Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning

Noor Nashid
University of British Columbia
Vancouver, Canada
nashid@ece.ubc.ca

Mifta Sintaha
University of British Columbia
Vancouver, Canada
msintaha@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, Canada
amesbah@ece.ubc.ca

***Abstract*—Large language models trained on massive code corpora can generalize to new tasks without the need for task-specific fine-tuning. In few-shot learning, these models take as input a prompt, composed of natural language instructions, a few instances of task demonstration, and a query and generate an output. However, the creation of an effective prompt for code-related tasks in few-shot learning has received little attention. We present a technique for prompt creation that automatically retrieves code demonstrations similar to the developer task, based on embedding or frequency analysis. We apply our approach, CEDAR, to two different programming languages, statically and dynamically typed, and two different tasks, namely, test assertion generation and program repair. For each task, we compare CEDAR with state-of-the-art task-specific and fine-tuned models. The empirical results show that, with only a few relevant code demonstrations, our prompt creation technique is effective in both tasks with an accuracy of 76% and 52% for exact matches in test assertion generation and program repair tasks, respectively. For assertion generation, CEDAR outperforms existing task-specific and fine-tuned models by 333% and 11%, respectively. For program repair, CEDAR yields 189% better accuracy than task-specific models and is competitive with recent fine-tuned models. These findings have practical implications for practitioners, as CEDAR could potentially be applied to multilingual and multitask settings without task or language-specific training with minimal examples and effort.**

***Index Terms*—Large Language Models, Transformers, Few-shot learning, Program repair, Test assertion generation**

I. INTRODUCTION

Learning-based techniques have been applied to a wide array of source-code related tasks such as program repair [1]–[7] and assertion generation [8]–[10]. While task-specific ML models can replace hard-coded rules and heuristics, building large datasets of examples [11] and (re-)training the model involves significant effort and does not generalize beyond the given code processing task or programming language.

Very large neural networks, such as BERT [12] and T5 [13] trained for language understanding and generation have achieved great results for various tasks in recent years. However, they still require a significant number of task-specific training examples to *fine-tune* the model and can generally support a handful of programming languages. In addition, some of the model parameters must be updated to fit the task, adding more complexity to the model fine-tuning.

More recently, large language models such as GPT-3 [14], trained on large corpora of data at a very large scale, have been shown to generalize to new tasks without task-specific fine-tuning. These models take textual input, which is composed of natural language instruction, (optionally) a handful of examples of task demonstration, and a query that is defined as the prompt. This notion of learning from the desired task description, along with a few examples, is called *prompt-based few-shot learning* [15]. By employing prompts, these large language models are shown to be effective in different tasks that the model is not explicitly trained on, without the need for large-scale task-specific data collection or model parameter updating.

A number of large pre-trained language models for code generation [16]–[18] have been proposed, which primarily focus on prompts to generate code from natural language descriptions. Large language models such as CODEX are employed by GITHUB COPILOT [1] for code completion tasks. Recent studies assess how the adoption of code completion with large language models could be helpful to developer productivity [19]–[21], or solving coding interview problems and competitive programming [22]–[24], or how to employ feedback from external sources such as generated tests to improve the quality of generated code [25], [26]. There are also efforts to employ large language models to patch simple bugs [27], [28].

In this paper, we focus on the application of prompt-based few-shot learning on code-related tasks with the following questions to address: (a) Can few-shot learning be applied and generalized to specific code-related tasks? (b) How does few-shot learning compare to task-specific or fine-tuned models? (c) What are the ingredients of an effective prompt for code-related tasks? and (d) How to choose effective examples as demonstrations for code-related tasks?

We investigate how to build effective few-shot learning prompts for different code-processing tasks. We propose a novel technique for selecting a few demonstrations from a large pool of code examples by applying retrieval-based techniques based on embedding or frequency analysis. We implemented our approach in a framework called CEDAR (**C**ode **E**xample **D**emonstration **A**utomated **R**etrieval). We apply CEDAR on two different tasks, namely test assertion generation and program repair. We present an evaluation in

[1]https://copilot.github.com

which we use CODEX to instantiate few-shot learning with various numbers (e.g., zero-shot, one-shot, or n-shot) and forms (e.g., random vs systematic, or with vs without natural language descriptions) of code-related prompts. Our results show that with just a few examples, chosen systematically with our retrieval-based technique, a large language model employed for few-shot learning achieves significant results with an accuracy of 76% and 52% for test assertion generation and program repair tasks, respectively. We also compare our results against state-of-the-art learning-based models and see significant improvements with task-specific and fine-tuned models.

In this paper, we make the following contributions:

- A systematic approach of prompt-based few-shot learning for two code-related tasks, namely test assertion generation and program repair.
- First work to propose code-related prompts and retrieval-based methods (based on embedding and frequency) for demonstration selection in few-shot learning, to the best of our knowledge.
- Our framework CEDAR [29], which is available.
- An evaluation of the efficacy of our prompt-based demonstration retrieval approach in comparison with state-of-the-art techniques. For the program repair task, by adopting the retrieval-based technique, CEDAR outperforms the state-of-the-art task-specific model by a significant margin of 188.94% and the fine-tuned model by 4.91%. For assertion generation, retrieval-based demonstration selection outperforms task-specific and fine-tuned models by 333.47% and 11.05%, respectively.

This work has implications for practitioners and tool designers. Our results show that by combining retrieval-based techniques for prompt selection, CEDAR can potentially be applied to a wide range of tasks and programming languages for practical tool building without the need for large-scale task-specific data collection, model training, or fine-tuning.

II. APPROACH

Our goal is to devise an effective prompt that could help large language models with different code-related tasks. We first define the notion of a prompt for code generation tasks. Then we present our prompt construction technique, called CEDAR, which employs different techniques for code demonstration selection.

A. Problem Definition: Prompt Creation

In few-shot learning, a prompt needs to be provided for a given code-related processing task. We define a prompt, $\mathcal{P}$ = $\{\ x_{\text{test}} + \mathcal{CD} + \mathcal{NL}\ \}$ where x_{test} is a developer query to be inferred, $\mathcal{CD}$ is a set of code demonstrations $\mathcal{CD} = \{(x_i, y_i)\}_{i=1}^{n}$ of input code sequence (x_i) and desired output sequence (y_i), and $\mathcal{NL}$ is a natural language template; also $size(\mathcal{P}) \leq$ context-window, i.e., the prompt fits within the context window limit of the language model.[2] We call the instantiation of code demonstration examples, $\mathcal{E}$ together with the natural language query, the *ingredients* of a prompt.

Now for a new developer query x_{test}, the challenge is to devise a code retrieval technique $CR(x_{\text{test}}, \mathcal{CD})$, such that it will select a subset of code demonstration examples $\mathcal{E} = \{(x_j, y_j)\}_{j=1}^{m} \subset \mathcal{CD}$, where $m \leq n$ such that the chosen examples are similar to the task in x_{test} and fit within the context window limit of the language model. Given a large language model $\mathcal{LLM}$, during inference, a well-created prompt, $\mathcal{P}$ should lead to the desired target output sequence with high accuracy and not incur significant overload in terms of processing time or resource usage so that the retrieval technique can be adopted in real-time.

B. Demonstration Retrieval

Figure 1 depicts our overall approach. At a high level, the input to CEDAR is a set of code demonstrations, and the output is a text-based prompt ($\mathcal{P}$) that is composed of task description with or without natural language instructions, and instances of selected code demonstrations. Next, we describe the major components of CEDAR.

The first step is to build a prompt to elicit an intended response from the language model. The developer's intention could be described in the form of natural language text with or without including instances of code demonstrations. While natural language could provide the $\mathcal{LLM}$ with a cue, adding code demonstrations could help the model interpret the intended response that the developer expects unambiguously. To achieve this *demonstration builder* has two components, namely (1) the *demonstration retriever*, and (2) the *template selector*. For an unseen test example, x_{test} , *demonstration retriever* extracts similar code usage examples from the demonstration pool. A few well-written demonstration instances showing what the task entails can help the $\mathcal{LLM}$ to understand the expected behavior.

Our approach to selecting examples involves finding demonstrations that are similar to the task specified in the developer query. To that end, we employ information retrieval techniques [30] to obtain top-K relevant demonstrations from the pool demonstrations that match with the given query. In Figure 1, we use an accurately predicted sample from each task as our running example. Here, the *incomplete code* represents a real sample query from the test set for each task, and the retrieved similar code represents a single matched $\mathcal{CD}$ based on the query, x_{test}. We employ a sparse retrieval model [31] to perform lexical matching and compute relevance scores and rank code demonstrations ($\mathcal{CD}$) based on the query (x_{test}). Additionally, we explore dense retrieval techniques using sentence transformers [32] to retrieve relevant documents based on embedding vectors. Next, we describe our retrieval-based demonstration selection step.

[2]Language models limit the amount of contextual information that could be fed it to the model; the context window for CODEX is limited to 8,000 tokens.

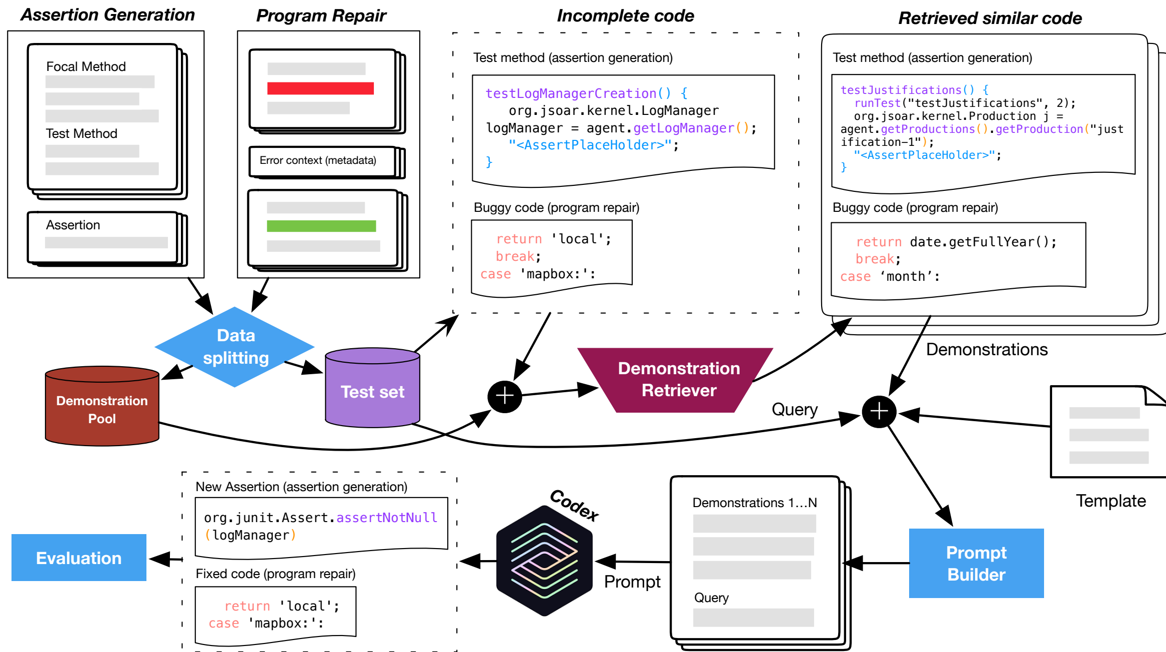

Fig. 1: Overview of our approach CEDAR.

Retrieval-based Demonstration Selection. In this step, we use a neural search and a frequency-based retrieval technique to select demonstrations based on similarity with the query. Therefore, the number of $\mathcal{CD}$s for each query, x_{test} may vary based on the number of similar demonstrations found for the query. However, since there is a context window limit in CODEX for a given prompt, we select the maximum number of similar demonstrations that can fit within the limit (including the query).

- *Embedding (*SROBERTA*):* For embedding-based search, we use the sentence transformer model [32] to encode code snippets as vectors. Specifically, we employ the pre-trained `st-codesearch-distilroberta-base`[3] model that was trained on the CodeSearchNet dataset [33]. In the assertion generation task, for a given query, the test method snippet is fed into the sentence transformer model to generate the embedding vector. Then cosine similarity metric is used to retrieve top-N test code snippets closest in the vector space to this query, from the set of code demonstrations, $\mathcal{CD}$. For program repair, given a query (x_{test}) for an error category, CEDAR performs a similarity search based on the buggy code snippet from the code demonstrations for that specific error category.
- *Frequency (*BM-25*):* We use the sparse retrieval method BM-25 [31] which is an extension of *TF-IDF* [34], to find demonstrations for each test sample with the most similar relevance score. In assertion generation, BM-25 ranks $\mathcal{CD}$s based on the test code snippet similarity for a given query, x_{test}. On the other hand, for program repair BM-25 ranks buggy code snippets from $\mathcal{CD}$s within that bug type.

Template Selection. A prompt can be composed with and without natural language instructions. The inclusion of natural language instructions in the prompt can provide the model with additional information about the developer's intent, but it also increases the risk of prompt engineering, due to the inherently experimental nature of using natural language [35], [36]. In the template selection step, we build prompts with and without natural language instructions for the best-performing strategy in each task to assess the effect of natural language instructions.

C. Prompt Builder and Model Invocation Layer

For a given code-related task, in this step, we build a prompt by composing (a) code demonstrations, $\mathcal{CD}$ (b) developer query, x_{test} and (c) natural language instructions. To select demonstrations, we take examples from the *demonstration retriever* component. Then we select a task-specific template and combine these three elements to build the final prompt.

For few-shot learning, we can employ different $\mathcal{LLM}$s. In this work, CEDAR employs CODEX, a $\mathcal{LLM}$ based on GPT-3. It is trained on a massive code corpus containing examples from many programming languages such as JavaScript, Python, C/C++, and Java. CODEX is released by OpenAI[4] and powers GITHUB COPILOT– an AI pair programmer that generates whole code snippets, given a natural language description as a prompt. Similar to GPT-3, CODEX is trained to predict the next token as an autocomplete task, given the past tokens as

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

[4]https://openai.com

Input

Code Demonstrations

```
### METHOD_UNDER_TEST:
getProduction(java.lang.String) {
  return productionsByName.get(name);
}

### UNIT_TEST
testJustifications() {
  runTest("testJustifications", 2);
  org.jsoar.kernel.Production j = agent.getProductions()
                               .getProduction("justification-1");
  "<AssertPlaceHolder>";
}

[METHOD_UNDER_TEST]: getProduction
[UNIT_TEST]: testJustifications

### generate assertion
org.junit.Assert.assertNull(j)
END_OF_DEMO
```

Demonstration 1

Demonstration 2

...

Demonstration N

Query

```
### METHOD_UNDER_TEST:
getLogManager() {
  return logManager;
}

### UNIT_TEST
testLogManagerCreation() {
  org.jsoar.kernel.LogManager logManager =
            agent.getLogManager();
  "<AssertPlaceHolder>";
}

[METHOD_UNDER_TEST]: getLogManager
[UNIT_TEST]: testLogManagerCreation

### generate assertion
```

Focal Method

Test Method

Context

Instruction

Output

```
org.junit.Assert.assertNotNull(logManager)
```

Fig. 2: Assertion Generation Prompt Template.

Input

Code Demonstrations

```
### Fix ESLint error in the following JavaScript code:
### Buggy JavaScript
  return date.getFullYear();
  break;
case 'month':

"rule_id" : no-unreachable
"evidence": Unreachable code.
"warning_line": break

### Fixed JavaScript
  return date.getFullYear();
case 'month':

END_OF_DEMO
```

Demonstration 1

Demonstration 2

...

Demonstration N

Query

Instruction

```
### Fix ESLint error in the following JavaScript code:
### Buggy JavaScript
  return 'local';
  break;
case 'mapbox:':

"rule_id" : no-unreachable
"evidence": Unreachable code.
"warning_line": break;

### Fixed JavaScript
```

Error Snippet

Error Context

Context

Output

```
  return 'local';
case 'mapbox:':
```

Fig. 3: Program Repair Prompt Template.

context over a large code corpus. We have chosen CODEX as it is a popular $\mathcal{LLM}$ on source code and has been studied in the literature [21], [27], [28], [37], [38]. We invoke the model with the constructed prompt using CODEX API.

Once the model generates a code suggestion, CEDAR sanitizes the output and performs an evaluation. Different tasks require different evaluation metrics, and there might be a need for more task-specific evaluation metrics for the automated evaluation of a given task. CEDAR provides generic evaluation metrics such as accuracy, edit distance, LCS, and latency as described in Section III-B. The final output from CEDAR is a sanitized model suggestion and evaluation results.

D. Prompt Templates for Code-Related Tasks

Prompt Design for Assertion Generation. Figure 2 illustrates a prompt template for assertion generation. The input prompt contains two sections – *code demonstrations,* $\mathcal{CD}$ and the *query*, x_{test}. The natural language instructions are denoted by the *blue text* in the template. Each demonstration is separated with a delimiter `END_OF_DEMO`. For assertion generation, each demonstration consists of the focal method, the test method containing an `<AssertPlaceholder>`, and the expected assertion. The query contains the context of a unit test (focal method and test method) followed by the instruction. This is essentially an autocomplete task where the query is an incomplete example used to prompt the model. Finally, the expected output for a given prompt is a single line of assertion.

Prompt Design for Program Repair. Figure 3 illustrates the running example for program repair in a prompt template. Similar to the previous template, the input prompt contains the same two sections ($\mathcal{CD}$ and x_{test}); the *blue text* indicates natural language instructions in the template and each demonstration is separated with `END_OF_DEMO` as a delimiter. For program repair, each demonstration consists of the error code snippet, error context (i.e., ESLint rule name, error message/evidence, warning line snippet), and the fixed code snippet. The query contains the natural language instruction, followed by the context of the buggy code snippet. As it is an autocomplete task, the comment `Fixed JavaScript` is used to signal the model to generate a correct code snippet. Finally, the

expected output for a given prompt is a multi-line code snippet addressing the bug.

III. Evaluation

To assess the effectiveness of CEDAR we address the following research questions:

- **RQ1**: How effective is retrieval-based prompt creation?
- **RQ2**: How does CEDAR's accuracy compare to state-of-the-art models?

For running our experiments, we use CODEX model `code-davinci-002`. We set the temperature as 0 to get a well-defined answer from CODEX. We run all the experiments on an Intel(R) Xeon(R) CPU 2.50GHz machine with 62 GB RAM. The running OS platform was RHEL 8. For the implementation of BM-25 we have used gensim[5]. which is a widely used package for similarity retrieval with large corpora. For the vector search, we employ vdblite[6] which is a database for vector similarity search. In the following subsections, we outline the results of the experiments that we designed to answer each of the research questions.

A. Datasets

We compare CEDAR with ATLAS [8] for assertion generation and TFIX [39] for program repair. To compare with these task-specific models, we select *code demonstrations, $\mathcal{CD}$*, from the corresponding training datasets. A task-specific model such as TFIX [39] and ATLAS [8] need a large training dataset. In contrast, for CEDAR the goal is to select a few instances of demonstrations from the TFIX and ATLAS training dataset. In this work, to retrieve demonstrations, we use the training dataset from respective tasks as the demonstration pool, but conceptually, this demonstration pool can be created from any project(s) containing a similar format of contextual information and query for a given task.

ATLAS Dataset. We use the dataset from ATLAS [8] to verify the ability of CEDAR in generating assert statements for Java code. Each data point contains a focal method and a test method as context for generating a single assertion for the given test method, known as Test-Assert Pairs (TAPs). These test methods only use the JUnit test framework, which is a popular unit testing framework in Java. ATLAS [8] contains both abstracted, AG_{abs} and raw source code, AG_{raw}; for our evaluation, we only use the raw source code. The dataset contains eight categories of assertions; `assertEquals`, `assertTrue`, `assertNotNull`, `assertThat`, `assertNull`, `assertFalse`, `assertArrayEquals`, `assertSame`. The TAPs from the training set are used as the pool for extracting demonstrations, and all the TAPs in the testing set are used for the evaluation. The total number of samples in AG_{raw} is 188,154, out of which 18,815 instances are used for the evaluation of our approach. The training set size for this dataset is 150,523, which is used as the demonstration pool for selecting $\mathcal{CD}$s.

[5]https://github.com/RaRe-Technologies/gensim

[6]https://pypi.org/project/vdblite

TFIX Dataset. We use the dataset from TFIX [39] to evaluate the ability of CEDAR to repair JavaScript code. This dataset contains static linting errors collected from the top 500k GitHub public repositories based on the number of stars and consists of 52 error types from ESLint[7]. Each data point contains a buggy code snippet (usually 2-4 lines), and an error context from ESLint that provides information about the error type, message, and warning line. In their paper, the authors of TFIX [39] use two types of datasets to evaluate their model, namely *clean test* and *random test*. We use *clean test* dataset for querying our model as it has been used for comparison with other models in [39], unlike *random test*. Similar to the ATLAS dataset, we use the training set for extracting demonstrations. The total number of samples in this dataset is 104,804. Following the same split from TFIX, we choose 10% of the samples as x_{test} during inference. From the remaining sample size, 90% is selected for fine-tuning and 10% for validation in TFIX. We use the samples that were used for fine-tuning to choose the $\mathcal{CD}$s.

B. Evaluation metrics

Accuracy exact match (%). This metric is used to determine the percentage of samples where the inferred output matches lexically with the expected output.

Accuracy plausible match (%). We use this metric to determine the percentage of samples where the predicted output is similar to the expected output.

LCS (%). Longest Common Subsequence (LCS) is the ratio of the longest common subsequence between the predicted output and the expected output. We calculate this as a percentage. This metric has been used in sentence ordering of NLP tasks [40].

Edit distance (ED.). This metric determines the number of edit operations required for the inferred output to match the expected output. These edit operations can be addition, deletion or modification. The lower the edit distance, the closer the predicted output is to the actual output. This metric has been used in learning-based code-related work as a proxy to measure developer effort [41], [42].

Assertion method matched (AMM) %. This metric is only used for the assertion generation task, where we calculate the percentage of match between the expected assertion type-/method and predicted assertion type/method. This metric has been employed in previous work [8].

Inference time. This is the amount of time in seconds to predict the output, given a prompt.

C. Baselines

We compare CEDAR to several baseline approaches that select random code snippets in the demonstration retriever. In the random strategy, we choose random examples from the demonstration pool.

[7]https://eslint.org

TABLE I: Results for demonstration selection strategy in code-related tasks.

Task	Strategy	Types	Examples	EM Acc.(%)	PM Acc.(%)	LCS(%)	ED.	AMM.(%)	Inf. time(s)
Assertion Generation	Random	Zero-shot	0	0.00	14.42	15.34	36.62	29.16	1.33
		One-shot	1	44.41	47.64	64.97	57.83	70.24	3.64
		8-shot/category	8	50.33	53.69	62.65	17.94	74.48	2.54
		8-shot	8	50.57	53.34	68.77	19.10	74.12	2.41
		20-shot	20	49.99	52.73	74.64	20.53	73.80	3.16
	CEDAR	Embedding	7	75.79	77.40	88.50	11.35	88.49	1.78
		Frequency	**6**	**76.55**	**78.32**	**88.99**	**13.18**	**89.31**	**2.20**
Program Repair	Random	Zero-shot	0	27.33	32.57	68.02	98.30	-	3.35
		One-shot	1	28.54	37.57	67.93	87.13	-	3.95
		52-shot/category	52	33.47	44.10	67.05	23.34	-	3.83
		52-shot	52	34.65	45.02	67.64	23.71	-	3.86
		60-shot	60	39.88	49.80	68.22	23.09	-	3.85
	CEDAR	Embedding	48	46.12	55.63	73.99	21.47	-	3.78
		Frequency	**41**	**51.72**	**60.55**	**75.57**	**32.97**	-	**4.22**

For the baseline, we consider the search space to be random when selecting demonstration samples to be used in the prompt. To this end, we vary the number of samples by no examples (zero-shot), one example (one-shot), N examples selected randomly (random N-shot), and N examples based on category (N-shot per category):

- *Zero-shot:* This prompt does not contain any code demonstration, $\mathcal{CD}$; instead, the model is directly queried with natural language instruction without any examples.
- *One-shot:* This prompt contains only one $\mathcal{CD}$ followed by a query.
- *N-shot - 1 example per category:* This prompt contains N random examples from each category for that representative task. The goal is to show the $\mathcal{LLM}$ with examples from each category for that code generation task. For assertion generation, there are 8 different assertion types as described in III-A. We chose 1 example from each assertion type, hence it is 8-shot for assertion generation. For program repair, as there are 52 different error types, we select 52-shot to include one instance from each error category.
- *N-shot:* We select code demonstration examples, $\mathcal{E}$, in two ways to understand: (a) *role of examples per category*: while for the previous strategy, we chose 1 example per category, here we use N number of examples randomly without the explicit consideration of category. As a result, the number of examples for assertion generation and program repair remains the same at 8 and 52, respectively; (b) *effect of including more examples*: we provide more examples to the $\mathcal{LLM}$ to understand their impact. We chose 20 examples per prompt for assertion generation that fit within the context window limit. For a given developer query (x_{test}) for program repair, we select random 60-examples from the same error category within the context window. These are the maximum number of code demonstrations used across all strategies.

D. *Effectiveness of the Prompt (RQ1)*

To study the effectiveness of a prompt, we experiment with various input configurations. As discussed in Section II, we build the prompt using extracted demonstrations from the training set and query the model using data points from the test set. We do this for each of the datasets and vary both the demonstration selection strategy and template. The demonstration template is varied by excluding or including natural language instructions in the prompt. We experiment with the two types of demonstration selection strategy – random and retrieval based. We execute the random strategies over the whole test dataset twice and report the averages. First, we determine the best demonstration strategy needed for prompt-based learning. After identifying the best-performing demonstration strategy, we investigate the effect of natural language instructions in the prompt.

Results. Tables I and II show the results for each of the code-related tasks. We discuss the results for each task below.

Assertion Generation: In Table I, we observe that in the **zero-shot** setting, the exact match accuracy (EM Acc.) is 0% and plausible match accuracy (PM Acc.) is 14.42%. However, the inference time (1.33 seconds) and percentage of assertion matched (29.16%) is the lowest for a zero-shot setting. The average LCS is only 15.34% on average. In the **one-shot** setting, the accuracy increased significantly to 44.41% for the exact match and 47.64% for the plausible match. The inference time for generating the assertion also increased to 3.64 seconds. The average edit distance for one-shot increased (to 57.83) compared to the edit distance for the zero-shot setting (36.62). The LCS (%) is 4.2 times higher than the zero-shot setting i.e. the actual assertion order is 64.97% similar to the expected assertion order in the one-shot setting. The percentage of assertion matched is 2.4 times more than the zero-shot setting (70.24%). In the **category-specific 8-shot** setting, the accuracy increased to 50.33% (exact match) and 53.69% (plausible match), respectively. The LCS (%) decreased to 62.65%, while the average edit distance decreased to 17.94. The percentage of matched assertions increased to 74.48%, and the inference time was 2.54 seconds,

TABLE II: Results for demonstration template in code-related tasks.

Task	Template	Examples	EM Acc.(%)	PM Acc.(%)	LCS(%)	ED.	AMM.(%)	Inf. time(s)
Assertion Generation	w/o NL instructions	6	76.53	78.27	89.00	13.68	89.37	2.95
	w/NL instructions	**6**	**76.55**	**78.32**	**88.99**	**13.18**	**89.31**	**2.20**
Program Repair	w/o NL instructions	53	49.31	58.33	74.41	36.25	-	3.27
	w/NL instructions	**41**	**51.72**	**60.55**	**75.57**	**32.97**	-	**4.22**

which is lower than the one-shot setting. We observe the best performance in the random strategy from the **8-shot** setting, where the demonstrations have been selected from the overall dataset. Here, the accuracy for the exact match increased to 50.57% and 53.34% accuracy for the plausible match. The LCS (%) also increased to 68.77%, which is also higher than the one-shot setting. Additionally, the edit distance is low, requiring only 19.10 edit operations on average, to reach the expected assertion in cases where it does not match. The average inference time is also lower than the 8-shot per category setting. However, after adding more examples in the random **20-shot** setting, the exact match and plausible match accuracy slightly decreased to 49.99% and 52.73%, respectively. While the LCS (%) increased to 74.64%, the average inference time (3.16 seconds), edit distance (20.53), and assertion method matched (73.80%) declined slightly with the inclusion of more examples.

The retrieval-based strategy of CEDAR yielded the best results in assertion generation. In this strategy, the average number of $\mathcal{CD}$s for SROBERTA and BM-25 is 7 and 6, respectively. The demonstration selection based on SROBERTA embedding significantly improved the performance with an exact match accuracy of **75.79%** and plausible match accuracy of 77.40%. The average LCS (%) increased to 88.50%, and assertion matched (%) increased to 88.49% which is significantly higher than the best-performing random demonstration selection strategy. The inference time and edit distance is also very low (1.78 seconds and 11.35 respectively). With the frequency-based (BM-25) retrieval strategy, we observe an improvement over the SROBERTA retrieval, with an accuracy of **76.55%** (exact match) and 78.32% (plausible match). The LCS (%) and assertion method matched (%) also increased to 88.99% and 89.31%, respectively. However, both the edit distance (13.18) and inference time (2.20 seconds) are slightly higher than SROBERTA retrieval strategy. This entails that both retrieval-based techniques - SROBERTA and BM-25 provide competitive results in assertion generation task.

We also evaluated the effect of the demonstration template by running the best performing strategy, BM-25 based retrieval without natural language instructions. In the assertion generation task, the impact of natural language instruction is negligible, as shown in Table II. The exact match (76.53%) and plausible match (78.27%) accuracy without natural language instructions is slightly lower compared to the template with NL instructions. The average edit distance (13.68) and inference time (2.95 seconds) is higher than the template containing natural language instructions. However, we observe a slight improvement in the percentage of assertion method matched (89.37%) and average LCS percentage (89.00%).

Program Repair: Also, for program repair, we observe that the retrieval-based demonstration selection outperforms randomly selecting demonstrations. In random **zero-shot** learning, the exact match accuracy is 27.33%, however, when considering the plausible match, the accuracy increases to 32.57%. The inference time is 3.35 seconds which is higher than the assertion generation task. This is because of generating multi-line code fixes, as opposed to single-line assertions in the previous task. The edit distance is the highest (98.30%), and the LCS percentage is 68.02%. **One-shot** learning yielded a slight improvement in both exact match accuracy (28.54%), plausible match accuracy (37.57%), LCS percentage (67.93%), and a lower edit distance of 87.13%. The inference time is higher than the zero-shot setting by 0.60 seconds. In **52-shot per category**, the exact match accuracy increased even further to 33.47% and plausible match accuracy increased to 44.10%. However, the LCS percentage reduced slightly to 67.05%. The average edit distance and inference time decreased significantly to 23.64 and 3.83 seconds, respectively. In the random **52-shot** setting, the exact match, and plausible match accuracy increased slightly to 34.65% and 45.02%. The LCS % increased to 67.64% whereas the edit distance and inference time remained similar to 52-shot per category. In the **60-shot** setting, the performance improved moderately with 39.88% exact match and 49.80% plausible match. The LCS percentage, edit distance, and inference time improved slightly to 68.22%, 23.09, and 3.85 seconds, respectively.

In the retrieval-based strategy, SROBERTA retrieval yielded **46.12%** in exact matches and 55.63% in plausible match accuracy, which is significantly higher than the random strategy. The LCS (%) increased to 73.99%, and the average edit distance decreased to 21.47. The inference time was reduced slightly to 3.78 seconds. With BM-25 retrieval, the accuracy for both exact and plausible match increased even further to **51.72%** and 60.55% respectively. The LCS (%) with 75.57% is the highest among all other strategies. However, the average edit distance and inference time increased to 32.97% and 4.22 seconds. For SROBERTA, the average number of $\mathcal{CD}$s is 48, and with BM-25, it is 41.

When varying the demonstration template in program repair, we observe the best performance with natural language instructions as shown in Table II. Unlike the assertion generation task, where we saw similar performance when varying the template, this task yielded an improvement with the addition of natural language instructions. The accuracy is lower for

TABLE III: Comparison with state-of-the-art in code-related tasks.

Task	Approach	Model type	Exact Match Acc. (%)
Assertion Generation	ATLAS	Task-specific	17.66
	Mastropaolo et al. [9]	Fine-tuned	57.60
	Mastropaolo et al. [10]	Fine-tuned	68.93
	CEDAR	**Few-shot learner**	**76.55**
Program Repair	HOPPITY	Task-specific	7.90
	COCONUT	Task-specific	11.70
	SEQUENCER	Task-specific	17.90
	TFIX	Fine-tuned	49.30
	CEDAR	**Few-shot learner**	**51.72**

both exact (49.31%) and plausible matches (58.33%) in the template without natural language instructions. The LCS (%) is slightly lower (74.41%) and the average edit distance increases to 36.25. However, the inference time of 3.27 seconds, is lower compared to the template containing natural language instructions. We use the results achieved with BM-25 retrieval, using natural language instructions in the template as our default approach in CEDAR, for both tasks going forward.

E. Comparison with State-of-the-Art (RQ2)

We compare CEDAR with state-of-the-art learning based-models that have been trained on the same dataset (i.e., ATLAS AG_{raw} and TFIX dataset) and evaluated in assertion generation and program repair tasks. We use both task-specific models ATLAS [8] for assertion generation and HOPPITY [43], SEQUENCER [44], COCONUT [45] for program repair as well as the recent fine-tuned T5 models Mastropaolo et al. [9], [10] (assertion generation) and TFIX [39] (program repair). We show the best reported top-1 exact match accuracy percentage of these approaches in Table III.

Assertion Generation: In Table III, the accuracy in top-1 for the task-specific model, ATLAS, AG_{raw} is 17.66%. ATLAS uses sequence-to-sequence learning through a recurrent neural network (RNN) encoder-decoder model to learn test assertion statements within test methods. It uses the focal method and test method as context for sequence-to-sequence learning to generate a single assertion statement. Recently two other approaches by Mastropaolo et al. [9], [10] outperformed the NMT-based model of ATLAS by a significant margin (57.60% and 68.93%, respectively). Both of these approaches use a T5 transformer model that has been pre-trained on a large database of source code and natural language text and then fine-tuned in single-task and multi-task settings using datasets for each code-related task. They used the same dataset from ATLAS, AG_{raw} and AG_{abs} for evaluating their performance in assertion generation. CEDAR outperforms both the fine-tuned models and task-specific models by a large margin (76.55%) without the need for pre-training or fine-tuning.

Program Repair: In this task, we use the results reported in [39] for each of state-of-the-art task-specific and fine-tuned models that have been trained and evaluated on the TFIX dataset containing JavaScript ESLint errors. As shown in Table III, HOPPITY had the lowest accuracy of 7.90%, followed by 11.70% exact match accuracy in COCONUT. SEQUENCER performed better than COCONUT with an accuracy of 17.90%. TFIX significantly outperformed all four of these models by fine-tuning a large T5 transformer model. TFIX had been fine-tuned on 94K samples for the task of generating JavaScript code fixes. Nevertheless, CEDAR outperforms all these models with an (exact match) accuracy of 51.72% without requiring any pre-training or fine-tuning step and on a much smaller example set.

IV. DISCUSSION

A. Effect of Relevant Contextual Information

CEDAR employs prompt-based learning on code-related tasks, to assist developers by suggesting assert statements or bug fixes. Our results show that tuning the prompt effectively is key to achieving high accuracy in specific downstream tasks. In both assertion generation and program repair, we observe that employing a random strategy to determine $\mathcal{CD}$s, can still outperform traditional task-specific deep learning-based techniques (as shown in Section III-E (RQ2)). This indicates the superiority of the usage of large pre-trained language models in source code processing tasks. However, our results for RQ1 in Section III-D show that the retrieval-based selection of $\mathcal{CD}$s significantly improves the accuracy of predictions for both these code-related tasks, and is able to outperform not only the task-specific models but also the fine-tuned models e.g. TFIX and Mastropaolo et al. [9], [10]. We achieve this by using BM-25 as the ranking function to select relevant examples from the demonstration pool. For instance, in Figure 3 (program repair), the prompt template contains relevant code demonstrations pertaining to the buggy code, error type, warning line, and evidence. If these $\mathcal{CD}$s were chosen randomly, it would be difficult to ensure the selection of similar and relevant demonstrations to reason about a fix. Similarly, in Figure 2 (assertion generation), the $\mathcal{CD}$s were selected similar to the test method of the query, x_{test}. Although the assertion type in the demonstration is not syntactically similar (`assertNull`), the model is still able to leverage the relevant unit test and focal methods in the $\mathcal{CD}$ to generate the assertion. We also observe that using natural language instructions improves the performance as it helps the model in understanding the task at hand more effectively (as shown in Table II). Furthermore, we note that the inclusion of more demonstrations does not necessarily improve the accuracy of predictions. In Table I, we see a

significant jump in the random strategy of assertion generation when including one $\mathcal{CD}$, however, after adding 8 $\mathcal{CD}$s, the performance only slightly improved and adding 20 examples slightly deteriorated the performance. In the program repair task, we observe a similar pattern, although with a slight improvement in the 60-shot setting. However, by using the retrieval-augmented strategy, having 6 and 41 examples on an average for respective tasks still outperforms the best-performing random strategy, which establishes quality over quantity as a key factor. Therefore, contextual relevancy in prompts plays a significant role in improving the effectiveness of few-shot learning.

B. Qualitative analysis of code-related tasks

We randomly sampled the incorrect predictions made by CEDAR to determine the quality of the output and the reason for the incorrect matches. In the following subsections, we qualitatively assess the results for each of the tasks along with examples.

```
// stripping extra parenthesis
assertTrue(((list.size()) > 0)) // expected
assertTrue((list.size()) > 0) // generated

// matching args by ignoring order
assertEquals(reparsed, header) // expected
assertEquals(header, reparsed) // generated

// matching assertEquals(bool, ...) and assertTrue or
    assertFalse
assertEquals(true, result) // expected
assertTrue(result) // generated
```

Listing 1: Example of plausible matches in assertion generation.

Assertion Generation. CEDAR outperforms the task-specific model ATLAS [8] by a significant margin (as shown in Table III) as well as the fine-tuned T5 models [9], [10]. In our experiments, we introduced another metric called *plausible match* to evaluate the similarity between the generated assertion by CEDAR and the expected assertion. This means that we are able to capture cases, where the output might have extra parenthesis, the order of assertion method arguments are ignored, and when the expected assertion uses `assertTrue(...)` `assertFalse(...)` while the generated assertion uses `assertEquals(Boolean, ...)` (and vice versa). We show examples of these exact cases in Listing 1.

```
assertTrue(max.equals(answer)) // expected
assertEquals (answer, max) // generated

assertEquals((-1), comparison) // expected
assertTrue(comparison < 0) // generated

assertTrue((fileSizeAfter < fileSizeBefore)) // expected
assertTrue((fileSizeBefore > fileSizeAfter)) // generated
```

Listing 2: Example of semantically correct matches in assertion generation.

When sampling the incorrect exact or plausible match predictions, we found that not all the assertions generated were incorrect; many of the predictions were semantically correct, or sometimes better (i.e., more explicit and readable) than the expected assertions. Listing 2 shows some examples of these semantically correct assertions that we manually evaluated to be a match. These samples could not be detected with our accuracy metrics. As a result, the reported accuracy is a lower bound of the output generated by CEDAR. These predictions could potentially be captured using semantic-based code clone detection techniques. Currently, our evaluation approach handles Type 1–3 (Textual) code clone variants, therefore, incorporating a code clone detection technique that handles Type 4 (Functional) [46] code clones could potentially improve the results further.

```
// Buggy code (Unexpected 'this'.)
}).call(this);

}).call(this); // warning line

module.exports = OptionsExpression; // expected

server.listen(pathOrPort); // generated
```

Listing 3: Example of warning line completely replaced.

Program Repair. As shown in Table III, CEDAR outperformed the state-of-the-art task-specific learning based models such as COCONUT and SEQUENCER by a significant margin (342.05% and 188.94% respectively) and the fine-tuned model TFIX (4.91%) as well. This accuracy comparison is based on the exact string match, however, TFIX used another metric called *Error removal* to evaluate predictions. This metric is similar to our plausible match accuracy, where we count the prediction as correct if the warning line has been addressed by removing/replacing the error with the fix. Additionally, in the plausible match metric, we sanitize the output by removing extra spaces or parenthesis (similar to the assertion generation PM Accuracy) to match the expected output to the generated output.

```
// Buggy code (Unused variable.)
    feed,
    remoteTrackers
}) => {

remoteTrackers // warning line

// expected
    feed,
    updateTorrentTrackers,
    remoteTrackers
}) => {

// generated
    feed,
}) => {
```

Listing 4: Example of mismatch between error context and fix.

The dataset of TFIX [39] contained about 2-4 lines of context including the warning line along with ESLint error information. We randomly assessed a few incorrect predictions by CEDAR in the program repair task and found that many of the buggy code context contained incomplete or too little contextual information to generate a fix. As a matter of fact, many of the expected fixes were completely new lines,

which would have been impossible for a developer to generate without seeing enough relevant context of the warning line. Listing 3 and 4 show two samples demonstrating these cases.

In Listing 3, we notice that the expected fix is a completely new line that replaces the warning line. Although the error message provides an indication of the bug related to the warning line, the fix generated by the model and the expected fix are unrelated to both the error message and the warning line. In Listing 4, the predicted fix by CEDAR, which removes the line, seems plausible because of the error message which states that the warning line contains an unused variable. However, the expected fix introduces a new variable instead of addressing the warning line. Such an error is difficult to fix without the presence of more contextual statements surrounding the warning line.

C. Neural Coding Assistance Tool for Developers

CEDAR could be envisioned as a neural code assistance tool or an IDE plugin to facilitate code-related tasks for developers. A key feature of CEDAR is its generalizability and efficiency. The overall processing time for a single code suggestion, which includes the prompt building (200 milliseconds) and inference (2.20 seconds for assertion generation and 4.22 for program repair) is around 2.4 to 4.4 seconds, depending on the task. CEDAR can also be used for multiple code-related tasks and supports a variety of programming languages. Unlike the task-specific models [2], [43]–[45], [47], which require large training set and support only a single task, CEDAR requires only a handful of systematically curated code demonstrations to reason about a code query. Fine-tuned models have been proposed in the literature to support multiple tasks without creating a model from scratch. These models are essentially $\mathcal{LLM}$s which are tweaked to support multi-task learning using transfer learning. However, it still requires a training step with dataset tailored to the task, which can take a few hours depending on the dataset complexity, model architecture, and hyperparameters. Such models are yet to be fully incorporated in a development setting due to its high setup cost. In contrast, CEDAR provides better results within 2-4 seconds for each query with minimal effort, which is an important factor to aid developers in their tasks.

Task-specific or fine-tuned models are bound by the dataset that they have been trained on. Certain languages, such as JavaScript and Python, may contain syntactical or API changes in the latest versions. This may lead to outdated syntax usage or the exclusion of legacy API in the generated code. CEDAR dynamically curates the code demonstrations for each query, and as such, the demonstration pool could potentially be collected from a dynamic source, as opposed to a static pool of examples. As part of future work, we plan to augment our demonstration pool to support relevant GitHub commits or project-specific mining of $\mathcal{CD}$s in real-time.

V. THREATS TO VALIDITY

Potential data leakage. CODEX is trained on open-source code repositories, and it is possible that there is a data leakage, i.e., the testing dataset was included in its training. The training dataset used by CODEX is not publicly available. As such, there is a potential threat that the model's output is generated due to memorization. However, we observe that (a) the model does not perform well with zero-shot prompting, indicating it has not memorized the test data, (b) for both the random selection and CEDAR, we use the exact same model and test dataset, showing that our retrieval-based prompt generation is effective in eliciting better responses and improving the performance of the $\mathcal{LLM}$. We see a similar trend for both tasks, assertion generation, and program repair.

Evaluation metrics. In this paper, we relied on evaluation metrics such as *exact match* that is computed by strict string equality matching. However, as observed in Section IV-B, CEDAR could provide code suggestions that do not exactly match with the developer written code, yet are functionally equivalent. As a result, our comparison presents the lowest bound of effectiveness from CEDAR. Nevertheless, exact accuracy match is widely adopted as an evaluation metric by other learning-based source code processing techniques [43]–[45], [48].

Large language models on source code. In this study, we evaluated the effectiveness of few-shot learning using CODEX [18]. There are other large language models on source code namely INCODER [49], CODET5 [50], among others [16], [17], [51]–[57]. An empirical controlled study such as REPTORY [42] addressing all the factors such as model architecture, hyperparameters, and vocabulary size that could affect the accuracy of code generation could certainly be useful; however, such a study is not the goal of this paper. In this work, we investigate whether few-shot learning could be leveraged for two different source code processing tasks.

Generalization on different source code-related tasks. CEDAR is applied to two source code-related tasks in this study. The results indicate that CEDAR is effective in assertion generation and fixing bugs and could be competitive with task-specific models. Large language models such as CODEX were not specifically trained for the task of assertion generation or learning to apply generic code repair. Despite this, CEDAR yielded competitive results using few-shot learning. This encouraging finding implies that few-shot learning could be equally effective and competitive with other task-specific learning-based models.

Reproducibility. We have made CEDAR's implementation publicly available [29]. We have also included instructions for reproducing our experimental results.

VI. RELATED WORK

Foundation models in deep learning. Many researchers have fined-tuned general-purpose pre-trained models to a specific downstream task [58], [59]. Prominent examples of pre-trained language models are ELMo [60], BERT [12], RoBERTA [61], and T5 [13]. A more recent improvement is the introduction of GPT-3, which is an autoregressive $\mathcal{LLM}$ that is pre-trained to predict the next word in a sequence. These

large foundation models are found to be effective for prompt-based learning [15] and can be generalized to a wide range of tasks, such as text summarization, question answering, and dialogue (e.g., ChatGPT[8]).

Applications of language models on source code. A number of pre-trained $\mathcal{LLM}$s for code generation have been proposed [16]–[18]. Recent studies examine the impact of code completion with $\mathcal{LLM}$s on software development [62], [63], and developer productivity [19]–[21]. There is a growing interest in determining whether generated code can introduce security vulnerabilities [38], [64], reducing unauthorised neural code learning [65], [66] from open-source code, and how to use $\mathcal{LLM}$s to repair vulnerabilities [67], [68]. Deep reinforcement learning is used in CodeRL [69] to improve the performance of $\mathcal{LLM}$s. There have been efforts to exploit feedback from generated tests [25], [26] and incorporate user interaction [70] to enhance the code quality generated from a language model. There are also applications of $\mathcal{LLM}$ on program synthesis [71], [72]. Compared to these works, CEDAR examines how $\mathcal{LLM}$s could be employed in a few-shot setting with retrieval-augmented techniques for different tasks.

CODEX and PolyCoder have been used to patch programs [27], [28] in the Quixbugs [73] dataset, which is perhaps the most similar existing work. However, they employed (a) simpler problem settings instead of a more generic program repair that represents real bugs, (b) focused on a single task, and (c) did not provide means to create an effective prompt for prompt-based learning. In this work, we focus on developing a prompt with retrieval-based techniques that could elicit a better response from the $\mathcal{LLM}$.

Task-specific models on source code processing. There is a large body of literature [74]–[78] that applies learning based task-specific models to different source code processing tasks. Different task-specific learning-based program repair techniques [1]–[4], [6], [42]–[45], [79]–[82] have been proposed that employ different neural architectures and a varied source code representation. Similarly, task-specific neural models have shown promise in generating test assertions [8], [83], [84]. Unlike these task-specific models, CEDAR demonstrates how $\mathcal{LLM}$ could be employed with careful demonstration selection to achieve competitive results, without building and training a neural network. CEDAR is not tied to a specific task or programming language and could be generalized to multiple tasks and multiple programming languages.

Pre-trained and fine-tuning based learning on source code. Following the success of large pre-trained models and fine-tuned models for many NLP tasks, these techniques are also explored in the domain of source code [52], [57], [85], [86]. Pre-trained models are explored on bug fixing [10], [87]–[89], assertion generation [10], [90], code review [91]–[93]. In contrast, we show that by leveraging retrieval-based demonstration selection for few-show learning, CEDAR can achieve competitive results with pre-trained models.

[8]https://chat.openai.com/chat

VII. CONCLUSION

The design of an effective prompt is pivotal for few-shot learning. However, the steps to create an effective prompt are sparsely explored for source code-related tasks. We present CEDAR, the first work to apply an automated retrieval-based demonstration selection strategy in code-related few-shot learning. Our evaluation of CEDAR shows that it can generate exact matches with 76% and 52% for test assertion generation and program repair tasks, respectively. Our technique outperforms the best-performing state-of-the-art models for assertion generation and program repair by up to 11% and 5%, respectively. As part of our future work, we plan to extend CEDAR to more code-related tasks. In addition, we plan to incorporate our approach as an IDE plugin to assist developers with their software engineering tasks. ChatGPT, a recent large language model for dialogue, has shown promise in various code generation tasks. However, it is also known to produce incorrect or nonsensical output, known as hallucinations. In future work, we plan to investigate whether a retrieval-based approach to select in-context examples can mitigate these limitations.

REFERENCES

[1] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," in *37th International Conference on Machine Learning*. JMLR.org, 2020.

[2] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[3] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 505–509.

[4] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021, p. 341–353.

[5] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *44th International Conference on Software Engineering*. ACM, 2022, p. 511–523.

[6] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *44th International Conference on Software Engineering*. ACM, 2022, p. 1506–1518.

[7] H. Hata, E. Shihab, and G. Neubig, "Learning to generate corrective patches using neural machine translation," *arXiv:1812.07170*, 2019.

[8] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *ACM/IEEE 42nd International Conference on Software Engineering*. ACM, 2020, p. 1398–1409.

[9] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 336–347.

[10] A. Mastropaolo, N. Cooper, D. N. Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using transfer learning for code-related tasks," *IEEE Transactions on Software Engineering*, pp. 1–20, 2022.

[11] X.-W. Chen and X. Lin, "Big data deep learning: Challenges and perspectives," *IEEE Access*, vol. 2, pp. 514–525, 2014.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*. ACL, 2019, pp. 4171–4186.

[13] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, 2020.

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

[15] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *arXiv:2107.13586*, 2021.

[16] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *6th ACM SIGPLAN International Symposium on Machine Programming*. ACM, 2022, p. 1–10.

[17] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv:2203.13474*, 2022.

[18] "Codex model," *https://beta.openai.com/docs/models/codex-series-private-beta*, 2022.

[19] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*. ACM, 2022.

[20] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *6th ACM SIGPLAN International Symposium on Machine Programming*. ACM, 2022, p. 21–29.

[21] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, Z. Ming *et al.*, "Github copilot ai pair programmer: Asset or liability?" *arXiv:2206.15331*, 2022.

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

[23] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub Copilot's code suggestions," in *19th ACM International Conference on Mining Software Repositories (MSR)*, 2022, pp. 1–5.

[24] S. Lertbanjongam, B. Chinthanet, T. Ishio, R. G. Kula, P. Leelaprute, B. Manaskasemsak, A. Rungsawang, and K. Matsumoto, "An empirical evaluation of competitive programming ai: A case study of alphacode," *arXiv:2208.08603*, 2022.

[25] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago *et al.*, "Competition-level code generation with alphacode," *arXiv:2203.07814*, 2022.

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

[27] J. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs?: An evaluation on quixbugs," in *International Workshop on Automated Program Repair (APR)*. IEEE, 2022, pp. 69–75.

[28] S. D. Kolak, R. Martins, C. Le Goues, and V. J. Hellendoorn, "Patch generation with language models: Feasibility and scaling behavior," in *Deep Learning for Code Workshop*, 2022.

[29] "CEDAR," https://github.com/prompt-learning/cedar, 2022.

[30] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[31] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Found. Trends Inf. Retr.*, vol. 3, no. 4, p. 333–389, 2009.

[32] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in *Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. ACL, 2019, pp. 3982–3992.

[33] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv:1909.09436*, 2019.

[34] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*. Citeseer, 2003, pp. 29–48.

[35] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–7.

[36] V. Liu and L. B. Chilton, "Design guidelines for prompt engineering text-to-image generative models," in *CHI Conference on Human Factors in Computing Systems*. ACM, 2022.

[37] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, "Improving automatically generated code from codex via automated program repair," *arXiv:2205.10583*, 2022.

[38] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *43rd IEEE Symposium on Security and Privacy, SP 2022, May 22-26, 2022*. IEEE, 2022, pp. 754–768.

[39] B. Berabi, J. He, V. Raychev, and M. T. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *ICML*, 2021, pp. 780–791.

[40] J. Gong, X. Chen, X. Qiu, and X. Huang, "End-to-end neural sentence ordering using pointer network," *CoRR*, 2016.

[41] Y. Ding, B. Ray, P. Devanbu, and V. J. Hellendoorn, "Patching as translation: The data and the metaphor," in *35th International Conference on Automated Software Engineering*. ACM, 2020, p. 275–286.

[42] M. Namavar, N. Nashid, and A. Mesbah, "A controlled experiment of different code representations for learning-based bug repair," *Empirical Software Engineering Journal*, 2022.

[43] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations*, 2020.

[44] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[45] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2020, p. 101–114.

[46] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, pp. 470–495, 2009.

[47] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *42nd International Conference on Software Engineering*. ACM, 2020, p. 602–614.

[48] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *ACM on Programming Languages*, vol. 2, no. OOPSLA, 2018.

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

[50] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *2021 Conference on Empirical Methods in Natural Language Processing*. ACL, 2021, pp. 8696–8708.

[51] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. ACL, 2020, pp. 1536–1547.

[52] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," in *60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. ACL, 2022, pp. 7212–7225.

[53] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, ICLR*. OpenReview.net, 2021.

[54] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv:2102.04664*, 2021.

[55] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. Gonzalez, and I. Stoica, "Contrastive code representation learning," in *2021 Conference on Empirical Methods in Natural Language Processing*. ACL, 2021, pp. 5954–5971.

[56] L. Phan, H. Tran, D. Le, H. Nguyen, J. Annibal, A. Peltekian, and Y. Ye, "CoTexT: Multi-task learning with code-text transformer," in *1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, 2021, pp. 40–47.

[57] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. ACL, 2021, pp. 2655–2668.

[58] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. De Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, "Parameter-efficient trans-

fer learning for NLP," in *36th International Conference on Machine Learning*, vol. 97. PMLR, 2019, pp. 2790–2799.

[59] Z. Lin, A. Madotto, and P. Fung, "Exploring versatile generative language model via parameter-efficient transfer learning," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. ACL, 2020, pp. 441–459.

[60] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*. ACL, 2018, pp. 2227–2237.

[61] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv:1907.11692*, 2019.

[62] A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn, "What is it like to program with artificial intelligence?" *arXiv:2208.06213*, 2022.

[63] S. Imai, "Is github copilot a substitute for human pair-programming? an empirical study," in *44th International Conference on Software Engineering (ICSE-Companion)*, 2022, pp. 319–321.

[64] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "An empirical cybersecurity evaluation of github copilot's code contributions," *arXiv e-prints*, pp. arXiv–2108, 2021.

[65] S. Fu, F. He, Y. Liu, L. Shen, and D. Tao, "Robust unlearnable examples: Protecting data privacy against adversarial learning," in *International Conference on Learning Representations*, 2022.

[66] Z. Sun, X. Du, F. Song, M. Ni, and L. Li, "Coprotector: Protect open-source code against unauthorized training usage with data poisoning," in *ACM Web Conference 2022*. ACM, 2022, p. 652–660.

[67] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Can openai codex and other large language models help us fix security bugs?" *arXiv:2112.02125*, 2021.

[68] A. Grishina, "Enabling automatic repair of source code vulnerabilities using data-driven methods," in *44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 275–277.

[69] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *arXiv:2207.01780*, 2022.

[70] S. K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang, and J. Gao, "Interactive code generation via test-driven user-intent formalization," *arXiv:2208.05950*, 2022.

[71] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," in *44th International Conference on Software Engineering*. ACM, 2022, p. 1219–1231.

[72] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv:2108.07732*, 2021.

[73] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 2017, p. 55–56.

[74] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk, "A systematic literature review on the use of deep learning in software engineering research," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, 2022.

[75] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *ACM Comput. Surv.*, 2021.

[76] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, 2018.

[77] T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Comput. Surv.*, vol. 53, no. 3, 2020.

[78] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, and F. Sarro, "A survey on machine learning techniques for source code analysis," *arXiv:2110.09610*, 2021.

[79] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology*, 2019.

[80] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, "Deepdelta: Learning to repair compilation errors," in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, p. 925–936.

[81] N. Noor, M. Sintaha, and A. Mesbah, "Embedding context as code dependencies for neural program repair," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2023, p. 12 pages.

[82] M. Sintaha, N. Noor, and A. Mesbah, "Dual slicing-based context for learning bug fixes," *Transactions on Software Engineering and Methodology (TOSEM)*, p. 27 pages, 2023.

[83] R. White and J. Krinke, "Reassert: Deep learning for assert generation," *arXiv:2011.09784*, 2020.

[84] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 2130–2141.

[85] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "Coditt5: Pretraining for source code and natural language editing," *arXiv:2208.05446*, 2022.

[86] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Conference on Empirical Methods in Natural Language Processing*. ACL, 2021, pp. 8696–8708.

[87] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *36th International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 443–455.

[88] M. Lajko, V. Csuvik, and L. Vidacs, "Towards javascript program repair with generative pre-trained transformer (gpt-2)," in *International Workshop on Automated Program Repair (APR)*. IEEE, 2022, pp. 61–68.

[89] M. Lajkó, D. Horváth, V. Csuvik, and L. Vidács, "Fine-tuning gpt-2 to patch programs, is it worth it?" in *Computational Science and Its Applications – ICCSA 2022 Workshops*. Springer-Verlag, 2022, p. 79–91.

[90] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv:2009.05617*, 2020.

[91] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *44th International Conference on Software Engineering*. ACM, 2022, p. 2291–2302.

[92] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," in *3rd ACM/IEEE International Conference on Automation of Software Test*. ACM, 2022, p. 54–64.

[93] L. Li, L. Yang, H. Jiang, J. Yan, T. Luo, Z. Hua, G. Liang, and C. Zuo, "Auger: Automatically generating review comments with pre-training models," *arXiv:2208.08014*, 2022.