# De-Hallucinator: Iterative Grounding for LLM-Based Code Completion

ARYAZ EGHBALI, University of Stuttgart, Germany
MICHAEL PRADEL, University of Stuttgart, Germany

Large languages models (LLMs) trained on datasets of publicly available source code have established a new state-of-the-art in code completion. However, these models are mostly unaware of the code that already 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. Although the API information is available to IDEs, the input size limit of LLMs prevents code completion techniques from including all relevant context into the prompt. This paper presents De-Hallucinator, an LLM-based code completion technique that grounds the predictions of a model 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 LLMs often predict code that resembles the desired completion, but that fails to correctly refer to already existing APIs. De-Hallucinator automatically identifies project-specific API references related to the code prefix and to the model's initial predictions and adds these references into the prompt. Our evaluation applies the approach to the task of predicting API usages in open-source Python projects. We show that De-Hallucinator consistently improves the predicted code across four state-of-the-art LLMs compared to querying the model only with the code before the cursor. In particular, the approach improves the edit distance of the predicted code by 23–51% and the recall of correctly predicted API usages by 24–61% relative to the baseline.

## 1 INTRODUCTION

Large language models (LLMs) have proven effective in many natural language [Brown et al. 2020] and programming tasks [Bareiß et al. 2022; Chen et al. 2021; Jain et al. 2022; Poesia et al. 2022; Xia and Zhang 2022; Xu et al. 2022]. One of the most promising tasks is code completion, as evidenced by the rapid adoption of LLM-based code completion tools, such as Copilot[1] and Tabnine[2]. Since these models are trained on large corpora of code, they usually generate syntactically correct code, and sometimes even the exact code that the developer intends to write.

A crucial factor for obtaining the desired completions is the *prompt*, i.e., the input given to the LLM. State-of-the-art LLMs build on transformers [Vaswani et al. 2017], 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 [Liu et al. 2021; Nashid et al. 2023; Shrivastava et al. 2023].

For code completion, the prompt typically consists of the incomplete code, and optionally, some additional context. The incomplete code can be at any level of granularity, such as a partial line, an incomplete function, or an incomplete file. The context can be any textual information that may help the model to predict a suitable completion [Jain et al. 2022; Schäfer et al. 2023]. For example, a commonly used prompt is a fixed-size window of code just before the cursor [Chen et al. 2021; Xu et al. 2022], i.e., code that a developer has already written and that may contain hints about what variables, functions, etc. to use in the completion.

[1] https://github.com/features/copilot
[2] https://www.tabnine.com/

Authors' addresses: 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.

DataStore.py

```
class DataStore():
  """Data structure for storing documents."""
  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]:
    """Returns all documents that contain the keyword."""
      return [d for d in self.documents if keyword in d]
  ...
```

utils.py

```
...
def relevance(document: str, keyword: str) -> float:
  """Returns the relevance of the document to the keyword."""
  return document.count(keyword) / len(document)
...
```

UI.py

```
...
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
  """Returns the top_k most relevant documents that contain the keyword sorted by relevance."""
  docs = ds.find_by_keyword(keyword)
  return sorted(docs, key=lambda d: relevance(d, keyword), reverse=True)[:top_k]
...
```

Fig. 1. The desired completion of search is highlighted in gray.

```
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
  """Returns the top_k most relevant documents that contain the keyword sorted by relevance."""
  docs = ds.find_by_keyword(keyword)
  return sorted(docs, key=lambda doc: doc.relevance , reverse=True)[:top_k]
```

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

Despite the impressive success of LLM-based code completion, 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 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 [Nguyen and Nadi 2022], or perhaps even worse, it may reimplement some functionality that is already present in the code base.

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.

```python
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
  """Returns the top_k most relevant documents that contain the keyword sorted by relevance."""
  docs = ds.find_by_keyword(keyword)
  docs_scores = [(doc, compute_relevance_score(doc, keyword) ) for doc in docs]
  sorted_docs_scores = sorted(docs_scores, key=lambda x: x[1], reverse=True)
  return [doc_score[0] for doc_score in sorted_docs_scores[:top_k]]
```

Fig. 3. The completion of search by ChatGPT highlighted in gray, and the wrong API usage highlighted in red.

Requesting an LLM, e.g., CodeGen [Nijkamp et al. 2022], 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 `doc.relevance`). A larger model, e.g., ChatGPT, faces similar challenges and implements `search` as shown in Fig. 3. As before, the code is partially correct but invokes an API not available in the project (a function `compute_relevance_score`). The underlying problem for both models is that they are not aware of the project-specific APIs that should be used to complete the code, and hence, the LLMs simply hallucinate 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. 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 prompts and an iterative form of LLM-based code completion. Our approach uses three types of prompts, which provide increasingly suitable context information. At first, De-Hallucinator queries the LLM with the conventional prompt that contains only the preceding code. Because the preceding code alone often is insufficient to obtain a correct completion, the approach then augments the prompt with API references that are most similar to the preceding code, which makes the second prompt type. However, also this prompt may fail to find a suitable completion, because the preceding code might not be similar to the desired API, or there are other APIs more similar to the preceding code than the correct one. Since LLMs often predict code that resembles the desired completion, but fail to correctly refer to project-specific APIs, the completion from the previous prompt types often resembles the desired API. Hence, to construct the third type of prompt, De-Hallucinator leverages these hints about what code the model intends to predict to retrieve suitable project-specific APIs, which are then added to the third type of prompt.

The result of these queries to the LLMs is a sequence of completions likely to be increasingly suitable. In a practical deployment, these completions could be shown to a user as a ranked list, or be further filtered by a validation mechanism, such as running a test suite.

Our approach addresses the two challenges described above. To address Challenge 1, De-Hallucinator adds information about project-specific APIs into the prompt. As a result, the model avoids (re)inventing code and instead correctly invokes existing APIs. The idea of augmenting an LLM with well-grounded facts relates to work on grounding of language models for natural languages [Ahn et al. 2022; Gu et al. 2022; Roy 2005]. To the best of our knowledge, we are the first to apply grounding to the problem of predicting API usages in code. To address Challenge 2, De-Hallucinator uses the model itself to gather hints about what kind of context information is most suitable to help the model make a better prediction. This iterative approach complements prior work that tries to guess the most suitable context from the incomplete code alone [Ding et al. 2022; Shrivastava et al. 2023]. Our work also relates to other retrieval-based prompt augmentation techniques, which try to find suitable few-shot examples [Nashid et al. 2023] . In contrast to existing retrieval-based prompt augmentation, De-Hallucinator retrieves API references instead of few-shot examples and addresses the task of code completion.

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 completion 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 four state-of-the-art LLMs for code, namely CodeGen [Nijkamp et al. 2022], CodeGen 2.5 [Nijkamp et al. 2023], UniXcoder [Guo et al. 2022a], and StarCoder+ [Li et al. 2023a]. Although, conceptually, the approach can be applied to any programming language, our evaluation focuses on Python as it is one of the most popular languages [3] and a common target of prior work on code completion [Chen et al. 2021; Guo et al. 2022a; Li et al. 2018, 2023a; Nijkamp et al. 2023, 2022; Svyatkovskiy et al. 2019; Zhang et al. 2023b]. Compared to querying the model with a prompt that contains only the code before the missing lines, we find that De-Hallucinator enables the model to provide more accurate predictions. In particular, we show a relative improvement of 23.28-50.64% in edit distance, of 12.12-27.48% in normalized edit similarity, and of 23.90-60.98% in recall of correctly predicted API usages.

In summary, this paper contributes the following:

- Empirical motivation showing that the problem of predicting API usages affects a large portion of failed code completion tasks.
- A technique for addressing this problem using off-the-shelf, unmodified LLMs.
- A novel algorithm that combines retrieval-augmented prompting 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 four state-of-the-art LLMs, De-Hallucinator offers more accurate completions than querying the same models with a fixed prompt.

## 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 hallucinates code that resembles the desired completion, but that fails to correctly refer to an API. To assess the importance of this limitation, 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,

[3] https://octoverse.github.com/2022/top-programming-languages

which we collect by (i) randomly selecting ten Python projects from a curated list of open-source projects [4] 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 of the authors independently classify the 50 pairs, with an inter-rater agreement (Cohen's kappa) of 0.76, which is considered excellent [Fleiss 1981]. The two authors then discuss the few cases with initial disagreement. The initial disagreement in each of the cases proved to be due to a human error, such as missing the difference between similar identifier names, `high` and `highpass`, or similar literal values, `3` and `8`. After resolving these errors, the authors 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 examples in Fig. 2 and Fig. 3. 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.

### 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 2,048 (GPT-3 models) and 32,768 (GPT-4-32k) 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. 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 an LLM-based code completion approach that uses retrieval of relevant APIs to improve the prompt for additional queries to the model. We call the approach De-Hallucinator, as it reduces the hallucinations of the LLM by providing relevant API references to ground the model. After defining the problem that De-Hallucinator addresses (Section 3.1), we provide an overview of the approach and present its main algorithm (Section 3.2). Sections 3.3 to 3.6 then present each of the components of De-Hallucinator in detail.

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

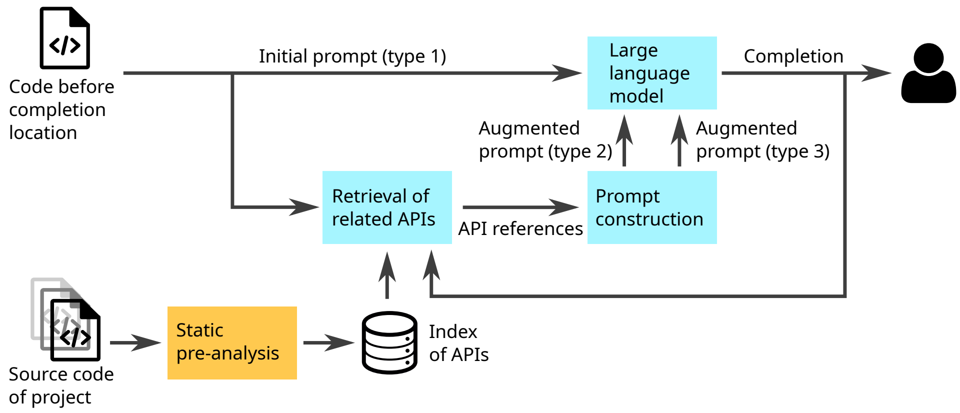

Fig. 4. Overview of De-Hallucinator.

### 3.1 Problem Definition

Before describing our approach, we define the problem that De-Hallucinator addresses. Given an incomplete piece of code $c$, where some code is missing at a cursor location $l$, the problem is to predict code $c'$ to be placed into $c$ at location $l$. Specifically, we consider the problem where $c'$ consists of one or more missing lines that contain an API usage, and where $c$ is all the code in the project except for the code between $l$ and the end of the current function.

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. Note that the API usage does not have to start directly at the cursor location, but only to start in the next line. 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.

To address this problem, we assume to have a generative language model $m$ trained on a large corpus of code. The code $c$ is part of a project $p$, which may contain a lot of already written code that $c'$ could refer to. However, we assume that the existing code in $p$ largely exceeds the maximum input size accepted by $m$. We further assume that the existing code in $p$ has not been part of $m$'s training data, which matches the practical usage scenario of using a single large-scale model as the basis of a code completion tool used by many projects.

### 3.2 Overview and Main Algorithm

Figure 4 gives a high-level overview of the approach, and Fig. 5 shows each step of the approach on our running example from Fig. 1. Using the code before the cursor location as an initial prompt, De-Hallucinator queries the model to get an initial completion. Because the model is not aware of project-specific APIs, the completion is likely to not make best use of these APIs. For the example in Fig. 5, the initial completion refers to a non-existing API `doc.relevance`.

To help the model predict a better completion, De-Hallucinator retrieves a ranked list of related API references in two ways. The first is to retrieve API references similar to the initial prompt. The second approach is to retrieve them based on similarity to the preceding code and the initial completion of the model. In Fig. 5, this step retrieves a reference to the `relevance` function defined in `utils.py`. Note that in this example the retrieved API reference is the same when retrieved based on the initial prompt and based on the initial completion. For efficient retrieval of APIs, De-Hallucinator analyzes the project in advance and indexes all APIs in the project. The approach

Initial prompt

```
...
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
  """Returns the top_k most relevant documents that contain the keyword sorted by relevance."""
  docs = ds.find_by_keyword(keyword)
```

Initial completion

```
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
  """Returns the top_k most relevant documents that contain the keyword sorted by relevance."""
  docs = ds.find_by_keyword(keyword)
  return sorted(docs, key=lambda doc: doc.relevance , reverse=True)[:top_k]
```

Most relevant API reference

```
relevance(document: str, keyword: str) -> float # Returns the relevance of the document to the
    keyword.
```

Augmented prompt

```
# API Reference:
# relevance(document: str, keyword: str) -> float # Returns the relevance of the document to the
    keyword.
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
  """Returns the top_k most relevant documents that contain the keyword sorted by relevance."""
  docs = ds.find_by_keyword(keyword)
```

Second completion

```
def search(ds: DataStore, keyword: str, top_k: int) -> List[str]:
  """Returns the top_k most relevant documents that contain the keyword sorted by relevance."""
  docs = ds.find_by_keyword(keyword)
  return sorted(docs, key=lambda doc: relevance(keyword, doc), reverse=True)[:top_k] # <- Correct
```

Fig. 5. Step-by-step progression of De-Hallucinator on the example in Fig. 1.

then uses the related API references to construct an augmented prompt, and queries the model again. In the example, the second query returns the correct completion, as shown at the bottom of Fig. 5. De-Hallucinator continues the retrieval based on the previous completion by the model until either reaching a fixed point, i.e., the model generates the same completion twice in a row, or until exhausting a configurable maximum number of queries, $k$.

**??** 1 shows the pseudo-code of the main algorithm of De-Hallucinator. The call to *retrieveRelevantAPIrefs* corresponds to the "Retrieval of related APIs" component in Fig. 4, and the call to *constructPrompt* matches the "Prompt construction" component in the figure. The following presents these two components, as well as the static pre-analysis that enables an efficient retrieval of related APIs, in detail.

### 3.3 Static Pre-Analysis

To ensure that the retrieval of API references does not unnecessarily slow down the code completion, 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.

*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,

**Algorithm 1:** Main algorithm of De-Hallucinator.

```
Data: Incomplete code $c$, model $m$, budget $k$
Result: List $\mathbb{C}$ of completions
$prompt = c$;
$c' = m(prompt)$;
$prevCompletion = c'$;
$\mathbb{C} = [c']$;
$APIrefs = retrieveRelevantAPIrefs(c)$;
$prompt = constructPrompt(c, APIrefs)$;
$c' = m(prompt)$;
$\mathbb{C} = \mathbb{C} + [c']$;
for $i \in \{1, ..., k-1\}$ do
    $c' = m(prompt)$;
    $\mathbb{C} = \mathbb{C} + [c']$;
    if $prevCompletion == c'$ then
        break;
    end
    $prevCompletion = c'$;
    $APIrefs = retrieveRelevantAPIrefs(c + c')$;
    $prompt = constructPrompt(c, APIrefs)$;
end
return $\mathbb{C}$
```

  - 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 `__init__` function.

Table 1 shows some of the API references extracted from our example project.

To extract API references from a project, the preprocessing phase of De-Hallucinator builds on CodeQL.[5] CodeQL is a query language and framework for statically analyzing source code, which provides a declarative approach for querying a code base. The framework uses static analysis to create a database and run the queries. Using CodeQL allows adding support for other languages with minimal effort. We implement two CodeQL queries, one for function references, and the other for class references and attribute references. Alternatively to our approach for gathering API references, an IDE-based implementation could reuse information about the current project that is computed by the static code indexing performed by an IDE anyway.

Because an effective code completion technique must provide suggestions quickly, De-Hallucinator indexes the extracted API references for fast retrieval, which takes two steps. The first step of the indexing is to embed API references into a vector space. Formally, we need an embedding function,

[5] https://codeql.github.com/

Table 1. Examples of API references extracted from the project in Fig. 1.

| Source | API reference | Type of API reference |
|---|---|---|
| DataStore.py in Fig. 1 | `DataStore.find_by_keyword(self, keyword: str) -> List[str] # Returns all documents that contain the keyword.` | Function reference |
| utils.py in Fig. 1 | `relevance(document: str, keyword: str) -> float # Returns the relevance of the document to the keyword.` | Function reference |
| DataStore.py in Fig. 1 | `class DataStore() # Data structure for storing documents.` | Class reference |
| DataStore.py in Fig. 1 | `DataStore.documents` | Attribute reference |

$E$, 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 $c_2$. To this end, we embed the API references into a vector space using Sentence-BERT [Reimers and Gurevych 2019], a BERT-based model designed for measuring the semantic similarity between sentences. We use a variant of this 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. Because our approach uses the embedding function as a black-box, any other embedding model or similarity-preserving vector representation [Wainakh et al. 2021] could also be used instead, e.g., GloVe [Pennington et al. 2014], BERT [Devlin et al. 2017], or FastText [Bojanowski et al. 2017]. The second step, after embedding the API references, is to index 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.

### 3.4 Retrieval of Related APIs

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 this end, De-Hallucinator uses a similarity metric that compares lines of code in the input with project-specific API references. Specifically, we embed each line in the input using the same pre-trained SentenceBERT model as in Section 3.3, 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. More formally, assuming that $E$ is the SentenceBERT model, we calculate $v_l = E(l)$ for each line $l \in \textit{completion}$, and then normalize the vectors into $v_l / |v_l|$. Next, we find the closest API reference of each line by querying the Ball Tree constructed in Section 3.3. 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 $l \in \textit{completion}$ 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.

Getting back to our running example, consider the third section in Fig. 5. It shows the API reference from our example project that the retrieval component finds to be the top-most relevant ($n = 1$) for both the incomplete code (initial prompt) and the initial prediction of the LLM. Our implementation considers multiple relevant API references, i.e., $n > 1$, which we omit in the example for brevity.

[6] https://huggingface.co/flax-sentence-embeddings/st-codesearch-distilroberta-base

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

```
# API Reference:
# <most relevant API reference>
# <2nd most relevant API reference>
# ...
# <nth most relevant API reference>
<clipped original prefix code>
```

Fig. 6. Template for the augmented prompt.

### 3.5 Prompt Construction

Given the code-to-complete and a list $R$ of API references that may enable the LLM to accurately complete the 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 prompt structure we use is shown in Fig. 6. The API references are shown as a block of commented lines at the beginning of 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. This commented block is followed by the original prompt clipped to fit in the prompt size.

For our running example, the fourth section in Fig. 5 shows the prompt for function `search` in our example, augmented with the API reference. Given the augmented prompt, the same LLM that predicted the code in Fig. 2, completes this function correctly in the last section of Fig. 5.

### 3.6 Integration with the LLM

De-Hallucinator is designed with minimal assumptions about the underlying LLM. The approach considers the model to be a black box that we query with a string that contains incomplete code and that returns a string with a suggested completion of the code. After receiving the completion, De-Hallucinator post-processes it to make the completion syntactically correct and to remove any extra completions that are out of the scope of the current completion. For example, during the completion of a function body, when the model generates a long completion, it is possible that the completion closes the scope of the current function, and starts defining a new function, or write code outside of the function body. Since our approach is focused towards API usages, the post-processing aims to mitigate these issues by cutting off the generated completion at the end of the line that completes the API usage.

## 4 IMPLEMENTATION

We implement the De-Hallucinator approach into a Python-based tool. For statically extracting API references from the current project, the implementation builds on CodeQL, and because we target Python code, on the Python language support of CodeQL, which offers easy access to the classes, functions, etc. in a code base. To access the LLMs, we use the HuggingFace transformers library,[8] making our implementation compatible with many publicly available models. 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.

## 5 EVALUATION

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

[8] https://huggingface.co/transformers/

Table 2. List of Python projects used for the evaluation.

| Project (owner/name) | Commit | Description | LoC | Stars* |
|---|---|---|---|---|
| graphql-python/graphene | 57cbef6 | GraphQL wrapper | 9,484 | 7.7k |
| geopy/geopy | ef48a8c | Client for geocoding web services | 10,000 | 3.9k |
| nvbn/thefuck | ceeaeab | CLI tool to correct commands | 12,181 | 77.1k |
| aaugustin/websockets** | ba1ed7a | Websocket server & client library | 14,186 | 4.5k |
| arrow-py/arrow | 74a759b | Library for easier use of date & time | 14,402 | 8.3k |
| lektor/lektor | be3c8cb | Static CMS framework | 16,852 | 3.7k |
| Parsely/streamparse | aabd9d0 | Python interface for job scheduling and distribution | 26,214 | 1.5k |
| Supervisor/supervisor | ca54549 | Process control for Unix-like systems | 29,860 | 7.8k |
| mwaskom/seaborn | f9827a3 | Statistical data visualization library | 37,367 | 10.7k |
| psf/black | ef6e079 | Python code formatter | 106,005 | 32.2k |
| scikit-learn/scikit-learn | f3c6fd6 | Machine learning framework for Python | 193,863 | 54.0k |

* As of May 1, 2023

** The project has been moved to python-websockets/websockets

**RQ1:** How much does De-Hallucinator improve code completions compared to querying the LLM with the default prompt only?
**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 completions?
**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 LLMs and Baseline.* We evaluate on four state-of-the-art LLMs: CodeGen [Nijkamp et al. 2022] with 2.7B parameters (Salesforce/codegen-2B-mono), CodeGen 2.5 [Nijkamp et al. 2023] with 7B parameters (Salesforce/codegen25-7b-mono), UniXCoder [Guo et al. 2022a] with 125M parameters (microsoft/unixcoder-base), and StarCoder+ [Li et al. 2023b] with 15.5B parameters (bigcode/starcoderplus). The reasons for selecting these models are (i) that, unlike the models offered by OpenAI, they are freely available and hence allows for building upon our work in the future, and (ii) 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.

*5.1.2 Dataset and Ground Truth.* With the goal of having a diverse set of projects in terms of size, domain, and popularity, we gather a dataset of eleven public Python projects from GitHub, shown in Table 2. These projects cover different domains, such as data visualization, machine learning, process management, job distribution, and geocoding. Likewise, the dataset has a wide range of project sizes and levels of popularity, as measured by GitHub stars.

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. Specifically, we use LibCST[9] to identify API calls that use an API from another file in the same project. 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 all local imports (i.e. starting with `.`) and imports of the package itself. 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. 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. Moreover, for StarCoder+ 26 completions result in non-ascii completions, which we also ignore. Overall, the evaluation dataset consists of 11 projects × 10 × 4 models = 440 code completion tasks.

During the static pre-analysis, we ensure to not index any information that would not be available during the code completion task. This is necessary because we run our CodeQL queries (Section 3.3) on a database that contains all of the code in the project, including the function that we want to complete. To avoid any leakage of information, the static pre-analysis does not extract any information from function bodies, except for `__init__` functions, which are analyzed to extract attribute references, and which we therefore exclude from the evaluation dataset.

*5.1.3 Metrics.* We evaluate the code completions in three ways:

- *Edit distance.* To quantify the number of edits a developer would have 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,[10] and then calculate the edit distance using NLTK's `edit_distance`.[11] That is, given predicted code $c_{pred}$, a ground truth $c_{correct}$, and the tokenizer function $tokens$, the metric is:

$$edit\_distance(tokens(c_{pred}), tokens(c_{correct}))$$

- *Normalized edit similarity.* Similar to previous work [Lu et al. 2022] 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:

$$1 - \frac{edit\_distance(tokens(c_{pred}), tokens(c_{correct}))}{max(|tokens(c_{pred})|, |tokens(c_{correct})|)}$$

- *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 example, given a line of code `docs = ds.find_by_keyword(keyword)` the corresponding API usage is `ds.find_by_keyword(keyword)`. The exact API match then is the percentage of exact matches between the prediction and the ground truth API usages. Let $api(c)$ be the API usages in code $c$, then the metric is:

$$\frac{|api(c_{pred}) \cap api(c_{correct})|}{|api(c_{correct})|}$$

[9] https://libcst.readthedocs.io/en/latest/

[10] https://huggingface.co/docs/transformers/model_doc/gpt2#transformers.GPT2TokenizerFast

[11] https://www.nltk.org/_modules/nltk/metrics/distance.html#edit_distance

Table 3. Effectiveness of De-Hallucinator compared to the baseline on four off-the-shelf LLMs. The **bold** numbers show statistically significant improvement over the baseline (Type 1). The numbers in parentheses show the relative improvement over the baseline (Type 1).

| Metric | Prompt type | UniXCoder (125M) | CodeGen v1 (2B) | CodeGen v2.5 (7B) | StarCoder+ (15B) |
|---|---|---|---|---|---|
| Edit distance (lower is better) | Type 1 | 52.39 | 40.04 | 47.21 | 44.60 |
| | Type 2 | **46.80** (10.67%) | **33.42** (16.53%) | **31.64** (32.98%) | **35.94** (19.42%) |
| | Type 3 | **25.86** (50.64%) | **30.72** (23.28%) | **30.09** (36.26%) | **33.50** (24.89%) |
| Normalized edit similarity (higher is better) | Type 1 | 33.40 | 43.64 | 43.93 | 33.24 |
| | Type 2 | **37.31** (11.71%) | **48.00** (9.99%) | **49.40** (12.45%) | **38.00** (14.32%) |
| | Type 3 | **42.58** (27.48%) | **48.93** (12.12%) | **50.15** (14.16%) | **39.66** (19.31%) |
| Exact API match (higher is better) | Type 1 | 4.77 | 7.12 | 8.33 | 5.68 |
| | Type 2 | 4.77 (0.0%) | **10.15** (42.56%) | **11.59** (39.13%) | **7.50** (32.04%) |
| | Type 3 | **5.91** (23.90%) | **11.06** (55.34%) | **13.41** (60.98%) | **7.50** (32.04%) |

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.

*5.1.4 Hardware.* We perform the experiments with the CodeGen v2.5 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

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 $k = 4$ iterations and add $n = 20$ API references into the prompt. As shown in Table 3, De-Hallucinator reduces the edit distance by 9.32 to 26.53 tokens, on average, which is between 23.28% and 50.64% relative improvement. This in turn translates to normalized edit similarity improvements of 12.12% to 27.48% relative to the baseline. Moreover, the approach relatively improves the exact API matches by 23.90% to 60.98%. For example, for the CodeGen v2.5 model, De-Hallucinator is able to predict 1.5 times more APIs correctly than the baseline. We evaluate the statistical significance of our improvements using the Wilcoxon test, with the Pratt method to deal with zero values. De-Hallucinator shows statistically significant improvements over the baseline consistently for all metrics and all models. The fact that the highest improvement is achieved for the exact API match metric shows that De-Hallucinator effectively achieves its goal of improving API usages in code completions. Given the importance of this problem in realistic code completion scenarios (Section 2), De-Hallucinator has the potential to significantly improve the productivity of developers.

*5.2.1 Examples.* Figure 7 shows the initial completion of CodeGen, with only the preceding code as prompt. In this example the model uses the `assert` statement instead of the custom `assert_equivalent` function, which is defined in the project. After augmenting the prompt with the API reference of this function, the same model correctly predicts the call to the existing API, as shown in Fig. 7.

```
...
def test_idempotent_any_syntatically_valid_python(
  src_contents: str, mode: black.FileMode
) -> None:
  ...
    dst_contents = black.format_str(src_contents, mode=mode)
  ...
  # And check that we got equivalent and stable output.
  assert dst_contents == src_contents     # <- baseline
  black.assert_equivalent(src_contents, dst_contents)  # <- ground truth
```

```
# API Reference:
# check_stability_and_equivalence( src_contents: str, dst_contents: str, *, mode: Mode) -> None #
    Perform stability and equivalence checks. Raise AssertionError if source and destination
    contents ar
# assert_equivalent(src: str, dst: str) -> None # Raise AssertionError if `src` and `dst` aren't
    equivalent.
...
def test_idempotent_any_syntatically_valid_python(
  src_contents: str, mode: black.FileMode
) -> None:
  ...
    dst_contents = black.format_str(src_contents, mode=mode)
  ...
  # And check that we got equivalent and stable output.
  assert_equivalent(src_contents, dst_contents) # <- De-Hallucinator
```

Fig. 7. Completion by CodeGen (baseline) highlighted in red, the ground truth highlighted in green, and the completion by De-Hallucinator using the augmented prompt highlighted in blue.

Figure 8 shows another 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. 8.

### 5.3 RQ2: Correct Retrieval of API References

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 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, 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, 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 4. Moreover, for CodeGen and CodeGen v2.5 there are five completion tasks where the completion from prompt of type 1 either misses the APIs or uses them incorrectly, but in the completions from type 2 or type 3 prompts, the API reference section of the prompt contains the correct API. The same happens for UniXcoder in four tasks and for StarCoder+ in two tasks. Overall, the results show that De-Hallucinator is able to successfully

```
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 modifi
...
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
```

Fig. 8. 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.

Table 4. Number of APIs correctly augmented in the prompt.

| | | API usages | | | | |
|---|---|---|---|---|---|---|
| Model | Tasks | Missing/wrong<br>in init. comp. | Expected API ref. added | | | |
| | | | Type 2 | Type 3 (k=2) | Type 3 (k=3) | Type 3 (k=4) |
| 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 |

augment the prompt with the correct API references in many cases. 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.

### 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 the main loop of De-Hallucinator, where we run the approach with $k \in \{2, 3, 4, 5\}$. As shown in Fig. 9, the first iteration provides significant improvement on all metrics, but the gain is reduced with further iterations. Higher values of $k$ are beneficial when the model cannot immediately predict a relevant completion, but upon presenting the first round of API references, the model responds with a more relevant completion. At the same time, even $k = 2$ 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 $k = 4$ for all other experiments in this paper, since $k = 5$ does not provide large enough improvements on our metrics.

Second, we study the impact of the maximum number $n$ of API references that we add to the prompt. Setting low values for $n$ can result in missing relevant context, whereas adding many API references reduces the space left for context before the cursor. We perform experiments with $n \in \{2, 10, 20, 40\}$. Figure 10 shows the results on our three metrics, with the similarity metrics increasing with $n$, and exact API match peaking between $n = 10$ and $n = 20$. As a default in the rest of the paper, we use $n = 20$.

### 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 preprocessing phase 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 on 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.

Retrieving relevant APIs and constructing the augmented prompt takes from 21 to 227 milliseconds per iteration. Finally, on our Nvidia T4 GPUs, the CodeGen v2.5 and the UniXcoder models take, on average, 66.7 seconds and 44.9 seconds for each completion, respectively, and on our Nvidia Tesla V100 GPU, the UniXcoder, CodeGen, CodeGen v2.5, and StarCoder+ models take, on average, 3.6 seconds, 18.6 seconds, 15.3 seconds, and 16.4 seconds per completion, respectively. 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 this time is not representative of the time a real-world code completion tool would take. 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 the (larger, but not freely available) models of OpenAI.

## 6 LIMITATIONS AND THREATS TO VALIDITY

One limitation of our work is that we assume an API to be available when completing 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 improve the completion. 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 have implemented and evaluated De-Hallucinator for Python, and although our general approach could be applied to any language, our conclusions are valid only for Python. The set of Python projects we use might not be representative of all Python projects, which we try to mitigate

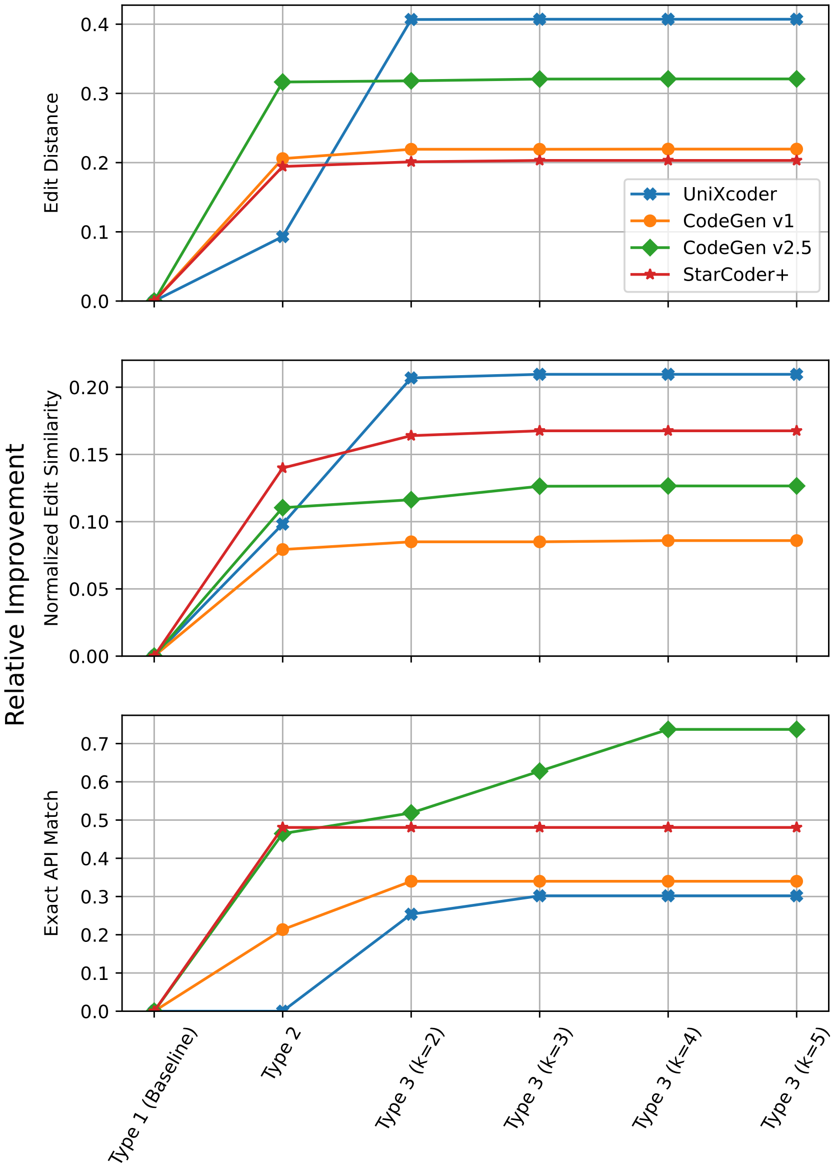

Fig. 9. Relative improvements over the baseline for the maximum number of iterations, *k*.

by selecting a diverse set of popular projects. Finally, we have evaluated De-Hallucinator with four LLMs, excluding the models of OpenAI because those models are not publicly available and may even be taken offline at short notice, which would prevent future work from reproducing and comparing against our work. Because these models, like the models we use in our evaluation, are unaware of project-specific APIs, we expect that our approach would also improve their completions.

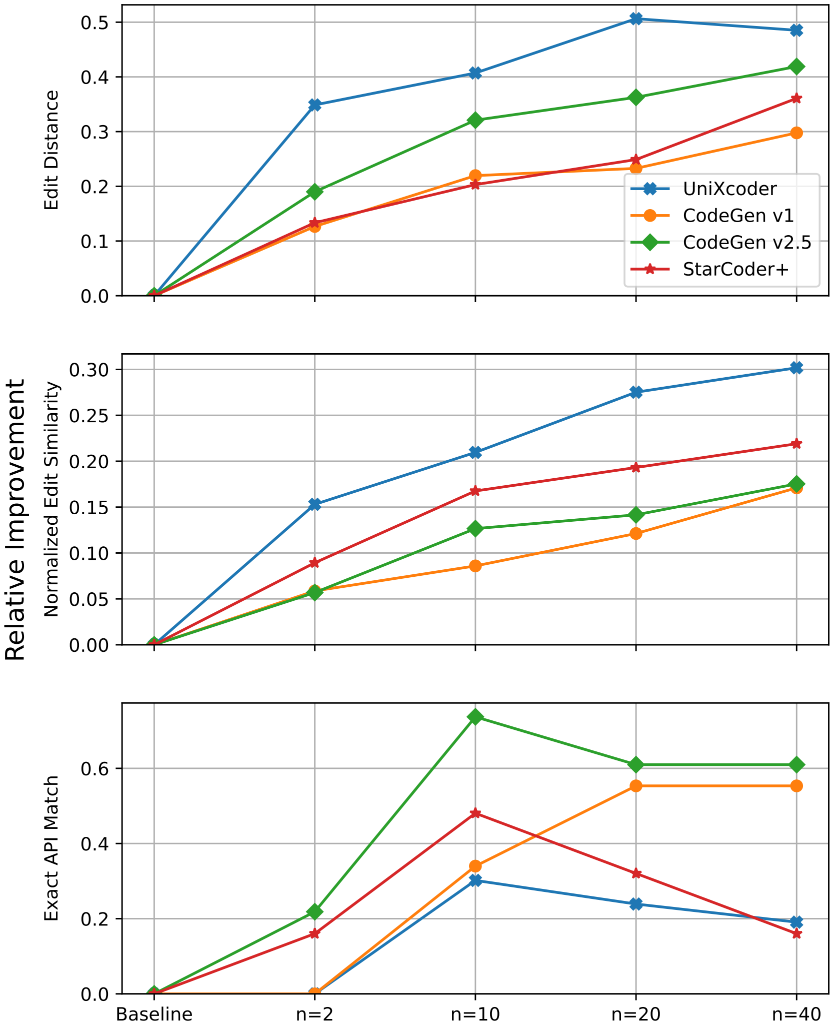

Fig. 10. Relative improvements of De-Hallucinator for different maximum API reference counts, *n*.

## 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.[Bruch et al. 2009]. More recently, statistical models are used, such as a pre-trained BERT model [Devlin et al. 2018] applied to code completion [Liu et al. 2020], and models trained for specific kinds of completions, e.g., API usages [Raychev et al. 2014] and test methods [Nie et al. 2023]. Grammars can help in improving statistical code completions, either by restricting what tokens could be predicted next [Poesia et al. 2022] or by generating code that leaves some syntax subtrees undefined [Guo et al. 2022b]. Hellendoorn et al. [Hellendoorn et al. 2019] study data-driven code completion techniques based on recorded real-world completion requests, with a focus on completing a single identifier at a time. Our work differs from all the above by providing project-specific API references 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. [Shrivastava et al. 2023] 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.[Ding et al. 2022] 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 [Shrivastava et al. 2023] uses the LLM to calculate the loss function for training a new model, and the second approach [Ding et al. 2022] 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. Moreover, De-Hallucinator introduces a dialog-based way of interacting with an LLM for completion.

An approach developed concurrently with ours [Zhang et al. 2023a] 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.

Agrawal et al. [Agrawal et al. 2023] modify the decoding stage of the LLM and filter out tokens that are invalid based on static analysis of the existing code. Their approach requires access to the decoder of the LLM, as opposed to our approach which treats the LLM as a black box.

*Combining LLMs and Retrieval.* Lu et al.[Lu et al. 2022] 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.[Nashid et al. 2023] propose a retrieval technique to find suitable examples for few-shot learning, but do not apply the idea to code completion. HyDE [Gao et al. 2023] 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.

*Improving LLM-Suggested Code.* To improve code suggested by LLMs, existing techniques for automated program repair [Le Goues et al. 2019] can be applied in a post-processing step [Fan et al. 2022]. Alternatively, the code predicted by a model can serve as input for initializing and guiding a component-based code synthesis algorithm [Rahmani et al. 2021]. 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 input given to the model.

*Querying LLMs Multiple Times.* Work on program repair queries a model multiple times until finding a suitable repair [Lutellier et al. 2020]. 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 to ensure low latency. Li et al.[Li et al. 2022] 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 differs from their approach by using the initial prediction to construct an improved prompt. Xia et al.[Xia and Zhang 2023] 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 [Pradel and Chandra 2022] has lead to various other applications beyond code completion. For example, neural models provide type predictions [Hellendoorn et al. 2018; Malik et al. 2019; Pradel et al. 2020; Wei et al. 2020], make predictions about code changes [Brody et al. 2020; Hoang et al. 2020], and enable code search [Gu et al. 2018; Sachdev et al. 2018]. LLMs are shown to be useful, e.g., for code mutation, test oracle generation, and test case generation [Bareiß et al. 2022; Kang et al. 2023; Schäfer et al. 2023], and for automated program repair [Jiang et al. 2023].

## 8 CONCLUSION

Motivated by the inability of current LLM-based code completion to correctly predict project-specific APIs, we present De-Hallucinator. Our approach combines the strengths of LLMs, retrieval-based code completion, and iteratively augmenting the prompt with relevant API references. Because De-Hallucinator treats the LLM as a black-box, does not rely on any fine-tuning, and does not require any additional training data, it can be applied to any LLM. Our evaluation on 440 code completion tasks shows that De-Hallucinator significantly improves the quality of code completions over the state-of-the-art baseline, with relative improvements of 23–51% in edit distance and 24–61% in the recall of correctly predicted API usages. Beyond code completion, the idea of using a model's initial predictions to guide retrieval efforts could be used in other software engineering tasks.

## DATA-AVAILABILITY STATEMENT

Our implementation and the dataset of incomplete functions is publicly available at https://github.com/AryazE/dehallucinator.

## REFERENCES

Lakshya Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K Lahiri, and Sriram Rajamani. 2023. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context. In *Thirty-seventh Conference on Neural Information Processing Systems*.

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).

Patrick Bareiß, 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

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

Shaked Brody, Uri Alon, and Eran Yahav. 2020. A Structural Model for Contextual Code Changes. In *OOPSLA*.

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 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

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.

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

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

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

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, 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).

Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. *Improving automatically generated code from Codex via Automated Program Repair.* Technical Report.

Joseph L Fleiss. 1981. *Statistical methods for rates and proportions.* John Wiley.

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 Naoaki Okazaki (Eds.). Association for Computational Linguistics, 1762–1777. https://aclanthology.org/2023.acl-long.99

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 933–944. https://doi.org/10.1145/3180155.3180167

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).

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022a. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. 2022b. Learning to Complete Code with Sketches. In *ICLR*. https://arxiv.org/abs/2106.10158

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 FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 152–162. https://doi.org/10.1145/3236024.3236051

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*.

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.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models meet Program Synthesis. In *ICSE*.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair, In ICSE. *arXiv preprint arXiv:2302.05020.*

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction, In ICSE. *arXiv preprint arXiv:2209.11515.*

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

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

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023a. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023b. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

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).

Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-Task Learning based Pre-Trained Language Model for Code Completion. In *ASE*.

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

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]

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

Rabee 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

Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *ICSE*.

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

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*.

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).

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).

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/10.3115/v1/D14-1162

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: 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=KmtVD97J43e

Michael Pradel and Satish Chandra. 2022. Neural software analysis. *Commun. ACM* 65, 1 (2022), 86–96. https://doi.org/10.1145/3460348

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: 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

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

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.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. https://doi.org/10.48550/ARXIV.1908.10084

Deb Roy. 2005. Semiotic schemas: A framework for grounding language in action and perception. *Artificial Intelligence* 167, 1-2 (2005), 170–205.

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.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive Test Generation Using a Large Language Model. *arXiv preprint arXiv:2302.06527* (2023).

Disha Shrivastava, 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.

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.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 6000–6010. http://papers.nips.cc/paper/7181-attention-is-all-you-need

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

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

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

Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. *CoRR* abs/2301.13246 (2023). https://doi.org/10.48550/arXiv.2301.13246 arXiv:2301.13246

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

Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. arXiv:2303.12570 [cs.CL]

Tianyi Zhang, Tao Yu, Tatsunori B. Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I. Wang. 2023b. Coder Reviewer Reranking for Code Generation. In *ICML*.