Can LLMs Keep Up with Library Changes? An Exploratory Study on LLM-Generated Code

1st Xiangrong Lin
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, Zhejiang, China
22321084@zju.edu.cn

2nd Jiakun Liu
Harbin Institute of Technology
Harbin, China
jiakunliu@hit.edu.cn

3rd Lingfeng Bao*,†
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, Zhejiang, China
lingfengbao@zju.edu.cn

***Abstract*—Selecting appropriate libraries is important when generating code, especially when the library evolves rapidly, e.g., libraries can be obsolete because of deprecated, vulnerable, and the emergence of better alternatives, and need to be updated in the repository. While large language models have shown impressive capabilities in library selection when generating the code, recent studies have not explored library selection when generating code that can also be implemented by obsolete libraries.**

To fill the gap, we explore whether LLMs can use appropriate libraries in the generated code when answering the Stack Overflow questions that were originally answered with obsolete libraries. We extract 20 obsolete libraries from the library migration history and 667 related Stack Overflow questions. Our results reveal that LLMs struggle with certain obsolete libraries undergoing security-driven and alternative-driven due to challenges such as serialization risks and the absence of well-documented migration paths, as well as data processing libraries involving complex format changes. In contrast, they perform well with long-unmaintained libraries and those related to Web and system utilities. Furthermore, questions explicitly mentioning obsolete libraries would instruct the LLM to directly generate the code with obsolete libraries rather than recommending up-to-date libraries. To improve the performance of LLM in selecting appropriate libraries, we explore different prompt refinement strategies, including explicitly showing deprecated libraries and suggesting up-to-date alternatives. Our findings show that refined prompts significantly enhance the ability of LLM to select appropriate libraries, offering valuable insights for optimizing LLM-driven code generation in evolving software ecosystems.

***Index Terms*—large language model, library selection, library migration**

I. Introduction

With the continuous advancement of artificial intelligence, Large Language Models (LLMs) have significantly advanced the software development process, including code generation [1]–[3], code completion [4]–[6], and bug fixing [7], [8]. Especially, LLMs have demonstrated capabilities in code-related Question-answering tasks, effectively assisting developers by generating code snippets answering questions [9], [10]. It improves the efficiency of developers when searching by providing relevant and context-aware recommendations. Despite the remarkable ability of LLMs to generate code, challenges still arise. The generated code often requires further refinement to address performance and readability. Researchers not only focus on the correctness of the generated code but also focus on other aspects, e.g., code effectiveness [11], [12], vulnerabilities [13], [14], and code smells [15].

Among them, one crucial aspect that is unexplored is whether LLMs can select appropriate libraries when generating code. This is because library selection plays a fundamental role in software development [16]. Due to deprecation [17], security vulnerabilities [18], and the emergence of more efficient or feature-rich alternatives [19], libraries are frequently deprecated, necessitating library migration [20] in real-world software development to ensure code reliability. However, LLMs are trained on historical data [21], their training corpus may include code that relies on deprecated libraries. As a result, LLM-generated code may also incorporate outdated or unsuitable libraries. A prior study has investigated the type, popularity, maintenance, and license of the libraries in the code generated by ChatGPT [22]. However, it is still unclear how LLMs handle libraries that are undergoing continuous updates and changes when generating code. Given the ever-evolving ecosystem of libraries, it is essential to examine whether LLMs can make informed choices, particularly for those libraries that are gradually being deprecated or migrated.

To this end, our study aims to fill this gap by analyzing the performance of LLMs in answering Stack Overflow questions, whose answers import the obsolete libraries. To systematically evaluate whether LLMs can reliably select contextually appropriate software libraries during code generation, we conduct an empirical study based on real-world programming tasks. We collect 20 obsolete libraries from the library migration history dataset, i.e., **PyMigBench** [23] and **SALM** [24]. Then we select 667 Stack Overflow questions related to these old libraries. Using this dataset, we explore the performance of recent popular SOTA, including *Qwen*, *LlaMA*, *DeepSeek*, *GPT-3.5-Turbo*, *GPT-4o*, to generate code to answer the questions.

Specifically, we provide each model with standardized prompts designed to elicit library recommendations while minimizing the influence of pre-existing code snippets. The generated code is then analyzed to determine whether the

*Corresponding author. Email: lingfengbao@zju.edu.cn
†Also affiliated with: Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security

2640-7574/26/$31.00 ©2026 IEEE
DOI 10.1109/SANER67736.2026.00088

models correctly select appropriate libraries. We analyze the impact of different migration reasons (such as long-term unmaintained libraries, security-driven migrations, and the emergence of better alternatives), the influence of library domains (including web development, data processing, and system utilities), as well as the effect of question descriptions (including functionality-driven questions and those explicitly mentioning deprecated libraries) in making informed library choices for questions related to obsolete libraries. After that, we propose strategies to enhance their recommendations. Finally, we answer research questions (RQs):

RQ1: How do different obsolescence factors and library domains affect LLMs' library selection performance?
We observe that LLMs struggle with security-driven and alternative-driven obsolete libraries, particularly due to serialization risks, undocumented migration paths, and complex format changes in data processing libraries. In contrast, they perform significantly better with long-unmaintained libraries and in domains like Web Development and System Utilities, where suitable replacements are more established.

RQ2: How does the question description affect LLMs' library selection performance?
Explicitly mentioning obsolete libraries in the question leads to lower precision compared to questions focused only on functionality needs. LLMs often struggle to override the influence of explicitly stated outdated libraries, frequently defaulting to using them rather than suggesting modern alternatives.

RQ3: Can prompt refinement improve LLMs' library selection performance?
Two prompt refinement strategies, including explicitly stating a library's deprecated status and providing recommendations for modern alternatives, improve performance across all tested models significantly.

To summarize, this paper makes the following contributions:

- We systematically investigate LLMs' performance in code generation for library migration-related questions.
- We provide practical prompt design guidelines to help users obtain better library recommendations from LLMs.
- We release a dataset containing LLM-generated code responses for these questions, facilitating future research in this area.[1]

II. Background and Related Work

In this section, we introduce the background and summarize the related work about library selection and migration and large language models.

A. Library Selection and Migration

In modern software development, libraries provide prewritten functionalities and tools, thereby assisting developers in enhancing efficiency [25]. Meanwhile, the consistency, compatibility, and scalability of libraries can enhance the stability of software in the CI/CD software development process [26]. On the other hand, using an unsuitable library may introduce vulnerabilities, degrade performance, or complicate maintenance, ultimately undermining code quality [27], [28].

Therefore, library selection plays an important role in software development. However, as the software ecosystem evolves, the libraries may become outdated [29], [30]. Additionally, libraries with known security vulnerabilities [31] or unresolved bugs [32] can pose significant risks to applications. Furthermore, developers may identify newer libraries that offer improved performance, enhanced usability, or more robust features [33]. This requires the library migration process [34], where developers replace outdated, vulnerable, or suboptimal libraries with more suitable alternatives to maintain software reliability and security. Therefore, effective library selection should account for these factors, ensuring that the chosen libraries remain well-maintained, secure, and aligned with current best practices in software development.

In this work, our aim is to understand the ability of LLMs in library selection, especially when addressing issues related to libraries that are gradually being migrated.

B. Large Language Model

Large language models (LLMs) have shown great promise in software engineering [35]. Zan et al. [36] generate code based on user-provided problem descriptions, with a focus on generating code snippets that rely on third-party libraries. By training on vast data, these models can understand and generate code snippets based on natural language inputs [37].

However, due to the training data of public code with outdated coding patterns [38], the LLMs' generated code may suffer from weaknesses [39]. Wang et al. [40] investigate the statuses and causes of deprecated API usages in LLM-based code completion. Therefore, the performance of LLMs in recommending appropriate libraries is also influenced.

Furthermore, LLMs are subject to the inherent challenge of knowledge latency. While models are trained on large datasets, the information they have is static and reflects the state of knowledge up until the point the training is completed. This means that LLMs may not be aware of recent updates, deprecated libraries, or newly introduced alternatives. When LLMs are asked about library migration-related problems, they must not only identify which libraries are outdated, insecure, or inefficient but also suggest more suitable alternatives based on the context of the migration.

III. Experiment Setup

Figure 1 shows the overview of our study. We first collect obsolete libraries and relevant Stack Overflow questions. Next, we filter out the unrelated library migrations and classify migrated libraries into different categories. Similarly, we also refine and classify SO questions, selecting those that specifically focus on library usage. With these data, we instruct LLM to generate code snippets to answer the questions. Libraries in the generated code are extracted to evaluate the performance of LLM in selecting appropriate libraries. Finally, we assess the performance of LLMs in library selection using predefined evaluation metrics.

[1] https://anonymous.4open.science/r/llm-data-15C3/

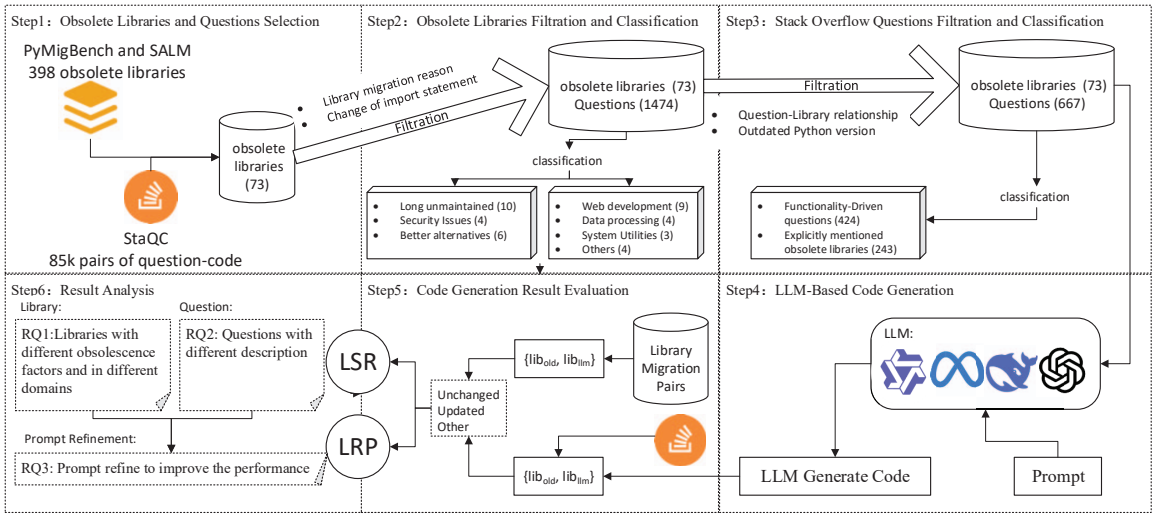

Fig. 1. The Experiment Workflow

A. Obsolete Libraries and Questions Selection

1) Datasets of Library Migration: We consider the libraries that are replaced by new libraries to be obsolete libraries. Therefore, we collected library migration instances from the following two datasets:

- **PyMigBench** [23] is a benchmark for Python library migrations. Islam et al. analyze the changes in the dependency files retrieved from GitHub repositories to identify potential library migration changes. They investigate additions, deletions, and modifications within these files, extracting patterns of library replacements, and verifying them through manual and automated processes to confirm instances of library migration. By 2024, the author has been continuously iterating and updating their dataset. The latest version of the dataset includes a total of 134 manually validated library migration pairs.
- **SALM** [24] provides a structured analysis of self-admitted library migrations across three major software ecosystems, i.e., Java, JavaScript, and Python, with a particular focus on their prevalence and unidirectional characteristics. Gu et al. designed an accurate NLP-powered heuristic-based mining method to identify migrations and their rationales directly from Git repositories. In our study, we focus on the Python ecosystem. From the SALM dataset, we extract 378 obsolete libraries.

We extract all library migration pairs in these two datasets and consider the union set of these two datasets as the final library migration pairs. During this process, duplicated pairs are removed. Finally, we collect 398 obsolete libraries that are replaced by new libraries.

2) Stack Overflow Question Selection: To evaluate LLMs' ability to select appropriate libraries in real-world coding scenarios, we use the questions and answers on Stack Overflow that are related to library selection.

The StaQC (Stack Overflow Question-Code) dataset [41], is the most diverse dataset for systematically mining high-quality question-code pairs from Stack Overflow. It focused on 'how-to-do-it' questions, as these questions reflect real-world scenarios where developers seek guidance on implementing specific functionalities. And then choose the standalone solution from all accepted answers. In our study, for simplicity, we mainly focus on the 85k question-answer pairs where there is only one standalone code block in the answer.

Then, we identify the questions that are related to the obsolete libraries in our library migration dataset. Given that Stack Overflow answers do not typically include dependency files such as requirements.txt or setup.py, we adopt a straightforward approach to extracting referenced libraries by analyzing import statements. Specifically, we identify libraries through direct import patterns such as "`import lib`" and "`from lib import module`" using the static analysis tool `ast` in Python, ensuring a reliable and efficient way to determine library usage within the provided code snippets.

Due to possible inconsistencies between import statements in code snippets and the actual library names, we consulted PyPI and the official Python documentation to perform consistency checks and ensure the accurate identification of migrated libraries. To ensure sufficient representation and statistical reliability, we only select the libraries with at least 15 associated questions. After this step, we identify 73 libraries that have a sufficient number of related questions for further analysis.

B. Obsolete Libraries Filtration and Classification

We exclude the libraries that are updated because of project-specific needs and those with unchanged import statements, ensuring a focus on migrations due to deprecation, security issues, or better alternatives.

We exclude library migrations driven by specific project requirements, such as performance or functionality improve-

ments, as these migrations are based on the unique needs of a project rather than inherent issues with the libraries themselves. For example, although a project might migrate from `argparse` to `click` due to enhanced usability for project needs, `argparse` remains a widely used and relevant library for argument parsing. Our study focuses on mainstream library migrations driven by factors related to the libraries themselves, including deprecation, security vulnerabilities, or the existence of more popular alternatives. Therefore, such library migration instances that are project-specific are excluded from the migration datasets.

We also exclude certain library migrations where the import statements remain unchanged. This could be because the older library is replaced by a more modern alternative, but the library name in the import statements stays the same. For instance, `pycrypto` and its successor `pycryptodome` share the same import statements (e.g., `from Crypto.Cipher import AES`), making it impossible to determine from the code snippet alone whether the updated library is being used, especially without access to configuration files like requirements.txt or setup.py. To address this, we exclude such cases from our analysis. This ensures that our evaluation focuses on migrations where the library transitions are explicitly visible in the code.

To implement the filtering criteria, we examined 73 libraries and their corresponding migration targets by comparing their import statements and assessing their community activity, with supporting evidence drawn from PyPI pages, official websites, and GitHub repositories to accurately identify obsolete libraries. After applying these filtering steps, we finalize a set of 20 obsolete libraries, with a total of 1,474 related questions.

At the same time, we categorize the libraries based on two key aspects: their reasons for migration and their functional domains. First, we classify libraries according to the primary factors driving their migration, including *long-term maintenance*, *security vulnerabilities*, and *the availability of better alternatives*. To achieve this, we examine official deprecation notices, security advisories, and community discussions to determine the specific reason for each migration. Furthermore, we read the README file of the repository and categorize the libraries based on the use scenario of the libraries as web development, data processing, and system utilities.

C. Stack Overflow Questions Filtration and Classification

After selecting the 20 finalized obsolete libraries, we further filter the 1,474 questions to ensure that the selected questions align with our research objective, evaluating LLMs' performance in handling library migration-related issues. We retrieve the content of each question using the `question_id` through the API provided by Stack Overflow at first. Then, we filter the Stack Overflow questions as follows:

On the one hand, some questions mention a specific library but are not directly concerned with its functionality. For example, the question **How to get difference between two *.sql files using difflib**[2] includes the `optparse` in the provided code snippet, but the focus of this question is likely on comparing SQL files using `difflib` rather than on command-line argument parsing, which is the primary purpose of the `optparse`. Therefore, the relevance of `optparse` to the question is minimal, as the library is not the main part of the problem. To refine the dataset, we examine whether the problem description is genuinely related to the intended functionality of the mentioned library, ensuring the retained questions are meaningful in the context of library selection and migration.

On the other hand, some questions explicitly require using an outdated Python version (e.g., **Python 2.7 - xlrd - Matching A String To a Cell Value**[3]), where the solution is constrained by compatibility rather than selecting a more modern or widely adopted library. Since our study focuses on evaluating LLMs in recommending more up-to-date libraries, including such questions would be misaligned with our experimental objectives. Therefore, we exclude questions where adhering to an older Python version is a strict requirement.

Due to the scale of the StaQC dataset, the number of relevant questions varies among different libraries. To mitigate the impact of the disparity in the number of questions among different libraries, for libraries with more than 50 related questions, e.g., `PySide` has 115 questions, we randomly select 50 questions from all their related questions.

In the end, we identify 20 suitable obsolete libraries and select 667 relevant questions from the StaQC dataset. The details are shown in Table I.

TABLE I
THE DETAILS OF THE 20 OBSOLETE LIBRARIES

Domain	Name	Num	Reason	Replacement
Web	BeautifulSoup	50	LT	bs4
	urlparse	34	LT	requests, urllib
	cgi	20	Sec	flask
	suds	21	Sec	zeep
	urllib	50	Alt	requests
	pycurl	13	Alt	requests
	twisted	50	Alt	asyncio
	mechanize	50	Alt	requests, selenium
	httplib	28	LT	http, httpx, requests
Data	xlwt	37	LT	openpyxl, pandas
	simplejson	17	Sec	json, ujson
	xlrd	46	LT	openpyxl, pandas
	pickle	46	Sec	dill, joblib, json
System	optparse	17	LT	argparse
	imp	16	LT	importlib
	distutils	34	Alt	setuptools
Others	MySQLdb	34	LT	mysql, pymysql
	nose	18	LT	pytest, unittest
	PySide	50	LT	PySide6, PyQt5
	pytz	36	Alt	zoneinfo, dateutil

LT—-Long-Term Unmaintained Libraries
Sec—-Security-Driven Migrations
Alt—-Better Alternatives Exist

[2]https://stackoverflow.com/questions/34848233/how-to-get-difference-between-two-sql-files-using-difflib

[3]https://stackoverflow.com/questions/17136711/python-2-7-xlrd-matching-a-string-to-a-cell-value

Prompt

You are an experienced software developer. Please assist me with the question: [QUESTION TITLE] and the question description: [QUESTION CONTENT].

- *You do not need to focus on the specific code implementation provided in the question. Instead, understand the required functionality and the challenges involved.*
- *Please generate a Python function that is secure, popular, and up-to-date.*

Output the code only, without any additional explanation.

D. LLM-Based Code Generation

1) LLM Selection: After collecting and refining the dataset, we proceed with code generation using various LLMs. We select multiple models, including *Qwen-Plus*, *LLaMA3.3-70B-Instruct*, *DeepSeek-V3*, *GPT-3.5-Turbo*, and *GPT-4o*, to generate responses to the questions. To mitigate the hallucination problem of LLMs, we perform three independent runs for each query.

These models are chosen based on their widespread use in code generation tasks, their representation of different architectures and training methodologies, and their ability to capture evolving library usage patterns. By including both closed-source and open-source models, we aim to provide a comprehensive evaluation of how different LLMs handle library selection in migrated-library-related scenarios.

2) Prompt Design: To ensure experimental consistency across various LLMs, we employ a standardized prompt format [42] that typically includes a clear task definition, relevant context, and explicit constraints to guide the model's response. Specifically, our prompt structure consists of three key components: a concise problem description to provide necessary context, an instruction directing the model to prioritize library choices over code details, and an emphasis on security and performance considerations to encourage more reliable recommendations. By structuring the prompt in this manner, we aim to reduce variance in model outputs and better assess their ability to make informed library selections.

E. Code Generation Result Evaluation

After generating the code using LLMs, we need to analyze the libraries used in the generated code. Specifically, we extract the libraries referenced in the code snippets generated by LLMs, denoted as lib_{llm}, using the method mentioned in Section III-A2. The library of the corresponding accepted answers is denoted as lib_{old}. Each lib_{old} has a corresponding set of lib_{new} in the collected library migration dataset. We then compare lib_{llm} with lib_{new} and classify the results into three categories:

- **Unchanged**: If lib_{llm} is the same as lib_{old}, the model fails to recommend a more appropriate alternative and retains the outdated or less optimal library.
- **Updated**: If lib_{llm} matches lib_{new}, indicating that the model successfully suggests an appropriate replacement.
- **Other Recommendation**: If lib_{llm} matches neither lib_{old} nor lib_{new}, suggesting that the model proposes a different library that is not in our predefined migration pairs.

The possible reason for **Other Recommendations** is that LLMs may interpret the problem differently from our predefined migration pairs. For example, consider the question **Can I connect to multiple databases in CherryPy?**[4], where the user's accepted answer uses `MySQLdb`. In this case, the expected migration might be to libraries like `PyMySQL`. However, an LLM suggests `sqlite3`, as it could interpret the problem based on the general concept of "database connections". Although the LLM's recommendation could still be valid in the context of the task at hand, `MySQLdb` and `sqlite3` do not constitute a valid migration pair because they serve different database management systems.

To evaluate the effectiveness of LLMs in making appropriate library recommendations, we introduce the following metrics:

- **Library Selection Relevance (LSR)**: measures whether LLMs select a library that matches either library (the replaced library or the new library) in the library migration pairs.

$$LSR = \frac{N_{updated} + N_{unchanged}}{N_{total}} \tag{1}$$

  This metric shows to what extent the LLM can hold an agreement with the real-world migration dataset in terms of the use of the library. The higher the value, the better the agreement between LLM and the real-world code.
- **Library Recommendation Precision (LRP)**: measures whether LLMs select an appropriate library when the LLM holds an agreement with the real-world migration dataset in terms of the use of the library.

$$LRP = \frac{N_{updated}}{N_{updated} + N_{unchanged}} \tag{2}$$

  Here, we focus on the proportion of selected appropriate libraries among all potential libraries in the real-world migration dataset. The higher the value, the better the performance of LLM.

After computing the relevant metrics, we perform a multi-dimensional analysis to evaluate the performance of LLMs.

IV. Experiment Results

To better understand how LLMs perform in selecting appropriate libraries, we analyze their recommendations from two main perspectives: libraries and problem descriptions. For libraries, we examine both the reasons behind their migration and their respective domains. Additionally, we perform prompt

[4]https://stackoverflow.com/questions/22824649/can-i-connect-to-multiple-databases-in-cherrypy

refinement to improve LLM performance. Specifically, we formulate and address the following three research questions:

- **RQ1.** How do different obsolescence factors and library domains affect LLMs' library selection performance?
- **RQ2.** How does the question description affect LLMs' library selection performance?
- **RQ3.** Can prompt refinement improve LLMs' library selection performance?

A. RQ1-Obsolete Libraries

Motivation. Intuitively, the ability of LLMs to select appropriate libraries should be related to the characteristics of the libraries. For example, the reason for migration and the domain of the library might have an impact on whether an LLM makes a suitable recommendation. To verify this assumption, we explore the library selection performance of LLMs on libraries with different characteristics. Specifically, we first evaluate the overall effectiveness of LLMs in selecting appropriate libraries. Then, we compare the performance of LLMs on different migration reasons and library domains.

Results. Table II shows the performance of LLMs on libraries with different migration reasons. We observe that the Library Selection Relevance (LSR) remains consistently high, around 90% across all models. This indicates that LLMs are generally effective at selecting libraries relevant to the given questions, and the LSR values show little variation concerning specific libraries. Moreover, LRP is around 70% among all LLMs. It shows that LLMs generally recommend appropriate libraries for a given task. There are differences between models, with closed-source models performing better than open-source ones. This suggests that proprietary data and more extensive training may provide some advantage in library recommendations, though the gap is not substantial.

Despite the overall moderate LRP, precision varies among different libraries. Some libraries are frequently recommended correctly, while others are not. This variation in performance highlights the need for a deeper analysis of how migration reasons and library domains influence LLMs' selection decisions. Understanding these factors can provide insights into their strengths and limitations, guiding improvements in LLM-based code generation and recommendation systems.

1) Different Obsolescence Factors: To assess whether LLMs can distinguish between different obsolescence scenarios and adjust their library recommendations accordingly, we analyze their performance based on various obsolescence factors.

Long-Term unmaintained Libraries. To answer the question about long-term unmaintained libraries, LLMs must identify whether a given library is no longer maintained or has been deprecated in favor of a more modern solution. When responding to questions about such libraries, the LLM must demonstrate an understanding of their obsolescence and recommend the appropriate alternative based on current best practices.

The results shown in table III indicate a contrast in LLM performance when dealing with questions related to outdated libraries versus libraries that are still actively evolving. Specifically, for libraries that have not been maintained for an extended period, the LPR reaches approximately 90%. In contrast, for libraries that remain in use but are gradually being replaced by newer solutions, the LPR is lower, around 56%.

LLMs perform well at identifying definitively outdated libraries, as their deprecation is well-documented and widely recognized. This stability enables accurate migration recommendations, often aligning with industry standards. In contrast, actively maintained libraries require more context-aware reasoning, making recommendations less straightforward.

Security-Driven Migrations. The libraries `simplejson`, `cgi`, `suds`, and `pickle`, each present distinct risks that impact their safe usage. Libraries with known security vulnerabilities present a unique challenge for LLMs when addressing migration-related questions. Unlike long-term unmaintained libraries, security-vulnerable libraries may still be actively maintained at the time of their vulnerability disclosure, meaning that the decision to migrate depends on factors such as the availability of patches, community responses, and the severity of the risk. These security risks necessitate not just migration but also an understanding of secure coding practices, which poses additional challenges for LLMs in generating appropriate recommendations.

The experimental results show varying correctness rates among libraries with known security vulnerabilities in the second part of Table II. The `simplejson` achieved a 98% precision, likely due to its clear migration path to the standard `json` module, which is widely recommended and more resistant to DoS attacks. The `cgi` and `suds` exhibited moderate precision of around 70%. The `cgi` module's deprecation is driven in part by its vulnerability to shell injection and lack of modern security protections, making the transition well-supported. Similarly, `suds` has a clear successor in `zeep`, which is a more secure alternative to handle XML external entity attacks.

However, `pickle` demonstrates the lowest LRP at only 37%, reflecting the complexity. Unlike other libraries, `pickle` does not have a universal alternative. Its replacement depends heavily on the specific use case. While `json`, `dill` and `joblib` can serve as safer serialization options, they lack `pickle`'s ability to serialize arbitrary Python objects, making the migration more nuanced.

Overall, these findings suggest that LLMs perform well when a library's deprecation has a clear and widely accepted alternative. However, they struggle with libraries like `pickle`, where security concerns necessitate not just a library change but also a fundamental shift in how data serialization is handled. This highlights the need for more contextual awareness in LLM-generated recommendations, particularly when dealing with security-sensitive library migrations.

Better Alternatives Exist. The existence of a better alternative often means that developers are encouraged to migrate, even though the original library may still function. However, the success of LLMs in recommending these newer libraries depends on how well they recognize and prioritize the superior

TABLE II
DETAILED PERFORMANCE IN SPECIFIC LIBRARIES AMONG DIFFERENT MIGRATION REASONS

Library	Total	Qwen		Llama		Deepseek		GPT-3.5-turbo		GPT-4o	
		%LSR	%LRP	%LSR	%LRP	%LSR	%LRP	%LSR	%LRP	%LSR	%LRP
optparse	17	76.47	100.00	88.24	93.33	82.35	100.00	82.35	100.00	82.35	100.00
MySQLdb	34	88.24	96.67	97.06	96.97	94.12	90.63	85.29	82.75	91.18	96.77
nose	18	83.33	46.67	88.89	25.00	83.33	53.33	83.33	73.33	77.78	64.29
PySide	50	98.00	97.96	92.00	97.83	98.00	100.00	86.00	97.67	98.00	100.00
xlwt	37	91.89	50.00	72.97	70.37	89.19	78.79	94.59	82.86	83.78	90.32
imp	16	75.00	100.00	87.50	100.00	87.50	100.00	87.50	100.00	75.00	100.00
BeautifulSoup	50	94.00	100.00	96.00	97.92	98.00	97.96	90.00	97.78	98.00	97.96
urlparse	34	82.35	100.00	91.18	100.00	85.29	100.00	76.47	100.00	82.35	100.00
httplib	28	95.24	96.00	100.00	92.86	82.14	100.00	92.86	96.15	85.71	100.00
xlrd	46	95.65	65.91	95.65	84.10	97.83	95.56	97.83	77.78	97.83	95.56
simplejson	17	58.82	100.00	70.59	100.00	58.82	100.00	58.82	90.00	58.82	100.00
cgi	20	60.00	41.67	80.00	93.75	70.00	78.57	80.00	68.75	65.00	76.92
suds	21	95.24	50.00	95.24	60.00	80.95	70.59	90.48	63.16	71.43	46.67
pickle	46	84.78	33.33	95.65	36.36	86.96	42.50	84.78	38.46	89.13	36.59
urllib	50	82.00	85.37	86.00	88.37	90.00	88.89	90.00	95.56	88.00	88.64
mechanize	50	100.00	82.00	100.00	82.00	100.00	80.00	96.00	93.75	100.00	98.00
distutils	34	97.06	90.91	82.35	57.14	79.41	81.48	88.23	90.00	73.53	92.00
pycurl	13	100.00	30.77	100.00	61.54	100.00	38.46	100.00	69.23	100.00	61.54
pytz	36	83.33	10.00	91.67	9.09	88.89	9.38	83.33	13.33	88.89	3.13
twisted	50	94.00	4.26	90.00	22.22	90.00	8.89	88.00	13.64	92.00	19.57
Total	667	88.76	68.75	90.55	72.85	89.36	75.00	87.86	75.94	87.86	77.82

TABLE III
COMPARISON BETWEEN LONG-TERM UNMAINTAINED AND STILL-MAINTAINED LIBRARIES

	Long-Term Unmaintained		Still-Maintained	
	%LSR	%LRP	%LSR	%LRP
Qwen	90.00	85.52	87.53	51.86
Llama	90.91	89.67	90.21	56.25
Deepseek	91.82	93.40	86.94	55.97
Gpt-3.5-Turbo	88.48	90.41	87.24	61.56
Gpt-4o	90.00	95.96	85.76	59.17

TABLE IV
THE LIBRARY DOMAIN CLASSIFICATION

	Web Development		Data Processing		System Utilities	
	%LSR	%LRP	%LSR	%LRP	%LSR	%LRP
Qwen	89.56	69.26	86.97	54.33	86.57	94.83
Llama	92.41	78.08	86.99	66.14	85.07	77.19
Deepseek	90.19	74.38	87.67	75.00	82.09	90.91
Gpt-3.5	89.24	78.37	88.36	68.21	86.57	94.83
Gpt-4o	89.24	78.72	86.99	75.60	76.12	96.08

alternatives.

The `requests` is a Python library designed for sending HTTP requests and facilitating interaction with web servers. Its popularity among developers is particularly notable in tasks involving web requests and API integration. In our experiment, the libraries relevant to the aforementioned domains included the `urllib`, `pycurl`, and `mechanize` libraries.

The high precision for questions related to three migrated libraries suggests that LLMs effectively recognize modern HTTP-handling alternative of `requests`, which is widely adopted. This can be attributed to the fact that their modern replacements are well-documented, widely used, and often recommended in official Python documentation. Similarly, `distutils` has been officially replaced by `setuptools`, making its migration path well-documented and easier for LLMs to correctly suggest the newer option.

Conversely, `twisted` and `pytz` showed the lowest precision. The `twisted`, an asynchronous networking framework, has no direct one-to-one replacement, requiring a shift to `asyncio`, which presents a more complex scenario because the migration involves not just swapping an API but also adapting to a fundamentally different asynchronous programming model. The `pytz`, on the other hand, has been largely replaced by the native `zoneinfo` module in Python 3.9+, but because `pytz` remains widely used in legacy code and some developers continue to rely on it, LLMs may struggle to confidently recommend `zoneinfo`, especially when the problem statement lacks clear context regarding Python version.

Finding 1: LLMs perform best with long-unmaintained libraries, while some security-driven and alternative-driven cases suffer from serialization risks and the lack of a clear, widely accepted alternative.

2) Different Library Domains: To thoroughly assess the ability to select appropriate libraries during code generation, it is crucial to analyze their performance in various problem domains. Different domains present distinct challenges in library selection due to varying rates of technological evolution, library lifecycle, and domain-specific best practices. The results are presented in IV.

Web Development. Problems in this domain typically involve tasks related to front-end and back-end development, including HTTP requests and responses, web scraping, web service integration, and server-side scripting. The web development ecosystem is dynamic and rapidly evolving, leading to a high turnover rate of libraries and frameworks. The migration trends in this domain reflect a broader industry shift toward higher-

level, developer-friendly libraries and frameworks.

Our experiments achieve about 75% precision, indicating a reasonably strong performance of the LLMs in adapting to the dynamic and fast-evolving ecosystem of web development. However, the gap from a perfect score suggests that challenges remain, particularly when handling less commonly used libraries such as `cgi` and `twisted`. The evolving nature of web standards and the frequent introduction of modern frameworks contribute to the model occasionally defaulting to older, outdated libraries when the prompt lacks clear and modern context.

Data Processing. This domain covers a wide range of problems related to data manipulation, serialization, and file handling, often focusing on scenarios such as reading and writing structured data and converting data formats.

The data processing ecosystem has undergone significant changes over the years, with many older libraries being replaced by more modern, efficient, and feature-rich alternatives. The evolution of data formats has driven a shift in data processing libraries from outdated solutions (e.g., `xlwt` to modern libraries `openpyxl`) that support contemporary formats `.xlsx`. Additionally, the security of data serialization is continuously driving the evolution of libraries.

In the Data Processing domain, the LRP is around 68%, highlighting the complexities and pitfalls. The lower performance can be attributed to several factors, including the LLMs' difficulty in adapting to changes in data formats and serialization standards. Particularly, the choice of serialization tools, such as avoiding `pickle` in favor of safer alternatives like `json`, `dill` or `joblib`, underscores the importance of security considerations that the model might not always prioritize accurately. And the differences among different models primarily stem from their varying sensitivity to the evolution of data formats.

System Utilities. In system utilities, libraries primarily focus on module management, command-line parsing, and package distribution. These libraries are all part of Python's standard library, but their relevance and usage have significantly evolved over time due to changes in Python's ecosystem, version updates, and shifts in community maintenance practices.

It shows the highest performance, with a LRP of 90%. This strong result suggests that the LLMs are highly effective in recognizing the deprecation of standard libraries and aligning with the recommended practices of newer Python versions. The transition from `imp` to `importlib`, from `optparse` to `argparse`, and the complete phasing out of `distutils` in favor of `setuptools` are well-reflected in the model's output. The high precision here is also due to the relatively stable and standardized evolution of system utilities in Python, as well as the clear guidance provided by Python's official documentation and PEP processes. And the low performance of Llama could indeed be attributed to the fact that `distutils` is officially removed in Python 3.12, but remains active and in use in earlier versions, such as Python 3.9, which is also commonly used now.

TABLE V
COMPARISON BETWEEN DIFFERENT QUESTION DESCRIPTION

	%LSR		%LRP	
	Type1	Type2	Type1	Type2
Qwen	84.20	97.14	82.35	48.09
Llama	88.68	93.83	84.57	53.51
Deepseek	86.79	93.83	86.68	56.14
GPT-3.5-turbo	86.32	90.53	84.97	60.91
GPT-4o	84.91	93.00	85.28	65.93

Type1:Functionality-Driven questions
Type2:Explicitly mentioned obsolete libraries

Finding 2: LLMs perform well in Web and System Utilities but struggle with Data Processing due to format changes and serialization issues.

B. RQ2- Question Description

Motivation. Problem descriptions in Stack Overflow vary significantly, some explicitly mention library-related concerns, while others focus on functionality without specifying a library. This variation may impact how LLMs interpret the question and select appropriate libraries. Therefore, we analyze the effect of different problem descriptions, comparing cases where library selection is explicitly guided versus those where the required library is only implied.

Methodology. In this study, library migration-related problems refer to programming questions or tasks in which the users' solution involves the use of a deprecated or migrated library. These problems may manifest in two primary forms:

- Functionality-Driven questions. The question description does not directly mention a specific library, but instead describes the desired functionality.
- Explicitly mentioned obsolete libraries. The statement explicitly references a specific library or API that has since been replaced or is no longer recommended.

This categorization allows for a structured analysis of how well LLMs can not only generate code but also make informed decisions when selecting libraries. To enhance the accuracy of our classification, we invite two experienced Python developers to perform double annotation of the question description. The inter-annotator agreement, measured by Cohen's kappa, reaches an impressive 0.91, indicating a high level of consistency between the two annotators. For cases where discrepancies arose, the final problem type is determined through further discussions between the authors and the two developers, ensuring a reliable classification.

Result. The detail of the results is shown in Table V.

1) Functionality-Driven questions: In such cases, LLMs infer the appropriate library from the described functionality, such as HTTP request handling. The prompts focus on expected outcomes rather than specifying libraries, relying on the model's knowledge to determine the best choice.

For example, a question might state, 'Scraping interactive website'[5] without mentioning any specific scraping libraries. LLMs are expected to identify appropriate libraries such as `requests` based on their understanding of web scraping techniques. One of the main challenges for LLMs in handling functional problems is ensuring that they select libraries that not only fulfill the functionality described but are also up-to-date and efficient. Similarly to the question mentioned above, LLMs might generate code using `urllib`, even when newer and more efficient options such as `requests` are available. Moreover, ambiguity in the descriptions of functional problems can lead to varying interpretations. If the question is vague or imprecise, the models might resort to outdated or less optimal libraries, even those that are not relevant.

In our analysis, we find that Functionality-Driven questions achieve a precision of around 84%, which reflects the LLMs' ability to focus on fulfilling the core functional requirements but also selecting the appropriate libraries. This higher precision suggests that when the prompt emphasizes the desired functionality without being too prescriptive about the libraries to be used, the LLMs perform significantly better. This allows the model to choose modern, actively supported libraries that are better aligned with current best practices, ensuring not only functionality but also efficiency, maintainability, and security.

2) Explicitly mentioned obsolete libraries: These questions occur when the problem statement directly refers to a specific library or API that is no longer recommended or has been replaced. This scenario demands that the LLMs not only recognize the deprecated status of the mentioned library but also proactively suggest a more suitable modern alternative. The challenge is heightened by the need for the model to balance the following user instructions with offering more effective and current recommendations, showcasing a deeper level of contextual awareness and critical decision-making.

Similarly to the question 'Python optparse defaults vs function defaults'[6], LLMs can choose to strictly follow the instructions of the prompt and use `optparse`, as was directly mentioned. However, this would lead to the usage of a deprecated library, as `optparse` has been officially replaced by `argparse` in modern Python versions. To avoid this issue, we adopt a strategy that focuses on functional requirements rather than exact libraries in the description of the question.

In contrast, Explicitly Mentioned Obsolete Libraries show a much lower precision of around 55%. This outcome highlights the model tends to prioritize the exact libraries mentioned in the prompt, even if those libraries are no longer optimal or widely supported. However, compared to the functionality-driven questions, its LSR is slightly higher. Functionality-driven questions, while focusing on the core task, may leave some room for ambiguity, particularly if the functional requirements are not described with sufficient clarity or precision.

[5]https://stackoverflow.com/questions/36438539/scraping-interactive-website

[6]https://stackoverflow.com/questions/1512242/python-optparse-defaults-vs-function-defaults

TABLE VI
THE LRP AFTER PROMPT REFINEMENT

	LLM	No_Refinement	Strategy_1	Strategy_2
twisted	Qwen	4.26	19.57	95.92
	Llama	22.22	74.29	100.00
	Deepseek	8.89	69.23	98.00
	GPT-3.5	13.64	83.33	97.83
	GPT-4o	19.57	91.18	97.92
pytz	Qwen	10.00	96.77	100.00
	Llama	9.09	100.00	100.00
	Deepseek	9.38	100.00	100.00
	GPT-3.5	13.33	34.48	100.00
	GPT-4o	3.13	100.00	100.00
Total	Qwen	68.75	90.05	97.21
	Llama	72.85	97.34	98.90
	Deepseek	75.00	95.92	98.17
	GPT-3.5	75.94	91.03	98.90
	GPT-4o	77.82	97.90	97.48

Within the Explicitly mentioned obsolete libraries category, the decline in precision is not uniform across all libraries. Some deprecated libraries, such as `PySide2`, have clear and consistent upgrade paths to `PySide6`. Since these successors retain a similar name, LLMs are likely to recognize them as appropriate choices. This results in a higher LRP for such cases compared to libraries that require more significant shifts.

At the same time, closed-source models such as GPT-4o, generally outperform open-source models like Qwen and Llama in handling explicitly mentioned deprecated libraries. This is likely due to their superior contextual understanding and better knowledge updates. In particular, closed-source models are more effective in recognizing deprecated libraries and suggesting modern replacements. In contrast, open-source models often struggle with such replacements.

Finding 3: LLMs perform better when questions focus on functionality, while explicit mentions of deprecated libraries cause hesitation between updating or following the prompt.

C. RQ3- Prompt Refinement

Motivation. To improve the accuracy of LLMs' library recommendations, we refine the prompt to provide clearer guidance. Based on our previous analysis, LLMs struggle with libraries lacking well-documented migration paths or facing serialization security risks. Explicitly mentioned obsolete libraries prompt may lead LLMs to overlook better alternatives. By explicitly emphasizing library selection criteria and reducing ambiguity, we help LLMs generate more precise and reliable recommendations aligned with real-world migration needs.

Result. We present the results of the LRP after prompt refinement strategies in Table VI, along with the outcomes for two low LRP libraries including `twisted` and `pytz`.

1) Indicating the deprecated status: In our first refinement strategy, we focus on explicitly indicating the deprecated status of certain libraries within the prompt. After indicating the deprecated status of libraries, we observe improvements in the

precision of LLMs. It indicates that the explicit indication of deprecated libraries helps guide LLMs toward more relevant library selections.

However, we also notice some specific patterns worth discussing. The lower LRP of *Qwen* in handling twisted-related problems suggests that the model struggles with balancing the explicitly mentioned obsolete library in the prompt against the modern library selection, particularly since many unresolved issues explicitly mentioned the `twisted`. The model *Qwen* demonstrates stronger adherence to explicitly stated libraries rather than dynamically adapting to the latest choices, which aligns with the findings of our RQ2 experiments. In the case of *GPT-3.5-Turbo*, its lower LRP on `pytz` might reflect the model could have been influenced by training data, which makes it more likely to suggest the obsolete library, despite its deprecation in newer Python versions. In comparison to more recent models like *GPT-4o*, which have been trained on more up-to-date data or fine-tuned to better recognize such transitions, *GPT-3.5-Turbo* may still exhibit a preference for older libraries when prompts are less clear or more ambiguous.

2) Indicating the recommended alternatives: In our second refinement strategy, we build on the first strategy by not only explicitly marking the deprecated status of certain libraries, but also proactively informing the LLMs about the currently popular and recommended alternatives.

The results demonstrate a significant improvement across all tested scenarios, indicating that providing explicit guidance on modern library choices effectively improves the LLMs toward selecting the appropriate libraries. This suggests that LLMs, while capable of recognizing obsolete libraries, benefit from additional context about the current software landscape to make optimal recommendations.

While these strategies enhance LLMs' ability to select appropriate libraries, they assume users are aware of deprecated libraries and suitable replacements. This places a burden on users to stay updated on library evolution. To mitigate this, LLMs should integrate dynamic retrieval mechanisms to provide up-to-date recommendations, reducing reliance on user knowledge and improving adaptability to evolving best practices.

Finding 4: Prompt refinement greatly improves performance but has limitations, requiring high-quality user input and revealing hesitation in some models, as well as the impact of training data.

V. Discussion

In this section, we first discuss several implications based on the findings in our study, and then describe the threats to validity.

A. Implications

Verifying Library Choices in LLM-Generated Code. Developers should carefully review library selections, especially for security-sensitive tasks like data serialization. Our study found cases where LLMs suggested deprecated libraries, underscoring the need for caution. LLM platforms can also integrate detection tools to recommend better alternatives, ensuring safer and more up-to-date code.

Enhancing training data with more migration cases specific to certain obsolete libraries. In our study, while LLMs performed well for most libraries, certain migration cases showed significantly lower precision. LLMs may struggle with specific migrations due to insufficient exposure to those migrations during training. By incorporating more representative migration examples, especially for challenging cases, LLMs could develop a deeper understanding of appropriate replacements, leading to more reliable recommendations.

Prompt design is crucial for library migration tasks. Users should describe functionality rather than specifying a library, especially if its status is uncertain. Explicitly stating deprecation status, alternatives, security concerns, and performance needs helps LLMs make better recommendations. Our results show that clear prompts lead to more accurate library selection.

Enhancing prompt interpretation mechanisms to reduce reliance on specific library names and strengthen their understanding of functional requirements. Our findings suggest that LLMs perform better when prompted with functionality-driven questions rather than explicit mentions of obsolete libraries. Incorporating techniques like Retrieval-Augmented Generation (RAG) [43] to fetch up-to-date library migration information before generating responses could further improve recommendation accuracy.

Our study provides insights into LLMs' performance in library migration but has limitations. It focuses on a specific set of libraries, missing broader migration trends. Future work could include more diverse libraries, other ecosystems, and larger datasets. The absence of configuration files in Stack Overflow posts also complicates dependency tracking, suggesting the need for real-world project integration. Further research should explore optimizing LLMs for user-specific contexts to enhance practical code recommendations.

B. Threats to Validity

Internal Validity Threats: Internal validity may be affected by potential errors or biases introduced during data collection, analysis, or interpretation. To mitigate this, multiple researchers were involved and the results were cross-checked, and Cohen's kappa coefficient was calculated to ensure consistency in data labeling. In addition, the analysis of experimental results depends on the collected library migration pairs, and any incompleteness in the migration data may influence the conclusions. To reduce this risk, the migration pairs were collected from well-established datasets, and the migratable target libraries were manually consolidated and supplemented to improve the completeness of the migration space.

External Validity Threats: External validity concerns arise from the possibility of LLMs generating hallucinations, or incorrect or irrelevant responses that are not grounded in reality. To address this, we incorporate multiple LLMs into the

study to reduce the impact of model-specific biases. Moreover, we conducted multiple rounds of questioning to further ensure the robustness of the results. In terms of the library migration cases, careful manual filtering is done to ensure that only relevant libraries and questions are selected for the study, minimizing the risk of irrelevant or misleading results.

VI. Conclusion

In this study, we explore how LLMs handle questions related to obsolete libraries, particularly focusing on how they select appropriate libraries during code generation. We investigate different factors that influence the effectiveness of LLMs in addressing these types of questions. First, the reasons behind library obsolescence and the domain of different libraries. Some libraries, such as those with serialization risks and lacking well-documented migration paths, pose significant challenges for LLMs. Second, the type of question description, whether functionality-driven or explicitly mentioning obsolete libraries, with the latter causing a decrease in precision due to the models' tendency to focus too much on the outdated libraries mentioned. Based on these findings, we proposed two strategies for refining the prompt to improve model performance: clearly stating the outdated status of libraries and providing modern alternatives. These strategies are effective in boosting the LLMs' ability to recommend suitable libraries and improve their overall performance in addressing library migration-related problems.

Acknowledgment

This work is supported by the Zhejiang Pioneer (Jianbing) Project (2025C01198), the National Science Foundation of China (No.62372398, No.72342025), and the Fundamental Research Funds for the Central Universities (No.226-2025-00067).

References

[1] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: code generation using transformer," in *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 1433–1443.

[2] 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 *ISSTA*. ACM, 2022, pp. 39–51.

[3] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No need to lift a finger anymore? assessing the quality of code generation by chatgpt," *IEEE Trans. Software Eng.*, vol. 50, no. 6, pp. 1548–1584, 2024.

[4] 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 *ICLR*. OpenReview.net, 2023.

[5] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI*. ACM, 2014, pp. 419–428.

[6] 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 *ICSE*. ACM, 2024, pp. 79:1–79:13.

[7] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *ISSTA*. ACM, 2024, pp. 819–831.

[8] B. A. Stoica, U. Sethi, Y. Su, C. Zhou, S. Lu, J. Mace, M. Musuvathi, and S. Nath, "If at first you don't succeed, try, try, again...? insights and llm-informed tooling for detecting retry bugs in software systems," in *SOSP*. ACM, 2024, pp. 63–78.

[9] L. Zhong and Z. Wang, "Can LLM replace stack overflow? A study on robustness and reliability of large language model code generation," in *AAAI*. AAAI Press, 2024, pp. 21 841–21 849.

[10] L. Li, S. Geng, Z. Li, Y. He, H. Yu, Z. Hua, G. Ning, S. Wang, T. Xie, and H. Yang, "Infibench: Evaluating the question-answering capabilities of code large language models," in *NeurIPS*, 2024.

[11] S. Dou, H. Jia, S. Wu, H. Zheng, W. Zhou, M. Wu, M. Chai, J. Fan, C. Huang, Y. Tao, Y. Liu, E. Zhou, M. Zhang, Y. Zhou, Y. Wu, R. Zheng, M. Wen, R. Weng, J. Wang, X. Cai, T. Gui, X. Qiu, Q. Zhang, and X. Huang, "What's wrong with your code generated by large language models? an extensive study," *CoRR*, vol. abs/2407.06153, 2024.

[12] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X. D. Le, and D. Lo, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, pp. 116:1–116:26, 2024.

[13] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *SP*. IEEE, 2022, pp. 754–768.

[14] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" in *CCS*. ACM, 2023, pp. 2785–2799.

[15] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos, "An empirical study of code smells in transformer-based code generation techniques," in *SCAM*. IEEE, 2022, pp. 71–82.

[16] E. L. Vargas, M. F. Aniche, C. Treude, M. Bruntink, and G. Gousios, "Selecting third-party libraries: the practitioners' perspective," in *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 245–256.

[17] J. Zhang, Q. Luo, and P. Wu, "A multi-metric ranking with label correlations approach for library migration recommendations," in *SANER*. IEEE, 2024, pp. 183–191.

[18] R. G. Kula, D. M. Germán, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? - an empirical study on the impact of security advisories on library migration," *Empir. Softw. Eng.*, vol. 23, no. 1, pp. 384–417, 2018.

[19] S. Kabinna, C. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: a case study for the apache software foundation projects," in *MSR*. ACM, 2016, pp. 154–164.

[20] M. Islam, A. K. Jha, I. Akhmetov, and S. Nadi, "Characterizing python library migrations," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 92–114, 2024.

[21] S. Dai, C. Xu, S. Xu, L. Pang, Z. Dong, and J. Xu, "Bias and unfairness in information retrieval systems: New challenges in the llm era," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 6437–6447.

[22] J. Latendresse, S. Khatoonabadi, A. Abdellatif, and E. Shihab, "Is chatgpt a good software librarian? an exploratory study on the use of chatgpt for software library recommendations," *CoRR*, vol. abs/2408.05128, 2024.

[23] M. Islam, A. K. Jha, S. Nadi, and I. Akhmetov, "Pymigbench: A benchmark for python library migration," in *MSR*. IEEE, 2023, pp. 511–515.

[24] H. Gu, H. He, and M. Zhou, "Self-admitted library migrations in java, javascript, and python packaging ecosystems: A comparative study," in *SANER*. IEEE, 2023, pp. 627–638.

[25] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empir. Softw. Eng.*, vol. 24, no. 1, pp. 381–416, 2019.

[26] S. Mujahid, R. Abdalkareem, and E. Shihab, "What are the characteristics of highly-selected packages? A case study on the npm ecosystem," *J. Syst. Softw.*, vol. 198, p. 111588, 2023.

[27] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in java projects," in *ICSME*. IEEE, 2020, pp. 35–45.

[28] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz, "Categorizing developer information needs in software ecosystems," in *Proceedings of the 2013 international workshop on ecosystem architectures*, 2013, pp. 1–5.

[29] A. A. Sawant, R. Robbes, and A. Bacchelli, "To react, or not to react: Patterns of reaction to API deprecation," *Empir. Softw. Eng.*, vol. 24, no. 6, pp. 3824–3870, 2019.

[30] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated python library apis are (not) handled," in *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 233–244.

[31] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *CCS*. ACM, 2017, pp. 2187–2200.

[32] M. Hu and Y. Zhang, "An empirical study of the python/c API on evolution and bug patterns," *J. Softw. Evol. Process.*, vol. 35, no. 2, 2023.
[33] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, and M. Zhou, "A multi-metric ranking approach for library migration recommendations," in *SANER*. IEEE, 2021, pp. 72–83.
[34] C. Chen, Z. Xing, Y. Liu, and K. O. L. Xiong, "Mining likely analogical apis across third-party libraries via large-scale unsupervised API semantics embedding," *IEEE Trans. Software Eng.*, vol. 47, no. 3, pp. 432–447, 2021.
[35] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 220:1–220:79, 2024.
[36] 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 *IJCAI*. ijcai.org, 2022, pp. 2369–2375.
[37] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the quality of github copilot's code generation," in *PROMISE*. ACM, 2022, pp. 62–71.
[38] J. He and M. T. Vechev, "Controlling large language models to generate secure and vulnerable code," *CoRR*, vol. abs/2302.05319, 2023.
[39] X. She, Y. Liu, Y. Zhao, Y. He, L. Li, C. Tantithamthavorn, Z. Qin, and H. Wang, "Pitfalls in language models for code intelligence: A taxonomy and survey," *CoRR*, vol. abs/2310.17903, 2023.
[40] C. Wang, K. Huang, J. Zhang, Y. Feng, L. Zhang, Y. Liu, and X. Peng, "How and why llms use deprecated apis in code completion? an empirical study," *CoRR*, vol. abs/2406.09834, 2024.
[41] Z. Yao, D. S. Weld, W. Chen, and H. Sun, "Staqc: A systematically mined question-code dataset from stack overflow," in *WWW*. ACM, 2018, pp. 1693–1703.
[42] 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," in *NeurIPS*, 2023.
[43] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.