From Misuse to Mastery: Enhancing Code Generation with Knowledge-Driven AI Chaining

1st Xiaoxue Ren
School of Software Technology
Zhejiang University
Hangzhou, China
xxren@zju.edu.cn

2nd Xinyuan Ye
School of Computing
Australian National University
Canberra, Australia
xinyuan.ye@anu.edu.au

3rd Dehai Zhao
CSIRO's Data61
Sydney, Australia
dehai.zhao@data61.csiro.au

4th Zhenchang Xing
CSIRO's Data61
& Australian National University
Sydney, Australia
zhenchang.xing@data61.csiro.au

5th Xiaohu Yang
College of Computer Science and Technology
Zhejiang University
Hangzhou, China
yangxh@zju.edu.cn

***Abstract*—Large Language Models (LLMs) have shown promising results in automatic code generation by improving coding efficiency to a certain extent. However, generating high-quality and reliable code remains a formidable task because of LLMs' lack of good programming practice, especially in exception handling. In this paper, we first conduct an empirical study and summarize three crucial challenges of LLMs in exception handling, i.e., incomplete exception handling, incorrect exception handling and abuse of try-catch. We then try prompts with different granularities to address such challenges, finding fine-grained knowledge-driven prompts works best. Based on our empirical study, we propose a novel Knowledge-driven Prompt Chaining-based code generation approach, name KPC, which decomposes code generation into an AI chain with iterative check-rewrite steps and chains fine-grained knowledge-driven prompts to assist LLMs in considering exception-handling specifications. We evaluate our KPC-based approach with 3,079 code generation tasks extracted from the Java official API documentation. Extensive experimental results demonstrate that the KPC-based approach has considerable potential to ameliorate the quality of code generated by LLMs. It achieves this through proficiently managing exceptions and obtaining remarkable enhancements of 109.86% and 578.57% with static evaluation methods, as well as a reduction of 18 runtime bugs in the sampled dataset with dynamic validation.**

***Index Terms*—Large Language Model, Code Generation, Knowledge-driven Prompt, API Misuse**

I. Introduction

Large Language Models (LLMs) have gained significant attention in the field of natural language processing (NLP) for their ability to generate coherent and contextually relevant text [1]–[6]. Recently, there has been growing interest in using LLMs for code generation. LLMs, such as Codex [7], AlphaCode [8], CODEGEN [9], and INCODER [10], use sophisticated algorithms to generate code based on natural language input, enabling developers to considerably automate the coding process. The use of LLMs in code generation holds great promise for improving software development processes by reducing the time and effort required for coding tasks and increasing developers' productivity [11], [12].

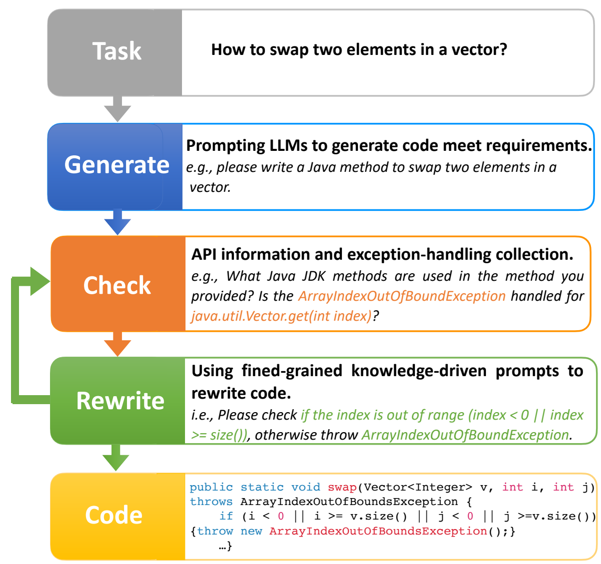


Fig. 1. High-level Overview of KPC-based Code Generation.

Nevertheless, generating high-quality and reliable code remains a formidable task [12]–[15]. One significant limitation of LLMs in code generation is the lack of good programming practice. LLMs are based on statistical patterns and lack the ability to understand the underlying logic and structure of programming languages. Therefore, they may generate code that is syntactically correct but not semantically correct, resulting in poor programming practices. Particularly, LLMs have challenges in generating reliable, maintainable and robust code in terms of exception handling, which has been proven to be essential in software development [16]–[18].

Three crucial exception-handling challenges of code generated by LLMs have been summarized by our empirical study in Section II-A, including ***incomplete exception handling***, ***incorrect exception handling***, and ***abuse of try-catch***. For example, code in Figure 2 (A) demonstrates an incomplete exception handling example, as both java.util.Vector.get(int index) and java.util.Vector.set(int index, E element) may encounter exceptions. Code in Figure 2 (B) shows an incorrect exception handling example, in which ArrayIndexOutOfBoundsException should be handled instead of IndexOutOfBoundsException. Moreover, the usage of try-catch statements in Figure 2 (C) is not considered to be the best practice for exception handling. Such challenges will result in serious consequences, including software crashes and reliability and security issues [19], [20].

In order to obtain ideal results from LLMs, prompt engineering [21]–[24] is one of the most effective solutions and has been widely studied. This technique focuses on designing proper prompts that guide LLMs to take desired actions, with the goal of improving the quality of output. In the context of software engineering, prompt engineering can help LLMs complete a series of development tasks with improved user experiences, reduced errors and support costs, and increased user adoption and satisfaction [23], [25]. In our work, a well-designed prompt can make LLMs understand the expected behavior of coding and find the best solution to handle exceptions in the code. With different prompts in the empirical study (see Section II-B), we can explore various possible scenarios and edge cases, and enable LLMs to iterate and address potential exceptions autonomously. Figure 2 (B), (C), and (D) show responses of LLMs with the different granularities of prompts, where we find fine-grained knowledge-driven prompts are most effective in preventing incomplete or incorrect exception handling, as well as the abuse of try-catch statement.

In this paper, we propose a novel **K**nowledge-driven **P**rompt **C**haining-based code generation approach, named KPC, which utilizes fine-grained exception-handling knowledge extracted from API documentation to assist LLMs in code generation. This approach follows the divide-and-conquer strategy and iterative coding practice, which improves the generated code from misuse to mastery by formulating a chain of modular prompts for LLMs. To this end, we first construct an API knowledge base from official API documentation [26]. Then, based on the original code generated by LLMs, we chain knowledge-driven prompts to check whether there exists any unhandled exception in the generated code. Whenever such exceptions are identified, we rewrite the code with fine-grained exception-handling knowledge prompts until all exceptions are properly handled, as shown in Figure 1.

The API knowledge base constructed for our KPC-based code generation approach includes 19,057 exceptions with corresponding conditions, covering 11,477 Java SDK & JDK APIs, which is available in our replication package [27]. We conduct experiments to evaluate the efficiency and effectiveness of our KPC-based code generation approach for exception handling. Our experiments are based on 3,079 Java coding tasks obtained from Java SDK & JDK API functional descriptions [26]. To evaluate the efficiency, we analyze the distributions of unhandled exceptions and checking-rewriting loops. We find that our KPC-based approach can efficiently handle exceptions for the majority of coding tasks using a small number of iterative checking-rewriting loops. Specifically, 90.35% of the generated code can be effectively handled within ten loops when dealing with exceptions. To evaluate the effectiveness, we use three evaluation methods, including both static and dynamic validations. For static validations, on one hand, we leverage the ability of LLMs to automatically evaluate the exception-handling practice of the generated code. On the other, we manually review a subset of the tasks. For dynamic validation, we employ EvoSuite [28] to generate test cases for detecting runtime bugs. The results of all three evaluation methods show that our KPC-based approach outperforms the state-of-the-art code generator (i.e., ChatGPT) in exception handling. Specifically, we observed improvements of 109.86% and 578.57% in the two static evaluation methods, and a reduction of 18 runtime bugs in the dynamic validation. In addition to these evaluations, we also conduct a user study to determine how our KPC-based approach can help developers in practice. Results show that our approach can effectively remind developers of exception-handling specifications, leading to a high level of correctness (75.00%) in handling potential exceptions.

In summary, we make the following contributions:

- We are the first to propose the fine-grained knowledge-driven prompt chaining approach (i.e., KPC) for LLMs in code generation tasks.
- We conduct an empirical study to investigate the challenges of LLMs in exception handling, and design the most appropriate prompts to address these challenges.
- Experimental results demonstrate that our KPC-based approach is highly efficient and effective in handling exceptions. And our user study also shows KPC can assist developers in effectively handling exceptions in practice.
- We open source our replication package [27], including the dataset, the source code of KPC, and experimental results, for follow-up works.

II. An Empirical Study on Exception Handing for LLMs Code Generation

Despite the great success of LLMs in the software engineering field, there still exist some issues that have not been well studied in the literature, leading to unexpected errors when solving tasks such as code generation in this work. Therefore, we conduct an empirical study to facilitate the understanding of knowledge-driven prompts chaining for LLMs in code generation by answering the following two research questions.

- **RQ1: What challenges do LLMs face in handling exceptions when generating code?** By analyzing the generated code from LLMs, we manually summarize the challenges related to exception handling.
- **RQ2: How to help LLMs address the challenges effectively?** After summarizing the challenges, we leverage prompt engineering to address them by trying different prompts related to exception handling.

A. Challenges of LLMs in Exception Handling (RQ1)

To explore the challenges that LLMs encounter in handling exceptions, we first collect 92 Java code generation tasks from Tutorialspoint [29], which may have potential exceptions. We then rephrase (see Section III-B1) the tasks as inputs of ChatGPT to generate Java code. Afterward, two of the authors collaborate to review the generated code manually with the goal of identifying the challenges that LLMs might encounter when handling exceptions in code generation tasks. Both authors have more than five years of experience in Java development and a thorough understanding of official Java API documentation. We summarize the challenges into three crucial aspects, which are elaborated as follows.

1) Incomplete exception handling: Exceptions are an essential part of programming that signal the occurrence of an error during program execution. It's common for a single code snippet to encounter multiple exceptions. However, failing to handle all of these exceptions can have severe consequences, such as crashes, data corruption, and security vulnerabilities. Unfortunately, results of the empirical study show that the incomplete exception handling phenomenon is very common in the code generated by ChatGPT, since there are 88.04% (81 out of 92) of the coding tasks have such issues. In fact, such a fault is very likely to be avoided because official documentation clearly describes the exception-handling specifications for each API. For example, the code in Figure 2 (A) involves the incomplete exception-handling issues for the two APIs, java.util.Vector.get(int index) and java.util.Vector.set(int index, E element), and we can find the exception-handling specification *"Throws: ArrayIndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())"* in Java API documentation [30], [31].

2) Incorrect exception handling: This issue occurs when an exception is handled incorrectly in the code, such as catching the wrong type of exceptions. According to the statistical results of our empirical study, 4.35% (4 out of 92) of the code examples generated by ChatGPT contain such issues. Although these fault examples account for a small part of the overall coding tasks, we cannot ignore this issue, especially for those code examples that are able to run without any error, which would be very hard to localize the root cause of the problem. Figure 2 (B) shows an example of incorrect exception handling, which catches IndexOutOfBoundsException. Referring to Java API documentation [30], [31], it is recommended to use ArrayIndexOutOfBoundsException instead of IndexOutOfBoundsException in this scenario, as ArrayIndexOutOfBoundsException is a subclass of IndexOutOfBoundsException and is specifically designed for cases where an array or vector index is out of bounds.

3) Abuse of try-catch: It refers to situations where try-catch statements are used excessively or inappropriately, e.g., using them to handle non-exceptional situations or to handle exceptions that should be handled by a specific piece of code. We observe from the empirical study that ChatGPT attempts to use try-catch statements to solve 5.43% (5 out of 92) of the coding tasks, which constitutes a small percentage of the

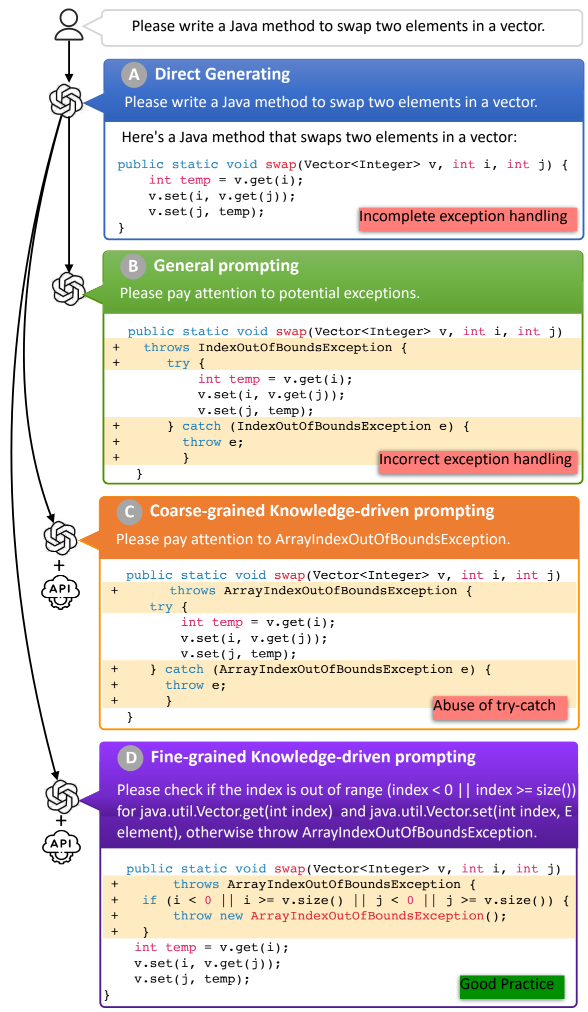


Fig. 2. A walkthrough example demonstrating the difference between direct generating (A), general prompting (B), and both coarse-grained (C) and fine-grained (D) knowledge-driven promoting, using the coding task from Tutorialspoint.

whole dataset, but we can not take this issue lightly, as this practice will reduce the readability and maintainability of the code. Typically, a try-catch statement enables code to run without errors, but it may not always lead to logically sound code. Hence, it's often necessary to find a more effective solution to handle exceptions. For example, the code in Figure 2 (C) implements the function of throwing exceptions if there exists any error with the input, but it lacks the ability to determine the explicit conditions to throw exceptions. Instead, using an if-condition checking mechanism implemented in Figure 2 (D), is generally considered to be a better practice in the software development process.

> **Answer to RQ1:** Code generated by LLMs mainly encounters three challenges, including *incomplete exception-handling*, *incorrect exception-handling* and *abuse of try-catch statements*.

B. Prompt Engineering for Exception Handling (RQ2)

As shown in RQ1, LLMs have great potential to solve software engineering tasks such as code generation, but the three challenges are still significant barriers to exception handling. It is believed that prompt engineering is the key to unlocking the magic of LLMs to generate high-quality solutions. However, prompt engineering involves much more than just developing prompts. It requires a deliberate and systematic approach to designing and refining prompts and the underlying data structures that control LLMs. In this RQ, we apply a three-granularity (i.e. general, coarse-grained and fine-grained) prompt design strategy, which adds exception-handling relevant information to prompts gradually, to investigate the impact of prompts to exception handling in LLM-based code generation tasks.

1) General prompting: The most straightforward way to make LLMs pay attention to the exceptions in code is asking them to do so directly, which is named as general prompting in our work. This method only uses generic warnings to raise concerns about exception to LLMs without providing exception types and conditions. Figure 2 (B) shows an example of general prompting, in which we give the prompt *"Please pay attention to potential exceptions."* to ChatGPT. By reviewing the code generated in the last round, ChatGPT realizes there exists potential exceptions involving "out of bounds" and solves this problem by catching `IndexOutOfBoundsException`. Although this method tries to solve problems with minimal effort, it can solve a part of the exception-handling issues. According to the experiment results in Table I, general prompting helps ChatGPT to reduce incomplete exception handling examples from 81 to 35. Nevertheless, this solution may introduce other problems, such as incorrect exception handling (Section II-A2) and abuse of try-catch (Section II-A3) because the general prompting fails to provide sufficient information about the types of exceptions in the code and appropriate measures to handle them. As presented in Table I, comparing general prompting with direct generating, the number of incorrect exception handling increases from 4 to 23, and the number of abusing try-catch goes up from 5 to 20. Without comprehensive exception-handling specifications, LLMs create low-quality solutions to the coding tasks, which go against the best practices in the software development process.

2) Coarse-grained prompting: Compared with general prompting which provides zero exception-handling information, coarse-grained prompting offers a part of such information to the input prompts of ChatGPT. Specifically, this strategy applies exception-handling specifications from API documentation and indicates what exception ChatGPT should pay attention to exactly. As shown in Figure 2 (C), the prompt is updated to *"Please pay attention to ArrayIndexOutOfBoundsException."* from the naive version. ChatGPT understands clearly that it should focus on the given exception and revises the code by catching `ArrayIndexOutOfBoundsException` with a `try-catch` block. Based on the statistical results in Table I, coarse-grained prompting achieves a significant reduction of incomplete exception-handling issues by 95.06% (from 81 to 4). Moreover, the number of incorrect exception handling is completely eliminated. Theoretically, if ChatGPT is provided with explicit exception information, there will be no chance to exist incomplete or incorrect exception handling. We investigate the reason why there still exist four incomplete exception-handling examples and find they use APIs whose exception-handling specifications are incomplete in the official documentation. For example of `java.lang.String.split(String regex)` [32], neither of the object string nor parameter regex can be `null`, otherwise `NullPointerException` will happen. Therefore, the prompts are not given such information, and the corresponding exceptions cannot be handled. We also observe some side effects of this strategy from Table I, in which the number of abusing `try-catch` statements by coarse-grained prompting increases from 5 to 41 compared with direct generating. The main reason lies in the lack of condition information of the exceptions, so ChatGPT is unable to handle them in appropriate manners.

3) Fine-grained prompting: From the previous prompting strategies, we find that the more detailed information given to prompts, the higher quality can ChatGPT solve the problems. Therefore, in the final edition of prompt engineering, we introduce all the exception-handling information obtained from official Java API documentation and propose fine-grained prompting strategy, which contains not only the exception types, but also corresponding conditions to trigger the exceptions. As shown in Figure 2 (D), the prompt is formulated as *"Please check if the index is out of range (index $<$ 0 || index $>=$ size()) for java.util.Vector.get(int index) and java.util.Vector.set(int index, E element), otherwise throw ArrayIndexOutOfBoundsException."*. ChatGPT leverages the key information from the prompt and provides the code solution by adding an `if-condition` statement to throw an `ArrayIndexOutOfBoundsException`, which is considered as a type of best practice in the software development process. Table I illustrates the experimental results of fine-grained prompting, from which we observe that this strategy solves the three challenges perfectly, with incorrect exception handling and abuse of `try-catch` to be completely eliminated. Similar to the result of coarse-grained prompting, there still exist three (instead of four) examples that have incomplete exception handling issues, which share the same reason stated previously. One of the coding tasks is solved by ChatGPT accidentally, and the primary cause is the uncertainty of ChatGPT's outputs. Even the same prompt could trigger different activities of this model, and we will discuss it in Section VII.

Answer to RQ2: Fine-grained prompts with specific exceptions and corresponding conditions enable LLMs to identify and handle exceptions accurately, which follows the best practice of the software development process.

III. The Approach

Figure 3 illustrates the overall framework of our approach, which consists of one off-line phase and one online phase: API knowledge base construction (i.e., step-0 in Figure 3) is off-line and KPC-based code generation (i.e., step-1, step-2 and step-3 in Figure 3) is online. Specifically, we first construct an API knowledge base from Java official documentation. Then,

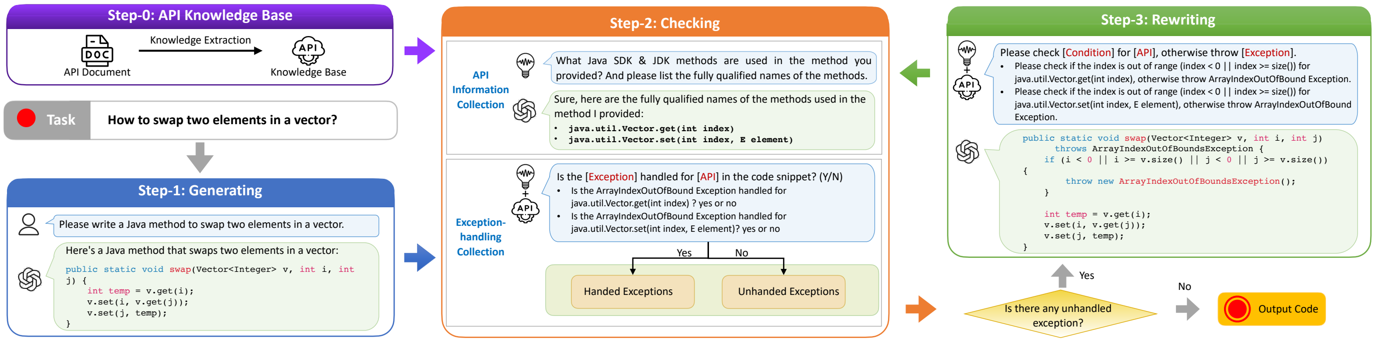


Fig. 3. The Overall Framework of KPC-based Code Generation

TABLE I
QUALITY STATISTICS OF GENERATED CODE WITH VARIOUS PROMPTS

Quality of code	Direct generating	General prompting	Knowledge-driven prompting	
			Coarse-grained	*Fine-grained*
Incomplete exception handling	81	35	4	3
Incorrect exception handling	4	23	0	0
Abuse of try-catch	5	20	41	0
Good practice	2	14	47	89

Note: There is no overlap among the three challenges, and we label them according to a priority of* ***incomplete exception handling*, ***incorrect exception handling*** *and* ***abuse of try-catch***.

we work on top of ChatGPT and design a Knowledge-driven Prompt Chaining (KPC)-based code generation approach to improve the code quality with regard to exception handling.

A. API Knowledge Base Construction

As investigated in Section II-B, fine-grained prompts that provide specific exceptions and corresponding conditions can significantly boost the ability of LLMs to generate effective solutions for exception handling, avoiding the fault of incomplete or incorrect handling. Previous works [33], [34] demonstrate the feasibility of extracting high-quality knowledge from the official documentation. We follow such methodologies to mine API exception handling knowledge from Java documentation.

1) Knowledge schema: We construct the API knowledge base as triples of <entity, relation, entity> and only considered method-level APIs. The entities have three types of components: **API**, **exception** and **condition**, and the relations between entities include: **throw** and **trigger**. Specifically, the API knowledge base only has two forms: an API throws exceptions, and a condition triggers an exception. For example, the task in Figure 2 is related to two APIs in our API knowledge base: java.util.Vector.get(int index) and java.util.Vector.set(int index, E element). Both of them throw ArrayIndexOutOfBoundsException, which is triggered by the condition of *"if the index is out of range (index < 0 || index $>=$ size())"* [30], [31].

2) Knowledge extraction: In this work, we use Java SDK & JDK API specification [26] to construct the API knowledge base. The API documentation is collected from online resources by a web crawling tool [35], and each crawled web page is treated as an API document. We only keep the semi-structured API declarations and exception-handling specifications, but ignore other contents on the crawled web pages (e.g., code snippets and other textual descriptions), as this work focuses on exception-handling issues brought by API calls. API declarations describe the API's fully qualified name, which acts as an index of a specific API. Exception-handling specifications are usually in the form of *"Throw: [Exception] - [Condition]."* and we can use a rule-based method to extract exceptions and conditions from them. Different from previous works [33], [34], which conduct a series of pre-processing such as splitting and tokenizing to the natural language descriptions of the conditions, no further operation is required in this work because of the semantic interpretation capability of LLMs applied by our method.

B. KPC-based Code Generation

Given a natural language described coding task as input, our method takes three steps (i.e., generating, checking-rewriting), which are all supported by prompting the LLM in an AI chain workflow, to output the code with all exceptions being handled (see Figure 3). The three steps share the same state-of-the-art (SOTA) LLM, namely ChatGPT, and we elaborate the AI chain workflow as shown in Figure 1.

1) Generating: Typically, a coding task is a sentence of natural language description in the form of "How to ...". The sentence is rephrased into "Please write a Java method to... " as a prompt in order to clarify the requirement for LLMs explicitly.

As one of the most representative techniques to promote the revolution of artificial general intelligence (AGI), ChatGPT performs well on a large variety of natural language as well as software tasks. For example, previous work [12] has demonstrated the ability of ChatGPT to generate high-quality code, inspiring us to take advantage of it to implement the generating step. This step is responsible for generating the initial version of code according to a given coding task. Furthermore, the context-aware semantic interpretation of ChatGPT offers the potential for inter-model collaboration and interaction, which will be applied in the checking-rewriting loop.

2) Checking: After the first step of generating, we obtain a piece of code to solve the task (e.g., code of step-1 in

Figure 3). However, there is a high possibility that the code contains defects such as unhandled exceptions, as the input coding task does not provide any domain-specific information about exception handling, which usually appears in API documentation and has been stored in our knowledge base. We adopt the constructed API knowledge base and formulate a checking step with two parts to collect the exception-handling results.

- **API Information Collection.** It has been shown that LLMs can achieve good performance on program understanding [36], which simplifies the process of API extraction. To obtain the APIs' fully qualified names, we present the generated code to ChatGPT and ask it with the prompt "What Java SDK & JDK methods are used in the method you provided? Please list the fully qualified names of the methods." This API information collection part of step-2 in Figure 3 illustrates an example of how to extract the two API fully qualified names, i.e., java.util.Vector.get(int index) and java.util.Vector.set(int index, E element), from the generated code. We also use the extracted API names to construct links with the API knowledge base for further analysis.
- **Exception-handling Collection.** We first create a prompt template "Is the [Exception] handled for [API] in the code snippets? (Y/N)", in which the [Exception] and [API] are placeholders. Then the prompt template is instantiated by filling placeholders with related APIs and their corresponding exceptions according to information in the API knowledge base. For example, the prompt template in Figure 3 (step-2) is instantiated into two sub-questions, i.e., "Is the ArrayIndexOutOfBoundException handled for java.util.Vector.get(int index)?" and "Is the ArrayIndexOutOfBoundException handled for java.util.Vector.set(int index, E element)?". Similar to the previous part (i.e., API information collection), we provide ChatGPT with code and ask it questions with the aim of acquiring the exception-handling information. This operation is conducted on each extracted API, and all the handled/unhandled exceptions are collected in this step.

3) Rewriting: This step is technically similar to generating and checking, which applies the proper prompt to make ChatGPT generate the required code. The main difference between them is the prompt creation strategy. Specifically, we design a new prompt template "Please check [Condition] for [API], otherwise throw [Exception]." for this step. If there exist any unhandled exceptions in the code collected in step-2, the prompt template is instantiated with the information from the API knowledge base. We integrate all the exceptions into one prompt and ask ChatGPT to handle the exceptions at one time and rewrite the code. For example, in Figure 3 (Rewriting), we instantiate the prompt template into "Please check if the index is out of range (index < 0 || index >= size()) for java.util.Vector.get(int index), otherwise throw ArrayIndexOutOfBoundsException. Please check if the index is out of range (index < 0 || index >= size()) for java.util.Vector.set(int index, E element), otherwise throw ArrayIndexOutOfBoundsException".

The checking-rewriting steps will keep going iteratively until no more unhandled exception remains in the generated code.

IV. Experimental Setup

A. Research Question

In the evaluation, we study the following research questions:

- **RQ3: How effective is our KPC in identifying and handling exceptions during code generation tasks?** To answer this question, we perform a detailed statistical analysis to determine the number of potential exceptions that may arise in the generated code, as well as the number of checking-rewriting loops required by KPC to address these exceptions.
- **RQ4: To what extent can our KPC effectively assist in handling exceptions in LLM code generation?** We compare our approach with the state-of-the-art (SOTA) code generator, i.e.,ChatGPT, and employ both static and dynamic evaluation methods to assess the improvements in exception-handling issues in the generated code.

B. Task Collection and Implementation

The coding tasks of our experiments are collected from Java SDK & JDK API specification [26]. Following previous work [15], [37], we also use the API functional descriptions as the coding tasks. However, not all tasks are considered in this work. We only considered the tasks that LLMs can generate code using Java SDK & JDK APIs with potential exceptions. Thus, by analyzing code generated by LLMs, we collect 3,079 Java coding tasks from 11,259 API functional descriptions.

Both our KPC-based code generation approach and our baseline (i.e., ChatGPT) are implemented on top of the GPT-3.5-turbo model [38], which has been proven to achieve the best performance in code generation [12].

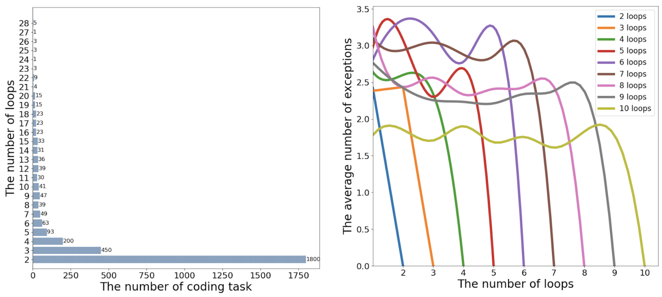
V. Evaluation

A. Efficiency of KPC (RQ3)

1) Method: Our KPC-based code generation approach is an iterative process, it is crucial to evaluate its efficiency in dealing with exception-handling issues. As our approach is implemented on top of model-3.5-turbo, whose response time cannot be artificially controlled, the best way to assess our approach's efficiency is to measure the number of exceptions that must be addressed in each loop during exception handling, and how many loops are necessary to complete the exception handling tasks.

Note that the initial round of checking-rewriting steps is considered as the first loop. Additionally, it's important to highlight that various exceptions that arise from the same API are identified as distinct exceptions.

2) Result: Figure 4(a) shows the distribution of checking-rewriting loops for all coding tasks completed by our KPC-based code generation approach. In our experiment, the number of checking-rewriting loops required ranges from a minimum of 2 to a maximum of 28. Among the 3,079 coding tasks, 90.35% (2,782) tasks could be completed within 10 loops, and 58.46% (1,800) tasks require only two loops for completion.



(a) Distribution of Checking-Rewriting Loops (b) Distribution of Unhandled Exceptions

Fig. 4. Distributions of KPC-based Code Generation

In this case, the results suggest that exceptions in the majority of coding tasks can be efficiently handled by our KPC within a limited number of checking-rewriting loops.

Figure 4(b) displays the average number of unhandled exceptions collected within each loop, which refers to the average number of unhandled exceptions per loop across all tasks that can be completed in the same number of loops. It is obvious that the average number of unhandled exceptions per loop ranges from 1.61 to 3.35, which indicates that each loop is simple and not required to handle too many exceptions.

We also observe that the average number of unhandled exceptions per loop can fluctuate (e.g., 5 loops and 6 loops in Figure 4(b)), which is because once the generated code is rewritten, new unhandled exceptions may be introduced. However, all unhandled exceptions can be resolved by the end of the loop, resulting in zero unhandled exceptions upon completion.

Answer to RQ3: Our KPC-based code generation approach is designed to efficiently handle the majority of exceptions using a small number of checking-rewriting loops, typically within 10 simple loops.

B. *Effectiveness of* KPC *(RQ4)*

1) Method: To evaluate the effectiveness of ChatGPT and our KPC-based approach in exception handling from both static and dynamic perspectives, we select three evaluation methods: LLMEva, CodeReview, and EvoSuite. LLMEva and CodeReview use static code analysis to evaluate the generated code, and EvoSuite generates test cases to make the dynamic validation.

Details of the three evaluation methods are as follows:

- **LLMEva:** It leverages LLMs' capability of comprehending code to evaluate the performance of generated code. Specifically, LLMEva evaluates the exception-handling performance of the generated code by posing the prompt to ChatGPT: "Can the code handle all exceptions in good practice? (Y/N)?".
- **CodeReview**: It is a manual code review method. Two of the authors independently label whether all exceptions in the generated code are handled in accordance with good practice. After completing respective labels, they compared their assessments and discussed any disagreements to arrive at a consensus. Note this process was conducted by the same authors who have conducted the empirical study before, which adds to the reliability and validity of our results.

TABLE II

THE NUMBER OF CODE WITH GOOD EXCEPTION-HANDLING PRACTICES

	# of Tasks	ChatGPT	KPC	Improve
LLMEva	3,079	1,056 (**34.30%**)	2,195(**71.29%**)	107.86%
CodeReview	384	56 (14.58%)	380 (98.86%)	578.57%
EvoSuite	384	353 (**91.93%**)	371 (**96.61%**)	↓ 18 bugs

- **EvoSuite [28]:** It is a state-of-the-art search-based software testing approach [28], [39] used for generating test cases for Java code, and it has been demonstrated to achieve high code coverage and improve bug detection capabilities. Generating test cases for code is an effective way to assess the quality of the code and ensure that it can avoid errors through appropriate exception handling. Many search algorithms have been proposed for EvoSuite, and we choose DynaMOSA [40] as the search algorithm, which optimizes multiple coverage targets simultaneously. Search algorithms have various parameters to set, but previous work [41] shows that parameter tuning SBST is extremely expensive and not necessary compared to default parameter values. Thus, we use the default settings of DynaMOSA. Moreover, considering our work focuses on exception handling in the generated code, we choose branch coverage and exception coverage as the optimization objective. To ensure the efficient allocation of resources, we established a time budget of two minutes per class for this process and repeated it 10 times. The executions is run on gnu/Linux system (Ubuntu 18.04.6 LTS) with 5.4.0-128-generic Linux kernel, 28-core 2.60GHz Intel(R) Xeon(R) Gold 6348 CPU and 1TB RAM.

For LLMEva, we evaluate results of all coding tasks in our experiment because the evaluation process is automated. While to tackle the time-consuming nature of both CodeReview and EvoSuite, we employ a statistical sampling method [42] to analyze MIN randomly selected instances of the coding tasks. $MIN = 384$ in this work, which ensures the estimated accuracy is in 0.05 error margin at 95% confidence level.

2) Result: Table II shows the number of code with good exception-handling practices generated by our KPC and ChatGPT, which can reflect their effectiveness of exception handling under different evaluation methods. Overall, our KPC demonstrates significant advantages across all three evaluation methods, with exception handling improvements of 107.86%, 578.57%, as well as a reduction of 18 real bugs, respectively.

According to LLMEva, 34.30% (1,056 out of 3,079) of the code generated by ChatGPT and 71.29% (2,195 out of 3,079) of the code generated by our KPC-based approach are proven to be with good exception-handling practice. As discussed in Section II-A, LLMs have several limitations in handling exceptions. The reliability of results evaluated by LLMEva is uncertain. However, based on the relative results (71.29% vs. 34.30%), it is evident that our KPC-based approach exhibits a significant improvement in exception handling compared to the code generated directly by ChatGPT.

The same results are more pronounced in manual code review. According to code review (i.e., the "CodeReview" row in Table II), only 14.58% of code generated by ChatGPT can meet good exception handling practice, while that of our KPC-based approach is as high as 98.86%. For manual code review, the Cohen's Kappa between the two authors is 0.98, which indicates they are with a significantly high agreement in the labeling results.

With careful observation, we find the main reason for the huge gap between the results of CodeReview and LLMEva is LLMs lack sufficient exception-handling knowledge and best practices for exception handling. Specifically, LLMs prefer to use general exceptions handling strategies, e.g, "RuntimeException", instead of specific exceptions, e.g., "ArrayIndexOutOfBoundsException". Using general exception handling can catch and handle multiple types of exceptions with a single catch block, but may result in the loss of important information needed to diagnose and fix specific errors, which is considered a bad practice in this work. By contrast, our KPC-based approach leverages fine-grained knowledge-driven prompts that include specific exceptions and their corresponding conditions. Our approach enhances the ability of LLMs to generate informative and accurate error messages. By providing developers with more detailed error messages, our approach can be a valuable tool for debugging.

Furthermore, the results of EvoSuite reveals that out of the 384 randomly selected tasks, a total of 31 ($384 - 353$) unique bugs across 10 runs can be detected in the code generated by ChatGPT. In comparison, 13 ($384 - 371$) unique bugs are detected in the code generated by our KPC-based approach. As the ultimate goal of exception handling is to minimize runtime errors, results from EvoSuite indicate that our KPC-based approach has the potential to help developers handle real bugs in their code. Although our approach only reduces 18 more real bugs in the sampled dataset compared with EvoSuite, we would like to remind the readers that EvoSuite is a sophisticated software tool that requires long-term research and significant amount of enginering effort to build and maintain. In contrast, by standing on the shoulder of the LLMs and through the design of an AI chain and knowledge-driven prompts, which is much simpler than EvoSuite, our approach achieves the performance on par with EvoSuite. This result sheds the light on the new opportunities to build software analysis tools on top of the LLMs.

Answer to RQ4: Our KPC-based approach can enhance the ability of LLMs to generate informative and accurate error messages, which can also help reduce runtime bugs in the generated code.

VI. User Study

We conduct a user study to evaluate the usefulness of our KPC-based code generation approach for helping developers write code efficiently and correctly.

A. User Study Design

1) Tasks and Procedure: As shown in Table III, we select six typical Java programming tasks from the dataset of empirical study (Section II) and evaluation (Section V), which includes two easy tasks (T2 and T6), two medium tasks (T1 and T5) and two hard tasks (T3 and T4). The participants are expected to use a certain kind of Java API to write code and implement a method that satisfies the requirement of each task. It is free to access any online resources, including API documentation and Q&A forums, except the code generator (e.g., CodeX and ChatGPT). Each task has a time limit of ten minutes, which means the user study can be finished within one hour. After the user study, we conduct a short survey by collecting feedback on each participant, which is used to understand their behaviors and improve our work in the future.

2) Participants: We recruit 12 participants from an IT company that has over 2,000 developers to attend the user study. The participants have 1 to 5 (on average of 3.2) years of Java development experience on both commercial and open-source projects. They are divided into three groups (i.e., G-1, G-2, G-3) equally, and each group consists of four participants. G-1 and G-2 are control groups, where G-1 is given only the coding tasks, while G-2 is given both the coding tasks and reference code generated by ChatGPT. G-3 is the experimental group that is given both the coding tasks and reference code generated by our KPC-based code generation approach.

3) Evaluation Metrics: We consider two key factors for evaluating the performance of participants: task completion time and answer correctness. Task completion time reflects how fast a participant can complete a coding task. Answer correctness represents whether the code submitted by a participant is actually an appropriate solution with good exception-handling practice to the task. The two authors first review the code to determine the correctness of it independently and discuss for final decisions if there exist different results. We compute Fleiss's Kappa [43] to examine the agreement between the two authors. If the submitted code is annotated as correct, the participant gets 1 mark, otherwise 0 mark. We use Wilcoxon signed-rank test [44] with Bonferroni correction [45] to determine if the performance variation across different groups is statistically significant. For example, if the corresponding Wilcoxon signed-rank test result (i.e., p-value) is less than 0.05, we can consider one group performs better than the other at the confidence level of 95%.

Furthermore, we ask participants in G-2 and G-3 a supplementary question, "To what extent does the reference code help you to complete the task successfully?". They are supposed to answer it by giving a score of 0 to 4 for each task, where 0 means helpless, and 4 means very helpful.

B. Result

Table III shows the results of the user study.

1) Correctness: The "Correctness" column in Table III indicates the accuracy of each task completed by different groups. For instance, a value of "1/4" means that only one out of four participants completed the coding task correctly.

We can see from Table III that without any reference to complete the coding tasks, G-1 only achieves an average correctness of 29.17%. All coding tasks have limited numbers of correct answers, with only one or two participants being able to complete them correctly. In contrast, G-2 and G-3 have reference code to complete the tasks, which obtain

TABLE III
CODING TASKS AND RESULTS OF USER STUDY

No.	Coding Task	Difficulty	Correctness			Time Consumption (second)			Usefulness	
			G-1	*G-2*	*G-3*	*G-1*	*G-2*	*G-3*	*G-2*	*G-3*
T1	Please write a Java method that removes the char at the specified position in this sequence.	Medium	1/4	3/4	4/4	84.5	267.5	131.3	3.5	4.0
T2	Please write a Java method to swap two elements in a vector using Java	Easy	2/4	2/4	4/4	82.3	225.0	148.8	3.0	4.0
T3	Please write a Java method that acquires a lock on the given region of this channel's file.	Hard	1/4	1/4	2/4	500.0	385.5	420.0	3.5	3.8
T4	Please write a Java method that creates a new instance of URLClassLoader for the specified URLs and default parent class loader.	Hard	0/4	1/4	2/4	253.0	248.0	480.0	3.8	3.5
T5	Please write a Java method that interrupts a running Thread in Java.	Medium	1/4	2/4	4/4	159.0	147.3	190.0	3.8	3.5
T6	Please write a Java method that split a string into a number of substrings in Java	Easy	2/4	2/4	2/4	95.3	180.5	107.5	3.3	2.8
Average	-	-	29.17%	45.83%	**75.00%**	**195.7**	242.3	246.3	3.5	**3.6**

relatively higher average correctness of 45.83% and 75.00%, respectively. This suggests the reference code can significantly assist participants in completing coding tasks, especially when the code is generated by KPC-based approach and provide sufficient information about exception handling. The Wilcoxon signed rank test shows that the differences of the correctness between G-1 and G-2, G-1 and G-3, and G-2 and G-3 are statistically significant at $p - value < 0.05$.

We gain the following research findings when evaluating the correctness of coding tasks. In cases of no reference code is provided (i.e., G-1), easy tasks such as T2 and T3 are more likely to be completed correctly, while hard tasks such as T3 and T4 are still big challenges for the participants to solve. However, the situation becomes more complex when a reference code is provided. In G-2, the reference code generated by ChatGPT may contain numerous problems with exception handling, which could mislead participants. Consequently, there is no improvement for half of the tasks, including T2, T3 and T6. On the other hand, the reference code generated by our KPC-based approach can provide more useful information for exception handling, achieving significant improvements in correctness. Specifically, all four participants successfully completed tasks T1, T2, and T5, showcasing the effectiveness of our approach. It is worth noting that even provided with reference code, participants in G-2 and G-3 may occasionally complete T6 incorrectly. We find the main reason is that both ChatGPT and our KPC fail to generate the correct reference code as the API documentation does not include exception-handling specifications for the APIs used in this task, which has been explained in Section II-B.

2) Time Consumption: The "Time Consumption" in Table III refers to the meantime that a group completes each task, in which G-1 takes the shortest time to finish the tasks with an average of 195.7 seconds. We find that the participants in G-1 tend to complete the coding tasks on their own without referring to any generated code. In contrast, participants in G-2 and G-3 are more likely to read the reference code carefully to extract useful information about the tasks, which costs them more time to finish the tasks, with average values of 242.3 seconds and 246.3 seconds, respectively. Although it takes about 25% more time to read the reference code, it is worth that the participants have more chances to solve the coding task correctly. The Wilcoxon signed-rank test shows the differences in the task completion time between G-1 and G-2, G-1 and G-3, and G-2 and G-3 are statistically significant at $p - value < 0.05$.

3) Usefulness and Interview: The "Usefulness" in Table III is a unique feature of G-2 and G-3, and we obtain similar scores of 3.5 and 3.6 for the two groups, respectively, which means the participants believe the code generated by both ChatGPT and KPC is helpful. We further analyze the interview records and have two interesting findings. Firstly, most of the developers have weak awareness of exception handling. For example, when we ask them "Is there any exception handling issues in the reference code?", they feel very hard to identify the problem accurately, even for those APIs that are frequently used. This phenomenon applies not only to junior developers but also to senior developers, who often overlook seemingly "minor" issues. That is precisely why developers can inadvertently make mistakes, and even impeccably designed software may still contain bugs. Second, ChatGPT is very "good" at generating flawed code that appears to be correct, thereby misleading developers. That's because ChatGPT was trained with a plethora of online resources, and the answers it generates partly reflect common practices in the software development community, but common practices aren't necessarily the right ones. Hence, our reliable knowledge-driven approach is therefore of paramount importance here.

VII. DISCUSSION

A. Limitations of KPC

As our KPC-based code generation approach is implemented on top of ChatGPT, several limitations need acknowledging.

1) Code understanding: ChatGPT's capability to understand code is limited because it is primarily designed for natural language processing and lacks direct experience with programming languages. Generally, ChatGPT is able to recognize and generate basic code syntax, e.g., variable declarations and conditional statements, but does not have a deep understanding of underlying concepts and semantic interpretation of programs. Therefore, writing comprehensive code to solve complex coding tasks remains a big challenge for ChatGPT.

In this work, leveraging prompts to interact with ChatGPT exists in most steps of our approach, e.g., collecting API and exception-handling information in step-2 (see Section III-B). ChatGPT's limitation in code understanding is one of the potential negative impacts on the performance of this work.

2) Trade-off between efficiency and effectiveness: Our KPC interacts with ChatGPT iteratively to deal with the exception handling issues in the code until the generated code is with good practice. According to the CodeReview (Section V-B) and user study (Section VI), we also realize that the more iterations KPC interacts with ChatGPT, the more complex the generated code will be, resulting in confusing and difficulties for developers to understand the code. Although increasing loops give our tool a higher probability of solving all exceptions in the coding tasks, this strategy ignores the requirement of efficiency and might not be a good practice in the industry. As demonstrated in RQ3, a reduced loop number within 10 loops is suggested for balancing efficiency and effectiveness.

B. Threats to Validity

1) Internal Validity: There might be inaccuracy when labeling the exception-handling practice for generated code (i.e., Section V-B) and scoring for users' answers (i.e., Section VI). Both annotators have more than five years of programming experience in Java programming. Additionally, two annotators check all the information about a vulnerability independently and discuss if they cannot reach an agreement.

2) External Validity: One of the threats is the exception-handling specifications in Official API documenation [26] are not comprehensive, which makes our KPC-based approach ignore some useful exception-handling solutions. In the future, we will mine exception-handling knowledge from other sources (e.g., GitHub and Stack Overflow) to continuously extend our API knowledge base. We may also leverage the LLM as a neural knowledge base [46] to consult it for API exception knowledge, as the LLM "sees" all kinds of formal and informal API documentation during pre-training. This will further simplify our approach and mitigate the reliance on high-quality API reference documentation.

VIII. Related Work

A. Large Language Model & Prompt Engineering

LLMs (e.g., BERT [47], BART [48] and GPT-3 [49]) have become ubiquitous in NLP field and achieved impressive performance in various tasks including question answering [1], content creation [2]–[4], logical reasoning [5], [6], software testing [50]–[52] and robotics [53], [54]. PEER [2] and Re^3 [3] decompose content creation as recursive plan, write and revision steps, which achieve strong performance across various domains and editing tasks. Lemieux et al. [50] propose CODAMOSA for test case generation, which conducts search-based software testing until its coverage improvements stall, then asks LLMs (i.e., Codex [7]) to provide example test cases for under-covered functions. Besides, LLMs offer a potential solution to automated graphical user interface (GUI) testing [52]. The robustness and performance of LLMs depend strongly on the quality of prompt and tremendous effort has been made on the prompt engineering [21]–[24]. Nashid et al. [25] present a retrieval-based prompt selection method to solve test assertion generation and program repair problems, which can potentially be applied to multilingual and multitask settings without task or language-specific training.

B. Code Generation

Following the success of LLMs at natural language tasks, the application of LLMs to code has generated significant interest. As a result, multiple models for generating code were developed, such as Codex [7], GitHub Copilot [55], AlphaCode [8], PolyCoder [56], InCoder [10] and CodeGen [57]. Codex [7] is a state-of-the-art code generation model that utilizes GPT-3 technology and an API interface for public access. GitHub Copilot [55] is powered by Codex and trained on public GitHub repositories, which support multiple programming languages such as Python, JavaScript, TypeScript, Ruby, and Go. Although LLMs' utility is apparent, the degree of their robustness remains uncertain, which leads to a wide range of research interests. Barke et al. [13] study how programmers interact with Copilot and provide recommendations for improving the usability of future AI programming assistants. Chen et al. [58] leverages one LLM to automatically generate code samples and test cases for them. Recently, ChatGPT [59] has garnered a significant amount of attention, and researchers try to leverage this powerful technique to generate high-quality code [12]. In this work, we focus on exception handling in the process of code generation to make the generated code samples more reliable.

C. Exception Handling

Exception handling practice has been widely studied in literature [17]–[19], [60], [61]. Padua et al. [18] conduct an empirical study to explore the relationship between exception handling practices and software quality on open-source Java and C# projects. They find that exception flow characteristics in Java projects have a significant relationship with post-release defects. Nguyen et al. [60] design a tool to predict the potential exception type that could occur in a given code snippet and recommend proper code to handle those exceptions. To implement automated exception handling, Zhang et al. [20] propose a novel neural approach to predict the locations of try blocks and automatically generate the complete catch blocks for an exception. Recent work [16] tends to apply LLMs to handle exceptions with explanations of the errors and suggestions on how to fix the error, and shows promising performance in this field. Similarly, Our work takes advantage of LLMs' few short learning abilities and obtains domain-specific knowledge from API documents to guide the exception code generation.

IX. Conclusion

This paper presents the first knowledge-driven prompt chaining based code generation approach. We first extract exception-handling specifications from official API documentation and construct an API knowledge base. Then, we use the fine-grained knowledge to construct knowledge-driven prompt chains to assist LLMs in considering exception-hand in code generation tasks. Our KPC has been proven to be highly efficient and effective in handling exceptions. The usefulness of our KPC-based approach has also been demonstrated in practice. However, we acknowledge that the capability of LLMs to analyze code and the availability of exception-handling specifications in API documentation are still limited. Thus, there is still much room for improvement in this field. In the future, we aspire to address these limitations and make significant contributions to the field of knowledge-driven code generation for handling exceptions.

X. Acknowledgements

We would like to thank the reviewers for their detailed comments and constructive suggestions. This research was partially supported by "the Fundamental Research Funds for the Central Universities"(226-2022-00064).

References

[1] S. Arora, A. Narayan, M. F. Chen, L. J. Orr, N. Guha, K. Bhatia, I. Chami, F. Sala, and C. Ré, "Ask me anything: A simple strategy for prompting language models," *arXiv preprint arXiv:2210.02441*, 2022.

[2] T. Schick, J. Dwivedi-Yu, Z. Jiang, F. Petroni, P. Lewis, G. Izacard, Q. You, C. Nalmpantis, E. Grave, and S. Riedel, "Peer: A collaborative language model," *arXiv preprint arXiv:2208.11663*, 2022.

[3] K. Yang, N. Peng, Y. Tian, and D. Klein, "Re3: Generating longer stories with recursive reprompting and revision," *arXiv preprint arXiv:2210.06774*, 2022.

[4] T. Wu, M. Terry, and C. J. Cai, "Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–22.

[5] A. Creswell, M. Shanahan, and I. Higgins, "Selection-inference: Exploiting large language models for interpretable logical reasoning," *arXiv preprint arXiv:2205.09712*, 2022.

[6] S. M. Kazemi, N. Kim, D. Bhatia, X. Xu, and D. Ramachandran, "Lambada: Backward chaining for automated reasoning in natural language," *arXiv preprint arXiv:2212.13894*, 2022.

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

[8] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[9] 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," *arXiv preprint arXiv:2203.13474*, 2022.

[10] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[11] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.

[12] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *arXiv preprint arXiv:2304.07590*, 2023.

[13] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, 2023.

[14] R. Gozalo-Brizuela and E. C. Garrido-Merchan, "Chatgpt is not all you need. a state of the art review of large generative ai models," *arXiv preprint arXiv:2301.04655*, 2023.

[15] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on github copilot," *arXiv preprint arXiv:2302.00438*, 2023.

[16] J. Leinonen, A. Hellas, S. Sarsa, B. Reeves, P. Denny, J. Prather, and B. A. Becker, "Using large language models to enhance programming error messages," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 563–569.

[17] E. A. Barbosa and A. Garcia, "Global-aware recommendations for repairing violations in exception handling," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 858–858.

[18] G. B. de Pádua and W. Shang, "Studying the relationship between exception handling practices and post-release defects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 564–575.

[19] D. Sena, R. Coelho, U. Kulesza, and R. Bonifácio, "Understanding the exception handling strategies of java libraries: An empirical study," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 212–222.

[20] J. Zhang, X. Wang, H. Zhang, H. Sun, Y. Pu, and X. Liu, "Learning to handle exceptions," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 29–41.

[21] Y. Gu, X. Han, Z. Liu, and M. Huang, "Ppt: Pre-trained prompt tuning for few-shot learning," *arXiv preprint arXiv:2109.04332*, 2021.

[22] X. Liu, K. Ji, Y. Fu, W. Tam, Z. Du, Z. Yang, and J. Tang, "P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2022, pp. 61–68.

[23] X. Chen, N. Zhang, X. Xie, S. Deng, Y. Yao, C. Tan, F. Huang, L. Si, and H. Chen, "Knowprompt: Knowledge-aware prompt-tuning with synergistic optimization for relation extraction," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 2778–2788.

[24] J. Liao, X. Zhao, J. Zheng, X. Li, F. Cai, and J. Tang, "Ptau: Prompt tuning for attributing unanswerable questions," in *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2022, pp. 1219–1229.

[25] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*, 2023.

[26] JavaDoc, "Java official sdk & jdk api documentation," https://docs.oracle.com/en/java/javase/17/docs/api/index.html, 2023.

[27] KPC, "Kpc replication package," https://github.com/goodchar123/KPC, 2023.

[28] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[29] Tutorialspoint, "Tutorialspoint," https://www.tutorialspoint.com/javaexamples/index.htm, 2023.

[30] JavaAPI, "The exception handling specification for java.util.vector.get(int index)," https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Vector.html#get(int), 2023.

[31] ——, "The exception handling specification for java.util.vector.set(int index, e element)," https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Vector.html#set(int,E), 2023.

[32] ——, "The exception handling specification for java.lang.string.split(string regex)," https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html#split(java.lang.String), 2023.

[33] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving api caveats accessibility by mining api caveats knowledge graph," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 183–193.

[34] X. Ren, J. Sun, Z. Xing, X. Xia, and J. Sun, "Demystify official api usage directives with crowdsourced api misuse scenarios, erroneous code examples and patches," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 925–936.

[35] beautiful-soup 4, "Beautiful soup 4," https://beautiful-soup-4.readthedocs.io/en/latest/, 2023.

[36] C. S. Xia, Y. Wei, and L. Zhang, "Practical program repair in the era of large pre-trained language models," *arXiv preprint arXiv:2210.14179*, 2022.

[37] H. Huang, M. Wen, L. Wei, Y. Liu, and S.-C. Cheung, "Characterizing and detecting configuration compatibility issues in android apps," in

2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021, pp. 517–528.

[38] Openai, "Gpt-3.5-turbo," https://platform.openai.com/docs/models/gpt-3-5, 2023.

[39] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," *Empirical Software Engineering and Verification: International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures*, pp. 1–59, 2012.

[40] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[41] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.

[42] R. Singh and N. S. Mangat, *Elements of survey sampling*. Springer Science & Business Media, 2013, vol. 15.

[43] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[44] F. Wilcoxon, *Individual comparisons by ranking methods*. Springer, 1992.

[45] E. W. Weisstein, "Bonferroni correction," *https://mathworld. wolfram. com/*, 2004.

[46] D. Rai, Y. Zhou, B. Wang, and Z. Yao, "Explaining large language model-based neural semantic parsers (student abstract)," *arXiv preprint arXiv:2301.13820*, 2023.

[47] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[48] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

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

[50] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *45th International Conference on Software Engineering, ser. ICSE*, 2023.

[51] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," *arXiv preprint arXiv:2209.11515*, 2022.

[52] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "Fill in the blank: Context-aware automated text input generation for mobile gui testing," *arXiv preprint arXiv:2212.04732*, 2022.

[53] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg, "Progprompt: Generating situated robot task plans using large language models," *arXiv preprint arXiv:2209.11302*, 2022.

[54] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar *et al.*, "Inner monologue: Embodied reasoning through planning with language models," *arXiv preprint arXiv:2207.05608*, 2022.

[55] Openai, "Github copilot," https://github.com/features/copilot, 2023.

[56] 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.

[57] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv e-prints*, pp. arXiv–2203, 2022.

[58] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[59] Openai, "Introducing chatgpt," https://openai.com/blog/chatgpt, 2023.

[60] T. Nguyen, P. Vu, and T. Nguyen, "Recommending exception handling code," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 390–393.

[61] X. Ren, X. Ye, Z. Xing, X. Xia, X. Xu, L. Zhu, and J. Sun, "Api-misuse detection driven by fine-grained api-constraint knowledge graph," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 461–472.