The complete list of the LLMs are presented in Table II.

- **CodeGen-350m, 2b, 6b**: CodeGen [3] is a family of large language models developed by Salesforce specifically for code generation. These models are trained on diverse programming languages and are designed to assist in writing code by providing intelligent code suggestions and completions.
- **DeepSeek-1.3b**: DeepSeek-Coder [6] is designed on top of advanced transformer-based models tailored for code-related applications. This model is trained on a diverse set of coding repositories, which include a variety of

programming languages and coding styles. It combines state-of-the-art machine learning techniques to provide robust code suggestions and completions.

- **StarCoder2-3b**: StarCoder2 [4] is an advanced language model optimized for coding tasks. It leverages large-scale pre-training on code datasets to understand programming languages deeply. StarCoder2 excels in code completion, bug detection, and code translation, making it a valuable tool for developers seeking to enhance their coding efficiency and accuracy.
- **CodeLlama-7b**: CodeLlama [5] is a specialized variant of Meta's LLaMA architecture [54], adapted for programming tasks. It is designed to support developers by providing code suggestions, completing code snippets, and assisting in debugging. CodeLlama is trained on a vast corpus of programming languages and can perform a variety of code-related tasks with high proficiency.
- **GPT-3.5**: GPT-3.5 [55] is a general-purpose language model developed by OpenAI. While it is not exclusively designed for coding, it possesses powerful code generation capabilities due to its extensive training on diverse text, including programming languages. GPT-3.5 can understand and generate code across various languages, making it a versatile tool for code completion, documentation, and coding assistance. When used for code-related tasks, an instruction (prompt) is often provided to guide the model to generate the desired code output.

In our implementation, the first six code LLMs were downloaded from Hugging Face [56]. For LLMs with Python-specific versions available, we utilized those versions to improve completion results for Python functions. Consequently, the versions used for CodeGen and CodeLlama were `CodeGen-{350M,2B,6B}-mono` and `CodeLlama-7b-Python-hf`, which were fine-tuned on additional Python corpora. For DeepSeek and StarCoder2, the versions employed were `deepseek-coder-1.3b-instruct` and `starcoder2-3b`, respectively. For the general purpose LLM, *i.e.,* GPT-3.5, we queried the model via its official online APIs [57], using the `gpt-3.5-turbo` version released in January 2024.

For the six LLMs specifically tailored for code-related tasks, *i.e.,* three versions of CodeGen, DeepSeek, StarCoder2, and CodeLlama, the constructed prompts can be directly fed into the models to generate completions. Meanwhile, for the general-purpose GPT-3.5, we provided an instruction to ensure accurate code completion for the constructed prompts. The instruction was: *"Complete and output the next line for the following Python function: $pmpt$"*. For all these LLMs, we utilized greedy decoding (*i.e.,* choosing the token with highest possibility at each decoding step) to generate one completion for each prompt in $\mathcal{O}$ or $\mathcal{U}$. The maximal output token limit was set to 50. The greedy decoding for GPT-3.5 was implemented through setting the temperature parameter to 0.

Formally, the procedure of LLM-based completion is defined as:

$$comp \leftarrow \textbf{LLM}(pmpt)$$

TABLE II: Evaluated Large Language Models (LLMs).

Model	# Parameters	Open-Source
CodeGen-350m	350 million	✓
CodeGen-2b	2 billion	✓
CodeGen-6b	6 billion	✓
DeepSeek-1.3b	1.3 billion	✓
StarCoder2-3b	3 billion	✓
CodeLlama-7b	7 billion	✓
GPT-3.5	175 billion	✗

D. Completion Result Annotation

For each sample ($pmpt$, $dep \rightarrow rep$), we examined the completions generated by LLMs and determine whether the studied API was predicted and whether the deprecated API or replacing API was predicted. Specifically, we extract the FQN of the API invocation in the predicted line using the same object type resolution and API alias resolution in Sec. III-B1. The annotation procedure is formally described as follows, $comp$ is the completion result:

$$\{good, bad, irrelevant\} \leftarrow \text{annotate}(comp)$$

Specifically, *bad* denotes that the LLM gives a completion suggestion using a deprecated API, *i.e.,* the FQN of an invocating API was matched to dep; *good* denotes the LLM gives a completion suggestion using a replacing API, *i.e.,* the FQN of an invocating API was matched to rep; *irrelevant* denotes that the LLM suggests neither of the mapping APIs. Moreover, if a completion was annotated as either *bad* or *good*, we treat it as *plausible*. This indicates that the LLM successfully understood the code context and selected a plausible API functionality.

We investigate the performance of the LLMs using the following metrics:

- **API Usage Plausibility (AUP):** This metric measures the portion of *plausible* completions, which were annotated as *good* or *bad*. AUP is defined as:

$$AUP = \frac{1}{|\mathcal{P}|} \sum_{pmpt \in \mathcal{P}} \mathbb{I}(\text{annotate}(\textbf{LLM}(pmpt)) \in \{good, bad\})$$

- **Deprecated Usage Rate (DUR):** This metric calculates the rate of *plausible* completions that were annotated as *bad*. DUR is defined as:

$$DUR = \frac{\sum_{pmpt \in \mathcal{P}} \mathbb{I}(\text{annotate}(\textbf{LLM}(pmpt)) = bad)}{\sum_{pmpt \in \mathcal{P}} \mathbb{I}(\text{annotate}(\textbf{LLM}(pmpt)) \in \{good, bad\})}$$

In these equations, $\mathcal{P}$ is the prompt set corresponding to $\mathcal{O}$ or $\mathcal{U}$, and $\mathbb{I}(\cdot)$ is a binary function that returns 1 if the passed argument is true and 0 otherwise.

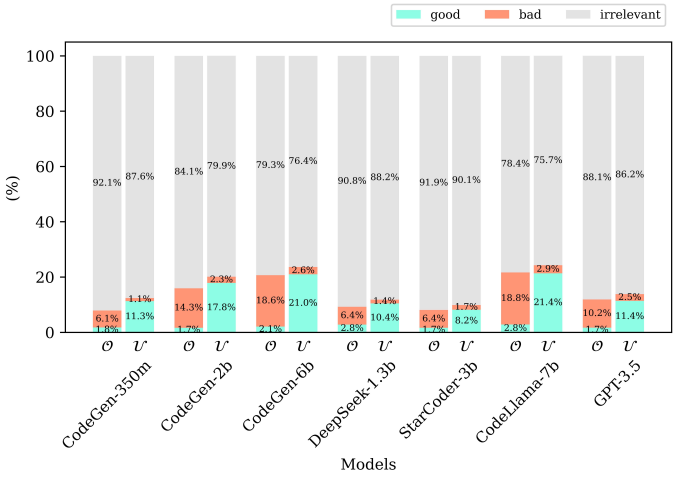

Fig. 3: Distribution of *good*, *bad*, and *irrelevant* Completions by LLMs for Prompts from $\mathcal{O}$ and $\mathcal{U}$

TABLE III: AUP and DUR Metrics for $\mathcal{O}$, $\mathcal{U}$, and Overall Dataset (*i.e.,* All = $\mathcal{O} \cup \mathcal{U}$)

Model	AUP (%)			DUR (%)		
	$\mathcal{O}$	$\mathcal{U}$	All	$\mathcal{O}$	$\mathcal{U}$	All
CodeGen-350m	8.6	14.2	12.3	77.7	9.0	24.9
CodeGen-2b	19.0	25.2	23.1	89.6	11.3	32.6
CodeGen-6b	26.1	30.9	29.3	90.0	11.1	34.2
DeepSeek-1.3b	10.2	13.4	12.3	69.7	11.6	27.2
StarCoder-3b	8.8	10.9	10.2	79.4	17.0	34.4
CodeLlama-7b	27.6	32.1	30.6	86.9	12.1	34.2
GPT-3.5	13.5	16.0	15.2	85.7	17.8	37.4

IV. Study Results

We present the experimental results and key findings on the status quo and root causes of deprecated and replacement API usage in LLM-based code completion. As illustrated in Figure 1, the results are categorized into three detailed aspects: Model Perspective (RQ1), Prompt Perspective (RQ2), and Library Perspective (RQ3). For each RQ, the results and findings are presented by first showing the status quo through *API Usage Plausibility* and *Deprecated Usage Rate*, followed by providing an in-depth *Root Cause Analysis*.

A. RQ1: Model Perspective for Status Quo and Cause Analysis

1) Status Quo Analysis: Figure 3 shows the distribution of *good* and *bad* completions by the different LLMs, and Table III presents the AUP and DUR metrics.

API Usage Plausibility. The completion distribution and the low AUP highlight that all the LLMs faced challenges in predicting *plausible* API usages for the given prompts. The AUP of the LLMs for overall dataset (*i.e.,* All = $\mathcal{O} \cup \mathcal{U}$) ranges from 10% to 30%, indicating that a majority of predictions were *irrelevant*. Among the LLMs, CodeLlama-7b achieved the highest AUP (30.6%), while StarCoder2-3b had the lowest overall AUP (10.2%). This may be attributed to the fact that StarCoder2 was not specifically fine-tuned on additional Python corpora, unlike other LLMs such as CodeLlama and CodeGen. Comparing the three versions of CodeGen suggests that the capacity of LLMs to predict *plausible* API usages increased with model size (*i.e.,* 12.3%, 23.1%, and 29.3% for CodeGen-350m, -2b, and -6b, respectively), given the model architecture and training data remain consistent. However, it's noteworthy that the largest LLM, GPT-3.5, did not achieve a high AUP (15.2%), possibly due to the instruction used not being finely tuned with advanced prompt engineering techniques such as chain-of-thought (COT) [58] and in-context learning (ICL) [59].

> **Finding 1:** All the evaluated LLMs faced challenges in predicting *plausible* API usages, with AUP ranging from 10% to 30%. Effectiveness of the LLMs in code completion generally improved with model size and language-specific fine-tuning.

Deprecated Usage Rate. The distribution shown in Figure 3, along with the DUR metric presented in Table III, indicates that all LLMs faced issues with using deprecated API usages.

The DUR of the LLMs for the overall dataset (*i.e.,* All = $\mathcal{O} \cup \mathcal{U}$) ranges from 25% to 38%, with larger models (*e.g.,* CodeGen-6b, CodeLlama-7b, and GPT-3.5) generally predicting more usages of deprecated APIs. Considering the differences among the LLMs, CodeLlama-7b and CodeGen-6b demonstrated the best balance between AUP and DUR. They achieved significant improvements in AUP compared to other LLMs, with a comparable DUR of 34.2%. Conversely, StarCoder2-3b and GPT-3.5 exhibited higher DUR (*i.e.,* 34.4% and 37.4%) despite having much lower AUP. These results indicate that the preference of LLMs for using deprecated or replacing APIs is not closely related to their capacity for predicting *plausible* completions.

> **Finding 2:** All the evaluated LLMs faced issues with deprecated API usages, with DUR ranging from 25% to 38%, and larger models exhibiting higher DUR. Among the LLMs, CodeLlama-7b and CodeGen-6b demonstrated the best balance between AUP and DUR.

2) Root Cause Analysis: From the model perspective, the causes of deprecated API usages can be divided into two main points:

Model Training: The LLMs were trained on large-scale code corpora, primarily collected from code repositories. As libraries evolved, both deprecated APIs and their replacements were used in software development, leading to training corpora containing instances of deprecated API usages. For instance, in this study, we collected 9,022 functions from open-source code corpora that used deprecated APIs. When trained on such data, LLMs might "memorize" these deprecated APIs and their usage contexts as part of the learned knowledge [60, 61]. The different training corpora also led to the different AUP and DUR of the LLMs.

Model Inference: During the inference stage, LLMs generated completions by predicting token probabilities based on their learned *prior* knowledge (*e.g.,* the memorized API usage

contexts) and applying token selection strategies (*e.g.,* greedy search or beam search). Given certain contexts, LLMs were likely to predict deprecated API usages due to the high token probabilities, without considering any *posterior* knowledge about API deprecations.

Finding 3: There are two primary reasons why LLMs predict deprecated APIs: the presence of deprecated API usages in corpora during model training, and the absence of posterior knowledge about API deprecations during model inference.

B. RQ2: Prompt Perspective for Status Quo and Cause Analysis

1) Status Quo Analysis: Table III presents the AUP and DUR metrics for prompts from the two datasets, *i.e.,* $\mathcal{O}$ and $\mathcal{U}$.

API Usage Plausibility. Between the prompts from the two different datasets, $\mathcal{O}$ and $\mathcal{U}$, there are some differences in AUP for all LLMs, with relative differences (*i.e.,* $(AUP^{\mathcal{U}} - AUP^{\mathcal{O}})/AUP^{\mathcal{O}}$) ranging from 16% to 65%. This disparity may be attributed to the imbalance in the number of outdated and up-to-dated functions in the LLMs' training corpora [62, 63]. Indirect evidence for this is that, in this study, the up-to-dated functions collected from open-source code repositories were about twice as many as the outdated functions (*i.e.,* 19,103 *vs.* 9,022), even though there was no collection preference. Given that LLMs were often trained on open-source code repositories, they likely learned more up-to-dated functions than outdated functions, leading to better AUP for the $\mathcal{U}$ dataset. Additionally, larger LLMs showcased smaller AUP differences (*e.g.,* 16% for CodeLlama-7b), possibly due to the better generalizability.

Finding 4: The LLMs showcased difference in AUP between the two datasets $\mathcal{O}$ and $\mathcal{U}$. This difference is possibly attributed to the different distribution of outdated and up-to-dated functions in the training corpora of LLMs.

Deprecated Usage Rate. When considering $\mathcal{O}$ and $\mathcal{U}$ separately, all LLMs consistently demonstrated extremely high deprecated usage rates for $\mathcal{O}$ (70%-90% DUR) and relatively low rates for $\mathcal{U}$ (9%-18% DUR). This significant difference is also evident in the distribution of *good* and *bad* completions shown in Figure 3. In fact, for *plausible* completions, the rate of *reference* API usages (*i.e.,* the usages used in the original functions) equals AUP for $\mathcal{O}$, while it is $(1 - AUP)$ for $\mathcal{U}$. Considering this rate, there is no noticeable difference between $\mathcal{O}$ and $\mathcal{U}$ (*i.e.,* around 70%-90% for both). This suggests that LLMs predicted the reference APIs for most prompts from both datasets, influenced by their differing characteristics. Nonetheless, the 9%-18% DUR for $\mathcal{U}$ indicates that LLMs still predicted the usage of deprecated APIs, even for the prompts from up-to-dated functions.

Finding 5: The LLMs consistently exhibited a significant difference in DUR between the two datasets, $\mathcal{O}$ and $\mathcal{U}$, with extremely high deprecated API usage rates for $\mathcal{O}$ (70%-90% DUR) and relatively low rates for $\mathcal{U}$ (9%-18% DUR).

2) Root Cause Analysis: The contextual characteristics of the input completion prompts can significantly influence LLMs' use of deprecated APIs. Since the completions were generated based on the input prompts, specific contexts can lead LLMs to use deprecated APIs. As presented above, the LLMs showed significantly different DUR for prompts from $\mathcal{O}$ and $\mathcal{U}$. Upon comparing the prompts from the two datasets, we found that contextual characteristics of the prompts, such as specific objects, control flows, and data flows, often lead the LLMs' predictions, *i.e.,* whether to use deprecated APIs or their replacements.

Finding 6: The contextual characteristics of the input prompts, such as the defined objects, control flows, and data flows, is a significant influence on the use of deprecated APIs.

C. RQ3: Library Perspective for Status Quo and Cause Analysis

We also conducted a detailed analysis to examine the LLMs' completions across different libraries.

1) Status Quo Analysis: The status quo from library perspective focuses on how the AUP and DUR metrics vary with different libraries.

API Usage Plausibility. The results of API usage plausibility are illustrated in the scatter plots depicted in Figure 4, where each data point signifies the AUP of an LLM for a specific library. Across the 8 libraries, most LLMs exhibited relatively low API usage plausibility for both $\mathcal{O}$ and $\mathcal{U}$, with AUP below 30%. Notable exceptions were Pandas and TensorFlow, where CodeLlama-7b, CodeGen-6b, and CodeGen-2b (represented by symbols " ★ ", " ⋋ ", and " ⋏ ", respectively) achieved better AUP (around or greater than 40%). This aligns with the results presented in Model Perspective, where CodeLlama-7b, CodeGen-6b, and CodeGen-2b achieved the best results for the overall dataset. On the other hand, among the libraries, completion prompts from SciPy and seaborn posed the most difficulty for LLMs in predicting *plausible* API usages, with AUP consistently below 15%.

Finding 7: Most LLMs exhibited relatively low API usage plausibility across the 8 libraries, with AUP below 30%. CodeLlama-7b, CodeGen-6b, and CodeGen-2b achieved better AUP for Pandas (about 45%-65%) and TensorFlow (around 40%).

Deprecated Usage Rate. The results are presented in the scatter plots shown in Figure 5, where each data point represents the DUR of a particular LLM for a specific library. The results reveal significant differences in the usage of deprecated APIs

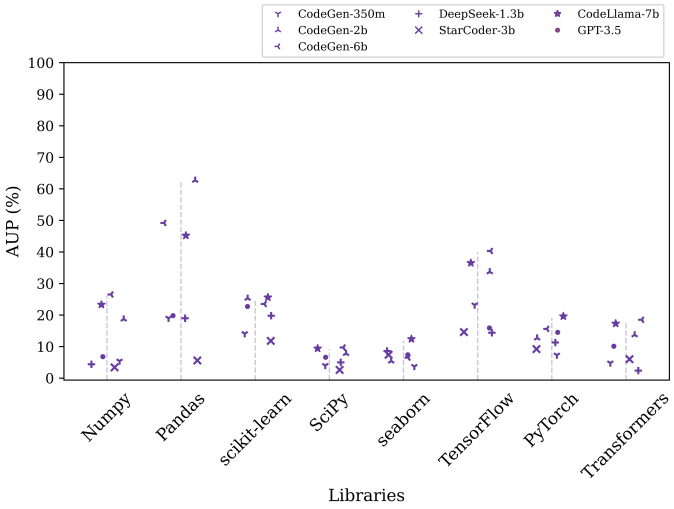

Fig. 4: AUP by Different LLMs across Eight Libraries

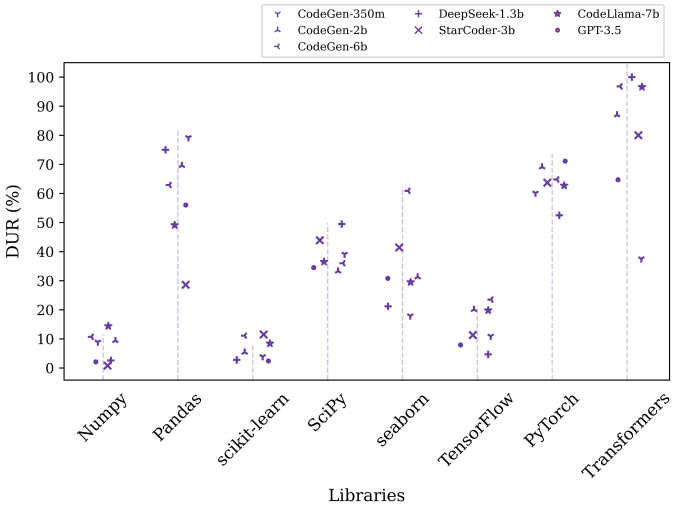

Fig. 5: DUR by Different LLMs across Eight Libraries

across the libraries, with DUR ranging from approximately 0% to 100%. Specifically, LLMs generally showed low DUR (around or below 20%) for Numpy, scikit-learn, and TensorFlow. In contrast, LLMs exhibited consistently high DUR for SciPy (approximately 30%-50%) and PyTorch (approximately 50%-70%), and unstable DUR for Pandas (approximately 30%-80%) and Transformers (approximately 40%-100%).

Finding 8: LLMs showed significant differences in the usage of deprecated APIs across various libraries, with DUR ranging from approximately 0% to 100%. LLMs exhibited consistently high DUR for SciPy and PyTorch, and unstable DUR for Pandas and Transformers.

2) Root Cause Analysis: After analyzing the completion prompts and LLMs' predictions, we found that the AUP differences between these libraries were primarily due to the characteristics of the API usage context. More specifically, the usage contexts of certain APIs followed common patterns and were surrounded by related APIs, allowing advanced LLMs like CodeLlama-7b to infer the desired API functionality based on the completion prompt. For example, many utility APIs in TensorFlow, such as the deprecated *tensorflow.compat.v1.initialize_all_variables* and its replacement *tensorflow.compat.v1.global_variables_initializer*, were often used in recognizable patterns that were easier for LLMs to predict. Conversely, some APIs were used more flexibly in diverse contexts and lacked obvious combinations with other APIs, leading to low AUP for LLMs in predicting such APIs. For instance, many APIs in SciPy, like the deprecated *scipy.misc.comb* and its replacement *scipy.special.comb*, can be used in diverse contexts to produce combinations for a data sequence. Since the data sequence can originate from numerous sources and be structured in various ways (*e.g.,* Numpy array and Pandas Series), it is challenging for LLMs to predict these APIs based on the completion prompt without additional hints.

The characteristics of API deprecations during library evolution also significantly impacted the use of deprecated APIs in LLM-based code completions. Some APIs were deprecated due to simple package refactoring, leading to similar usage patterns for the deprecated APIs and their replacements. For example, in the version 1.9.0 release of PyTorch, many APIs for tensor linear algebra were moved from the *torch* package to the *torch.linalg* package without changes to the API parameters or usage patterns (*e.g., torch.lstsq* $\rightarrow$ *torch.linalg.lstsq*). In such cases, LLMs found it more difficult to distinguish between deprecated APIs and their replacements, resulting in more frequent use of deprecated APIs in the predicted completions.

Finding 9: The characteristics of API deprecations during library evolution significantly impacted the use of deprecated APIs in LLM-based code completions. Minor changes between deprecated APIs and their replacements, such as simple package refactoring, often led to more pronounced issues with deprecated API usages.

V. Mitigation Approaches

Based on the findings regarding the causes of deprecated API usage, we proposed two lightweight mitigation approaches, which can serve as baselines for future investigation.

A. Motivation

As analyzed in the RQs, the causes of deprecated API usage in LLM-based completions can be attributed to Model Training, Model Inference, Prompt, and Library aspects. In this section, we further explore the feasibility of mitigating deprecated API usage issues from these four perspectives.

For model training, a direct mitigation method is to clean up the code containing deprecated APIs from the training corpora. However, this is impractical due to the constant evolution of libraries. New API deprecations would necessitate repeatedly rebuilding the training corpora and retraining the model. From the library perspective, we cannot control the evolution of the library and API deprecation. Therefore, mitigation efforts

Algorithm 1: Deprecation-Aware Code Completion

Input: Prompt $pmpt$, API Mappings $\mathcal{M}$
Output: Completion $comp$

1 $comp \leftarrow \textbf{LLM}(pmpt)$
2 **for** $(dep \rightarrow rep) \in \mathcal{M}$ **do**
3 **if** CONTAINS$(comp, dep)$ **then**
4 $comp \leftarrow$ FIX$(pmpt, comp, dep, rep)$
5 **break**

should focus on decoding strategies in model inference and prompt engineering.

B. Two Lightweight Approaches

Our basic idea is illustrated in Algorithm 1. Given an LLM, its generation process can generally be formulated as follows:

$$\mathbb{T}^O = \textbf{LLM}(\mathbb{T}^I),$$

where $\mathbb{T}^I$ and $\mathbb{T}^O$ are the input token sequence and output token sequence, respectively. In the context of LLM-based code completion, $\mathbb{T}^I$ corresponds to the input prompt $pmpt$, and $\mathbb{T}^O$ corresponds to the predicted completion $comp$ (line 1). When $comp$ contains a deprecated API dep (line 3), we need to perform fixing approaches, *i.e.,* the **FIX** procedure, to replace dep with the corresponding replacement rep (line 4). Note that during the **CONTAINS** procedure in line 3, API alias resolution is conducted similarly to the process described in Section III-B1.

The **FIX** procedure can be achieved by reconstructing input, through either (i) replacing the deprecated API tokens or (ii) inserting an additional replacing prompt, and then regenerating output:

- **Approach 1 - REPLACEAPI: Replacing Deprecated API Tokens then Regenerating.** As shown in Algorithm 2, the **REPLACEDEP** procedure involves removing the tokens corresponding to dep and any subsequent tokens from $comp$, then appending the tokens of rep (line 2). This results in a prefix $prefix$ that includes rep. The $prefix$ is concatenated with the $pmpt$, and the LLM generates the suffix (line 3), *i.e.,* arguments for rep and the remaining tokens to complete the code line. This concatenation forms the fixed completion $comp^*$.
- **Approach 2 - INSERTPROMPT: Inserting Additional Replacing Prompt then Regenerating.** As shown in Algorithm 3, the **CREATEDEPPMPT** procedure constructs an additional replacing prompt $pmpt'$ (line 2), formatted as inline comments to guide the LLM to use rep instead of dep. By expressing the replacing instruction into inline comments, the re-written prompt (*i.e.,* $pmpt \oplus pmpt'$) can be naturally processed by the LLM to continue writing the code. The replacing prompt is structured as follows, where "`{comp}`", "`{dep}`", and "`{rep}`" are placeholders for the original completion, deprecated API, and replacing API, respectively, and "..." represents the indentation to ensure syntax correctness.

Algorithm 2: Approach 1: Replacing Deprecated API Tokens then Regenerating

1 **Procedure FIX**$(pmpt, comp, dep, rep)$**:**
2 $prefix \leftarrow$ **REPLACEDEP**$(comp, dep, rep)$
3 $suffix \leftarrow \textbf{LLM}(pmpt \oplus prefix)$
4 $comp^* \leftarrow prefix \oplus suffix$
5 **return** $comp^*$

Algorithm 3: Approach 2: Inserting Additional Replacing Prompt then Regenerating

1 **Procedure FIX**$(pmpt, comp, dep, rep)$**:**
2 $pmpt' \leftarrow$ **CREATEREPPMPT**$(comp, dep, rep)$
3 $comp^* \leftarrow \textbf{LLM}(pmpt \oplus pmpt')$
4 **return** $comp^*$

```
...# {comp}
...# {dep} is deprecated, use {rep} instead and
    ↪ revise the return value and arguments.
```

The created $pmpt'$ is then concatenated with $pmpt$ and fed into the LLM to generate a new completion $comp^*$ (line 3).

C. Evaluation

We conducted experiments to address the following research question:

- **RQ4:** How effectively can the proposed approaches fix deprecated API usage in completions?

1) Evaluation Procedure: For each LLM, we selected up-to-dated samples from $\mathcal{U}$ where the LLM predicted bad completions using deprecated APIs, i.e.,

$$\mathcal{T} = \{(pmpt, dep \rightarrow rep) \in \mathcal{U} : \text{annotate}(\textbf{LLM}(pmpt)) = bad\},$$

to constitute the evaluation data. These samples were chosen because they have corresponding ground-truth completions (e.g., the line following the prompt in Figure 2), which are essential for assessing the effectiveness of the proposed fixing approaches.

For each sample, we employed the deprecation-aware code completion illustrated in Algorithm 1 with the two fixing approaches REPLACEAPI and INSERTPROMPT to prompt the LLM to generate a completion.

We used the following three metrics to assess the effectiveness of the proposed approaches:

- **Fixed Rate (FR)**: This metric indicates the proportion of *good* completions predicted by the fixing approaches.
- **Edit Similarity (ES)** [64]: This metric measures the similarity between the predicted completions and the ground-truth completions by analyzing the edit operations required to transform one into the other.
- **Exact Match (EM)**: This metric calculates the rate of predicted completions that exactly match the ground-truth completions after normalizing the return values of function calls (*i.e.,* replacing each element in return value with "_").

2) Results and Analysis: The evaluation results of the proposed fixing approaches are presented in Table IV.

REPLACEAPI. Using the REPLACEAPI fixing approach (Algorithm 2), all the LLMs achieve high fixed rates (FR), with values exceeding 85%. Failures in fixing are mainly due to syntax errors or incorrect function calls caused by erroneous tokens following the replaced APIs. For example, consider a bad completion: "meta_graph_def=tf.saved_model.loader.load(...)" predicted by the original completion procedure (line 1 of Algorithm 1). REPLACEAPI replaces the deprecated API *tf.saved_model.loader.load* with its replacement *tf.saved_model.load*, producing a $prefix$ of "meta_graph_def=tf.saved_model.load" (line 2 of Algorithm 2). However, CodeGen-2b then predicts a $suffix$ of "_meta_graph_def(...)" (line 3 of Algorithm 2), resulting in an erroneous function call: *tf.saved_model.load_meta_graph_def()*. This issue arises because the replacement operation in REPLACEAPI can disrupt the naturalness of the code context [65] and the LLMs' decoding process. An additional interesting finding is that for the three versions of CodeGen, the FR decreases as the model size increases. This suggests that larger models might be more sensitive to interventions in the decoding process.

The completions generated by LLMs using the REPLACEAPI approach exhibit high edit similarity (ES), exceeding 80%, and exact match rates between 30% and 50%, with these rates increasing alongside model size. The inaccuracies in the completions often involve incorrect return values and arguments for the replacing APIs. Incorrect return values arise because the replacing API might include different elements compared to the deprecated API, and the REPLACEAPI approach cannot resolve such inconsistencies in the $prefix$ (line 2 of Algorithm 2). The incorrect arguments are primarily due to the LLMs' limitations in correctly utilizing replacing APIs, especially those with complex argument lists.

INSERTPROMPT. When applying the INSERTPROMPT fixing approach (Algorithm 3), the LLMs exhibited significantly varied fixed rates, ranging from 25.7% to 97.2%. This variation suggests that larger models generally possess a stronger capacity to interpret and utilize inserted prompts formatted as inline comments. An exception to this trend is DeepSeek-1.3b, which achieved a notably high FR of 93.5%. This success can be attributed to using the `deepseek-coder-1.3b-instruct` version, which has robust zero-shot instruction-following capabilities. Moreover, the FR differences among various LLMs highlight their sensitivity to prompt construction [66, 67]. This sensitivity indicates that different LLMs may require specialized additional prompts in INSERTPROMPT for optimal performance.

Considering edit similarity and exact match, the completions generated by GPT-3.5 and DeepSeek-1.3b using the INSERTPROMPT approach do not perform as well as their fixed rates suggest. Despite being fine-tuned with an instruct-tuning corpus, they are not specifically fine-tuned on Python code. As a result, they can follow the instructions in the additional prompt to use replacing APIs but struggle to use those APIs correctly.

Comparison. The comparison between REPLACEAPI and INSERTPROMPT suggests that direct interventions in the decoding process are more effective than zero-shot prompt engineering. However, the results also reveal the potential of INSERTPROMPT. First, REPLACEAPI cannot be applied to black-box LLMs like GPT-3.5, as their decoding processes cannot be controlled by users. Second, the additional prompt employed by INSERTPROMPT was not carefully tuned for each LLM and was performed in a zero-shot manner. In the future, fine-tuning the LLMs with instructions specifically designed for fixing deprecated usage could enhance effectiveness.

TABLE IV: Evaluation Results of Proposed Approaches

Model	REPLACEAPI			INSERTPROMPT		
	FR (%)	ES (%)	EM (%)	FR (%)	ES (%)	EM (%)
CodeGen-350m	92.1	82.3	30.8	25.7	58.7	8.9
CodeGen-2b	88.2	84.6	38.8	66.1	66.0	23.3
CodeGen-6b	85.2	85.3	43.5	77.4	72.9	35.3
DeepSeek-1.3b	99.6	80.9	31.7	93.5	77.7	24.4
StarCoder-3b	90.9	85.0	42.2	85.3	72.2	29.1
CodeLlama-7b	99.5	85.7	48.1	95.5	82.0	43.3
GPT-3.5	–	–	–	97.2	76.2	20.5

Finding 10: The proposed REPLACEAPI effectively addresses deprecated API usage for all open-source LLMs, achieving fix rates exceeding 85%. The fixed completions also demonstrate acceptable accuracy compared to ground-truth completions. While INSERTPROMPT does not currently achieve sufficient effectiveness and accuracy in fixing completions containing deprecated API usage, it shows potential for future exploration.

VI. DISCUSSION

A. Implications

We provide implications for future research on the synergy of library evolution and LLM-driven software development.

Validating LLM-Generated Code Completions and Issuing Alerts for Deprecated API Usages. Our evaluation study reveals that LLMs frequently use deprecated APIs during code completion. Such deprecated API usages can be easily overlooked [19], potentially introducing bugs or security vulnerabilities into software projects. Therefore, implementing a validation mechanism for deprecated API usage in LLM-generated code completions is crucial to ensure the reliability of the code. Such a validation mechanism can be further integrated into the pose-processing of LLM-based code completion, such as issuing alerts to developers.

Fixing and Updating Outdated API Knowledge in LLMs by Model-Level Repair. The current fixing approaches mitigate the issues by intervening in decoding process and rewriting prompts, without addressing the outdated knowledge about deprecated API usages stored in the LLMs. Given the constant evolution of libraries, lightweight model repair techniques are potential solutions for fixing outdated knowledge. *Model Editing* is one such direction, which can be categorized into the following main categories: Memory-based approaches [68, 69,

70], Locating-then-editing approaches [71, 72, 73], and Meta-learning approaches [74, 75]. Compared to the proposed fixing approaches, model editing can directly update the outdated knowledge about deprecated API usages, even incorporating the information of entirely new replacing APIs (*i.e.,* the replacing APIs introduced after model training and unseen in the training corpora) into the LLMs.

Leveraging Retrieval-Augmented Generation for Up-to-dated Code Completion. As discussed in our study findings, a key cause of the deprecated API usage in LLM-based completions is the lack of posterior knowledge about API deprecations. Retrieval-Augmented Generation (RAG) is a suitable technique that can perfectly align with the need for posterior knowledge [76]. We can explore the possibility of adopting RAG to mitigate deprecated API usage in LLM-based code completion by retrieving related knowledge pieces, such as documentation and usage examples.

Designing Agent & Multi-Agent Systems for Incorporating Library Evolution into LLM-Driven Software Development. In modern software development driven by LLMs, the issues brought by library evolution are encountered not only in code completion. With advancements in AI agents [77, 78, 79], we can potentially develop autonomous agents or multi-agent systems capable of automatically discovering deprecated API usages, identifying correct replacements, upgrading dependent libraries, and fixing the code. To ensure comprehensive recognition of deprecated API usage and up-to-date fixes, we should design an effective multi-agent collaboration pipeline. One agent should scan the generated code to identify all pieces related to API usage. Another agent should continuously fetch information online, checking the latest official API documentation to aid in discovery and correction. Finally, a dedicated agent should be responsible for implementing the necessary library upgrades and code fixes. Such an agent system can fundamentally address the issues discussed in this article.

B. Threats to Validity

Internal Threats. The primary threat to the internal validity of our study is the soundness of the static analysis used for function location and result annotation. Given that Python is a dynamic programming language, the lightweight object type resolution method we employed may have missed some function calls of deprecated and replacing APIs during the matching process. In the future, we aim to address this issue by implementing advanced type inference techniques. Additionally, our study currently focuses on function-level API deprecation, overlooking parameter-level deprecations. Future research should investigate a broader range of deprecated API categories to provide a more comprehensive analysis.

External Threats. A primary threat to the external validity of our study lies in the choice of Python libraries and the evaluated LLMs. To mitigate this threat, we reused libraries examined in previous studies and introduced three popular deep learning libraries to ensure diversity and timeliness. For the LLMs, we selected models covering various architectures, model sizes, training corpora, and training strategies to ensure the generalizability of our findings. Another external threat is that the study was conducted solely on the Python language, which may limit the applicability of our findings to other languages such as Java and C#. In the future, we plan to explore the impact of library evolution on LLM-based code completion across a broader range of programming languages.

VII. Conclusion

In this work, we conducted an evaluation study to investigate the statuses and causes of deprecated API usages in LLM-based code completion. The study results all evaluated LLMs encounter challenges in predicting *plausible* API usages and face issues with deprecated API usages, influenced by the distinct code context characteristics of the prompts and the characteristics of API deprecations during the evolution of these libraries. We propose two lightweight fixing approaches to mitigate the deprecated API usages and can serve as baselines for future research. We also provide implications for the research directions for the combination of library evolution and LLM-driven code completion and software development.

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, and Others, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[2] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, S. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/pdf?id=hQwb-lbM6EL

[3] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/pdf?id=iaYcJKpY2B_

[4] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, and Others, "Starcoder: may the source be with you!" *CoRR*, vol. abs/2305.06161, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.06161

[5] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton-Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," *CoRR*, vol. abs/2308.12950, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.12950

[6] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming - the rise of code intelligence," *CoRR*, vol. abs/2401.14196, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.14196

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

[8] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 314–21 328, 2022.

[9] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 39–51.

[10] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1469–1481.

[11] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 172–184.

[12] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 858–868.

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

[14] S. Ugare, T. Suresh, H. Kang, S. Misailovic, and G. Singh, "Improving llm code generation with grammar augmentation," *arXiv preprint arXiv:2403.01632*, 2024.

[15] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[16] (2023) Github copilot. [Online]. Available: https://github.com/features/copilot

[17] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, "An empirical study on the impact of refactoring activities on evolving client-used apis," *Inf. Softw. Technol.*, vol. 93, no. C, pp. 186–199, 2018.

[18] M. Hu and Y. Zhang, "An empirical study of the python/c api on evolution and bug patterns," *Journal of Software: Evolution and Process*, vol. 35, no. 2, p. e2507, 2023.

[19] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated python library apis are (not) handled," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 233–244. [Online]. Available: https://doi.org/10.1145/3368089.3409735

[20] Api lifecycle stages. [Online]. Available: https://developers.meetmarigold.com/engage/terms/versioning-deprecation/#api-lifecycle-stages

[21] Pytorch: A python package that provides tensor computation and deep neural networks. [Online]. Available: https://pytorch.org/

[22] D. Zan, B. Chen, D. Yang, Z. Lin, M. Kim, B. Guan, Y. Wang, W. Chen, and J. Lou, "CERT: continual pre-training on sketches for library-oriented code generation," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, L. D. Raedt, Ed. ijcai.org, 2022, pp. 2369–2375. [Online].

Available: https://doi.org/10.24963/ijcai.2022/329

[23] K. Zhang, G. Li, J. Li, Z. Li, and Z. Jin, "Toolcoder: Teach code generation models to use API search tools," *CoRR*, vol. abs/2305.04032, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.04032

[24] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[25] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[26] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of transformer models for code completion," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4818–4837, 2021.

[27] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of bert models for code completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 108–119.

[28] H. Ding, V. Kumar, Y. Tian, Z. Wang, R. Kwiatkowski, X. Li, M. K. Ramanathan, B. Ray, P. Bhatia, S. Sengupta *et al.*, "A static evaluation of code completion by large language models," *arXiv preprint arXiv:2306.03203*, 2023.

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

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

[31] A. A. Sawant, M. Aniche, A. van Deursen, and A. Bacchelli, "Understanding developers' needs on deprecation as a language feature," in *ICSE*, 2018, pp. 561–571.

[32] A. A. Sawant, G. Huang, G. Vilen, S. Stojkovski, and A. Bacchelli, "Why are features deprecated? an investigation into the motivation behind deprecation," in *ICSME*, 2018, pp. 13–24.

[33] A. Mirian, N. Bhagat, C. Sadowski, A. P. Felt, S. Savage, and G. M. Voelker, "Web feature deprecation: a case study for chrome," in *ICSE-SEIP*, 2019, pp. 302–311.

[34] A. A. Sawant, R. Robbes, and A. Bacchelli, "To react, or not to react: Patterns of reaction to api deprecation," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3824–3870, 2019.

[35] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation? the case of a smalltalk ecosystem," in *FSE*, 2012, pp. 1–11.

[36] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: A threat to the success of android apps," in *ESEC/FSE*, 2013, pp. 477–487.

[37] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *ICSM*, 2013, pp. 70–79.

[38] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to api evolution? the pharo ecosystem case," in *ICSME*, 2015, pp. 251–260.

[39] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of 25,357 clients of 4+ 1 popular java apis," in *ICSME*, 2016, pp. 400–410.

[40] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *OOPSLA*, 2005, pp. 265–279.

[41] J. Henkel and A. Diwan, "Catchup! capturing and replaying refactorings to support api evolution," in *ICSE*, 2005, pp. 274–283.

[42] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.

[43] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for api evolution," in *ICSE*, 2009, pp. 599–602.

[44] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *ICSE*, 2008, pp. 471–480.

[45] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, 2005.

[46] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *ICSE*, 2010, pp. 325–334.

[47] K. Huang, B. Chen, L. Pan, S. Wu, and X. Peng, "Repfinder: Finding replacements for missing apis in library update," in *ASE*, 2021.

[48] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 102–106.

[49] T. Wu, W. Wu, X. Wang, K. Xu, S. Ma, B. Jiang, P. Yang, Z. Xing, Y.-F. Li, and G. Haffari, "Versicode: Towards version-controllable code generation," *arXiv preprint arXiv:2406.07411*, 2024.

[50] Most popular programming languages. [Online]. Available: https://www.orientsoftware.com/blog/most-popular-programming-languages/

[51] (2023) Pytorch documentation 1.9.0. [Online]. Available: https://pytorch.org/docs/1.9.0/

[52] Openai api reference. [Online]. Available: https://sourcegraph.com/search

[53] Pep 221 – import as. [Online]. Available: https://peps.python.org/pep-0221/

[54] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale,

and Others, "Llama 2: Open foundation and fine-tuned chat models," *CoRR*, vol. abs/2307.09288, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.09288

[55] Gpt-3.5 turbo. [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5-turbo

[56] Hugging face - host git-based models, datasets and spaces on the hugging face hub. [Online]. Available: https://huggingface.co/models

[57] Sourcegraph: Code search and an ai assistant with the context of the code graph. [Online]. Available: https://platform.openai.com/docs/api-reference/chat

[58] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[59] A. K. Lampinen, I. Dasgupta, S. C. Y. Chan, K. W. Mathewson, M. H. Tessler, A. Creswell, J. L. McClelland, J. Wang, and F. Hill, "Can language models learn from explanations in context?" in *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Y. Goldberg, Z. Kozareva, and Y. Zhang, Eds. Association for Computational Linguistics, 2022, pp. 537–563. [Online]. Available: https://doi.org/10.18653/v1/2022.findings-emnlp.38

[60] F. Petroni, T. Rocktäschel, S. Riedel, P. S. H. Lewis, A. Bakhtin, Y. Wu, and A. H. Miller, "Language models as knowledge bases?" in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Association for Computational Linguistics, 2019, pp. 2463–2473. [Online]. Available: https://doi.org/10.18653/v1/D19-1250

[61] B. Cao, H. Lin, X. Han, L. Sun, L. Yan, M. Liao, T. Xue, and J. Xu, "Knowledgeable or educated guess? revisiting language models as knowledge bases," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Association for Computational Linguistics, 2021, pp. 1860–1874. [Online]. Available: https://doi.org/10.18653/v1/2021.acl-long.146

[62] J. M. Johnson and T. M. Khoshgoftaar, "Survey on deep learning with class imbalance," *J. Big Data*, vol. 6, p. 27, 2019. [Online]. Available: https://doi.org/10.1186/s40537-019-0192-5

[63] X. Liu, J. Wu, and Z. Zhou, "Exploratory undersampling for class-imbalance learning," *IEEE Trans. Syst. Man Cybern. Part B*, vol. 39, no. 2, pp. 539–550, 2009. [Online]. Available: https://doi.org/10.1109/TSMCB.2008.2007853

[64] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: code generation using transformer," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 1433–1443. [Online]. Available: https://doi.org/10.1145/3368089.3417058

[65] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 837–847. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227135

[66] T. Gao, A. Fisch, and D. Chen, "Making pre-trained language models better few-shot learners," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Association for Computational Linguistics, 2021, pp. 3816–3830. [Online]. Available: https://doi.org/10.18653/v1/2021.acl-long.295

[67] Z. Jiang, F. F. Xu, J. Araki, and G. Neubig, "How can we know what language models know," vol. 8, 2020, pp. 423–438. [Online]. Available: https://doi.org/10.1162/tacl_a_00324

[68] E. Mitchell, C. Lin, A. Bosselut, C. D. Manning, and C. Finn, "Memory-based model editing at scale," in *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvári, G. Niu, and S. Sabato, Eds., vol. 162. PMLR, 2022, pp. 15 817–15 831. [Online]. Available: https://proceedings.mlr.press/v162/mitchell22a.html

[69] S. Murty, C. D. Manning, S. M. Lundberg, and M. T. Ribeiro, "Fixing model bugs with natural language patches," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Y. Goldberg, Z. Kozareva, and Y. Zhang, Eds. Association for Computational Linguistics, 2022, pp. 11 600–11 613. [Online]. Available: https://doi.org/10.18653/v1/2022.emnlp-main.797

[70] A. Madaan, N. Tandon, P. Clark, and Y. Yang,

"Memory-assisted prompt editing to improve GPT-3 after deployment," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Y. Goldberg, Z. Kozareva, and Y. Zhang, Eds. Association for Computational Linguistics, 2022, pp. 2833–2861. [Online]. Available: https://doi.org/10.18653/v1/2022.emnlp-main.183

[71] K. Meng, D. Bau, A. Andonian, and Y. Belinkov, "Locating and editing factual associations in GPT," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/6f1d43d5a82a37e89b0665b33bf3a182-Abstract-Conference.html

[72] K. Meng, A. S. Sharma, A. J. Andonian, Y. Belinkov, and D. Bau, "Mass-editing memory in a transformer," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/pdf?id=MkbcAHIYgyS

[73] X. Li, S. Li, S. Song, J. Yang, J. Ma, and J. Yu, "PMET: precise model editing in a transformer," in *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, M. J. Wooldridge, J. G. Dy, and S. Natarajan, Eds. AAAI Press, 2024, pp. 18 564–18 572. [Online]. Available: https://doi.org/10.1609/aaai.v38i17.29818

[74] N. D. Cao, W. Aziz, and I. Titov, "Editing factual knowledge in language models," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 6491–6506. [Online]. Available: https://doi.org/10.18653/v1/2021.emnlp-main.522

[75] E. Mitchell, C. Lin, A. Bosselut, C. Finn, and C. D. Manning, "Fast model editing at scale," in *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. [Online]. Available: https://openreview.net/forum?id=0DcZxeWfOPt

[76] P. S. H. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[77] S. Gronauer and K. Diepold, "Multi-agent deep reinforcement learning: a survey," *Artificial Intelligence Review*, vol. 55, no. 2, pp. 895–943, 2022.

[78] Y. Talebirad and A. Nadiri, "Multi-agent collaboration: Harnessing the power of intelligent llm agents," *arXiv preprint arXiv:2306.03314*, 2023.

[79] B. Ellis, J. Cook, S. Moalla, M. Samvelyan, M. Sun, A. Mahajan, J. Foerster, and S. Whiteson, "Smacv2: An improved benchmark for cooperative multi-agent reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 36, 2024.