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

Ziyao Zhang[1], Yanlin Wang[1]*, Chong Wang[2], Jiachi Chen[1], Zibin Zheng[1]
[1]Sun Yat-sen University, Zhuhai, China
[2]Nanyang Technological University, Singapore

Abstract—**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. The replication package including code, data, and experimental results is available at https://github.com/DeepSoftwareAnalytics/LLMCodingHallucination.**

I. Introduction

Code generation is an automation technology aimed at efficiently producing code from specifications described in natural language. This process significantly reduces the manual coding workload for developers [6], [9], [10], 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) [14]–[16], [18]–[25], [37], [44], [51] built upon the Transformer architecture [1]. 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 [15] and MBPP [17], 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 from detailed specifications [59]. To address this complexity, new benchmarks, such as CoderEval [59], ClassEval [60], and EvoCodeBench [61], 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 primarily focus on functional correctness as measured by test case pass rates and lack a thorough analysis of underlying failure causes. *To bridge this gap, this work aims to systematically investigate issues in practical LLM-based code generation from the perspective of hallucinations.*

Hallucination is a significant issue for state-of-the-art generative LLMs [7]. For general natural language tasks, LLM hallucinations have been explored to a certain extent [5], [7], [8], [69] and are typically categorized into three types: Input-Conflicting Hallucination, Fact-Conflicting Hallucination, and Context-Conflicting Hallucination [69]. In the domain of code generation, Liu et al. [75] conducted a study to analyze hallucinations in LLM-powered code generation and established a taxonomy that aligns with these three categories. While Liu et al.'s study provided insightful findings, it is based on benchmarks (i.e., HumanEval [15] and DS-1000 [76]) for *standalone* function/script generation instead. Our work, however, focuses on hallucinations within more practical and complex development contexts in repository-level generation scenarios. Additionally, their study primarily categorized hallucinations from a problem-presentation perspective to uncover fine-grained code-semantic issues, resulting in categories such as *Dead Code* and *Repetition*. In this work, we investigate hallucinations from a holistic perspective in terms of ***phenomena***, ***mechanism***, and ***mitigation***. We believe that our study can complement the findings by Liu et al., providing a broader understanding of hallucinations in LLM-based code generation.

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 (Root Causes):** What are the root causes of hallucinations in practical LLM-based code generation?

* Corresponding author.

To answer the questions, we experiment on six mainstream LLMs (ChatGPT [55], CodeGen [16], PanGu-α [44], StarCoder2 [3], DeepSeekCoder [51], and CodeLlama [14]) with the CoderEval dataset [59]. To obtain the hallucination taxonomy of practical LLM-based code generation, we manually perform open coding [65] 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 construct a retrieval library based on the repository in the development scenario of each generation task and obtain the code snippet that is beneficial to the current generation task as a prompt through the similarity detection between the task description in the generation task and the code snippet in the retrieval library. 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 types hallucinations in LLM code generation in real development scenarios and establish a hallucination taxonomy in LLM-based code generation.
- 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.
- We make the replication package available at https://github.com/DeepSoftwareAnalytics/LLMCodingHallucination, to support further studies in this field.

II. Background & Related Work

A. LLM-based Code Generation

For developers, a realistic scenario is to use a code repository to write code, which is very common in practice [66]. 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. 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 [59], 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 [59], ClassEval [60], and EvoCodeBench [61] 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.

B. 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 [68]. 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 [69]: (1) Input-Conflicting Hallucinations: When the text generated by the model deviates from the original input source, input-conflicting hallucinations will occur. This illusion 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 illusion 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 illusion. 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, there has been work to define hallucinations for LLMs in code generation tasks. This work [8] defined new hallucination standards for LLMs in code generation tasks and divided hallucinations into five main types, but this work ignores that LLMs in real-world code generation tasks will involve relevant knowledge unique to the software engineering field such as development environment, system resources, external constraints, code warehouses, etc. These factors often cause LLMs to fail in actual development. Problems such as low usability and low accuracy. In order to better explore the illusions that exist in LLMs in real development scenarios, our work obtained data sets in real development scenarios for empirical study, and defined new types of illusions, which opened up new ideas for subsequent research on illusions.

III. Evaluation Setup

A. Dataset

To better simulate practical development scenarios, we use a set of coding tasks from real-world Python repositories based on the CoderEval benchmark [59]. CoderEval comprises 230 Python code generation tasks, extracted from a diverse set of Python repositories. Each task consists of a natural language description, a ground-truth code snippet, and a set of test cases, along with the project environment context associated with the task.

B. 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** [55]: 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** [53]: 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** [44]: PanGu-α can perform code generation tasks in multiple languages. We use the PanGu-α-2.6B model.
- **DeepSeekCoder** [51]: DeepSeekCoder performs well in open source models across multiple programming languages and various benchmarks. We use the DeepSeekCoder-6.7B base model.
- **CodeLlama** [14]: 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** [18]: 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 nuclear sampling strategy and setting temperature to 0.6, following the same setting as CoderEval.

C. Taxonomy Annotation

In order to analyze the hallucination types in the LLM-generated code, we manually perform open coding [65] 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 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 test it in the actual development environment corresponding to the task to determine its correctness. On this basis, the two authors will compare the differences between the ground-truth and LLMs generated code snippets and discuss and record possible hallucination phenomena.

(2) Preliminary Taxonomy Construction. Secondly, we document possible hallucinations in the generated code and the location of the hallucination content. Several different hallucinations may occur within a single code snippet. All annotators are required to discuss the codes and define the code's hallucinatory taxonomy. In this process, we classify similar hallucination to create a preliminary taxonomy that illustrates the various hallucination types and their meanings in the code generated by LLMs.

(3) Full Taxonomy Construction. Finally, after obtaining the categorisation criteria, 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. If new types of hallucinations arise that are not covered by the current taxonomy, annotators are required to write descriptions of the hallucinations to allow further discussion to establish new types and enhance the taxonomy.

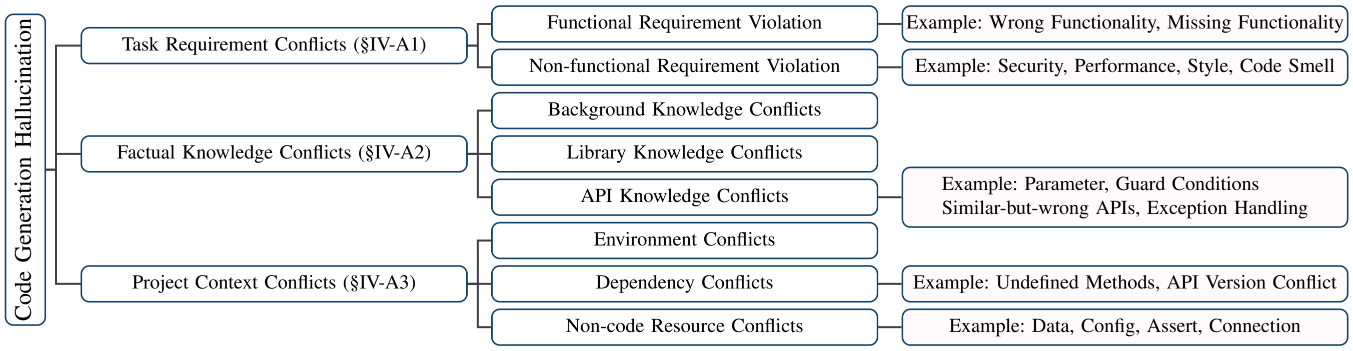

Code Generation Hallucination
Task Requirement Conflicts (§IV-A1)
Factual Knowledge Conflicts (§IV-A2)
Project Context Conflicts (§IV-A3)
Functional Requirement Violation
Non-functional Requirement Violation
Background Knowledge Conflicts
Library Knowledge Conflicts
API Knowledge Conflicts
Environment Conflicts
Dependency Conflicts
Non-code Resource Conflicts
Example: Wrong Functionality, Missing Functionality
Example: Security, Performance, Style, Code Smell
Example: Parameter, Guard Conditions
Similar-but-wrong APIs, Exception Handling
Example: Undefined Methods, API Version Conflict
Example: Data, Config, Assert, Connection

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

IV. Evaluation Results

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

A. RQ1: Hallucination Taxonomy

The overall LLM coding hallucination taxonomy we obtained from Section III-C 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 further divided into eight specific types. Note that our three primary categories align well with the hallucination types in the general domain [68]. 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. In the following, we present the detailed hallucination types in our taxonomy. Figure 2 shows the distribution of the hallucination types.

*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 [7]. 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.

<ins>Functional Requirement Violation (36.66%).</ins> 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 `LocalTime` based on the specific timezone `tz`. In the ground-truth code, this requirement is addressed by the

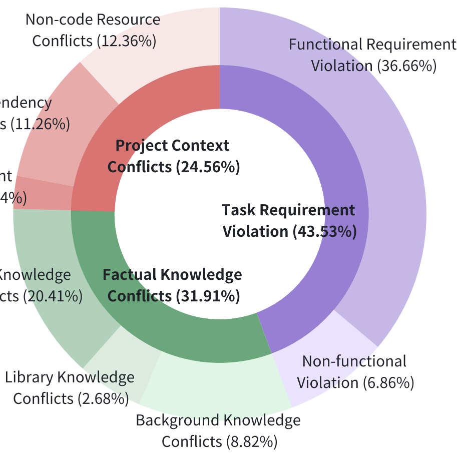
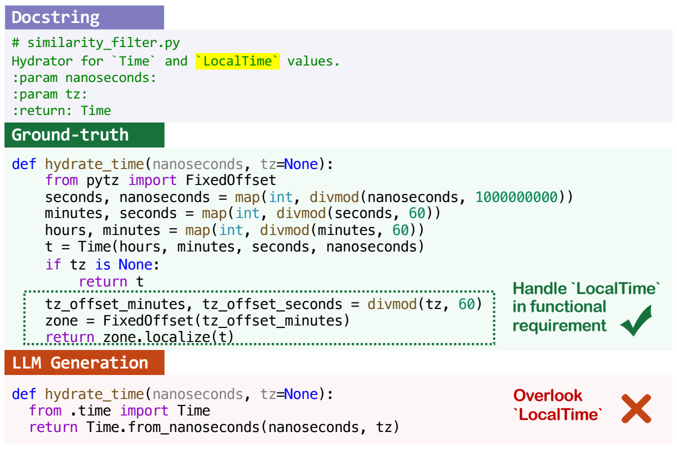


Fig. 2. Hallucination Distribution

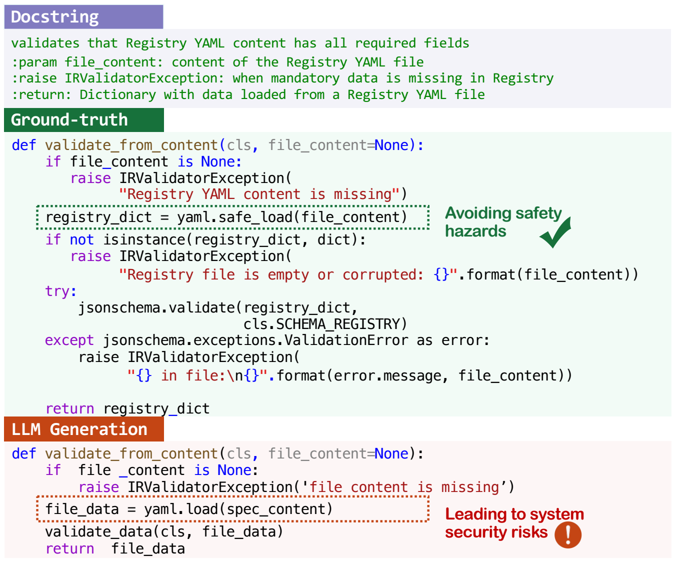


```
# similarity_filter.py
Hydrator for `Time` and `LocalTime` values.
:param nanoseconds:
:param tz:
:return: Time
```

```
def hydrate_time(nanoseconds, tz=None):
    from pytz import FixedOffset
    seconds, nanoseconds = map(int, divmod(nanoseconds, 1000000000))
    minutes, seconds = map(int, divmod(seconds, 60))
    hours, minutes = map(int, divmod(minutes, 60))
    t = Time(hours, minutes, seconds, nanoseconds)
    if tz is None:
        return t
    tz_offset_minutes, tz_offset_seconds = divmod(tz, 60)
    zone = FixedOffset(tz_offset_minutes)
    return zone.localize(t)
```

```
def hydrate_time(nanoseconds, tz=None):
  from .time import Time
  return Time.from_nanoseconds(nanoseconds, tz)
```

Fig. 3. Example: Functional Requirement Violation


Docstring

```
validates that Registry YAML content has all required fields
:param file_content: content of the Registry YAML file
:raise IRValidatorException: when mandatory data is missing in Registry
:return: Dictionary with data loaded from a Registry YAML file
```

Ground-truth

```
def validate_from_content(cls, file_content=None):
    if file_content is None:
       raise IRValidatorException(
             "Registry YAML content is missing")
    registry_dict = yaml.safe_load(file_content)
    if not isinstance(registry_dict, dict):
       raise IRValidatorException(
             "Registry file is empty or corrupted: {}".format(file_content))
    try:
        jsonschema.validate(registry_dict,
                            cls.SCHEMA_REGISTRY)
    except jsonschema.exceptions.ValidationError as error:
        raise IRValidatorException(
              "{} in file:\n{}".format(error.message, file_content))

    return registry_dict
```

LLM Generation

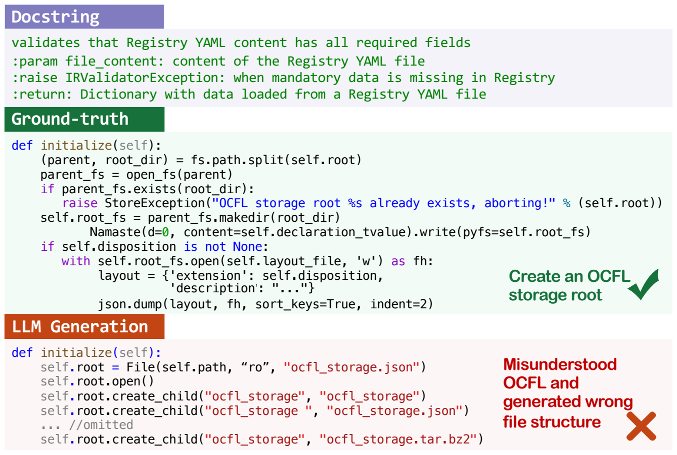
```
def validate_from_content(cls, file_content=None):
    if  file _content is None:
        raise IRValidatorException('file content is missing')
    file_data = yaml.load(spec_content)
    validate_data(cls, file_data)
    return  file_data
```

Fig. 4. Example: Non-functional Requirement Violation

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. These non-functional requirements are usually more implicit than functional requirements and are not described in the input natural language descriptions. 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 of the corresponding 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.


Docstring

```
validates that Registry YAML content has all required fields
:param file_content: content of the Registry YAML file
:raise IRValidatorException: when mandatory data is missing in Registry
:return: Dictionary with data loaded from a Registry YAML file
```

Ground-truth

```
def initialize(self):
    (parent, root_dir) = fs.path.split(self.root)
    parent_fs = open_fs(parent)
    if parent_fs.exists(root_dir):
       raise StoreException("OCFL storage root %s already exists, aborting!" % (self.root))
    self.root_fs = parent_fs.makedir(root_dir)
           Namaste(d=0, content=self.declaration_tvalue).write(pyfs=self.root_fs)
    if self.disposition is not None:
       with self.root_fs.open(self.layout_file, 'w') as fh:
            layout = {'extension': self.disposition,
                      'description': "..."}
            json.dump(layout, fh, sort_keys=True, indent=2)
```

LLM Generation

```
def initialize(self):
    self.root = File(self.path, “ro”, "ocfl_storage.json")
    self.root.open()
    self.root.create_child("ocfl_storage", "ocfl_storage")
    self.root.create_child("ocfl_storage ", "ocfl_storage.json")
    ... //omitted
    self.root.create_child("ocfl_storage", "ocfl_storage.tar.bz2")
```

Create an OCFL storage root
Misunderstood OCFL and generated wrong file structure

Fig. 5. Example: Background Knowledge Conflicts

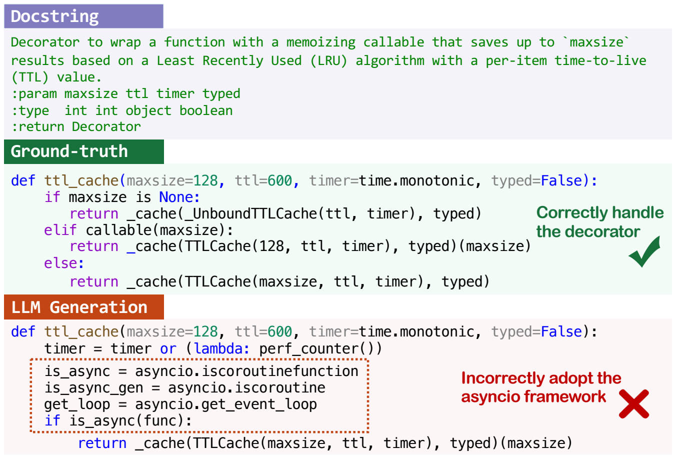
*2) **Factual Knowledge Conflicts (31.91%)**:* In the field of NLP, the term "factual conflicts" refers to content generated by LLMs that does not align with established knowledge or facts about the real world. Practical software development similarly relies on various types and levels of factual knowledge to produce correct code. Consequently, when LLMs fail to accurately understand and apply background knowledge [67], 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[2], 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. While the ground-truth code aligns with these specifications when initializing the OCFL storage root, the generated code might incorrectly focus on other OCFL aspects that are irrelevant to creating the storage root.

Library Knowledge Conflicts (2.68%). In modern software development, developers frequently employ frameworks or

[1]https://en.wikipedia.org/wiki/AUTOSAR
[2]https://ocfl.io/1.1/spec/#storage-root


Docstring
Ground-truth
Correctly handle the decorator
LLM Generation
Incorrectly adopt the asyncio framework

```
Decorator to wrap a function with a memoizing callable that saves up to `maxsize`
results based on a Least Recently Used (LRU) algorithm with a per-item time-to-live
(TTL) value.
:param maxsize ttl timer typed
:type  int int object boolean
:return Decorator
```

```
def ttl_cache(maxsize=128, ttl=600, timer=time.monotonic, typed=False):
    if maxsize is None:
       return _cache(_UnboundTTLCache(ttl, timer), typed)
    elif callable(maxsize):
       return _cache(TTLCache(128, ttl, timer), typed)(maxsize)
    else:
       return _cache(TTLCache(maxsize, ttl, timer), typed)
```

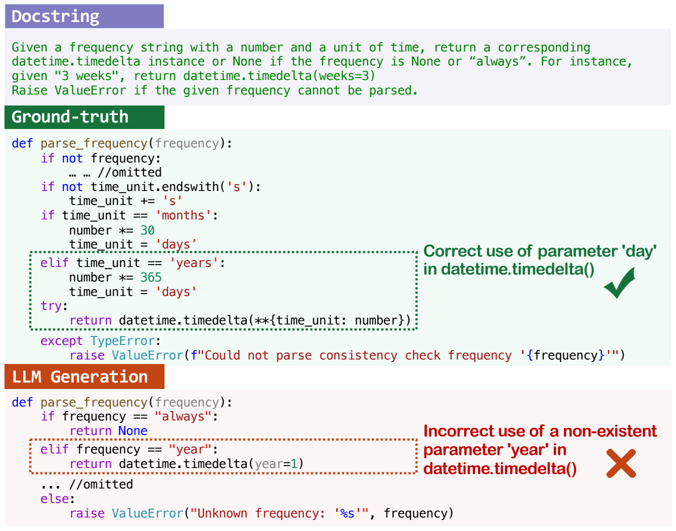
```
def ttl_cache(maxsize=128, ttl=600, timer=time.monotonic, typed=False):
    timer = timer or (lambda: perf_counter())
    is_async = asyncio.iscoroutinefunction
    is_async_gen = asyncio.iscoroutine
    get_loop = asyncio.get_event_loop
    if is_async(func):
        return _cache(TTLCache(maxsize, ttl, timer), typed)(maxsize)
```

Fig. 6. Example: Library Knowledge Conflicts

third-party libraries (e.g., Django[3] 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 the asynchronous processing 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. 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 [29]. 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, 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.

[3]https://www.djangoproject.com/


Docstring
Ground-truth
Correct use of parameter 'day' in datetime.timedelta()
LLM Generation
Incorrect use of a non-existent parameter 'year' in datetime.timedelta()

```
Given a frequency string with a number and a unit of time, return a corresponding
datetime.timedelta instance or None if the frequency is None or “always”. For instance,
given "3 weeks", return datetime.timedelta(weeks=3)
Raise ValueError if the given frequency cannot be parsed.
```

```
def parse_frequency(frequency):
    if not frequency:
        … … //omitted
    if not time_unit.endswith('s'):
        time_unit += 's'
    if time_unit == 'months':
        number *= 30
        time_unit = 'days'
    elif time_unit == 'years':
        number *= 365
        time_unit = 'days'
    try:
        return datetime.timedelta(**{time_unit: number})
    except TypeError:
        raise ValueError(f"Could not parse consistency check frequency '{frequency}'")
```

```
def parse_frequency(frequency):
    if frequency == "always":
        return None
    elif frequency == "year":
        return datetime.timedelta(year=1)
    ... //omitted
    else:
        raise ValueError("Unknown frequency: '%s'", frequency)
```

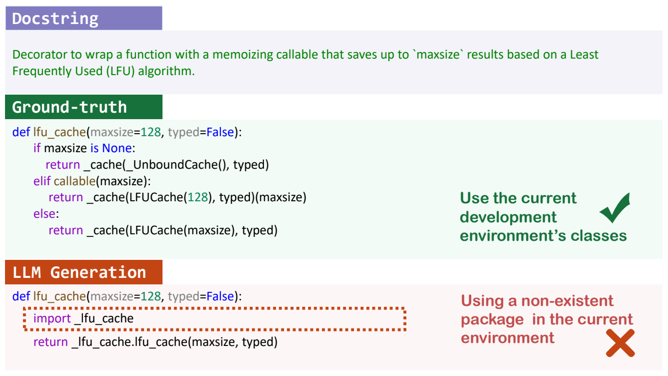
Fig. 7. Example: API Knowledge Conflicts

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`.

*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



Docstring

```
Decorator to wrap a function with a memoizing callable that saves up to `maxsize` results based on a Least
Frequently Used (LFU) algorithm.
```

Ground-truth

```
def lfu_cache(maxsize=128, typed=False):
    if maxsize is None:
      return _cache(_UnboundCache(), typed)
    elif callable(maxsize):
      return _cache(LFUCache(128), typed)(maxsize)
    else:
      return _cache(LFUCache(maxsize), typed)
```

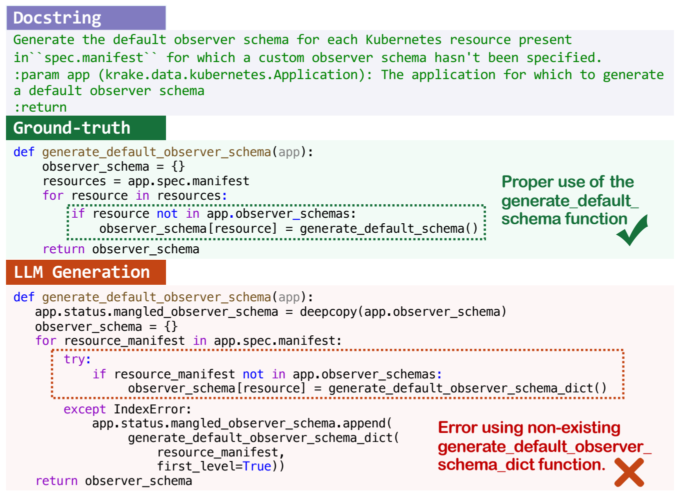
Use the current development environment's classes ✓

LLM Generation

```
def lfu_cache(maxsize=128, typed=False):
    import _lfu_cache
    return _lfu_cache.lfu_cache(maxsize, typed)
```

Using a non-existent package in the current environment ✗

Fig. 8. Example: Environment Conflicts



Docstring

```
Generate the default observer schema for each Kubernetes resource present
in``spec.manifest`` for which a custom observer schema hasn't been specified.
:param app (krake.data.kubernetes.Application): The application for which to generate
a default observer schema
:return
```

Ground-truth

```
def generate_default_observer_schema(app):
    observer_schema = {}
    resources = app.spec.manifest
    for resource in resources:
        if resource not in app.observer_schemas:
            observer_schema[resource] = generate_default_schema()
    return observer_schema
```

Proper use of the generate_default_schema function ✓

LLM Generation

```
def generate_default_observer_schema(app):
   app.status.mangled_observer_schema = deepcopy(app.observer_schema)
   observer_schema = {}
   for resource_manifest in app.spec.manifest:
       try:
           if resource_manifest not in app.observer_schemas:
               observer_schema[resource] = generate_default_observer_schema_dict()
       except IndexError:
           app.status.mangled_observer_schema.append(
               generate_default_observer_schema_dict(
                   resource_manifest,
                   first_level=True))
   return observer_schema
```

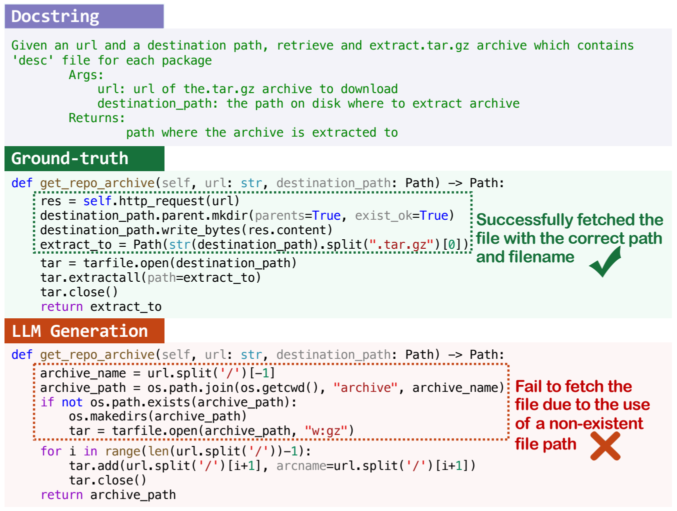
Error using non-existing generate_default_observer_schema_dict function. ✗

Fig. 9. Example: Dependency Conflicts

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 [59]. Due to the inability of LLMs to access the entire project 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 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.



Docstring

```
Given an url and a destination path, retrieve and extract.tar.gz archive which contains
'desc' file for each package
        Args:
            url: url of the.tar.gz archive to download
            destination_path: the path on disk where to extract archive
        Returns:
                path where the archive is extracted to
```

Ground-truth

```
def get_repo_archive(self, url: str, destination_path: Path) -> Path:
    res = self.http_request(url)
    destination_path.parent.mkdir(parents=True, exist_ok=True)
    destination_path.write_bytes(res.content)
    extract_to = Path(str(destination_path).split(".tar.gz")[0])
    tar = tarfile.open(destination_path)
    tar.extractall(path=extract_to)
    tar.close()
    return extract_to
```

Successfully fetched the file with the correct path and filename ✓

LLM Generation

```
def get_repo_archive(self, url: str, destination_path: Path) -> Path:
    archive_name = url.split('/')[-1]
    archive_path = os.path.join(os.getcwd(), "archive", archive_name)
    if not os.path.exists(archive_path):
        os.makedirs(archive_path)
        tar = tarfile.open(archive_path, "w:gz")
    for i in range(len(url.split('/'))-1):
        tar.add(url.split('/')[i+1], arcname=url.split('/')[i+1])
        tar.close()
    return archive_path
```

Fail to fetch the file due to the use of a non-existent file path ✗

Fig. 10. Example: Non-code Resource Conflicts

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.

B. 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

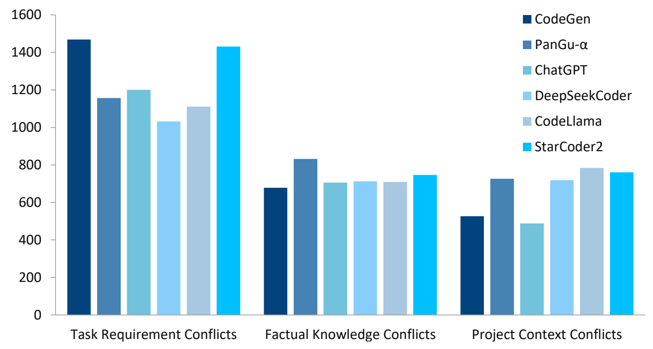


Fig. 11. Hallucination distribution of different models

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. 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.

C. RQ3: Root Cause Analysis

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

*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 [77], inefficient or insecure code implementations [46], misused API calls, outdated library documentation and usage [50], 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. 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.

*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 [49]. 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 [48]. 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.

*3) **Knowledge Acquisition Capacity**:* LLMs may learn incorrect knowledge and miss certain domain-specific knowledge due to the aforementioned training data quality issues. 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. 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 during inference. 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) **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 [1], 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 [78] or apply retrieval-augmented generation (RAG) based on repository-level retrieval corpora [62] 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.

V. Mitigation Approach

A. 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. There are many previous works investing LLM-based code generation [26]–[28], [28], [30], [31], [33]–[35], we draw inspiration from existing work [62] on repository-level code generation and 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.

B. RAG-based Mitigation

To implement the RAG method, we first collect all code repositories from the CoderEval dataset and follow RepoCoder's method [62] 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.

TABLE I
Experimental Results of Mitigation Method under Pass@1.

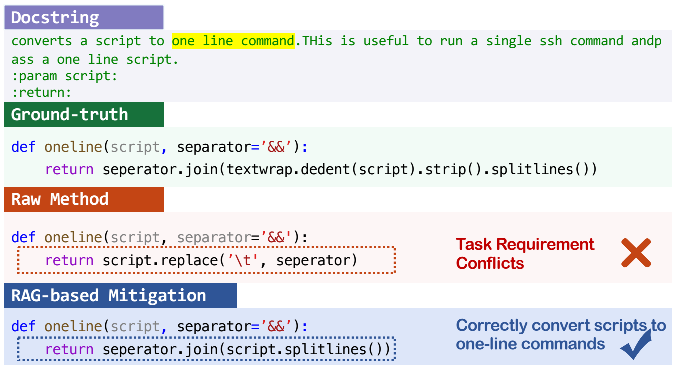
Model	Raw Method	RAG-based Mitigation
CodeGen	1.30%	2.61% (↑ 1.31%)
PanGu-α	0.04%	1.74% (↑ 1.70%)
DeepSeekCoder	3.04%	3.91% (↑ 0.87%)
CodeLlama	2.17%	5.22% (↑ 3.05%)
StarCoder2	0.04%	2.61% (↑ 2.57%)
ChatGPT	10.40%	12.61% (↑ 2.21%)

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.

C. 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 I, the Pass@1 scores of all six models are consistently 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 future work, there are more methods worth studying, such as model fine-tuning and multi-agent framework with tool using, etc.

To further illustrate the effectiveness of the hallucination mitigation, we conduct two case studies. As shown in Figure 12, 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 13, 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


Docstring
Ground-truth
Raw Method
Task Requirement Conflicts
RAG-based Mitigation
Correctly convert scripts to one-line commands

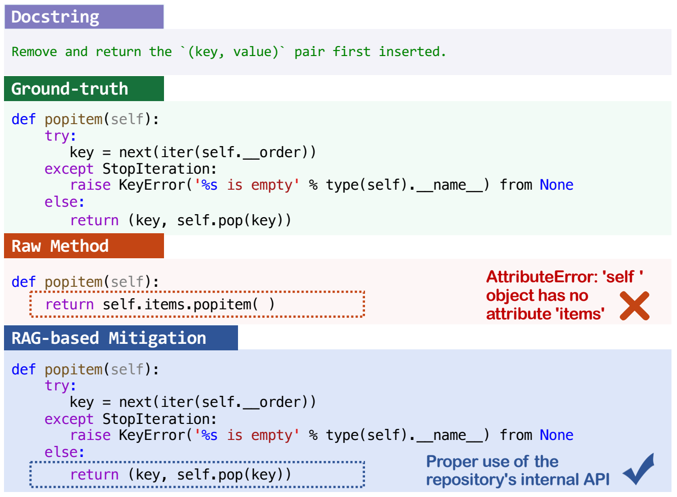
```
converts a script to one line command.THis is useful to run a single ssh command andp
ass a one line script.
:param script:
:return:
```

```
def oneline(script, separator='&&'):
    return seperator.join(textwrap.dedent(script).strip().splitlines())
```

```
def oneline(script, separator='&&'):
    return script.replace('\t', seperator)
```

```
def oneline(script, separator='&&'):
    return seperator.join(script.splitlines())
```

Fig. 12. Example: hallucination mitigation in Task Requirement Conflicts


Docstring
Ground-truth
Raw Method
AttributeError: 'self ' object has no attribute 'items'
RAG-based Mitigation
Proper use of the repository's internal API

```
Remove and return the `(key, value)` pair first inserted.
```

```
def popitem(self):
    try:
        key = next(iter(self.__order))
    except StopIteration:
        raise KeyError('%s is empty' % type(self).__name__) from None
    else:
        return (key, self.pop(key))
```

```
def popitem(self):
    return self.items.popitem( )
```

```
def popitem(self):
    try:
        key = next(iter(self.__order))
    except StopIteration:
        raise KeyError('%s is empty' % type(self).__name__) from None
    else:
        return (key, self.pop(key))
```

Fig. 13. Example: hallucination mitigation in Project Context Conflicts

mitigation, ChatGPT correctly implements the requirement using the `self.pop()` function.

VI. Discussion

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

Developing hallucination identification techniques: Through our study, we find 3 major categories of hallucinations in the LLM-based code generation. Some hallucinations can be detected by using static analysis (e.g., undefined variables) 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 [71], [72] 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 V, 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 [4], [49] and multi-agent systems [73] 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.

VII. 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.

VIII. 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.

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "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*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[2] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "Github copilot ai pair programmer: Asset or liability?" *Journal of Systems and Software*, vol. 203, p. 111734, 2023.

[3] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[4] K. D. Dhole, R. Chandradevan, and E. Agichtein, "An interactive query generation assistant using llm-based prompt modification and user feedback," *arXiv preprint arXiv:2311.11226*, 2023.

[5] H. Ye, T. Liu, A. Zhang, W. Hua, and W. Jia, "Cognitive mirage: A review of hallucinations in large language models," *arXiv preprint arXiv:2309.06794*, 2023.

[6] Z. Zhang, C. Chen, B. Liu, C. Liao, Z. Gong, H. Yu, J. Li, and R. Wang, "Unifying the perspectives of nlp and software engineering: A survey on language models for code," *arXiv preprint arXiv:2311.07989*, 2023.

[7] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *arXiv preprint arXiv:2311.05232*, 2023.

[8] S. Tonmoy, S. Zaman, V. Jain, A. Rani, V. Rawte, A. Chadha, and A. Das, "A comprehensive survey of hallucination mitigation techniques in large language models," *arXiv preprint arXiv:2401.01313*, 2024.

[9] 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.

[10] 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.

[11] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," *CoRR*, vol. abs/2302.04761, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.04761

[12] B. Paranjape, S. M. Lundberg, S. Singh, H. Hajishirzi, L. Zettlemoyer, and M. T. Ribeiro, "ART: automatic multi-step reasoning and tool-use for large language models," *CoRR*, vol. abs/2303.09014, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.09014

[13] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton-Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," *CoRR*, vol. abs/2307.09288, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.09288

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

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

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

[17] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[18] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. M. V, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Moustafa-Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" *CoRR*, vol. abs/2305.06161, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.06161

[19] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow," Mar. 2021, If you use this software, please cite it using these metadata. [Online]. Available: https://doi.org/10.5281/zenodo.5297715

[20] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model," https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[21] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "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*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 8696–8708. [Online]. Available: https://doi.org/10.18653/v1/2021.emnlp-main.685

[22] Y. Wang, H. Le, A. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "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*, H. Bouamor, J. Pino, and K. Bali, Eds. Association for Computational Linguistics, 2023, pp. 1069–1088. [Online]. Available: https://aclanthology.org/2023.emnlp-main.68

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

[24] Y. Li, D. H. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P. Huang, J. Welbl,

S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *CoRR*, vol. abs/2203.07814, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2203.07814

[25] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang, M. Pieler, U. S. Prashanth, S. Purohit, L. Reynolds, J. Tow, B. Wang, and S. Weinbach, "Gpt-neox-20b: An open-source autoregressive language model," *CoRR*, vol. abs/2204.06745, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2204.06745

[26] L. Guo, Y. Wang, E. Shi, W. Zhong, H. Zhang, J. Chen, R. Zhang, Y. Ma, and Z. Zheng, "When to stop? towards efficient code generation in llms with excess token prevention," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1073–1085.

[27] Y. Wang, T. Jiang, M. Liu, J. Chen, and Z. Zheng, "Beyond functional correctness: Investigating coding style inconsistencies in large language models," *arXiv preprint arXiv:2407.00456*, 2024.

[28] Z. Zheng, K. Ning, J. Chen, Y. Wang, W. Chen, L. Guo, and W. Wang, "Towards an understanding of large language models in software engineering tasks," *arXiv preprint arXiv:2308.11396*, 2023.

[29] C. Wang, J. Zhang, Y. Lou, M. Liu, W. Sun, Y. Liu, and X. Peng, "Tiger: A generating-then-ranking framework for practical python type inference," *arXiv preprint arXiv:2407.02095*, 2024.

[30] Y. Wang, Y. Wang, D. Guo, J. Chen, R. Zhang, Y. Ma, and Z. Zheng, "Rlcoder: Reinforcement learning for repository-level code completion," *arXiv preprint arXiv:2407.19487*, 2024.

[31] Y. Li, E. Shi, D. Zheng, K. Duan, J. Chen, and Y. Wang, "Repomincoder: Improving repository-level code generation based on information loss screening," in *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, 2024, pp. 229–238.

[32] C. Wang, Y. Lou, X. Peng, J. Liu, and B. Zou, "Mining resource-operation knowledge to support resource leak detection," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 986–998.

[33] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, "A survey of large language models for code: Evolution, benchmarking, and future trends," *arXiv preprint arXiv:2311.10372*, 2023.

[34] Y. Wang and H. Li, "Code completion by modeling flattened abstract syntax trees as graphs," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 16, 2021, pp. 14 015–14 023.

[35] L. Nie, J. Sun, Y. Wang, L. Du, S. Han, D. Zhang, L. Hou, J. Li, and J. Zhai, "Unveiling the black box of plms with semantic anchors: towards interpretable neural semantic parsing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 11, 2023, pp. 13 400–13 408.

[36] Y. Wang, L. Guo, E. Shi, W. Chen, J. Chen, W. Zhong, M. Wang, H. Li, H. Zhang, Z. Lyu *et al.*, "You augment me: Exploring chatgpt-based data augmentation for semantic code search," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023, pp. 14–25.

[37] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, L. K. Umapathi, C. J. Anderson, Y. Zi, J. Lamy-Poirier, H. Schoelkopf, S. Troshin, D. Abulkhanov, M. Romero, M. Lappert, F. D. Toni, B. G. del Río, Q. Liu, S. Bose, U. Bhattacharyya, T. Y. Zhuo, I. Yu, P. Villegas, M. Zocca, S. Mangrulkar, D. Lansky, H. Nguyen, D. Contractor, L. Villa, J. Li, D. Bahdanau, Y. Jernite, S. Hughes, D. Fried, A. Guha, H. de Vries, and L. von Werra, "Santacoder: don't reach for the stars!" *CoRR*, vol. abs/2301.03988, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2301.03988

[38] F. Christopoulou, G. Lampouras, M. Gritta, G. Zhang, Y. Guo, Z. Li, Q. Zhang, M. Xiao, B. Shen, L. Li, H. Yu, L. Yan, P. Zhou, X. Wang, Y. Ma, I. Iacobacci, Y. Wang, G. Liang, J. Wei, X. Jiang, Q. Wang, and Q. Liu, "Pangu-coder: Program synthesis with function-level language modeling," *CoRR*, vol. abs/2207.11280, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2207.11280

[39] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, T. Cohn, Y. He, and Y. Liu, Eds., vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547. [Online]. Available: https://doi.org/10.18653/v1/2020.findings-emnlp.139

[40] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[41] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

[42] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 7212–7225. [Online]. Available: https://doi.org/10.18653/v1/2022.acl-long.499

[43] K. Zhang, G. Li, J. Li, Z. Li, and Z. Jin, "Toolcoder: Teach code generation models to use apis with search tools," *arXiv preprint arXiv:2305.04032*, 2023.

[44] W. Zeng, X. Ren, T. Su, H. Wang, Y. Liao, Z. Wang, X. Jiang, Z. Yang, K. Wang, X. Zhang *et al.*, "Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation," *arXiv preprint arXiv:2104.12369*, 2021.

[45] K. Washio and Y. Miyao, "Code generation for unknown libraries via reading api documentations," *arXiv preprint arXiv:2202.07806*, 2022.

[46] J. H. Klemmer, S. A. Horstmann, N. Patnaik, C. Ludden, C. Burton Jr, C. Powers, F. Massacci, A. Rahman, D. Votipka, H. R. Lipford *et al.*, "Using ai assistants in software development: A qualitative study on security practices and concerns," *arXiv preprint arXiv:2405.06371*, 2024.

[47] D. Zan, B. Chen, Z. Lin, B. Guan, Y. Wang, and J.-G. Lou, "When language model meets private library," *arXiv preprint arXiv:2210.17236*, 2022.

[48] C. Spiess, D. Gros, K. S. Pai, M. Pradel, M. R. I. Rabin, S. Jha, P. Devanbu, and T. Ahmed, "Quality and trust in llm-generated code," *arXiv preprint arXiv:2402.02047*, 2024.

[49] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: Empowering llm-based code generation with intention clarification," *arXiv preprint arXiv:2310.10996*, 2023.

[50] L. Zhong and Z. Wang, "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, no. 19, 2024, pp. 21 841–21 849.

[51] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[52] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[53] 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.

[54] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[55] OpenAI, "Chatgpt: Optimizing language models for dialogue," https://openai.com/blog/chatgpt, 2022.

[56] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 269–280.

[57] W. Zou, J. Xuan, X. Xie, Z. Chen, and B. Xu, "How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects," *Empirical Software Engineering*, vol. 24, pp. 3871–3903, 2019.

[58] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An internet-scale software repository," in *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. IEEE, 2009, pp. 1–4.

[59] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[60] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.

[61] J. Li, G. Li, X. Zhang, Y. Dong, and Z. Jin, "Evocodebench: An evolving code generation benchmark aligned with real-world code repositories," *arXiv preprint arXiv:2404.00599*, 2024.

[62] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," *arXiv preprint arXiv:2303.12570*, 2023.

[63] M. Liu, T. Yang, Y. Lou, X. Du, Y. Wang, and X. Peng, "Codegen4libs: A two-stage approach for library-oriented code generation," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 434–445.

[64] D. Liao, S. Pan, Q. Huang, X. Ren, Z. Xing, H. Jin, and Q. Li, "Context-aware code generation framework for code repositories: Local, global, and third-party library awareness," *arXiv preprint arXiv:2312.05772*, 2023.

[65] S. H. Khandkar, "Open coding," *University of Calgary*, vol. 23, no. 2009, 2009.

[66] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, "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*, 2010, pp. 1–10.

[67] C. Wang, X. Peng, Z. Xing, and X. Meng, "Beyond literal meaning: Uncover and explain implicit knowledge in code through wikipedia-based concept linking," *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3226–3240, 2023.

[68] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.

[69] Y. Zhang, Y. Li, L. Cui, D. Cai, L. Liu, T. Fu, X. Huang, E. Zhao, Y. Zhang, Y. Chen *et al.*, "Siren's song in the ai ocean: a survey on hallucination in large language models," *arXiv preprint arXiv:2309.01219*, 2023.

[70] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[71] A. Agrawal, M. Suzgun, L. Mackey, and A. T. Kalai, "Do language models know when they're hallucinating references?" *arXiv preprint arXiv:2305.18248*, 2023.

[72] P. Manakul, A. Liusie, and M. J. Gales, "Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models," *arXiv preprint arXiv:2303.08896*, 2023.

[73] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," *arXiv preprint arXiv:2402.01680*, 2024.

[74] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. H. Chi, N. Schärli, and D. Zhou, "Large language models can be easily distracted by irrelevant context," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 210–31 227.

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

[76] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, "Ds-1000: A natural and reliable benchmark for data science code generation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 18 319–18 345.

[77] Z. Sun, L. Li, Y. Liu, X. Du, and L. Li, "On the importance of building high-quality training datasets for neural code search," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1609–1620.

[78] C. Wang, J. Zhang, Y. Feng, T. Li, W. Sun, Y. Liu, and X. Peng, "Teaching code llms to use autocompletion tools in repository-level code generation," *arXiv preprint arXiv:2401.06391*, 2024.

[79] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.