

De-Hallucinator: Mitigating LLM Hallucinations in Code Generation Tasks via Iterative Grounding

Aryaz Eghbali

Software Lab

University of Stuttgart

Stuttgart, Germany

aryaz.eghbali@iste.uni-stuttgart.de

Michael Pradel

Software Lab

University of Stuttgart

Stuttgart, Germany

michael@binaervarianz.de

ABSTRACT

Large language models (LLMs) trained on datasets of publicly available source code have established a new state of the art in code generation tasks. However, these models are mostly unaware of the code that exists within a specific project, preventing the models from making good use of existing APIs. Instead, LLMs often invent, or “hallucinate”, non-existent APIs or produce variants of already existing code. This paper presents De-Hallucinator, a technique that grounds the predictions of an LLM through a novel combination of retrieving suitable API references and iteratively querying the model with increasingly suitable context information in the prompt. The approach exploits the observation that predictions by LLMs often resemble the desired code, but they fail to correctly refer to already existing APIs. De-Hallucinator automatically identifies project-specific API references related to the model’s initial predictions and adds these references into the prompt. Unlike retrieval-augmented generation (RAG), our approach uses the initial prediction(s) by the model to iteratively retrieve increasingly suitable API references. Our evaluation applies the approach to two tasks: predicting API usages in Python and generating tests in JavaScript. We show that De-Hallucinator consistently improves the generated code across five LLMs. In particular, the approach improves the edit distance by 23.3–50.6% and the recall of correctly predicted API usages by 23.9–61.0% for code completion, and improves the number of fixed tests that initially failed because of hallucinations by 63.2%, resulting in a 15.5% increase in statement coverage for test generation.

1 INTRODUCTION

Large language models (LLMs) have proven effective in many natural language [7] and programming tasks [4, 9, 23, 46, 54, 61, 63]. Rapid adoption of LLM-based tools, such as Copilot¹ and Tabnine², shows practical productivity benefits [33, 66]. State-of-the-art LLMs build on transformers [58], which use self-attention to generate sequences of tokens in an auto-regressive process. That is, the model decides what token to predict next based on the tokens in the prompt and any already generated tokens. Hence, designing effective prompts, sometimes called prompt engineering, is a crucial part of developing a practical LLM-based technique [35, 39, 56].

Despite the impressive success of LLM-based code generation, these techniques are still at an early stage. In particular, we identify two key challenges faced by current approaches:

Challenge 1: Project-specific APIs. As LLMs are trained on huge code bases, they effectively capture typical language idioms and

¹<https://github.com/features/copilot>

²<https://www.tabnine.com/>

```
class DataStore():
    def __init__(self, file: str):
        with open(file, 'r') as f:
            self.documents = f.read().split('-----')
    ...
    def find_by_keyword(self, keyword: str) -> List[str]:
        return [d for d in self.documents if keyword in d]
    ...

```

```
def relevance(document: str, keyword: str) -> float:
    return document.count(keyword) / len(document)
...
```

```
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
    docs = ds.find_by_keyword(keyword)
    return sorted(docs, key=lambda d: relevance(d, keyword),
                  reverse=True)[:top_k]
...
```

Figure 1: The desired completion of `search` is highlighted in gray.

```
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
    docs = ds.find_by_keyword(keyword)
    return sorted(docs, key=lambda x: x.score, reverse=True)[:top_k]
```

Figure 2: The completion of `search` by CodeGen-2B-mono highlighted in gray, and the wrong API usage highlighted in red.

commonly used libraries. In contrast, a general-purpose model lacks knowledge of project-specific APIs, and may fail to correctly use existing functions and classes. In particular, this lack of knowledge may cause the model to “hallucinate” APIs that actually do not exist in the current code base [40], or perhaps even worse, it may reimplement some functionality that is already present in the code base. Developers perceive this lack of knowledge about project-specific APIs as an obstacle to using AI programming assistants [33].

As a running example, consider three files in a large project dealing with text documents, shown in Fig. 1. One file, `DataStore.py`, contains a class implementing a data structure that stores documents and provides a keyword-based search over the documents. Another file, `utils.py`, provides helper functions, one of which allows for measuring the relevance of a document to a keyword. In a third file, `UI.py`, the developer is working on a function, `search`, to search for the `top_k` documents that are most relevant to a keyword.

Requesting an LLM, e.g., CodeGen [43], to complete the `search` function given a prompt that contains all existing code in `UI.py` results in Fig. 2. The code is partially correct, but refers to a non-existing API (an attribute `x.score`). The underlying problem is that the models are not aware of the project-specific APIs that should be used to complete the code, and hence, the LLM simply hallucinates some plausible but ultimately wrong APIs.

Challenge 2: Prioritizing context. A naive solution to address Challenge 1 would be to simply add all of the code in the project into the prompt. However, LLMs have a fixed maximum sequence length, which restricts how many tokens one can provide to the model. Even with the recent increases in sequence length of LLMs, providing the most useful context can improve the output and reduce the costs. Choosing the most helpful context for a given completion task is crucial, but an inherently difficult problem, because the optimal context depends on the desired code, which is not known a-priori. While traditional code completion approaches typically have access to various kinds of information available in IDEs, such as the names and types of program elements, providing all this information, or even all the code of the project, to an LLM is impossible due to the limited prompt size.

This paper presents De-Hallucinator, which addresses the above challenges through a novel combination of retrieval-augmented generation (RAG) and an iterative form of LLM-based code generation. Our approach uses three types of prompts, which provide increasingly suitable context information. The *initial prompt* type is querying the LLM with the conventional prompt, i.e., without any retrieval. Retrieval-augmented generation (RAG) [28] proposes to retrieve relevant context based on the initial prompt to address both Challenges 1 and 2, which we refer to as the *RAG prompt* type. The idea of augmenting an LLM with well-grounded facts relates to work on grounding of language models for natural languages [2, 17, 52]. However, also this prompt may fail to generate factually correct code (i.e., without hallucinations), because the initial prompt might not have any similar code to the desired API, or there are other APIs more similar to the initial prompt than the correct one. We make the important observation that the generated code from the previous prompt types often resembles the desired API. Hence, we construct a new type of prompt called the *iterative prompt*. De-Hallucinator leverages the code that the model predicts to retrieve suitable project-specific APIs, which are then added to the iterative prompt for the next round of model prediction. The iterative prompt type complements prior work that tries to guess the most suitable context from the incomplete code alone [12, 56].

The presented approach offers several benefits. First, De-Hallucinator works with any off-the-shelf LLM trained on code because the approach treats the model as a black box. In particular, we do not require to train or fine-tune the model in any way, but simply exploit the fact that its predictions contain implicit hints about additional context the model would benefit from. Second, because APIs usually evolve only slowly, De-Hallucinator can pre-compute, and occasionally update in the background, the set of project-specific API references. As a result, the latency of code generation is not impacted by any expensive program analysis, which is important for practical adoption. Finally, the approach is fully transparent to developers, because the approach hides the iterative interaction with the LLM from the user and simply returns a ranked list of predictions.

We evaluate De-Hallucinator by applying the approach to code completion with four state-of-the-art LLMs for code, namely CodeGen [43], CodeGen 2.5 [42], UniXcoder [18], and StarCoder+ [30], and to test generation with GPT-3.5-turbo. Conceptually, the approach can be applied to any programming language, and our evaluation focuses on two popular languages, Python and JavaScript

as they are among the most popular languages³ and common targets of prior work on code completion [9, 18, 29, 30, 42, 43, 57, 65] and test generation [3, 54]. Compared to conventional prompts, we find that De-Hallucinator enables the models to provide more accurate predictions. In particular, we show a relative improvement of 23.3–50.6% in edit distance, and of 23.9–61.0% in recall of correctly predicted API usages for code completion. Moreover, we show relative improvement of 17.9% in number of passing tests, of 15.5% in coverage, and of 63.2% in the number of mitigated hallucinations.

In summary, this paper contributes the following:

- Empirical motivation showing that API hallucinations affect a large portion of failed code completion and test generation tasks.
- A technique for addressing this problem using off-the-shelf, unmodified LLMs.
- A novel algorithm that combines retrieval-augmented generation with an iterative method for constructing increasingly suitable prompts by using the hallucinations produced in earlier iterations to augment the context information provided in the prompts of future iterations.
- Empirical evidence that, across two code generation tasks, two programming languages, and five state-of-the-art LLMs, De-Hallucinator offers more accurate generations than conventional prompts.

2 PRELIMINARY STUDY

Before delving into our approach, we validate the motivation for this work by performing a preliminary study, which assesses the importance of the two challenges described in the introduction.

2.1 Project-Specific APIs

The main motivation for this work is our observation that LLMs often hallucinate code that resembles the desired code, but that fail to correctly refer to an API. To assess the importance of this limitation, we investigate the prevalence and causes of hallucinated APIs in the two code generation tasks focused in this paper. For code completion, we manually investigate and classify the reasons why an LLM fails to predict the desired completion. We perform this preliminary study on 50 function-level code completion tasks, which we collect by (i) randomly selecting ten Python projects from a curated list of open-source projects⁴ and (ii) by then randomly selecting five functions from each project. The only filtering we perform is to ignore functions with more than 25 lines, as these are likely out of reach for today’s LLMs. For each of the 50 functions, we query an LLM (CodeGen 2.5 with 7B parameters and 4-bit quantization) with the code before the beginning of the function body, including the function signature and any docstring, in the prompt.

Given the 50 pairs of an LLM-predicted function body and the ground-truth function body, we manually classify them based on two questions. First, is the prediction correct w.r.t. the ground truth, where “correct” includes exact matches and semantically equivalent code? Second, does the ground truth contain an API usage, e.g., a function call, that is missing in the prediction? Initially, two

³<https://octoverse.github.com/2022/top-programming-languages>

⁴<https://github.com/vinta/awesome-python>. We randomly sample ten application domains and then sample one project from each domain.

of the authors independently classify the 50 pairs, with an inter-rater agreement (Cohen’s kappa) of 0.76, which is considered excellent [14], and after discussing the discrepancies reach a consensus about all 50 pairs.

The final inspection results show that in 13 out of the 50 cases, the LLM either predicts exactly the expected function body or a function body that is semantically equivalent to the expected one. For 22 out of the 37 remaining cases, there is at least one API usage that the LLM fails to correctly predict, similar to the example in Fig. 2. In other words, the problem identified and addressed in this work affects 44% of all studied function-level code completion tasks, and even 59% of all tasks where the LLM alone fails to predict the expected code.

For test generation, we use error messages of crashing tests generated by TestPilot [54] on a diverse set of 12 JavaScript projects. We automatically count the number of generated tests that result in “* is not a function”, or “Cannot read properties of undefined” errors, which typically indicates hallucinations of non-existing APIs. We find that, on average, 16.4% of all generated tests fail because of the aforementioned errors, which indicates that hallucinations of non-existing APIs are a common problem in test generation as well.

2.2 Prioritizing Context

To validate the importance of the second challenge, we compare the amount of code in a single project to the prompt sizes of high-end LLMs. The models in the popular GPT series by OpenAI have prompt sizes between 4k (GPT-3.5 models) and 128k (GPT-4) tokens. In contrast, in a sample dataset of 50 Python projects, which are randomly selected from the same curated list of projects as above, there are 488,635 tokens per project, on average. Furthermore, the average project has around 13 files longer than 8,192 tokens, and 22 projects in our sample have at least one file longer than 32,768 tokens. This means that even knowing the exact file that contains the relevant context (e.g., based on heuristics, such as recently used files or similar file names) leaves us with more tokens than one could fit into the prompt of some models. Even for models with longer context window, considering a cost of 0.5\$ for 100k tokens means that the cost of long prompts would be impractical for regular use. In other words, simply adding all potentially relevant code to the prompt is not a viable solution, but we need to prioritize the context information.

3 APPROACH

This section describes our approach for iteratively retrieving relevant APIs to improve the prompts for code generation tasks. We call the approach De-Hallucinator, as it reduces the hallucinations of the LLM by providing relevant API references to ground the model. First, we provide an overview of the approach (Section 3.1), and then present each of the components of De-Hallucinator in detail (Sections 3.2 to 3.5).

3.1 Overview

3.1.1 Main Algorithm. Figure 3 gives an overview of the approach, which we use to illustrate the main algorithm. The top of the figure shows the traditional code generation process, where an LLM receives a prompt and generates code. We call this prompt the *initial*

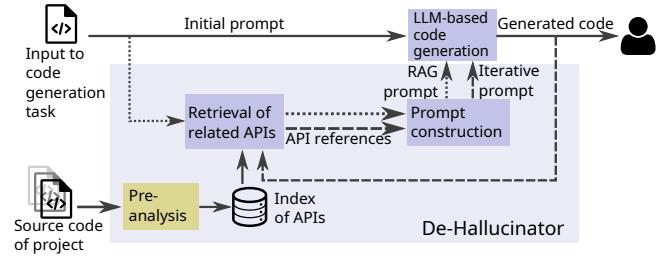


Figure 3: Overview of De-Hallucinator.

prompt. Because the model may not be aware of project-specific APIs, the output is likely to refer to some hallucinated APIs.

To help the model predict better code, De-Hallucinator refines the initial prompt using two techniques. Both of them add API references to the prompt, but they do so in different ways. First, as shown by the dotted lines, De-Hallucinator uses the initial prompt to retrieve related API references from the project. This approach is similar to retrieval-augmented generation (RAG) [28], and we refer to the resulting prompt as the *RAG prompt*.

Second, as shown by the dashed lines, De-Hallucinator uses the output of the model to retrieve related API references. This technique is unique to our approach, and it is based on the observation that the model’s output often resembles the desired code, but fails to refer to the correct APIs. Retrieving suitable API references based on the model’s output can be done multiple times, and hence, we refer to the resulting prompt as the *iterative prompt*. In general, De-Hallucinator repeats the iterative prompt refinement, i.e., the dashed loop in the figure, until exhausting a configurable maximum number k of queries to the model.

For efficient retrieval of APIs, De-Hallucinator analyzes the project in advance, as shown in the “Pre-analysis” component, and indexes all APIs of the project.

3.1.2 Example. Fig. 4 shows each step of the approach on our running example from Fig. 1. Given the initial prompt, the initial completion by the model refers to a non-existing API `x.score`. Then, for the RAG prompt the initial prompt is used for retrieval. In this step, as shown in Fig. 4, the reference to an already used function, `find_by_keyword` is retrieved. Consequently, the completion uses the wrong API, as the model still does not have knowledge of the relevance function. Next, using the initial completion by the model, De-Hallucinator retrieves a reference to the `relevance` function defined in `utils.py`. In the example, the iterative prompt results in the correct completion, as shown at the bottom of Fig. 4.

The following presents each component of De-Hallucinator in more detail, as well as how we instantiate the components for our two target tasks, code completion and test generation.

3.2 Pre-Analysis

3.2.1 General Idea. To ensure that the retrieval of API references does not unnecessarily slow down the code generation, De-Hallucinator has a preprocessing phase that indexes the current project for fast retrieval. We use *API reference* throughout this paper to refer to a piece of text extracted from the project’s code, which can be added to the prompt to provide further information about a project-specific API.

```

... Initial prompt ...
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
    docs = ds.find_by_keyword(keyword)

Initial completion ...
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
    docs = ds.find_by_keyword(keyword)
    return sorted(docs, key=lambda x:x.score, reverse=True)[:top\k]

Most relevant API reference ...
DataStore.find_by_keyword(self, keyword: str) -> List[str]

# API Reference: ...
# DataStore.find_by_keyword(self, keyword: str) -> List[str]
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
    docs = ds.find_by_keyword(keyword)

RAG prompt ...
# API Reference: ...
# DataStore.find_by_keyword(self, keyword: str) -> List[str]
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
    docs = ds.find_by_keyword(keyword)

RAG prompt completion ...
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
    docs = ds.find_by_keyword(keyword)
    return sorted(docs, key=lambda x:x.score, reverse=True)[:top\k]

Most relevant API reference ...
relevance(document: str, keyword: str) -> float

# API Reference: ...
# relevance(document: str, keyword: str) -> float
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
    docs = ds.find_by_keyword(keyword)

Iterative prompt ...
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
    docs = ds.find_by_keyword(keyword)
    return sorted(docs, key=lambda doc: relevance(keyword, doc),
                 reverse=True)[:top\k] # <- Correct

```

Figure 4: Step-by-step progression of De-Hallucinator on the example in Fig. 1.
Table 1: Examples of API references extracted from the project in Fig. 1.

Source	API reference	Type of API reference
DataStore.py	DataStore.find_by_keyword(self, keyword: str) -> List[str]	Function reference
utils.py	relevance(document: str, keyword: str) -> float	Function reference
DataStore.py	class DataStore()	Class reference
DataStore.py	DataStore.documents	Attribute reference

Definition 3.1 (API reference). An API reference is one of the following:

- A *function reference*, which consists of
 - the qualified name of the function,
 - the parameter names,
 - any available default values for arguments,
 - any available type annotations, and
 - any available function-level docstring.
- A *class reference*, which consists of
 - the qualified name of the class,
 - the parent class(es), and
 - any available class-level docstring.
- An *attribute reference*, which is the qualified name of a `self` attribute assigned to in the constructor.

3.2.2 *Example.* Table 1 shows some of the API references extracted from our example project. Note that the API references resemble real code to maintain compatibility with any model that is trained on source code.

3.2.3 *Application to Code Completion.* For the code completion task, we use CodeQL⁵ to statically extract APIs from the project source. The Python language support of CodeQL offers easy access to the classes, functions, etc. in a code base. Another benefit of CodeQL is that we can utilize databases created by GitHub for open-source projects.

3.2.4 *Application to Test Generation.* Since the APIs in JavaScript are in many cases only available once a module is instantiated, De-Hallucinator dynamically loads the modules and traverses them to extract API references.

Alternatively to our approaches for gathering API references, an IDE-based implementation could reuse information about the current project that is anyway computed by the static code indexing in an IDE.

3.3 Retrieval of Related APIs

3.3.1 *General Idea.* The retrieval module takes an input code piece and returns a ranked list of project-specific API references that are most similar to the input. To enable similarity-based search, De-Hallucinator embeds the extracted API references into a vector space. Formally, we need an embedding function, E , for which $E(c) = v_c \in \mathbb{R}^d$, such that, for two code pieces c_1 and c_2 , the cosine similarity of their embeddings, $v_{c_1} \cdot v_{c_2} / (|v_{c_1}| |v_{c_2}|)$, approximates the semantic similarity of c_1 and c_2 . Recently, many models have been trained for this task [1]. Because our approach uses the embedding function as a black-box, any embedding model or similarity-preserving vector representation [59] can be used with De-Hallucinator, e.g., GloVe [45], BERT [11], or FastText [5].

Given the embeddings of the API references, De-Hallucinator retrieves API references that are most similar to the input code piece. To this end, the approach embeds the input code piece and searches for the n API references that minimize the cosine distance to the input code piece. The parameter n specifies the number of API references to include in the prompt.

3.3.2 *Example.* Getting back to our running example, consider the third section in Fig. 4. It shows the API reference from our example project that the retrieval component finds to be the top-most relevant to the incomplete code (initial prompt). Even though our implementation retrieves n relevant API references, we show only one in the example for brevity.

3.3.3 *Application to Code Completion.* For code completion, we split the given input code piece into lines, then retrieve the most relevant APIs for each line, and finally merge the n most similar API references into single sorted list. The reason for retrieving the API references similar to full lines of code, as opposed to only API usages, is that we want the completion to avoid re-implementing existing code. Therefore, if there exists some API similar to a line of code that does not use any APIs, we want the approach to be able to retrieve suitable API references to generate the correct completion. We embed the API references into a vector space using Sentence-BERT [51], a BERT-based model designed for measuring the semantic similarity between sentences. We use a variant of this

⁵<https://codeql.github.com/>

model that is pre-trained on code.⁶ The model maps sentences, or in our case lines of code, into a dense vector representation of size 768.

After embedding the API references, the approach indexes the normalized vectors ($v_c / \|v_c\|$ for all $c \in$ API references) in a Ball Tree.⁷ This index allows for fast retrieval of nearest neighbors. During the retrieval, we embed each line in the input using the same pre-trained SentenceBERT model used for indexing the API references, and then normalize the vectors. The normalization is done to turn the Euclidean distance used by the Ball Tree into cosine similarity, which is commonly used. Next, we find the closest API reference of each line by querying the Ball Tree constructed in the pre-analysis step. The result is a list R_l of API references, sorted by their similarity to the line l in the input. To obtain a single ranked list of API references, we merge the lists R_l across all $l \in completion$ based on their similarity scores. This finally yields a single list R of API references, of which we use the top- n as additional context to add into the prompt.

3.3.4 Application to Test Generation. For test generation, we extract all API usages in a previously generated test using regular expression matching and embed them. Next, we retrieve the most relevant APIs for every API usage and return a single list. Here we retrieve API references based on API usages, and not based on lines. The reason is that as there are no ground truths for the test generation task, and the main goal is to generate passing tests. Therefore, fixing a wrong API usage is more important than avoiding re-implementation of existing code. The embedding used for indexing the API references and during retrieval is a BERT-based model available on HuggingFace.⁸ The reason for using a different model than the one used for code completion is that not all models on HuggingFace are compatible with TypeScript, so we choose a compatible model. Using mean pooling, we embed the code pieces into a vector of size 768. We use the same model for both indexing and retrieval, where during indexing each API signature is embedded and stored in a list, and during retrieval the API usage is embedded. During retrieval, we perform a linear search for the most relevant API references. Since the size of JavaScript projects are much smaller than the Python projects, there is not much benefit in using the Ball Tree data structure.

3.4 Prompt Construction

3.4.1 General Idea. Given the input and a list of at most n API references that may enable the LLM to accurately generate code, De-Hallucinator constructs an augmented prompt for querying the model. The prompt is designed in a way that resembles “normal” code, i.e., the kind of data that the LLM has been trained on. The API references are augmented as a block of commented lines to the prompt. These lines start with API Reference:, and the following lines contain the relevant API references in decreasing order of similarity to the lines in the input of the retrieval module.

3.4.2 Example. For our running example, the “Iterative prompt” section in Fig. 4 shows the prompt for function search in our example, augmented with the API reference. Given the iterative prompt, the same LLM that predicted the code in Fig. 2, completes this function correctly in the last section of Fig. 4.

3.4.3 Application to Code Completion. In the code completion task, we prepend the prompt with the API references, as it minimally disrupts the structure of the existing code. See our running example for illustration.

3.4.4 Application to Test Generation. On the other hand, for test generation, as TestPilot already includes additional information into the prompt, we append the API references to the end of the additional context section. Figure 6 shows how the prompt is augmented with the API references in JavaScript, and the model’s success in generating a correct test based on that prompt.

3.5 Integration with the LLM

De-Hallucinator is designed with minimal assumptions about the underlying LLM, and the tool that solves the code generation task. The approach considers the LLM to be a black box that we query with a string, which then returns one or multiple strings with suggested code pieces. Hence, we do not fine-tune the LLM, or train a model to preform retrievals, which makes De-Hallucinator applicable to more scenarios. To that end, the instantiation of De-Hallucinator in both code completion and test generation tasks are compatible with any LLM, and our experiments with five LLMs show the flexibility of the approach.

4 IMPLEMENTATION

The general ideas behind De-Hallucinator are language-agnostic, and the approach can be applied to different programming languages and to different code prediction tasks. We present two implementations, one for code completion in Python and one for test generation in JavaScript. The code completion implementation is in Python and builds on top of the HuggingFace transformers library. Adapting our implementation to other models requires only to adjust the prompt size of the model and to select other parameters passed to its API. The test generation implementation is in TypeScript and builds on top of the state-of-the-art LLM-based test generator TestPilot [54]. Like TestPilot, we use GPT-3.5-turbo as the LLM. TestPilot generates tests by going through the functions in the package under test, and for each function tries to generate a test with a simple input, which consists of a test header and the signature of the function under test. Then, a set of “prompt refiners” modify the prompt to include usage snippets, error messages, and the body of the function under test. We implement two new prompt refiners, one for the RAG prompts, and one for iterative prompts. These two refiners are only activated when a test fails with an error that is likely caused by a hallucinated API, which we detect by checking if the error message contains “is not a function” or “of undefined”.

5 EVALUATION

To evaluate the effectiveness and efficiency of our approach, we perform experiments that answer the following research questions:

⁶<https://huggingface.co/flax-sentence-embeddings/st-codesearch-distilroberta-base>

⁷<https://scikit-learn.org/stable/modules/neighbors.html#ball-tree>

⁸<https://huggingface.co/jinaai/jina-embeddings-v2-base-code>

- RQ1:** How much does De-Hallucinator improve the generated code compared to the baseline approaches?
- RQ2:** How effective is De-Hallucinator at adding the correct API references to the prompt?
- RQ3:** How do the hyperparameters of De-Hallucinator affect the results?
- RQ4:** How efficient is De-Hallucinator, and how much do the different steps of the approach contribute to the running time?

5.1 Experimental Setup

5.1.1 Tasks. Our evaluation targets two tasks, code completion and test generation. For code completion, we define the task as completing an incomplete function at the beginning of a line with an API usage, given the preceding code and the existing code in the project. This problem definition matches the common scenario of a developer implementing a function in an existing project, where the code to be written should use a project-specific API. For example, suppose that the cursor in Fig. 1 is at the beginning of the code marked with gray background. Everything above the cursor is our incomplete code c , and the problem is to predict the marked code c' , which refers, e.g., to the project-specific relevance API. For test generation, the task is to generate tests for a given JavaScript package. This problem setting has been well established by previous work [3, 54].

5.1.2 LLMs and Baseline.

Code Completion. We evaluate the code completion task on four state-of-the-art LLMs: CodeGen [43] with 2.7B parameters (Salesforce/codegen-2B-mono), CodeGen 2.5 [42] with 7B parameters (Salesforce/codegen25-7b-mono), UniXCoder [18] with 125M parameters (microsoft/unixcoder-base), and StarCoder+ [31] with 15.5B parameters (bigcode/starcoderplus). The reason for selecting these models is that they cover a variety of parameter sizes, model architectures, and pre-training processes. We leave all parameters of the models at their defaults, except for the maximum new tokens parameter, which we set to 256 to allow for longer completions. As a baseline, we query the models with a prompt that contains all the code preceding the cursor. In case this prompt exceeds the maximum prompt size of 2,048 tokens, we truncate the prompt from the beginning.

Test Generation. For the test generation task, we build upon TestPilot and use GPT-3.5-turbo-0125 for the LLM as the OpenAI GPT models are already integrated into TestPilot and require minimal effort to run. As a baseline, we use the original TestPilot implementation with a one-hour time limit per package, 130k token limit per package, and four completions per prompt with temperature 0.1. To make the comparison fair, we set the token limit of De-Hallucinator to the amount of tokens used by the baseline, and the number of completions to four with temperature 0.1. This prevents De-Hallucinator being advantaged with generating more tokens. Moreover, we cache the model outputs so that the same prompts return the same completion for De-Hallucinator and the baseline.

5.1.3 Datasets.

Code Completion. With the goal of having a diverse set of projects in terms of size, domain, and popularity, we gather a dataset of

Table 2: List of projects used for evaluation.

Project (owner/name)	Commit	LoC	Stars*
Python projects used for code completion			
graphql-python/graphene	57cbef6	9,484	7.8k
geopy/geopy	ef48a8c	10,000	4.3k
nvbn/thefuck	ceeaeb	12,181	83.3k
aaugustin/websockets**	ba1ed7a	14,186	5k
arrow-py/arrow	74a759b	14,402	8.6k
lektor/lektor	be3c8cb	16,852	3.8k
Parsely/streamparse	aabd9d0	26,214	1.5k
Supervisor/supervisor	ca54549	29,860	8.3k
mwaskom/seaborn	f9827a3	37,367	12.1k
psf/black	ef6e079	106,005	37.6k
scikit-learn/scikit-learn	f3c6fd6	193,863	58.5k
JavaScript projects used for test generation			
node-red/node-red	29ed5b2	60	18.8k
winstonjs/winston	c63a5ad	496	22.2k
prettier/prettier	7142cf3	916	48.5k
tj/commander.js	83c3f4e	1,134	26.3k
js-sdsl/js-sdsl	055866a	1,198	0.7k
goldfire/howler.js	003b917	1,319	23.1k
websockets/ws	b73b118	1,546	21.2k
handlebars-lang/handlebars.js	8dc3d25	2,117	17.8k
petkaantonov/bluebird	df70847	3,105	20.4k
hapijs/joi	5b96852	4,149	20.7k
Unitech/pm2	a092db2	5,048	40.9k
11ty/eleventy	e71cb94	5,772	16.4k

* As of June 5, 2024

** Has been moved to python-websockets/websockets

eleven public Python projects from GitHub, shown in Table 2 for the code completion task. We construct a dataset of API-related code completion tasks by removing API usages from the benchmark projects and by considering the removed code as the ground truth to be predicted by a model. For each such API call, we remove the lines containing the call. If a call spans multiple lines, we remove all of them. To prevent data leakage from imports of the API in the ground truth, we also remove API-related imports. Next, we check if the off-the-shelf LLMs can predict the exact code as in the original file using the code preceding the cursor as the prompt. If an LLM predicts exactly the original code, we ignore this API usage for the evaluation, as there is no need to further improve the prediction and to avoid any potential memorizations. We continue with this process for each of the four models, until we have ten code completion tasks for each of the eleven projects. During this process, we ignore 18, 51, 76, and 31 completions for UniXcoder, CodeGen, CodeGen v2.5, and StarCoder+, respectively. Overall, the code completion evaluation dataset consists of 11 projects \times 10 \times 4 models = 440 code completion tasks.

Test Generation. For the test generation task, we also gather a diverse set of 12 JavaScript projects from GitHub, shown in Table 2. These projects cover a variety of domains, such as website generation, code formatting, and process management. Since TestPilot cannot generate tests for ES modules, we only consider JavaScript

projects that are CommonJS packages. The rest of the setup is the same as in the TestPilot paper [54].

5.1.4 Metrics.

Code Completion. We evaluate the code completions in three ways:

- *Edit distance.* To quantify the number of edits a developer needs to apply after receiving a code completion, we measure the edit distance between the predicted code and the ground truth. This metric provides a sense for how many token edits are saved when using De-Hallucinator. We compute edit distance using the Levenshtein distance at the subtoken level. For each pair of completion and ground truth, we tokenize the code pieces with a GPT-2 fast tokenizer,⁹ and then calculate the edit distance using NLTK’s `edit_distance`.¹⁰
- *Normalized edit similarity.* Similar to previous work [36] we also compute the normalized edit similarity. To this end, we normalize the absolute edit distance (computed as above) to the length of the longer of the two token sequences, and then turn the result into a similarity metric.
- *Exact API match.* Since the goal of De-Hallucinator is to predict better API usages, we measure how many of all desired API usages are predicted exactly as in the ground truth. To identify the API usages in the lines of code to complete, we extract function calls, including the access path to the function, and the parameters.

For all the above metrics, we report the best completion obtained among k completions of De-Hallucinator. Measuring the *best@ k* matches a common usage scenario where a developer inspects a ranked list of code completion suggestions, and picks the first that matches the developer’s expectations.

Test Generation. For the test generation task, we use the following three metrics:

- *Number of passing tests.* We count the number of passing tests as a proxy for code without any hallucinations.
- *Coverage.* We measure the statement coverage in the passing tests.
- *Number of fixed hallucinated tests.* We measure the number of tests that initially crash with “* is not a function” or “Reading property of undefined” errors and that subsequently De-Hallucinator turns into passing tests.

5.1.5 Hardware. We perform the experiments with the CodeGen v2.5 model and the GPT-3.5-turbo-0125 model on a machine equipped with two Nvidia T4 GPUs, each having 16GB of memory. The experiments with the UniXcoder, CodeGen, and the StarCoder+ models are performed on a machine with a single Nvidia Tesla V100 with 32GB of memory. Each machine has a 48-core Intel Xeon CPU clocked at 2.20GHz.

5.2 RQ1: Effectiveness of De-Hallucinator

Code Completion. In the first set of experiments, we investigate to what extent De-Hallucinator improves code completions compared to the baseline. By default, we run De-Hallucinator with $k = 3$

⁹https://huggingface.co/docs/transformers/model_doc/gpt2#transformers.GPT2TokenizerFast
¹⁰https://www.nltk.org/_modules/nltk/metrics/distance.html#edit_distance

Table 3: Effectiveness of De-Hallucinator in code completion compared to the baseline on four off-the-shelf LLMs. The bold numbers show statistically significant improvement over the initial prompt. The numbers in parentheses show the relative improvement over the baseline.

Prompt type	UniXCoder 125M	CodeGen v1 2B	CodeGen v2.5 7B	StarCoder+ 15B
Edit distance (lower is better):				
Initial	52.4	40.0	47.2	44.6
RAG	46.8 (10.7%)	33.4 (16.5%)	31.6 (33.0%)	35.9 (19.4%)
Iterative	25.9 (50.6%)	30.7 (23.3%)	30.1 (36.3%)	33.5 (24.9%)
Edit similarity (lower is better):				
Initial	33.4	43.6	43.9	33.2
RAG	37.3 (11.7%)	48.0 (10.0%)	49.4 (12.5%)	38.0 (14.3%)
Iterative	42.6 (27.5%)	48.9 (12.1%)	50.2 (14.2%)	39.7 (19.3%)
Exact API match (higher is better):				
Initial	4.8	7.1	8.3	5.7
RAG	4.8 (0.0%)	10.2 (42.6%)	11.6 (39.1%)	7.5 (32.0%)
Iterative	5.9 (23.9%)	11.1 (55.3%)	13.4 (61.0%)	7.5 (32.0%)

Table 4: Effectiveness of De-Hallucinator in test generation compared to TestPilot. The bold numbers show statistically significant improvement over the baseline. The numbers in parentheses show the relative improvement over the baseline.

Prompt type	Passing tests	Coverage	Fixed hallucinations
Initial (TestPilot)	64.8	32.1	19.3
RAG & iterative	66.3 (3.1%)	33.7 (3.6%)	43.2 (94.0%)
Iterative	76.4 (17.9%)	37.0 (15.5%)	31.4 (63.2%)

iterations and add $n = 20$ API references into the prompt. As shown in Table 3, De-Hallucinator reduces the edit distance by 9.3 to 26.5 tokens, on average over the completions by the initial prompt, which is a relative improvement between 23.3% and 50.6%. This in turn translates to normalized edit similarity improvements of 12.1% to 27.5% relative to the baseline. Moreover, the approach relatively improves the exact API matches by 23.9% to 61.0%. For example, for the CodeGen v2.5 model, De-Hallucinator is able to predict 1.5 times more APIs correctly than the baseline. The approach shows statistically significant (using the Wilcoxon test and Pratt method) improvements over the baseline consistently for all metrics and all models. For example, Figure 5 shows a scenario where De-Hallucinator improves the completion. In this case the correct function is used by the model in the first try, but the order of parameters is wrong. By providing the API reference in the prompt, De-Hallucinator predicts the correct API usage, as shown in Fig. 5.

Test Generation. In the second set of experiments, we evaluate the effectiveness of De-Hallucinator in test generation. We use $k = 3$ and $n = 3$ as default parameters for these experiments. Table 4 shows statistically significant (using Wilcoxon test and Pratt method) improvements for code coverage and fixed hallucinations by De-Hallucinator. Since RAG prompts use the initial prompt for retrieval, and because the initial prompt just contains a single signature, it is not as useful as iterative prompts. This is also reflected in Table 4 as lower coverage compared to only using iterative prompts.

```

async def schedule_formatting(sources: Set[Path], fast: bool,
    write_back: WriteBack, mode: Mode, report: "Report", loop:
    asyncio.AbstractEventLoop, executor: "Executor") -> None:
    """Run formatting of `sources` in parallel using the provided
    `executor`. (Use ProcessPoolExecutors for actual parallelism.)
    `write_back`, `fast`, and `mode` options are passed to
    :func:`format_file_in_place`.

    """
    cache: Cache = {}
    if write_back not in (WriteBack.DIFF, WriteBack.COLOR_DIFF):
        cache = read_cache(mode)
    sources, cached = filter_cached(sources, cache) # <- baseline
    sources, cached = filter_cached(cache, sources) # <- ground truth

# API Reference:
# filter_cached(cache: Cache, sources: Iterable[Path]) ->
#     Tuple[Set[Path], Set[Path]] # Split an iterable of paths in
#     `sources` into two sets. The first contains paths of files that
#     modify
...
async def schedule_formatting(sources: Set[Path], fast: bool,
    write_back: WriteBack, mode: Mode, report: "Report", loop:
    asyncio.AbstractEventLoop, executor: "Executor") -> None:
    """Run formatting of `sources` in parallel using the provided
    `executor`. (Use ProcessPoolExecutors for actual parallelism.)
    `write_back`, `fast`, and `mode` options are passed to
    :func:`format_file_in_place`.

    """
    cache: Cache = {}
    if write_back not in (WriteBack.DIFF, WriteBack.COLOR_DIFF):
        cache = read_cache(mode)
    sources, cached = filter_cached(cache, sources) # <- De-Hallucinator

```

Figure 5: Completion by CodeGen highlighted in red, the ground truth, highlighted in green, and the completion by De-Hallucinator after augmenting the prompt with relevant APIs highlighted in blue.

Note that TestPilot uses some prompt refiners, such as retrying with error message, and including usage snippets and function bodies. These refiners can mitigate some hallucinations, but not as much as De-Hallucinator, as shown in Section 5.2. For example, Figure 6 shows a test generated by TestPilot, which uses non-existing APIs, but after providing the API reference in the prompt De-Hallucinator generates a test using the correct API.

5.3 RQ2: Correct Retrieval of API References

Code Completion. To better understand the effectiveness of De-Hallucinator, we investigate how often the approach successfully augments the prompt with the correct API references. Answering this question for all code completion tasks and all LLMs is difficult, because comparing the API references to the API usages is non-trivial due to different ways of importing APIs and passing arguments. Instead, for code completion, we manually inspect a sample of 20 completion tasks per LLM and count the number of times an API used in the ground truth is successfully added to the prompt by De-Hallucinator.

In our inspected samples for code completion, there are in total between 22 and 25 API usages (a completion can contain multiple API usages). The new prompt generated by our approach contains the correct API between two and six times, as shown in Table 5. Moreover, for CodeGen and CodeGen v2.5 there are five completion tasks where the completion from the initial prompt either misses the APIs or uses them incorrectly, but in the completions from RAG or iterative prompts, the API reference section of the prompt contains the correct API. The same happens for UniXcoder in four

```

let mocha = require('mocha');
let assert = require('assert');
let js_sds1 = require('js-sds1');
// class js_sds1.Deque(t = [], i = 1 << 12)
describe('test js_sds1', function() {
    it('test js-sds1.Deque', function(done) {
        let deque = new js_sds1.Deque([1, 2, 3, 4, 5]);
        assert.equal(deque.size(), 5);
        assert.equal(deque.front(), 1);
        assert.equal(deque.back(), 5);
        deque.push_back(6);
        assert.equal(deque.size(), 6);
        assert.equal(deque.back(), 6);
        deque.push_front();
        assert.equal(deque.size(), 7);
        assert.equal(deque.front(), 0);
        assert.equal(deque.pop_back(), 6);
        assert.equal(deque.size(), 6);
        assert.equal(deque.pop_front(), 0);
        assert.equal(deque.size(), 5);
        done();
    })
})
}

let mocha = require('mocha');
let assert = require('assert');
let js_sds1 = require('js-sds1');
// class js_sds1.Deque(t = [], i = 1 << 12)
// API Reference:
// js-sds1.Deque.prototype.pushFront(t)
// js-sds1.Deque.prototype.front()
// js-sds1.Queue.prototype.front()
describe('test js_sds1', function() {
    it('test js-sds1.Deque', function(done) {
        let deque = new js_sds1.Deque([1, 2, 3]);
        assert.equal(deque.front(), 1);
        deque.pushFront();
        assert.equal(deque.front(), 0);
        done();
    })
})

```

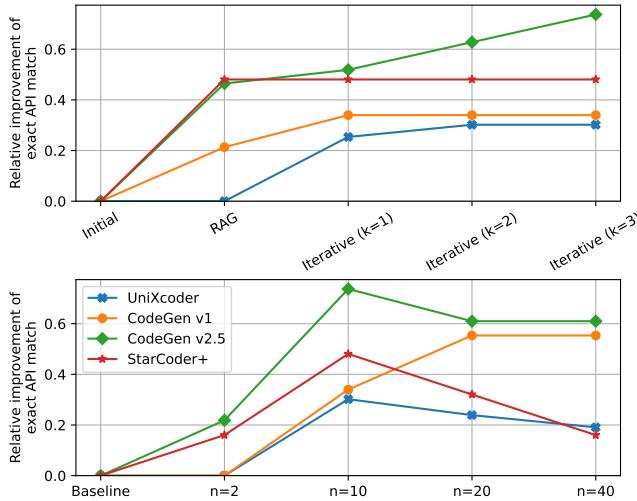
Figure 6: Test generated by TestPilot using GPT-3.5-turbo (top), and the iterative prompt by De-Hallucinator resulting in correct usage of APIs (bottom).

Table 5: Number of APIs correctly augmented in the prompt.

Model	Tasks	API usages				
		Missing /wrong	Expected API ref.	RAG	Iter. 1	Iter. 2
UniXcoder	20	18	4	5	5	5
CodeGen v1	20	17	5	6	6	6
CodeGen v2.5	20	15	5	5	6	6
StarCoder+	20	17	2	3	3	3

tasks and for StarCoder+ in two tasks. For cases where the approach fails to add the correct API reference into the prompt, the main reason is that the initial completion has low relevance w.r.t. the ground truth.

Test Generation. Since for the test generation task there are no ground truths for the generated code pieces, manual inspection is infeasible. Instead, we measure the number of passing tests and the number of non-crashing failing tests from an iterative prompt, as a proxy of success for the retrieval. Note that an iterative prompt is only created when an initial prompt causes a crash with one of the specified errors in Section 4. We observe that from the 622 instances where an iterative prompt is generated, 16.7% of iterative prompts result in a passing test, and 26.8% of iterative prompts result in a

Figure 7: Effects of k and n for code completion.

test without hallucinations. These results are also in line with the manual inspections of the code completion task above.

Overall, the results show that De-Hallucinator is able to successfully augment the prompt with the correct API references in many cases.

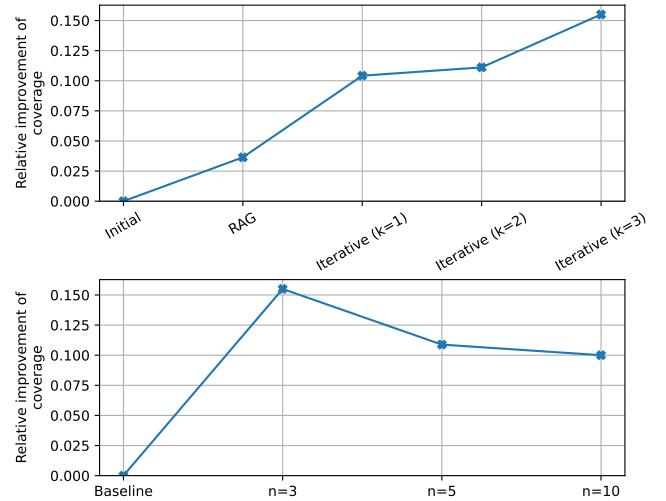
5.4 RQ3: Impact of the Hyperparameters

This research question evaluates the impact of the two main parameters of our approach. First, we consider the number k of iterations of iterative prompting. For both code completion and test generation, we run the approach with $k \in \{1, 2, 3\}$. As shown in Fig. 7, the first iteration provides significant improvement over the baseline, but the gain is reduced with further iterations. Higher values of k are beneficial when the model cannot immediately predict a relevant code, but upon presenting the first round of API references, the model responds with more relevant outputs. At the same time, even $k = 1$ provides clear improvements over the baseline, which makes De-Hallucinator useful even in scenarios where the cost of querying the model is high. We choose $k = 3$ for all other experiments in this paper.

Second, we study the impact of the maximum number n of API references that we add to the prompt. Depending on the task, the optimal number of API references in the prompt varies. Setting low values for n can result in missing relevant context, whereas adding many API references can confuse the model, while also costing context space. We perform experiments with $n \in \{2, 10, 20, 40\}$ for code completion, and $n \in \{3, 5, 10\}$ for test generation. Figure 7 shows the results of code completion, with exact API match peaking between $n = 10$ and $n = 20$. As a default in the rest of the paper, we use $n = 20$ for code completion. Figure 8 shows the effects of n on coverage of the generated tests. The peak in this case is at $n = 3$, which we use as the parameter for other experiments in this paper.

5.5 RQ4: Efficiency

The following evaluates the efficiency of the approach and how much each of De-Hallucinator’s components contributes to its running time. The pre-analysis step of code completion using CodeQL

Figure 8: Effects of k and n for test generation.

takes, on average, under one second per 1,000 lines of code in a project. For the projects in our dataset, it takes at most 80 seconds, and most projects need at most 26 seconds for the whole preprocessing phase. In a production-level implementation, our CodeQL-based approach could be replaced by using static information that is available in an IDE anyway, which is likely to further reduce the computational effort. Moreover, updating the indexed API references, e.g., when the code base evolves, can be done at low frequency in the background. For test generation, the pre-analysis takes between 0.6 seconds and 20 seconds with an average of 3.5 seconds. Because the JavaScript projects in our dataset are smaller than the Python projects, the pre-analysis is faster for test generation.

Retrieving relevant APIs and constructing the augmented prompt for one iteration takes from 21 to 227 milliseconds for code completion, and from 0.1 to 17 milliseconds for test generation. The time to query the LLMs ranges between 1.3 seconds (for the remotely deployed GPT-3.5) and 66.7 seconds (for CodeGen v2.5 running on our local Nvidia T4 GPU), on average per query. These numbers are roughly the same for the baseline and for querying the model with De-Hallucinator-augmented prompts. It is important to note that a production-level deployment would run the LLM on a GPU cluster, which typically answers queries within tens to hundreds of milliseconds, as evidenced by tools like Copilot and Tabnine.

6 LIMITATIONS AND THREATS TO VALIDITY

We assume an API to be available when generating the code that uses it. However, in some cases, a developer may first write an API usage and then implement the API. In such cases, De-Hallucinator would be unable to retrieve the API reference, and hence, could not provide any benefits. To address this limitation, one could configure De-Hallucinator to abstain from repeatedly querying the LLM if the similarity between the initial completion and the retrieved API references is below a threshold. We implement and evaluate De-Hallucinator for Python and JavaScript, and although our general approach could be applied to any language, our conclusions are valid only for these languages. The set of projects we use might

not be representative of all projects, which we try to mitigate by selecting a diverse set of popular projects.

7 RELATED WORK

Data-Driven Code Completion. The idea to augment traditional type-based code completion in a data-driven manner was introduced by Bruch et al.[8]. More recently, statistical models are used, such as a pre-trained BERT model [10] applied to code completion [34], and models trained for specific kinds of completions, e.g., API usages [50] and test methods [41]. Grammars can improve statistical code completions, either by restricting the tokens to predict [46] or by generating code that leaves some syntax subtrees undefined [19]. Hellendoorn et al. [21] study data-driven code completion based on recorded real-world completion requests, with a focus on completing single identifiers. Our work differs from all the above by providing project-specific API references based on previous completions as an input to a code completion model.

Code Completion with LLMs and Local Context. Motivated by the observation that LLMs lack project-specific information, Shrivastava et al. [56] propose a repository-level prompt generation technique to select the best context from a set of predefined contexts to solve the task of line completion. Their method relies on training a separate model that takes a context window around the incomplete line as input, and outputs a ranking for additional contexts. The training routine uses the LLM (in their case Codex) to calculate the loss function. Ding et al.[12] describe a similar method, called CoCoMIC, to address the challenge of project-specific APIs. They utilize a custom static analyzer, CCFinder, that initially creates a context graph of program components in the project, and allows retrieval of relevant contexts to complete a statement. They then fine-tuned CodeGen-2B-mono by adding the cross-file contexts to the input. Both of the above are tightly coupled with the underlying LLM: The first approach [56] uses the LLM to calculate the loss function for training a new model, and the second approach [12] changes the model’s weights during fine-tuning. In contrast, De-Hallucinator queries the LLM as a black-box, and hence, can be easily applied to other models.

An approach developed concurrently with ours [64] includes fragments of project-specific code in the prompt to improve the LLM’s predictions. Similar to our work, they also query the model iteratively. Unlike De-Hallucinator, their approach retrieves existing code fragments, and not API signatures. Since their approach relies on existing code fragments, it can only improve predictions when a project-specific API has already been used before and when this existing usage resembles the desired prediction, whereas our approach applies to all usages of project-specific APIs.

Combining LLMs and Retrieval. Lu et al.[36] use conventional retrieval methods to find similar code pieces in a pre-defined code database and add them to the prompt as dead code. Although this approach improves the quality of code completions by the LLMs, it does not address the challenge of project-specific APIs. Nashid et al.[39] propose a retrieval technique to find suitable examples for few-shot learning, but do not apply the idea to code completion. HyDE [15] prompts an LLM to generate hypothetical textual documents for a given query, and then retrieves real documents that

have an embedding similar to the hypothetical documents. Their work shares the observation that LLM predictions may be factually inaccurate, e.g., in our case by referring to non-existing APIs, while being similar to a factually correct document. By addressing this problem via retrieval, their approach is limited to producing already existing documents, whereas De-Hallucinator generates new code using an augmented prompt.

Automated Test Generation. Before the era of LLMs, random feedback-directed test generation became practical through Randoop [44]. LambdaTester [55] integrates higher order functions into test generation, and Nessie [3] targets asynchronous callbacks. TestPilot [54] uses LLMs and prompt refinement to generate human-readable regression tests. CodaMosa [27] uses LLMs to increase the coverage of automatic test generators when stuck in a plateau.

Improving LLM-Suggested Code. To improve code suggested by LLMs, existing techniques for automated program repair [26] can be applied in a post-processing step [13]. Alternatively, the code predicted by a model can serve as input for initializing and guiding component-based code synthesis [49]. The above work and ours shares the observation that completions from LLMs often share code elements with the desired code. Instead of improving code in a post-processing step, De-Hallucinator nudges an LLM toward producing better completions by improving the prompt.

Querying LLMs Multiple Times. Work on program repair queries a model multiple times until finding a suitable repair [37]. They repeatedly query the model with the same prompt and may trigger thousands of queries, whereas De-Hallucinator continuously augments the prompt and queries the model only a few times. Li et al.[32] propose querying a model with multiple mutations of the given code, and to then use the completion that is closest to the “average” completion. De-Hallucinator instead uses the initial prediction to construct an improved prompt. Xia et al.[62] introduce conversational program repair, which iteratively improves a prompt by adding test failures observed when executing the predicted code. In contrast, we do not require tests or executions, but only information that is statically available in a typical IDE.

Other Work on Models of Code. The impressive abilities of neural models of code [47] has lead to various other applications beyond code completion. For example, neural models provide type predictions [20, 38, 48, 60], make predictions about code changes [6, 22], and enable code search [16, 53]. LLMs are shown to be useful, e.g., for code mutation, test oracle generation, and test case generation [4, 25, 54], and for automated program repair [24].

8 CONCLUSION

Motivated by the inability of current LLM-based code generation approaches to correctly predict project-specific APIs, we present De-Hallucinator. The approach exploits the observation that LLMs often predict code that is similar to the desired code, but factually incorrect. We address the hallucination problem by iteratively augmenting the prompt with increasingly relevant API references. Our evaluation on two tasks, code completion and test generation, shows that De-Hallucinator significantly improves the quality of generations over the state-of-the-art baselines.

DATA-AVAILABILITY STATEMENT

Our implementation, datasets, and evaluation scripts are publicly available at <https://github.com/AryazE/dehallucinator> and <https://github.com/AryazE/testpilot>.

REFERENCES

- [1] [n. d.]. HuggingFace code embedding models. https://huggingface.co/models?pipeline_tag=feature-extraction&sort=trending&search=code. Accessed: 2024-06-05.
- [2] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. 2022. Do as I can, not as I say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691* (2022).
- [3] Ellen Artega, Sebastian Harner, Michael Pradel, and Frank Tip. 2022. Nessie: Automatically Testing JavaScript APIs with Asynchronous Callbacks. In *ICSE*.
- [4] Patrick Bareïf, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. *CoRR* abs/2206.01335 (2022). <https://doi.org/10.48550/arXiv.2206.01335> arXiv:2206.01335
- [5] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *TACL* 5 (2017), 135–146. <https://transacl.org/ojs/index.php/tacl/article/view/999>
- [6] Shaked Brody, Uri Alon, and Eran Yahav. 2020. A Structural Model for Contextual Code Changes. In *OOPSLA*.
- [7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS*. <https://proceedings.neurips.cc/paper/2020/hash/1457c0dbfc94967418fb8ac142f64a-Abstract.html>
- [8] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 213–222.
- [9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde 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>
- [10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 <http://arxiv.org/abs/1810.04805>
- [11] Jacob Devlin, Jonathan Uesato, Rishabh Singh, and Pushmeet Kohli. 2017. Semantic Code Repair using Neuro-Symbolic Transformation Networks. *CoRR* abs/1710.11054 (2017). arXiv:1710.11054 <http://arxiv.org/abs/1710.11054>
- [12] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context. *arXiv preprint arXiv:2212.10007* (2022).
- [13] Zhiyi Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. *Improving automatically generated code from Codex via Automated Program Repair*. Technical Report.
- [14] Joseph L Fleiss. 1981. *Statistical methods for rates and proportions*. John Wiley.
- [15] Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2023. Precise Zero-Shot Dense Retrieval without Relevance Labels. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, ACL 2023, Toronto, Canada, July 9-14, 2023, Anna Rogers, Jordan L. Boyd-Graber, and Naoki Okazaki (Eds.). Association for Computational Linguistics, 1762–1777. <https://aclanthology.org/2023.acl-long.99>
- [16] Xiaodong Gu, Hongyu Zhang, and Sungjun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE* 933–944. <https://doi.org/10.1145/3180155.3180167>
- [17] Yu Gu, Xiang Deng, and Yu Su. 2022. Don't Generate, Discriminate: A Proposal for Grounding Language Models to Real-World Environments. *arXiv preprint arXiv:2212.09736* (2022).
- [18] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).
- [19] Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. 2022. Learning to Complete Code with Sketches. In *ICLR*. <https://arxiv.org/abs/2106.10158>
- [20] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT*. 152–162. <https://doi.org/10.1145/3236024.3236051>
- [21] Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 2019. When Code Completion Fails: a Case Study on Real-World Completions. In *ICSE*.
- [22] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 518–529.
- [23] Naman Jain, Skanda Vaidyanathan, Arun Shankar Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram K. Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models meet Program Synthesis. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE*. 1219–1231. <https://doi.org/10.1145/3510003.3510203>
- [24] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 1430–1442. <https://doi.org/10.1109/ICSE48619.2023.00125>
- [25] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *45th IEEE/ACM International Conference on Software Engineering, ICSE*. 2312–2323. <https://doi.org/10.1109/ICSE48619.2023.00194>
- [26] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. <https://doi.org/10.1145/3318162>
- [27] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th International Conference on Software Engineering, ser. ICSE*.
- [28] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
- [29] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Jérôme Lang (Ed.). ijcai.org, 4159–4165. <https://doi.org/10.24963/ijcai.2018/578>
- [30] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
- [31] 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 Zebara, Ming-Ho Yee, Logesh Kumar Umaphathi, 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, Wenhai Yu, Swayam Singh, Sasha Lucioni, 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
- [32] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Shuai Wang, and Cuiyun Gao. 2022. CCTEST: Testing and Repairing Code Completion Systems. *arXiv preprint arXiv:2208.08289* (2022).
- [33] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (<conf-loc>, <city>Lisbon</city>, <country>Portugal</country>, </conf-loc>) ICSE '24*. Association for Computing Machinery, New York, NY, USA, Article 52, 13 pages. <https://doi.org/10.1145/3597503.3608128>

- [34] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-Task Learning based Pre-Trained Language Model for Code Completion. In *ASE*.
- [35] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *CoRR* abs/2107.13586 (2021). arXiv:2107.13586 <https://arxiv.org/abs/2107.13586>
- [36] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. arXiv:2203.07722 [cs.SE]
- [37] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNut: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. <https://doi.org/10.1145/3395363.3397369>
- [38] Rabec Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 304–315. <https://doi.org/10.1109/ICSE.2019.00045>
- [39] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Retained Few-Shot Learning. In *ICSE*.
- [40] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot’s Code Suggestions. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 1–5. <https://doi.org/10.1145/3524842.3528470>
- [41] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *ICSE*. arXiv preprint arXiv:2302.10166.
- [42] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. arXiv preprint arXiv:2305.02309 (2023).
- [43] 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 (2022).
- [44] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *International Conference on Software Engineering (ICSE)*. IEEE, 75–84.
- [45] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1532–1543. <https://doi.org/10.31115/v1/D14-1162>
- [46] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Syncromesh: Reliable Code Generation from Pre-trained Language Models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. <https://openreview.net/forum?id=KmfVD97J43e>
- [47] Michael Pradel and Satish Chandra. 2022. Neural software analysis. *Commun. ACM* 65, 1 (2022), 86–96. <https://doi.org/10.1145/3460348>
- [48] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Type-Writer: Neural Type Prediction with Search-based Validation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. 209–220. <https://doi.org/10.1145/3368089.3409715>
- [49] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal program inference: a marriage of pre-trained language models and component-based synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. <https://doi.org/10.1145/3485535>
- [50] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 44.
- [51] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. <https://doi.org/10.48550/ARXIV.1908.10084>
- [52] Deb Roy. 2005. Semiotic schemas: A framework for grounding language in action and perception. *Artificial Intelligence* 167, 1-2 (2005), 170–205.
- [53] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, 31–41.
- [54] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. <https://doi.org/10.1109/TSE.2023.3334955>
- [55] Marija Selakovic, Michael Pradel, Rezwana Karim Nawrin, and Frank Tip. 2018. Test Generation for Higher-Order Functions in Dynamic Languages. In *OOPSLA*.
- [56] Disha Srivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*. PMLR, 31693–31715.
- [57] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2727–2735.
- [58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 6000–6010. <http://papers.nips.cc/paper/7181-attention-is-all-you-need>
- [59] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. IdBench: Evaluating Semantic Representations of Identifier Names in Source Code. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 562–573. <https://doi.org/10.1109/ICSE43902.2021.00059>
- [60] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. <https://openreview.net/forum?id=Hkx6hANtwH>
- [61] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 959–971. <https://doi.org/10.1145/3540250.3549101>
- [62] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. *CoRR* abs/2301.13246 (2023). <https://doi.org/10.48550/arXiv.2301.13246>
- [63] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. *CoRR* abs/2202.13169 (2022). arXiv:2202.13169 <https://arxiv.org/abs/2202.13169>
- [64] Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. arXiv:2303.12570 [cs.CL]
- [65] Tianyi Zhang, Tao Yu, Tatsunori B. Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I. Wang. 2023. Coder Reviewer Reranking for Code Generation. In *ICML*.
- [66] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2024. Measuring GitHub Copilot’s Impact on Productivity. *Commun. ACM* 67, 3 (feb 2024), 54–63. <https://doi.org/10.1145/3633453>