LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation

ZIYAO ZHANG, Sun Yat-sen University, China
CHONG WANG, Nanyang Technological University, Singapore
YANLIN WANG*, Sun Yat-sen University, China
ENSHENG SHI, Huawei Cloud Computing Technologies Co., Ltd, China
YUCHI MA, Huawei Cloud Computing Technologies Co., Ltd, China
WANJUN ZHONG, Sun Yat-sen University, China
JIACHI CHEN, Sun Yat-sen University, China
MINGZHI MAO, Sun Yat-sen University, China
ZIBIN ZHENG, Sun Yat-sen University, China

Code generation aims to automatically generate code from input requirements, significantly enhancing development efficiency. Recent large language models (LLMs) based approaches have shown promising results and revolutionized code generation task. Despite the promising performance, LLMs often generate contents with hallucinations, especially for the code generation scenario requiring the handling of complex contextual dependencies in practical development process. Although previous study has analyzed hallucinations in LLM-powered code generation, the study is limited to standalone function generation. In this paper, we conduct an empirical study to study the phenomena, mechanism, and mitigation of LLM hallucinations within more practical and complex development contexts in repository-level generation scenario. First, we manually examine the code generation results from six mainstream LLMs to establish a hallucination taxonomy of LLM-generated code. Next, we elaborate on the phenomenon of hallucinations, analyze their distribution across different models. We then analyze causes of hallucinations and identify four potential factors contributing to hallucinations. Finally, we propose an RAG-based mitigation method, which demonstrates consistent effectiveness in all studied LLMs.

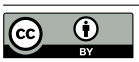
CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Repository-Level Code Generation, Hallucination, Large Language Models

ACM Reference Format:
Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and

*Yanlin Wang is the corresponding author.

Authors' Contact Information: Ziyao Zhang, zhangzy373@mail2.sysu.edu.cn, Zhuhai Key Laboratory of Trusted Large Language Models, Sun Yat-sen University, Zhuhai, China; Chong Wang, chong.wang@ntu.edu.sg, Nanyang Technological University, Nanyang Avenue, Singapore; Yanlin Wang, wangylin36@mail.sysu.edu.cn, Zhuhai Key Laboratory of Trusted Large Language Models, Sun Yat-sen University, Zhuhai, China; Ensheng Shi, shiensheng@huawei.com, Huawei Cloud Computing Technologies Co., Ltd, Beijing, China; Yuchi Ma, mayuchi1@huawei.com, Huawei Cloud Computing Technologies Co., Ltd, Shenzhen, China; Wanjun Zhong, zhongwj25@mail2.sysu.edu.cn, Sun Yat-sen University, Guangzhou, China; Jiachi Chen, chenjch86@mail.sysu.edu.cn, Zhuhai Key Laboratory of Trusted Large Language Models, Sun Yat-sen University, Zhuhai, China; Mingzhi Mao, mcsmmz@mail.sysu.edu.cn, Zhuhai Key Laboratory of Trusted Large Language Models, Sun Yat-sen University, Zhuhai, China; Zibin Zheng, zhzibin@mail.sysu.edu.cn, Zhuhai Key Laboratory of Trusted Large Language Models, Sun Yat-sen University, Zhuhai, China.



ACM 2994-970X/2025/7-ARTISSTA022
https://doi.org/10.1145/3728894

Mitigation. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA022 (July 2025), 23 pages. https://doi.org/10.1145/3728894

1 Introduction

Code generation aims at efficiently producing code from specifications described in natural language. This process significantly reduces the manual coding workload for developers [7, 11, 51], allowing them to focus more on solving advanced technical challenges and engaging in innovative tasks. Recent developments have introduced a variety of large language models (LLMs) [5, 8–10, 14, 16, 25, 26, 32, 34, 39, 43, 44, 48] built upon the Transformer architecture [38]. These models, trained on extensive code corpora, can automatically generate code from natural language inputs and have shown high efficacy in code generation. For example, GPT-4 has achieved state-of-the-art results on evaluation benchmarks such as HumanEval [10] and MBPP [6], demonstrating high functional correctness, particularly in generating *standalone* functions based on detailed specifications.

However, in practical development scenarios, the requirements for code generation are more complex than simply generating standalone functions [46]. To address this complexity, new benchmarks such as CoderEval [46] and EvoCodeBench [24] have been proposed to better reflect ***real-world repository-level*** development scenarios. Evaluations based on these benchmarks have revealed that LLMs face challenges in generating *non-standalone* functions with contextual dependencies, such as calls to user-defined functions and project-defined data protocol. While these benchmarks provide valuable insights into the effectiveness of LLMs in practical code generation, they mainly emphasize *functional correctness* of the LLM-generated code—measured by test case pass rates—without a comprehensive examination of the underlying mechanism of failure.

To bridge this gap, this work aims to systematically investigate issues in practical LLM-based code generation from the perspective of hallucinations. Hallucination is one of the most significant issues for state-of-the-art generative LLMs [19]. For general natural language tasks, LLM hallucinations have been explored to a certain extent [19, 37, 45, 50] and are typically categorized into three types: Input-Conflicting Hallucination, Fact-Conflicting Hallucination, and Context-Conflicting Hallucination [50]. In the domain of code generation, Liu et al. [27] conducted a study to analyze hallucinations in LLM-generated code and established a taxonomy that includes *Intent Conflicting*, *Context Deviation*, and *Knowledge Conflicting*. While Liu et al.'s study provided insightful findings, it still faces two potential limitations. First, as this study relies on benchmarks (i.e., HumanEval [10] and DS-1000 [23]) focusing on *standalone* function or script generation, the resulting taxonomy may not fully align with practical repository-level code generation scenarios involving non-standalone dependencies. For instance, project contexts such as user-defined dependencies and non-code resources (e.g., configuration files) are not taken into account. Second, their study primarily categorized *fine-grained* hallucinations in LLM-generated code and attempted to detect them, without thoroughly analyzing potential causes or exploring mitigation feasibility. In contrast, we investigate hallucinations in ***repository-level code generation*** within more practical development contexts, adopting a holistic perspective that encompasses ***phenomena***, ***mechanism***, and ***mitigation***.

In this work, we conduct an empirical study to uncover the status quo and root causes of hallucinations in LLM-based code generation within real-world projects. The study aims at answering the following research questions (RQs):

- **RQ1 (Hallucination Taxonomy):** What are the specific manifestations of hallucinations in practical code generation, and how are they distributed?
- **RQ2 (LLM Comparison):** How do different LLMs compare in terms of hallucination occurrences and patterns?

- **RQ3 (Potential Cause Discussion):** What are the potential causes of hallucinations in practical LLM-based code generation?

To answer the questions, we experiment on six mainstream LLMs (ChatGPT [33], CodeGen [32], PanGu-α [48], StarCoder2 [28], DeepSeekCoder [16], and CodeLlama [34]) with the CoderEval dataset [46]. To obtain the hallucination taxonomy of practical LLM-based code generation, we manually perform open coding [21] on the LLM-generated code. Specifically, we first extract 10% of the coding tasks from the CoderEval dataset in the initial stage. Then, from the initial annotation and discussion, we obtain preliminary taxonomy. Finally, we obtain the fully hallucination taxonomy with iterative labelling the remaining 90% coding tasks and continuously refining the taxonomy in the process. After obtaining the taxonomy, we conduct extensive analysis based on the research questions aforementioned.

Findings. Our study reveals the following findings. ① LLM hallucinations in code generation can be divided into three major categories (Task Requirement Conflicts, Factual Knowledge Conflicts, and Project Context Conflicts) with eight subcategories: Functional Requirement Violation, Non-Functional Requirement Violation, Background Knowledge Conflicts, Library knowledge Conflicts, API Knowledge Conflicts, Environment Conflicts, Dependency Conflicts, and Non-code Resource Conflicts. ② We analyze the hallucination distribution in different LLMs and find that Task Requirement Conflicts are the most prevalent type of hallucination across all models. ③ We identify four potential factors that cause hallucinations: training data quality, intention understanding capacity, knowledge acquisition capacity, and repository-level context awareness.

Mitigation. Based on the findings, we explore a lightweight mitigation approach based on retrieval augmented generation (RAG) and evaluate its effectiveness.

In this approach, we take two main steps. First, we construct a retrieval library. This library is built based on the code repository relevant to the development scenario of each code generation task. Second, we identify useful code snippets by detecting the similarity between the task description of the current generation task and the code snippets stored in the retrieval library. The code snippet that has the highest relevance to the current task, as determined by this similarity check, is then used as a prompt to guide the code generation process. Experimental results show that this lightweight mitigation can consistently improve the performance of all studied LLMs.

In summary, this paper makes the following contributions:

- We conduct an empirical study to analyze the hallucinations in LLM-based code generation within real development scenarios and establish a hallucination taxonomy.
- We elaborate on the phenomenon of hallucinations, analyze the distribution of hallucinations on different models.
- We further analyze causes of hallucinations and identify four possible factors.
- We propose a RAG-based mitigation approach based on the causes of hallucinations and experiment on various LLMs to study its effectiveness.

2 Background & Related Work

2.1 LLM-based Code Generation

For developers, a realistic scenario is to use a code repository to write code, which is very common in practice [15]. For example, due to security and functionality considerations, companies often only build code warehouses internally. The code repository provides many private APIs that are not seen by the language model and are not public on any code hosting platform. Therefore, it is worth exploring whether pre-trained language models can adapt to real development needs and generate correct and efficient code. Previous studies such as MBPP [6] and HumanEval [10] have evaluated LLMs for standalone functions, which do not depend on functions in other files. However,

in real-world development scenarios, the development of a function not only relies on the text description and function signature of the function, but also requires calling a custom API in the code repository. Such non-independent functions are commonly found in real-world generation scenarios. By analyzing the 100 most popular projects written in Java and Python on GitHub [46], previous work found that dependent functions account for more than 70% of the functions in open source projects. In order to better simulate real development scenarios and to check the correctness of LLMs, CoderEval [46], ClassEval [13], and EvoCodeBench [24] collected code snippets and text descriptions from real code repositories and used test cases to check the correctness of the code repositories in their corresponding environments. However, the performance of the model on these benchmarks is extremely poor. LLMs cannot generate correct code based on the problem description, and the model prefers to generate independent code segments rather than using existing functions in the current development scenario.

2.2 Hallucinations in LLMs

In the field of natural language processing (NLP), hallucination refers specifically to situations where the content produced by a language model in the process of generating text is inconsistent with the given input or expected output environment, lacks meaning, or violates the facts [20]. This kind of phenomenon is particularly prominent in text generation models, especially in tasks such as text completion, summary generation, and machine translation. The output of the model must maintain a high degree of consistency and authenticity to ensure its practicality and reliability. Hallucination phenomena can be divided into the following categories according to their nature [50]: (1) Input-Conflicting Hallucinations: When the text generated by the model deviates from the original input source, input-conflicting hallucinations will occur. This hallucination may result from the model's incorrect parsing or inaccurate internal representation of the input information, causing the output content to deviate from the intent and context of the source input. (2) Context-Conflicting Hallucinations: This type of hallucination occurs when the text generated by the model is contradictory or inconsistent with its previously generated content. Contextual conflict hallucinations reflect the model's challenges in maintaining textual coherence and consistency, which may be due to the model's insufficient processing of contextual information or limitations of its memory mechanism. (3) Fact-Conflicting Hallucinations: When the content generated by LLM is inconsistent with established knowledge or facts in the real world, fact-conflicting hallucinations will occur. This hallucination reveals the model's inadequacy in understanding and applying knowledge about the external world, and may be caused by limitations in model training data, lags in knowledge updates, or limitations in the model's reasoning capabilities.

However, there is a lack of research on hallucination phenomena in the field of code generation. Although there have been a large number of LLM-based methods to optimize code generation tasks, these works do not have a clear definition of the code generation hallucination. The presence of hallucination problems can be detrimental to the overall quality of the generated code. This may not only affect the performance and maintainability of the code, but may also lead to unexpected errors and security vulnerabilities, thus posing a threat to the stability and security of the software. In order to make up for the gaps in the definition of hallucination problems, Liu et al. [27] conducted a study based on a data set of standalone code generation to define hallucinations for LLMs in code generation tasks. They divided hallucinations in LLM-based code generation into five main types, but they ignored that LLMs in real-world code generation tasks will involve complex repository-level contexts such as development environment, system resources, external constraints, code warehouses, etc. These factors often cause LLMs to fail in actual development, leading to problems such as low usability and low accuracy. We believe that focusing solely on generating standalone functions is inadequate for developers in real-world development scenarios, often requiring the

integration of code into a comprehensive code repository by using LLMs. In order to better explore hallucinations that exist in LLMs in practical development scenarios, our work obtained data sets in real development scenarios for empirical study, and defined new types of hallucinations, which opened up new ideas for subsequent research on hallucinations.

3 Evaluation Setup

3.1 Dataset

Existing work only discusses the hallucination in the function-level development scenario, but does not discuss hallucinations of LLMs in practical development scenarios. To better simulate practical development scenarios, we use a set of coding tasks from real-world Python repositories based on CoderEval [46]. It comprises 230 Python code generation tasks. Each task consists of a natural language description, a ground-truth code snippet, and a set of test cases, along with the project context.

3.2 Studied LLMs

We utilize several mainstream LLMs to perform code generation for the studied programming tasks. The LLMs being used cover both open-source and closed-source models and span various parameter sizes, listed as follows.

- **ChatGPT** [33]: ChatGPT is a versatile text generation model for multilingualism with powerful code generation capabilities, we use the GPT-3.5-Turbo in our experiments.
- **CodeGen** [31]: CodeGen is a family of auto-regressive language models for program synthesis with several different versions. To better accomplish the generation task, we use the CodeGen-350M-Mono model.
- **PanGu-α** [48]: PanGu-α can perform code generation tasks in multiple languages. We use the PanGu-α-2.6B model.
- **DeepSeekCoder** [16]: DeepSeekCoder performs well in open source models across multiple programming languages and various benchmarks. We use the DeepSeekCoder-6.7B base model.
- **CodeLlama** [34]: CodeLlama is a set of pre-trained and fine-tuned generative text models ranging in size from 7 to 34 billion parameters. We use the CodeLlama-7b-Python-hf model.
- **StarCoder2** [25]: StarCoder2 is a family of open code-oriented models for large languages, providing three scales of models, we use the StarCoder2-7B model.

For each task, we use the LLMs to generate 10 code snippets by employing the nucleus sampling strategy [18] and setting temperature to 0.6, following the same setting as CoderEval.

3.3 Taxonomy Annotation

In order to analyze the hallucination types in the LLM-generated code, we manually perform open coding [21] on the generated code to obtain the hallucination taxonomy.

(1) Initial Open Coding. Firstly, in the initial open-coding stage, we select 10% of the 230 coding tasks in CoderEval Python dataset for preliminary manual analysis. We randomly collect 23 generative tasks from CoderEval, we employ CodeGen, Pangu-α, ChatGPT, DeepSeekCoder, CodeLlama, and StarCoder2, with each model generating ten code snippets for each code generation task, culminating in a total of 1,380 code snippets to be analysed for hallucination taxonomy framework. For each code snippet, we further run its test cases in the actual development environment corresponding to the task to determine its correctness. Next, two annotators manually review the LLM-generated code snippets that fail the tests, identifying specific code issues by referring to the ground truth and execution results. Note that an LLM-generated snippet may fail test cases due to multiple code issues. Ultimately, this stage identifies a total of seventeen types of issues, such as

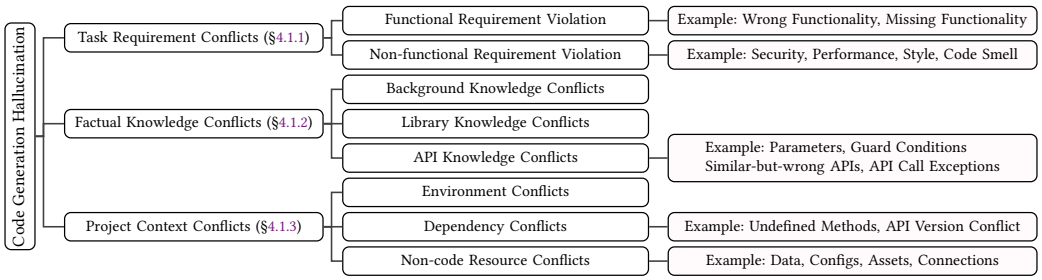


Fig. 1. Taxonomy of Hallucinations in LLM-based Code Generation.

Missing Functionality, *Code Smell*, *Similar-but-Wrong APIs*, *Data Resource Conflicts* and *Undefined Methods*, after deduplication.

(2) Preliminary Taxonomy Construction. Secondly, based on the specific issues identified, we conduct a manual analysis to group related code issues into broader hallucination types. For example, code issues involving *API Parameters*, *API Guard Conditions*, *Similar-but-Wrong APIs*, and *API Call Exceptions* are grouped under the type *API Knowledge Conflicts*, as they all represent API misuses and conflicts with factual API knowledge. This grouping process, conducted by two authors in consultation with the original annotators, results in eight hallucination types. Building on this grouping, we further categorize the types into three higher-level categories that generally align with definitions in the NLP domain. Finally, we establish a preliminary hallucination taxonomy comprising three major categories divided into eight subcategories, each including some code issues identified in the previous stage.

(3) Full Taxonomy Construction. Finally, after obtaining the taxonomy categorization, the remaining code snippets will be independently annotated by three newly invited volunteers with extensive Python programming experience, two with more than ten years of experience and one with four years of programming experience. The goal of the annotation is to identify issues in the LLM-generated code snippets and assess whether these issues fit within the established preliminary taxonomy. If any previously unknown code issues are identified, we will discuss whether existing hallucination (sub)categories can accommodate them or if a new type should be added to the taxonomy. Ultimately, we identify two new code issues *Asset Conflicts* and *Connection Conflicts*, but no new hallucination categories or subcategories.

3.4 Hallucination Statistics

During the process of taxonomy annotation, we record the specific code issues identified in each LLM-generated code snippet and count the frequency of each hallucination based on the relationships between hallucination (sub)categories and specific code issues. *It is important to note that for the code snippets that contain multiple code issues, all hallucinations present will be included in the statistics.*

4 Evaluation Results

In this section, we present the evaluation results and answer the three aforementioned research questions.

4.1 RQ1: Hallucination Taxonomy

The overall LLM coding hallucination taxonomy we obtained from Section 3.3 is presented in Figure 1. Through manual annotation, we identify three primary hallucination categories: *Task Requirement Conflicts*, *Factual Knowledge Conflicts*, and *Project Context Conflicts*, which can be

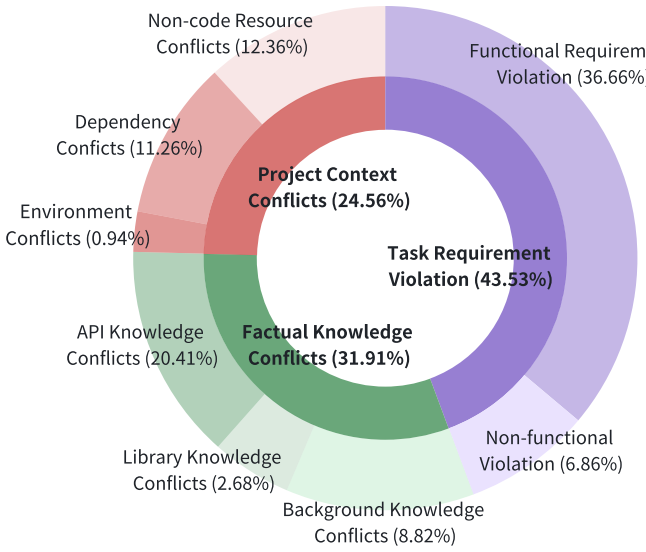


Fig. 2. Hallucination Distribution.

further divided into eight specific types. Note that our three primary categories align well with the hallucination types in the general domain [20]. Task requirement conflicts correspond to *input-conflicting hallucinations* in the general domain, indicating that the generated code does not meet the functional or non-functional requirements of the coding tasks. Factual knowledge conflicts correspond to *knowledge-conflicting hallucinations* in the general domain, indicating that the generated code does not comply with background knowledge, library/framework knowledge, or API knowledge. Project context conflicts correspond to *context-conflicting hallucinations* in the general domain, indicating that the generated code incorrectly uses project contexts, including environments, dependencies, and resources.

Figure 2 shows the distribution of the hallucination types, where the denominator is the sum of all annotated hallucination instances across the six LLMs. We present the detailed hallucination types in our taxonomy as follows.

4.1.1 Task Requirement Conflicts (43.53%). In the general domain, input-conflicting hallucinations occur when the answers generated by LLMs deviate from the original intentions of user inputs [19]. In the context of code generation tasks, the primary intentions of inputs typically revolve around the functional and non-functional requirements of the coding tasks. When the code generated by LLMs does not align with these requirements, hallucinations related to Task Requirement Conflicts occur. Specifically, these conflicts can be categorized into two types: *Functional Requirement Violation* and *Non-functional Requirement Violation*.

Functional Requirement Violation (36.66%). Functional requirements are typically expressed in natural language and describe the desired functionality of the generated code. When these requirements are not correctly and comprehensively understood, the resulting code may fail to meet expected functionality, leading to logic bugs (such as unexpected execution behaviors) or runtime errors (such as the `KeyError` during dictionary access). More specifically, the functional requirement mismatch can be subdivided into two typical types: *Wrong Functionality* and *Missing Functionality*. For example, as illustrated in Figure 3, the functional requirement involves handling

Fig. 3. Example: Functional Requirement Violation.

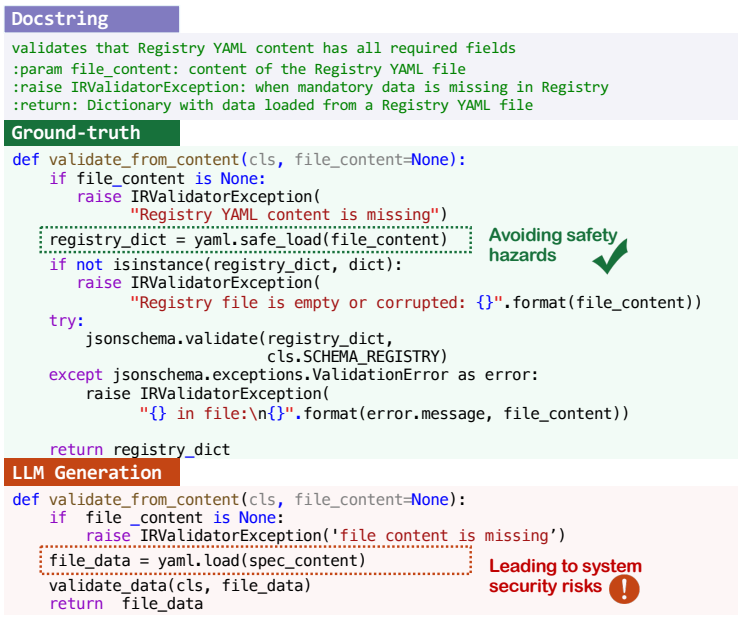


Fig. 4. Example: Non-functional Requirement Violation.

`LocalTime` based on the specific timezone `tz`. In the ground-truth code, this requirement is addressed by the lines highlighted in the green rectangle. However, the code generated by PanGu-α overlooks this requirement, resulting in a hallucination of Functional Requirement Violation.

Non-functional Requirement Violation (6.86%). Besides functional requirements, developers often have non-functional requirements for the generated code, such as security concerns or performance considerations. Although these non-functional requirements are often more implicit than functional ones and are not explicitly outlined in the user descriptions [41], they reflect common default requirements in real-world software development.

Our open coding annotation reveals that non-functional requirements in coding tasks can be mainly divided into the following aspects: *Security*, *Performance*, *Style*, and *Code Smell*. Generated code that violates these non-functional requirements may introduce safety risks or increase the maintenance complexity to the project.

Specifically, on the security side, the generated code may introduce vulnerabilities such as unsanitized inputs, which can lead to insecure deserialization or SQL injection attacks. As shown in Figure 4, the ground-truth code uses the `safe_load` function to safely read YAML files. In contrast, the LLM-generated code utilizes the `load` function, thereby introducing a potential security risk. Regarding performance, the generated code may lack optimization for execution efficiency, for example, by using inefficient loop structures that lead to unnecessary overhead in computing and memory resources. Style violations often occur when the generated code fails to follow established programming conventions or style guides, such as inconsistent naming conventions or inappropriate code layout, which can negatively affect code readability and maintainability. Code smell violations include issues such as overly complex functions or excessive use of global variables, which increase the complexity and potential risks associated with future maintenance.

4.1.2 Factual Knowledge Conflicts (31.91%). In the field of NLP, the term "factual conflicts" refers to content generated by LLMs that does not align with established knowledge or facts about the real world. Practical software development similarly relies on various types and levels of factual knowledge to produce correct code. Consequently, when LLMs fail to accurately understand and apply background knowledge [40], library/framework knowledge, or API knowledge, hallucinations on *Factual Knowledge Conflicts* arise. We further divide this hallucination category into three types: *Background Knowledge Conflicts*, *Library Knowledge Conflicts*, and *API Knowledge Conflicts*.

Background Knowledge Conflicts (8.82%). Background Knowledge Conflicts are a common issue when using large language models. These conflicts refer to the situation that the generated code is inconsistent with existing domain-specific knowledge, potentially rendering the code invalid or introducing logic bugs and risks. For instance, in automotive software development, if the generated code fails to adhere to certain industry standards (e.g., AUTOSAR [1]), it can result in significant compliance issues or safety risks.

Background knowledge typically includes *Domain Concepts* (e.g., specific data formats or protocols) and related *Standards and Specifications* (e.g., standard parameters or configurations). For example, Figure 5 shows an example about OCFL (Oxford Common File Layout), a specification for data storage and transformation. According to the official description [3], an OCFL storage root must contain a "Root Conformance Declaration" following the "NAMASTE" specification and may include a file named `ocfl_layout.json` to describe the root layout arrangement. However, the generated code might incorrectly focus on other OCFL aspects that are irrelevant to storage root.

Library Knowledge Conflicts (2.68%). In modern software development, developers frequently employ frameworks or third-party libraries (e.g., Django [2] for web applications) to expedite the development process by reusing the features or functionalities that these frameworks or libraries provide. When utilizing these frameworks or libraries, LLMs may encounter factual errors that lead to unexpected behaviors or even security risks. For example, As depicted in Figure 6, the task requires the model to generate a decorator that caches the return value of the function upon each invocation. In the code generated by the DeepSeekCoder model, the APIs from the `asyncio` framework are utilized. This framework is designed for asynchronous processing, and the model's misuse of this framework poses unexpected behaviors to the developed application.

API Knowledge Conflicts (20.41%). API Knowledge Conflicts are a common hallucination in LLM-generated code caused by various types of API misuses, such as parameter errors, improper guard conditions, similar-but-incorrect/deprecated API usage, and improper exception handling.

Fig. 5. Example: Background Knowledge Conflicts.

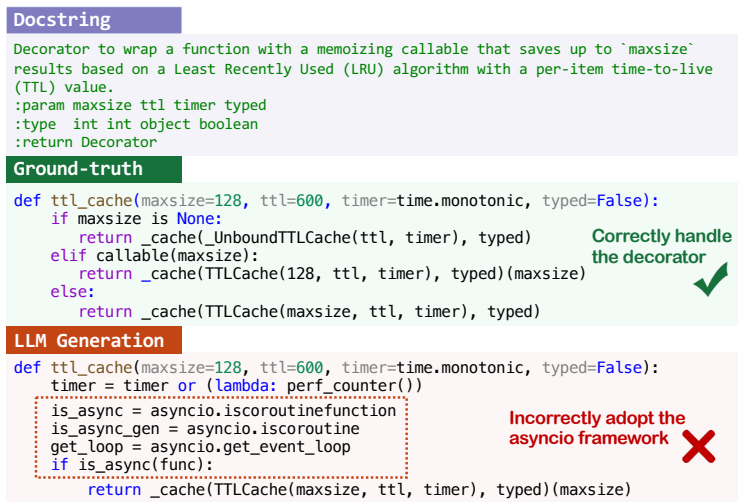


Fig. 6. Example: Library Knowledge Conflicts.

For example, parameter errors can occur when inappropriate parameter types or values are used in the generated code, causing API calls to fail or return unexpected results. This case is especially common in dynamically typed programming language such as Python. Improper guard conditions mean that the generated code does not correctly implement pre-condition checks. If the validity of the pre-conditions of certain APIs is not verified before calling them (e.g., file existence), runtime errors may occur. In terms of similar-but-wrong/deprecated API usage, LLMs may mistakenly choose APIs with similar functions but different applicable scenarios. Although this choice is syntactically correct, it cannot meet actual application needs. Improper exception handling involves generating code that fails to properly handle potential exceptions throwed by certain APIs, which can cause the program to crash or behave abnormally when faced with an error condition. This kind of API knowledge conflict will not only directly lead to program functional errors, but may also affect the stability of the system and the usability of the code.

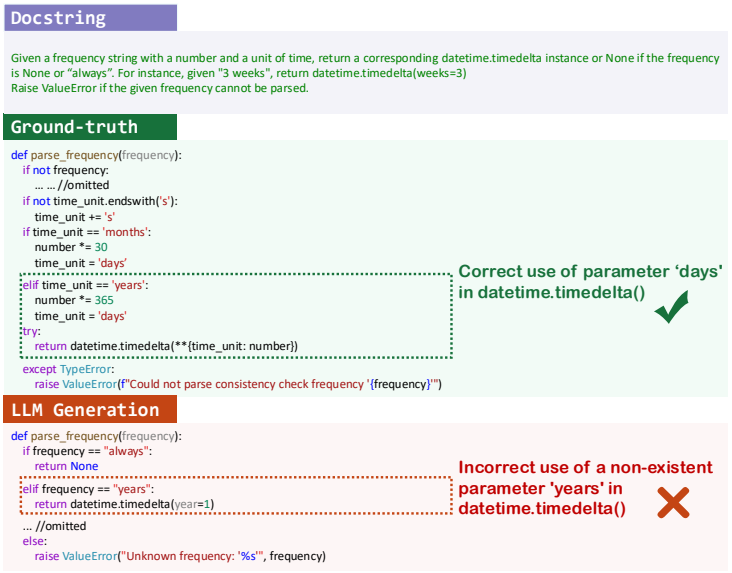


Fig. 7. Example: API Knowledge Conflicts.

We present an example in Figure 7. In this generation task, CodeGen correctly identifies the task intent and utilizes the `datetime.timedelta()` function. However, the code snippet generated by CodeGen uses a non-existing parameter `year`.

4.1.3 Project Context Conflicts (24.56%). *Project Context Conflict* hallucination refers to the phenomenon where the code generated by LLMs is inconsistent with the specific context of a given project. In a sense, this type of hallucination is also a type of factual conflict, where facts within the current project context are violated. The key difference is that *Factual Knowledge Conflicts* involve common facts (e.g., libraries and APIs) that are publicly accessible, while *Project Context Conflicts* pertain to facts that are specific to the corresponding project, which are generally unavailable for public access. *Project Context Conflicts* are often caused by LLMs not aware of such project-specific facts when generating code. This hallucination can be divided into *Environment Conflicts*, *Dependency Conflicts*, and *Non-code Resource Conflicts*.

Environment Conflicts (0.94%). In the process of software development, conflicts between the generated code and the development environment are common, especially regarding version differences in platforms, operating systems, drivers, languages, compilers/interpreters, frameworks, and libraries. When generating code, such environmental concerns are often not considered, leading to problematic code if there are environment-sensitive operations. For example, if the generated code uses language features (e.g., `f`-string expressions) from higher Python versions that are not supported by the current development environment, a conflict arises. For example, Figure 8 shows a code snippet generated by CodeGen that attempts to use the package `_lfu_cache`, which does not exist in the current environment.

Dependency Conflicts (11.26%). Dependency Conflicts arise when the generated code relies on undefined or unimported dependencies, such as user-defined attributes and functions. This often results in errors such as undefined variables or no-member errors. In practical software development, 70% of functions are non-standalone and depend on entities defined elsewhere in the project or imported from third-party libraries [46]. Due to the inability of LLMs to access the entire project

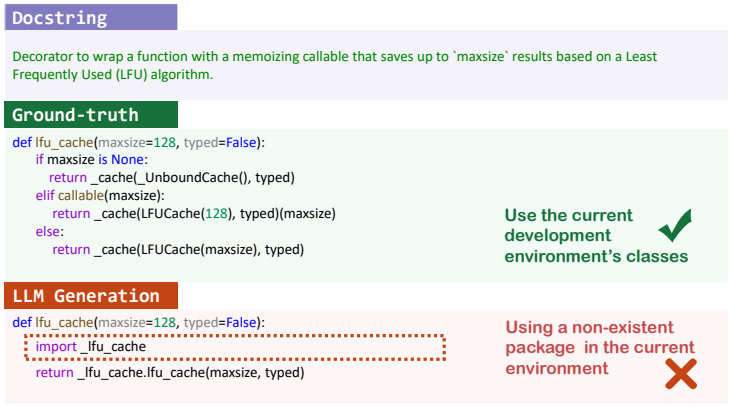


Fig. 8. Example: Environment Conflicts.

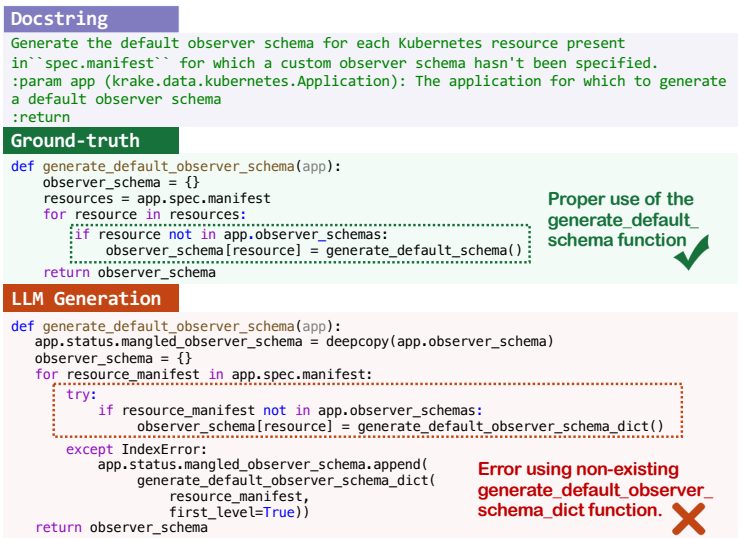


Fig. 9. Example: Dependency Conflicts.

context, they often resort to using non-existent APIs, functions, attributes, and variables when dealing with non-standalone functions.

For example, Figure 9 illustrates a scenario involving a user-defined function `generate_default_observer_schema_dict()`. In this case, the PanGu-α erroneously uses a function with a similar but incorrect name, `generate_default_schema()`, which does not exist in the project. This leads to a Dependency Conflict, as the code fails to execute correctly due to the missing definition.

Non-code Resource Conflicts (12.36%). Non-code Resource Conflicts can be further categorized into four main types: *Data*, *Configs*, *Assets*, and *Connections*. Each type of conflict can undermine the correctness and reliability of the system. Data conflicts often involve mishandling of data formats, fields, or content. For example, if the generated code incorrectly parses a data file or attempts to access a non-existent data field, it can lead to runtime errors or data inconsistencies. Config conflicts arise from incorrect settings or options in configuration files. This might include using undefined configuration fields or options, which can prevent the generated code from properly applying

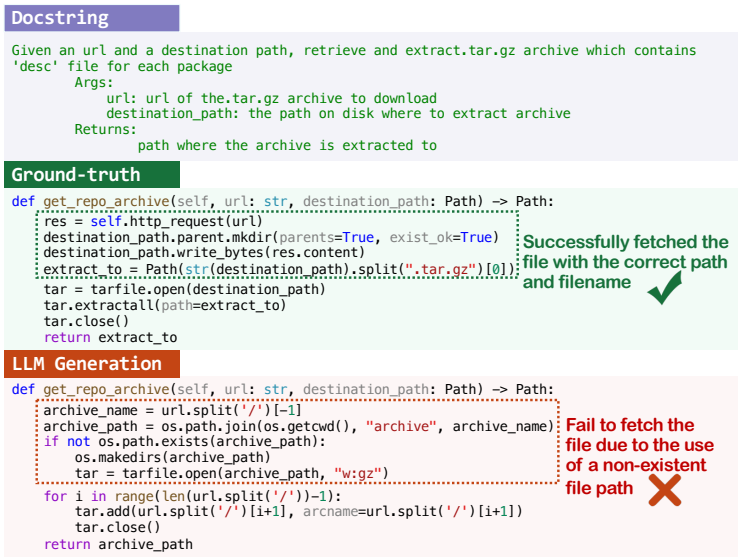


Fig. 10. Example: Non-code Resource Conflicts.

the configuration and affect system behavior. Asset conflicts here involve improper handling of asset files and their properties. For instance, if the generated code fails to set the correct size and resolution for images or videos, it can result in display issues or severe bugs, such as application crashes. Connection conflicts relate to wrong settings of various connection resources, such as incorrect IP addresses, port numbers, or database tables. These issues often lead to failed connections or operations being performed on the wrong server or database, potentially causing data leaks or security incidents.

For example, Figure 10 illustrates the generation task hopes that LLM can generate a function for a given URL and target path to retrieve and extract the `tar.gz` compressed package containing each package's "description" file. However, in the code snippet generated by the model, the model adds "archive" as a path in the target path, which causes the code snippet to point to a non-existent file path. This will not allow the `tar.gz` compressed package to be correctly obtained, resulting in a program error.

> **RQ1 Summary:** We have established a hallucination taxonomy in LLM-based code generation, comprising three main categories (i.e., *Task Requirement Conflicts*, *Factual Knowledge Conflicts*, and *Project Context Conflicts*) with eight subtypes. Among these, *Task Requirement Conflicts* are the most frequently occurring category.

4.2 RQ2: LLM Comparison

Based on the obtained hallucination taxonomy for LLM-based code generation, we further analyze the hallucination distribution comparison across different models. Figure 11 shows the distribution of the number of hallucinations of different models based on the breakdown analysis of the three hallucination types. We find that *Task Requirement Conflicts* are the most common hallucination type for all models, while *Factual Knowledge Conflicts* and *Project Context Conflicts* remain at approximately the same frequency.

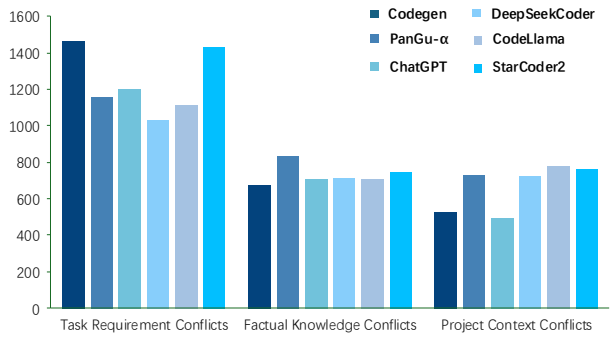


Fig. 11. Hallucination distribution of different models

Additionally, we find that CodeGen and StarCoder2 exhibit a notably higher frequency of hallucinations related to *Task Requirement Conflicts*, whereas DeepSeekCoder and CodeLlama demonstrates the lowest occurrence. This variation may be related to the models' ability to understand task requirements, potentially influenced by factors such as the model size or the training corpora. For instance, DeepSeekCoder and CodeLlama are trained on diverse corpora including both extensive code and text data, while CodeGen and StarCoder2 are primarily trained on code-related data. In terms of *Factual Knowledge Conflicts*, PanGu-α demonstrates the highest frequency of factual hallucinations. This can be attributed to its extensive training on Chinese corpora, which may have led to a relatively limited exposure to factual knowledge, such as specific domain concepts, expressed in English.

RQ2 Summary:
Task Requirement Conflicts are the most prevalent type of hallucination across all models, with CodeGen and StarCoder2 showing a notably higher frequency of this type compared to others.

4.3 RQ3: Potential Cause Discussion

In this research question, we conduct further analysis on the possible root causes of the hallucinations in practical LLM-based code generation.

4.3.1 Training Data Quality. The quality of the training data is a crucial factor in the development of LLMs, as it significantly affects models' inference capabilities. Recent LLMs are often trained on large-scale code corpora typically collected from open-source repositories. However, the quality of these repositories is not always assured, leading to the inclusion of low-quality data in the training corpora. Such issues include mismatches between docstrings and code [36], inefficient or insecure code implementations [22], misused API calls, outdated library documentation and usage [52], and a lack of domain diversity. When LLMs are trained on such corpora, they may unintentionally incorporate these flaws into their knowledge base, leading to hallucinations in code generation. As shown in Figure 4, LLMs may generate code that uses unsafe APIs, reflecting problematic patterns commonly found in the training data (e.g., there are 256k lines of Python code in the GitHub repository that use `yaml.load()` instead of the safer `yaml.safe_load()` API). This indicates that the model may have been affected by low-quality data during the training phase. Most hallucinations associated with *Task Requirement Conflicts* and *Factual Knowledge Conflicts* can be, to a certain extent, attributed to data quality issues in the training corpora. This highlights the importance of building a high-quality code-related training data to reduce hallucinations in code generation.

4.3.2 Intention Understanding Capacity. Although LLMs have shown great potential in code generation, they still face challenges in accurately capturing and interpreting specific user intentions and needs [30]. This limitation can result in generated code that is functionally or non-functionally

inaccurate, thereby affecting the overall effectiveness and trustworthiness of LLM-based code generation [35]. The core advantage of LLMs lies in their excellent pattern recognition capabilities, but this is also the source of their limitations. LLMs tend to generate code based on common patterns observed in the training data rather than from a deep understanding of the specific requirements context. As shown in Figure 3, the task description requires the LLM to handle `LocalTime`, which is ignored in the LLM-generated code. This example highlights LLMs' inadequacy in comprehensively interpreting the intentions behind requirements. Furthermore, LLMs also show limitations in handling subtle requirements involving complex logic or multi-step operations [47]. Due to a poor understanding of the overall scope and potential limitations of the task, LLM-generated code may only address part of the requirements or perform poorly in handling edge cases. This can result in generated code snippets that seem correct on the surface but fail to meet specific business logic or functional requirements in practice. As shown in Figure 4, although the code generated by LLMs functionally matches the description of the problem, such code will be more vulnerable to attacks in real development scenarios.

4.3.3 Knowledge Acquisition Capacity. LLMs may learn incorrect knowledge and miss certain domain-specific knowledge due to the aforementioned training data quality issues. There is a disparity in the distribution of data across different domains within the training dataset. The computation domain constitutes a substantial 63% of the total data, while the network domain is markedly less represented, comprising only 8% [53]. For example, as shown in Figure 5, the task description needs a piece of code for generating a data format that satisfies the OCFL storage specification, but LLM generates incorrect code, possibly due to its lack of the OCFL-related knowledge. Moreover, as software development techniques evolve, such as library updates, relevant knowledge developed after model training period cannot be acquired by LLMs. Unlike human developers who can continuously learn and integrate latest information during development, LLMs are limited to the knowledge available at the time of training. This limitation in LLMs' knowledge acquisition capacity leads to hallucinations related to incorrect or outdated factual information in the generated code. This highlights the need for a knowledge acquisition mechanism, such as retrieval augmented generation (RAG), to allow LLMs to update, correct, and supplement the knowledge they have learned.

4.3.4 Repository-level Context Awareness. Feeding all project contexts, including code, documents, and non-code resources, into an LLM for repository-level code generation is challenging and impractical. This is because LLMs, typically based on the Transformer architecture [38], have token number limits (e.g. 8k or 12k tokens) and experience quadratic computation growth as the number of tokens increases. Additionally, including all project contexts can introduce a significant amount of irrelevant information, hindering LLMs' ability to focus on the most relevant context for code generation. Therefore, it is crucial to develop methods that make LLMs aware of the project contexts (project-specific memory) that are precisely related to the current coding task. Recent works attempt to integrate static analysis tools [42] or apply retrieval-augmented generation (RAG) based on repository-level retrieval corpora [49] to address such context awareness issues.

RQ3 Summary: By further analyzing the causes of hallucinations, we identify four possible contributing factors: training data quality, intention understanding capacity, knowledge acquisition capacity, and repository-level context awareness. Deficiencies in any of these factors can lead to hallucinations in practical development scenarios.

5 Mitigation Approach

5.1 Motivation

The aforementioned root causes of hallucinations in code generated by LLMs can be traced back to three main factors at the inference stage: incorrect or insufficient understanding for task requirements, the lack of factual knowledge pertinent to the generation tasks, and the inability to access the necessary code and non-code resources from the repository. These limitations create substantial challenges for LLMs in code generation in practical development settings. Drawing inspiration from existing work [49] on repository-level code generation, we explore the feasibility of applying retrieval-augmented generation (RAG) to mitigate hallucinations. The idea is that by providing LLMs with code snippets relevant to the current task, they can better understand the requirements and gain awareness of specific factual knowledge and project contexts.

5.2 RAG-based Mitigation

To implement the RAG method, we first collect all code repositories from the CoderEval dataset and follow RepoCoder's method [49] to construct the retrieval corpora. Specifically, for each repository, we apply a sliding window to scan all the source files in it. This scanning process extracts consecutive lines of code based on a predefined window size. The sliding window moves by a fixed number of lines (slicing step) at each iteration to ensure complete coverage of the code. We adhere to RepoCoder's parameter settings, with a window size of 20 lines and a sliding step of 2 lines. To prevent answer leakage, code lines containing or following the ground-truth code are excluded from the scanning process. Once all files are processed, a retrieval corpus of code snippets is generated for the repository.

We employ a sparse bag-of-words (BOW) model for our retrieval mechanism, which simplifies gauging similarity between textual data. This model transmutes both the query and the candidate code snippets into sets of tokens, which are compared using the Jaccard index. The Jaccard index measures the similarity between two sets by dividing the size of their intersection by the size of their union, we choose the code snippet that retrieves the top ten scores each time to return as the prompt for the LLMs.

5.3 Evaluation

We evaluate the effectiveness of the RAG-based mitigation method with the six LLMs: CodeGen, PanGu-α, ChatGPT, DeepSeekCoder, CodeLlama, and StarCoder2 on the CodeEval dataset. We compared our RAG-based mitigation method with the Raw method. In the Raw method, we only provide LLMs basic docstrings and function signatures. In the RAG-based mitigation, when providing docstrings and function signatures, we will obtain ten related code snippets from the above-constructed retrieval library through a similarity algorithm as prompts and provide them to LLMs. We use the Pass@1 metric to assess the functionality correctness of the generated code snippets according to test cases. As shown in Table 1, the Pass@1 scores of all six models are slightly improved with the RAG-based mitigation method. Note that the performance improvement in our experiments is modest, as the mitigation method we explored is preliminary. We consider this experiment as an pilot study to explore the potential effectiveness of RAG-based mitigation. In

Table 1. Experimental results of mitigation method under Pass@1.

Model	Raw Method	RAG-based Mitigation
CodeGen	1.30%	2.61% (↑ 1.31%)
PanGu-α	0.04%	1.74% (↑ 1.70%)
DeepSeekCoder	3.04%	3.91% (↑ 0.87%)
CodeLlama	2.17%	5.22% (↑ 3.05%)
StarCoder2	0.04%	2.61% (↑ 2.57%)
ChatGPT	10.40%	14.78% (↑ 4.38%)

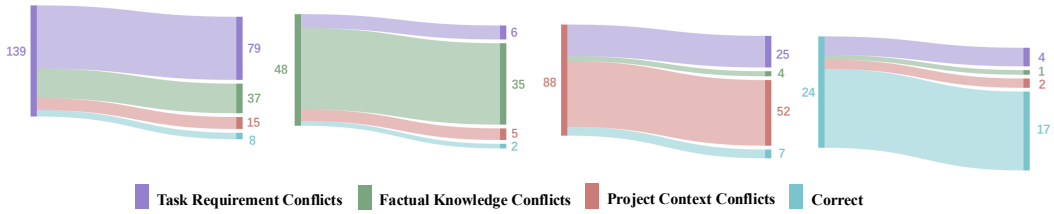


Fig. 12. Hallucination mitigation of ChatGPT.

future work, there are more methods worth studying, such as model fine-tuning and multi-agent framework with tool using, etc.

We also investigate how hallucinations evolve before and after employing RAG-based mitigation. We manually track the hallucinations in the code snippets generated by ChatGPT in both the RAW Method and RAG-based Mitigation. Figure 12 presents the analysis results in four sub-figures, each representing the evolution of a specific category of hallucination. The results reveal several interesting findings. First, among the three categories of hallucinations, the RAG-based mitigation shows the least impact on *Factual Knowledge Conflicts* (see the second sub-figure). This is intuitive, as the RAG in our experiments is utilized to retrieve code contexts from the current working repositories rather than from external knowledge sources such as API documentation or online knowledge bases like Wikipedia. Consequently, it does not introduce substantial factual knowledge regarding background, libraries, or APIs. Second, employing the RAG-based method can mitigate hallucinations related to *Task Requirement Conflicts* and *Project Context Conflicts* to a greater extent than those related to *Factual Knowledge Conflicts* (see the first and third sub-figures). Specifically, RAG has successfully resolved approximately 8% of *Project Context Conflicts* and about 6% of *Task Requirement Conflicts*. This improvement can be attributed to the code snippets retrieved using the current task descriptions, which enrich the information regarding the requirements and provide project-specific context. Third, while the RAG-based approach mitigates some hallucinations, it can also introduce new ones or alter existing hallucination types. For instance, approximately 30% of code generation tasks that were accurately handled by the RAW Method are now mishandled by the RAG-based Mitigation (see the last sub-figure) due to the introduction of new hallucinations. This may occur because the retrieved code snippets can sometimes act as noise rather than providing informative context, highlighting the importance of effective retrieval algorithms when implementing RAG-based mitigation.

To further illustrate the effectiveness of the hallucination mitigation, we conduct two case studies. As shown in Figure 13, in the Raw method, which only provides a docstring and a function signature, CodeGen incorrectly uses the `replace` function and fails to convert scripts to one-line commands. In contrast, with the RAG-based method, CodeGen correctly uses the `splitlines` function, aligning with the ground-truth and successfully addressing the requirement. In addition, the RAG-based method can also effectively mitigate *Project Context Conflicts*. As shown in Figure 14, in the Raw method, ChatGPT attempts to use the `self.items.popitem()` API, which does not exist in the repository, leading to hallucinated generation. In contrast, with the RAG-based mitigation, ChatGPT correctly implements the requirement using the `self.pop()` function.

6 Discussion

We provide implications for future research on the hallucinations in practical LLM-based code generation.

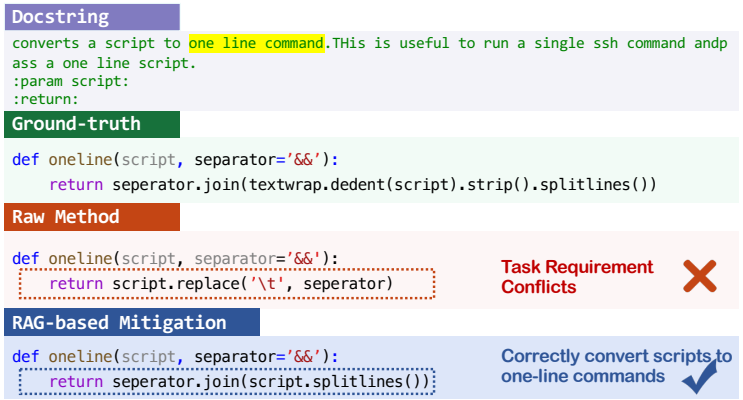


Fig. 13. Example: hallucination mitigation in Task Requirement Conflicts.

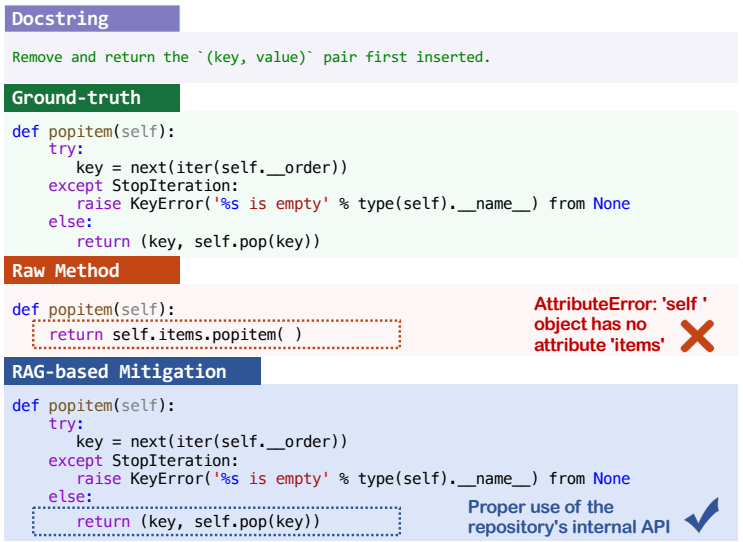


Fig. 14. Example: hallucination mitigation in Project Context Conflicts.

Developing hallucination identification techniques: Through our study, we find three major categories of hallucinations in the LLM-based code generation. Some hallucinations like *Dependency Conflicts*, *Environment Conflicts* and *API Knowledge Conflicts* can be detected by using static analysis (e.g., undefined variables or wrong API methods) or dynamic test execution (runtime errors or test failures), making it relatively easy for developers to recognize and locate the relevant code issues. However, certain hallucinations, such as incomplete functionality and security issues, are very difficult for developers to detect and correct, as they can likely pass static checks and all test cases. As a result, LLM-generated code containing these hallucinations may be introduced into development projects and even real production environments, leading to unreliable software systems and severe security risks. Existing hallucination localization approaches [4, 29] based on LLM self-feedback methods can detect hallucinations to a certain extent. However, these approaches heavily rely on the current model's capabilities and cannot address the fundamental limitations imposed by the training corpora. Therefore, in future work, researchers may consider developing more effective techniques to quickly and precisely identify and localize hallucinations in LLM-generated code.

Developing more effective hallucination mitigation techniques: In Section 5, we explore the feasibility of applying a lightweight RAG-based method to mitigate hallucinations in LLM-based code generation. While the method demonstrates effectiveness in mitigating hallucinations such as undefined attributes, the potentials of RAG need to be further explored. For example, we only construct retrieval corpus using current code repository, leading to the augmented information is insufficient to mitigate many hallucinations such as background knowledge conflicts. In the future, we can integrate more comprehensive knowledge sources like online search engines, API documents, and StackOverflow discussions. In addition to RAG techniques, other methods such as input query refinement [12, 30] and multi-agent systems [17] can also be leveraged to achieve an iterative process of (i) clarifying task requirements, (ii) generating code, (iii) running test cases, and (iv) mitigating hallucinations. To achieve this, we need to design the appropriate interaction protocols between agents and relevant tools (e.g., search engines and static analysis tools) and apply suitable prompting strategies.

7 Threats to Validity

External Validity. Threats to external validity mainly concern the generalizability of our findings. We focused on Python when exploring the taxonomy and root causes of hallucinations in LLM-based code generation due to its simplicity and ease of use. Constructing hallucination taxonomies for other programming languages and comparing them with our current taxonomy is a valuable future direction. Another potential threat is the limited scale of the adopted CoderEval dataset, which contains only 230 coding tasks. To mitigate this, we selected six LLMs and had each generate 10 code snippets for each task to ensure a sufficient number of annotations.

Internal Validity. Threats to internal validity primarily concern the manual annotation process in taxonomy construction. A key issue is the absence of formal inter-rater reliability measure for annotating hallucinations. To address this, discrepancies were discussed and resolved in annotator meetings to ensure a consistent annotation protocol, with each identified hallucination receiving a mutually agreed-upon label. Additionally, to ensure consistency in our findings, one author reviewed all labeled data. Another potential threat is model bias during the annotation process. To mitigate this, we mixed the generation results of the six models before annotation.

Construct Validity. Threats to construct validity are related to evaluating our hallucination mitigation approach. To alleviate these threats, we conducted experiments on six models using test cases available in the CoderEval dataset, a standard method for evaluating the correctness of generated code.

8 Conclusion

In this paper, we conduct an empirical study on code-generated hallucinations of large models in the practical development scenarios and through a full manual analysis, we construct a taxonomy of hallucinations and follow up with further hallucination classifications. Based on the hallucinations found, we provide a deeper discussion of the causes of hallucinations and the distribution of hallucinations in different LLMs. At last, we implement a RAG-based approach for hallucination mitigation and further discuss potential hallucination mitigation approaches.

9 Data Availability

We provide the replication package for this study at https://github.com/DeepSoftwareAnalytics/LLMCodingHallucination.

Acknowledgments

This work is supported by CCF-Huawei Populus Grove Fund CCF-HuaweiSE202403. This work is supported by the Guangdong Basic and Applied Basic Research Foundation (2023A1515012292) and CCF - Sangfor 'Yuanwang' Research Fund.

References

[1] [n. d.]. *AUTOSAR - Wikipedia Page*. Retrieved Nov. 1, 2024 from https://en.wikipedia.org/wiki/AUTOSAR

[2] [n. d.]. *Django - The web framework for perfectionists with deadlines*. Retrieved Nov. 1, 2024 from https://www.djangoproject.com/

[3] [n. d.]. *OCFL - Specifications*. Retrieved Nov. 1, 2024 from https://ocfl.io/1.1/spec/#storage-root

[4] Ayush Agrawal, Mirac Suzgun, Lester Mackey, and Adam Tauman Kalai. 2023. Do Language Models Know When They're Hallucinating References? *arXiv preprint arXiv:2305.18248* (2023).

[5] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy-Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! *CoRR* abs/2301.03988 (2023). https://doi.org/10.48550/ARXIV.2301.03988 arXiv:2301.03988

[6] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[7] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[8] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. *CoRR* abs/2204.06745 (2022). https://doi.org/10.48550/ARXIV.2204.06745 arXiv:2204.06745

[9] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow*. https://doi.org/10.5281/zenodo.5297715 If you use this software, please cite it using these metadata..

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

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

[12] Kaustubh D Dhole, Ramraj Chandradevan, and Eugene Agichtein. 2023. An interactive query generation assistant using LLM-based prompt modification and user feedback. *arXiv preprint arXiv:2311.11226* (2023).

[13] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861* (2023).

[14] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=hQwb-lbM6EL

[15] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. 2010. An empirical investigation into a large-scale Java open source code repository. In *Proceedings*

of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. 1–10.

[16] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[17] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680* (2024).

[18] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751* (2019).

[19] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2023. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *arXiv preprint arXiv:2311.05232* (2023).

[20] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *Comput. Surveys* 55, 12 (2023), 1–38.

[21] Shahedul Huq Khandkar. 2009. Open coding. *University of Calgary* 23, 2009 (2009).

[22] Jan H Klemmer, Stefan Albert Horstmann, Nikhil Patnaik, Cordelia Ludden, Cordell Burton Jr, Carson Powers, Fabio Massacci, Akond Rahman, Daniel Votipka, Heather Richter Lipford, et al. 2024. Using AI Assistants in Software Development: A Qualitative Study on Security Practices and Concerns. *arXiv preprint arXiv:2405.06371* (2024).

[23] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*. PMLR, 18319–18345.

[24] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories. *arXiv preprint arXiv:2404.00599* (2024).

[25] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023). https://doi.org/10.48550/ARXIV.2305.06161 arXiv:2305.06161

[26] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *CoRR* abs/2203.07814 (2022). https://doi.org/10.48550/ARXIV.2203.07814 arXiv:2203.07814

[27] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. 2024. Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. *arXiv preprint arXiv:2404.00971* (2024).

[28] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173* (2024).

[29] Potsawee Manakul, Adian Liusie, and Mark JF Gales. 2023. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. *arXiv preprint arXiv:2303.08896* (2023).

[30] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. 2023. ClarifyGPT: Empowering LLM-based Code Generation with Intention Clarification. *arXiv preprint arXiv:2310.10996* (2023).

[31] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[32] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=iaYcJKpY2B_

[33] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt.
[34] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). https://doi.org/10.48550/ARXIV.2308.12950 arXiv:2308.12950
[35] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Susmit Jha, Prem Devanbu, and Toufique Ahmed. 2024. Quality and Trust in LLM-generated Code. *arXiv preprint arXiv:2402.02047* (2024).
[36] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering*. 1609–1620.
[37] SM Tonmoy, SM Zaman, Vinija Jain, Anku Rani, Vipula Rawte, Aman Chadha, and Amitava Das. 2024. A comprehensive survey of hallucination mitigation techniques in large language models. *arXiv preprint arXiv:2401.01313* (2024).
[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html
[39] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.
[40] Chong Wang, Xin Peng, Zhenchang Xing, and Xiujie Meng. 2023. Beyond literal meaning: Uncover and explain implicit knowledge in code through wikipedia-based concept linking. *IEEE Transactions on Software Engineering* 49, 5 (2023), 3226–3240.
[41] Chong Wang, Xin Peng, Zhenchang Xing, Yue Zhang, Mingwei Liu, Rong Luo, and Xiujie Meng. 2023. Xcos: Explainable code search based on query scoping and knowledge graph. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–28.
[42] Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2024. Teaching Code LLMs to Use Autocompletion Tools in Repository-Level Code Generation. *arXiv preprint arXiv:2401.06391* (2024).
[43] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 1069–1088. https://aclanthology.org/2023.emnlp-main.68
[44] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/V1/2021.EMNLP-MAIN.685
[45] Hongbin Ye, Tong Liu, Aijia Zhang, Wei Hua, and Weiqiang Jia. 2023. Cognitive mirage: A review of hallucinations in large language models. *arXiv preprint arXiv:2309.06794* (2023).
[46] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
[47] Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. When language model meets private library. *arXiv preprint arXiv:2210.17236* (2022).
[48] Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, et al. 2021. PanGu-α: Large-scale Autoregressive Pretrained Chinese Language Models with Auto-parallel Computation. *arXiv preprint arXiv:2104.12369* (2021).
[49] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023).
[50] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren's song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219* (2023).
[51] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989* (2023).
[52] Li Zhong and Zilong Wang. 2024. Can LLM Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 21841–21849.

[53] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877* (2024).

Received 2024-10-31; accepted 2025-03-31