Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning

Mingyang Geng
gengmingyang13@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Shangwen Wang
wangshangwen13@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Dezun Dong
dong@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Haotian Wang
wanghaotian13@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Ge Li
lige@pku.edu.cn
Key Lab of High Confidence Software Technology, Peking University
Beijing, China

Zhi Jin
zhijin@pku.edu.cn
Key Lab of High Confidence Software Technology, Peking University
Beijing, China

Xiaoguang Mao
xgmao@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Xiangke Liao
xkliao@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

ABSTRACT

Code comment generation aims at generating natural language descriptions for a code snippet to facilitate developers' program comprehension activities. Despite being studied for a long time, a bottleneck for existing approaches is that given a code snippet, they can only generate one comment while developers usually need to know information from diverse perspectives such as what is the functionality of this code snippet and how to use it. To tackle this limitation, this study empirically investigates the feasibility of utilizing large language models (LLMs) to generate comments that can fulfill developers' diverse intents. Our intuition is based on the facts that (1) the code and its pairwise comment are used during the pre-training process of LLMs to build the semantic connection between the natural language and programming language, and (2) comments in the real-world projects, which are collected for the pre-training, usually contain different developers' intents. We thus postulate that the LLMs can already understand the code from different perspectives after the pre-training. Indeed, experiments on two large-scale datasets demonstrate the rationale of our insights: by adopting the in-context learning paradigm and giving adequate prompts to the LLM (e.g., providing it with ten or more examples), the LLM can significantly outperform a state-of-the-art supervised learning approach on generating comments with multiple intents. Results also show that customized strategies for constructing the prompts and post-processing strategies for reranking the results can both boost the LLM's performances, which shed light on future research directions for using LLMs to achieve comment generation.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Maintaining software**; *Software evolution.*

KEYWORDS

Code Summarization, Large Language Model, In-Context Learning

ACM Reference Format:
Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning . In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/xxxxxxx.xxxxxxx

1 INTRODUCTION

Code comment generation (a.k.a. code summarization) targets the ambition of automatically generating a concise and fluent natural language description of source code. It is considered as a critical

† Shangwen Wang and Dezun Dong are the corresponding authors.
Shangwen Wang and Xiaoguang Mao are with the Key Laboratory of Software Engineering for Complex Systems.

This work is supported by the National Key Research and Development Program Project "Heterogeneous Computing Fusion of Cross-Domain Resources" No.2022YFB4501702.

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/xxxxxxx.xxxxxxx

way to facilitate program comprehension since developers usually forget or have no time to write such comments, and thus holds the potential of boosting software development and maintenance activities. During the years, a number of studies have been devoted into advancing the state of the art in this domain [4, 28, 32]. For instance, information retrieval techniques, which focus on extracting some important tokens from the code, are used in the early stage [25, 58], followed by some recent works applying advanced deep learning techniques on this task, such as the neural machine translation (NMT) model [5, 28].

Despite the achieved tremendous progress in this domain, one critical problem that downgrades the practicality of existing code comment generation approaches is that they can only generate comments describing one aspect of a given code snippet (and thus a one-to-one mapping). In practice, however, developers often write comments with diverse intents to summarize the code from different perspectives (e.g., what is the main functionality of the code and how can we use it). For instance, Zhai *et al.* [75] manually checked comments from real-world projects and identified six categories of intents hidden in the comments (as shown in Table 1). Mu *et al.* [47] did the statistics of top-starred Java projects on GitHub and found that around 67% of the methods contain more than one intent in their comments. The above observations indicate that what developers really need is a one-to-many mapping (i.e., generating multiple comments that summarize the given code from different perspectives), which is referred to as the **multi-intent comment generation** task in this paper.

To tackle the aforementioned task, Mu *et al.* [47] proposed an approach named DOME, where an attention mechanism is used to focus on different parts of code for different intents. However, DOME is based on supervised learning, which limits its effectiveness due to the amount of data available for training. To address the data shortage problem, we propose to borrow the weapon of large language models (LLMs) [8], which are pre-trained on a data corpus of a very large scale in the self-supervised manner and have captured a lot of domain knowledge during such a process. The application of LLMs to the multi-intent comment generation task is motivated by two factors. Firstly, LLMs designed for the code domain are typically pre-trained using code and its associated pairwise comments to establish semantic connections between programming language and natural language [19, 67]. For example, the commonly used pre-training task, masked language modeling [15, 19, 24], is specifically intended to align programming language and natural language representations. Secondly, existing research has shown that code comments from real-world projects, which form the training corpus for LLMs, often contain multiple intents [47]. As a result, during pre-training, LLMs are trained to understand code from various perspectives, potentially allowing them to capture different code semantics. Thus, by fully exploiting the capabilities of pre-trained LLMs, we can achieve good performances on the multi-intent comment generation task.

Recently, in-context learning has been shown to be an effective way to exploit the domain knowledge hidden in the LLMs [8, 11, 48, 60], since the format of the inputs to the model can be consistent to that during the pre-training process. Inspired by these studies, we aim to investigate the feasibility of addressing the multi-intent comment generation task with in-context learning. Generally, in-context learning requires to provide a prompt to the model which is composed of a natural language instruction describing the detailed information of the task, (optionally) a handful of examples demonstrating how the task could be well done, and a query that is required to be addressed. Therefore, a follow-up question is that, with in-context learning, how can we obtain better results from the LLMs (e.g., if it is possible by designing prompts that can guide the LLMs towards the desired output). To provide empirical evidence on the aforementioned questions, we investigate the following aspects in this study: (a) Can the LLMs support to accomplish the multi-intent comment generation task using the in-context learning paradigm? (b) Can we improve the performance of the LLMs by designing customized demonstration selection strategies? and (c) Can we improve the performance of the LLMs by designing customized strategies to post-process the obtained results?

To that end, we perform extensive experiments on two large-scale Java language datasets, which are Funcom [36] and TLC [30]. We use the OpenAI Codex model as the representative LLM because of its superior performances on several code intelligence tasks [48, 54]. Our study makes the following important findings:

- F1: When the LLM is not adequately prompted (i.e., the number of demonstration examples is less than 10), the potential of the LLMs may not be fully exploited and the effectiveness is sub-optimal compared with that of the state-of-the-art supervised learning approach, DOME; in contrast, when the number of demonstration examples reaches ten, the LLM is more adequately prompted and its performance exceeds that of the DOME approach.
- F2: Demonstration selection strategies can help LLMs better understand the on-going task and thus enhance their effectiveness to a large extent: when the number of examples is ten and the code snippets which are most similar to the target one are used as the demonstration examples, the BLEU values of Codex can be increased by 97% and 131% on the two datasets, respectively, compared with random selection.
- F3: The outputs of LLMs can be reranked based on simple heuristics to achieve further performance enhancement: compared with the experiment setting mentioned above, the BLEU values of Codex can be improved by 9.9% and 9.6%, respectively, on the two datasets if the comment of the corpus code which is similar to the target one can be used for guiding the output reranking.

Our study demonstrates that LLMs can potentially be applied to multi-intent comment generation since it builds strong performance baselines on this task, which should be considered by tool designers in future evaluation. Further implications include that devising better demonstration selection strategies as well as reranking strategies are both promising research directions.

2 BACKGROUND AND RELATED WORKS

2.1 Comment Generation

Automatic code comment generation, which aims at summarizing code with concise natural language descriptions, is a critical task to

Table 1: The intent taxonomy of code comments [12, 75].

Category	Definition	Example
What	Describes the functionality of a method	"Checks if the tile units at the given coordinates are displayed on the screen"
Why	Explains the reason why a method is provided or the design rationale of the method	"Prepare to start making calls to the currently registered callbacks"
How-to-use	Describes the usage or the expected set-up of using a method	"Code executed before the intercepted method"
How-it-is-done	Describes the implementation details of a method	"Ends the current table, discards it and pops the top of the stack to be the new current table"
Property	Asserts properties of a method including pre-conditions or post-conditions of a method	"Returns true if the value is a string that matches a regex"
Others	Unspecified or ambiguous comments	"I am done with the model, free the resources "

facilitate program comprehension. Many approaches have been proposed to construct a set of manually-defined complex rules, based on which comments can be generated following specific templates [25, 27]. With the recent advancement of the deep learning, a hot line of researches has suggested applying deep neural networks (DNNs) to this task. By modeling code as the input and comment as the output, such neural comment generation (NCG) approaches automatically learn a function, which is usually a DNN model such as the neural machine translation model, that can produce the output given the input. Such a DNN model is learned using existing large-scale code-comment pairwise data. CodeNN [32] is an early attempt in this direction that uses only code token sequences, followed by various approaches that utilize the AST structure [4, 28, 29], API knowledge [30], type information [9], global context [7, 26, 66], reinforcement learning [22, 62, 65], multi-task learning [72], dual learning [68, 73], pre-trained language models [19, 21, 67], and hybrid approaches [69, 77]. In addition, a number of works also focus on generating latest and informative comments based on outdated comments (a.k.a comment updating) [39, 40].

The aforementioned approaches, however, can only generate comments describing one aspect of a given code snippet, which limits their practicality since developers usually express multiple intents when commenting the code [12, 47, 75]. That is to say, merely generating comments describing a specific aspect of a code snippet (e.g., the functionality of the code) may not meet the developers' requirements about comprehensively summarizing the code (e.g., how to use the code). Specifically, according to the previous studies [12, 47, 75], developers usually have six categories of intents when commenting the code, i.e., *what*, *why*, *how-to-use*, *how-it-is-done*, *property*, and *others*. In Table 1, we list the detailed definition and example for each category. The fact that developers usually express multiple intents in the comments cast threats to the practicality of existing single-intent comment generation techniques. To address this challenge, Mu *et al.* [47] propose a developer-intent driven code comment generation approach DOME, which aims to produce a comment coherent with a given intent. It works by leveraging the attention mechanism guided by the given intent to focus on the most relevant information from the code. To our best knowledge, DOME is so far the only existing technique that can generate diverse comments given different categories of intents.

2.2 Large Language Models

Large language models (LLMs) trained on massive corpora of unlabelled data have been shown to perform well on a wide range of tasks, including natural language generation, semantic parsing, and code generation [8, 16, 56]. The reason for their strong power can be concluded as they do not need task-specific training data and can be pre-trained on tremendous in-the-wild data in a self-supervised manner (a.k.a. pre-training), so that sufficient domain knowledge can be captured. The pioneer of this direction, the GPT model [55], was firstly proposed in 2018. After that, a number of follow-up studies continuously enhance the state-of-the-art performances by adjusting the model architecture (e.g., BERT [16]) or increasing the total amount of parameters (e.g., GPT-3 [8]).

Codex, released by OpenAI, is an LLM based on the GPT-3 architecture (i.e., contains a Transformer-based decoder) [2]. It powers GitHub Copilot, an AI pair programmer that generates the whole code function given a natural language description. Codex is trained on a massive code corpus containing code-comment pairwise examples from many programming languages including Python, JavaScript, C/C++, Go, Perl, PHP, Ruby, Swift, TypeScript, SQL and Shell. Similar to GPT-3, Codex adopts the auto-regressive manner during the pre-training, in which given a sequence of code/-comment tokens, it is trained to predict the next token and the predicted token is recursively used as the input for the next prediction until the end of the sequence. In our study, we use Codex as the representative LLM since it is a popular LLM in the software engineering domain and has been widely studied in the literature [10, 14, 18, 34, 49, 52, 54, 78].

2.3 In-Context Learning

Previously, to apply a pre-trained model on downstream tasks, users need to further train it on the labelled data of downstream tasks in a supervised manner (a.k.a. fine-tuning) [16, 43]. Compared with training a model from scratch, this paradigm can exploit the knowledge learned by the pre-trained model and thus achieve better performance [38, 44]. Such a paradigm, however, mainly has two limitations. First, the data used for pre-training and fine-tuning are in different formats, which makes the learned knowledge of the model cannot be fully leveraged during the fine-tuning process [63]. Second, the fine-tuning process can be extremely time-consuming and resource-intensive, especially when it comes to large language models which usually contain billions of parameters [8].

To address the aforementioned limitations, **in-context learning** is recently proposed and quickly becomes a research hotspot after that [8]. Such a paradigm denotes that a few training examples and/or task descriptions together with a developer query that needs to be answered are sent into a large language model to produce a response of the query, without any parameter update. Basically, in the in-context learning paradigm, a prompt needs to be provided

for a code intelligence task, e.g., code summarization. By employing prompts, large language models are shown to be effective in different tasks that the model is not explicitly trained on, without the need of task-specific data [63].

Generally, the rationale of the in-context learning is that since large language models have been trained on corpora of a very large scale, they must have absorbed much domain knowledge and are thus expected to generalize well to unseen tasks without fine-tuning [8]. Our study shares a similar motivation. Specifically, considering that (1) large language models, e.g., Codex, are trained on a large-scale corpus containing tremendous amount of code-comment pairwise data from real-world, and (2) the real-world comments usually contain different categories of developers' intents, we postulate that the large language models are capable of understanding the code from different perspectives and thus hold the potential to generate comments with diverse intents given a code snippet. By using the in-context learning, such potentials of LLMs can be exploited.

3 STUDY DESIGN

3.1 Research Questions

The goal of our study is to investigate the effectiveness of large language models on multi-intent comment generation using the in-context learning paradigm. To this end, we propose to answer the following research questions.

- **RQ1: What is the effectiveness of Codex on multi-intent comment generation using zero-shot, one-shot, and few-shot learning?** As the very first RQ, we aim at investigating the feasibility of addressing the multi-intent comment generation problem with in-context learning. Specifically, we do not use any customized design and only select code demonstrations randomly. Our target is to investigate how effective is the vanilla in-context learning compared with the state-of-the-art DOME approach. The results can also reflect to what extent the number of demonstrations (i.e., zero-shot, one-shot, and few-shot) affect the effectiveness.
- **RQ2: Can the effectiveness be improved by retrieval-based demonstration selections?** Some recent works have demonstrated that the quality of the demonstrations in the prompt can significantly impact the effectiveness of in-context learning [45, 48, 60]. Inspired by these studies, we propose to investigate whether customized demonstration selection approaches can help improve the model's performance. Specifically, to answer this question, we design two retrieval-based approaches that select code examples similar to the code specified in the developer query, and evaluate their effectiveness.
- **RQ3: Can the effectiveness be improved by reranking strategies?** A large language model experiences a sampling process to obtain the outputs [11, 49, 61, 78]. That is to say, a developer can obtain different results from the model for the identical input. In this RQ, we further investigate the feasibility of boosting the model's performance in a post-processing manner: by first obtaining a number of results and then reranking them through a pre-defined heuristic. Answering such a question can provide guidance for applying the approach in practice: it can make us clear about to what extent we can obtain more qualified results by sampling multiple outputs.

3.2 The Prompt Template for Multi-Intent Comment Generation

Formally, a prompt is defined as $P = \{x_{\text{test}} + \mathcal{CD} + \mathcal{NL}\}$, where $\mathcal{NL}$ is a natural language template, $\mathcal{CD} = \{(x_i, y_i)\}_{i=1}^{n}$ is a set of code demonstrations composed by input code sequence (x_i) and desired output sequence (y_i), and x_{test} is a developer query to be inferred. Specifically, if $i = 0$ which means there is no code demonstration, the setting is known as *zero-shot learning*; if $i = 1$ which means there is only one code demonstration, the setting is known as *one-shot learning*; and *few-shot learning* means there is a number of code demonstrations. Also, there is a constraint that $\text{size}(\mathcal{P}) \leq$ context-window, which means the prompt should fit within the context window limit of the language model. [1]

Figure 1 illustrates a prompt template for the multi-intent comment generation task. The input prompt contains two sections: the code demonstrations $\mathcal{CD}$ and the query x_{test} . The natural language instructions are denoted by the lines starting with the special token "#". In the first line of the prompt, we first tell the model the specific programming language it is working on (e.g., Java) and then the desired intent of the comment, as highlighted in the red, is specified by following the definitions shown in Table 1. In concrete, for the "what" intent, we add the prompt "Describe the functionality of the method"; for the "why" intent, we add the prompt "Explain the reason why the method is provided or the design rationale of the method"; for the "how-to-use" intent, we add the prompt "Describe the usage or the expected set-up of using the method"; for the "how-it-is-done" intent, we add the prompt "Describe the implementation details of the method"; for the "property" intent, we add the prompt "Assert properties of the method including pre-conditions pr post-conditions of the method". In this example, the illustrated prompt aims at generating a comment that fulfills the "what" intent. The first line is then followed by a number of code demonstrations that can help the LLM to understand the expected behavior and each demonstration contains one code snippet and one corresponding comment within the desired intent category. Each code demonstration is separated with a delimiter "###". Finally, the model is asked to output the desired comment of the query code, which is shown at the bottom of the figure.

3.3 Demonstration Retrieval

Note that the code demonstrations used in RQ1 are randomly selected from a corpus. While in RQ2, we aim at investigating whether customized demonstration selection can enhance the effectiveness. Therefore, we design two strategies to retrieve similar code demonstration examples from the corpus whose comments' intents belong to the desired category. The rationale is that a few demonstrations that are similar to the target one may help the model better understand the desired behaviour [45, 48, 60]. The whole process of such a paradigm is shown in Figure 2: given a code snippet and the required intent category, we select code examples that are similar to the target one and use the retrieved code together with their

[1] 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

```
1  ---------- begin of prompt ----------
2  # You are an expert Java programmer, please describe the functionality of the method:
3                                                          Natural Language Instruction
4  # Code
5  public Representation readOrListResourceSet() throws NotFoundException{
6          String resourceSetId = getResourceSetId();
7          if (resourceSetId == null || resourceSetId.isEmpty()) {
8              return listResourceSets();
9          } else {
10             return readResourceSet(resourceSetId);
11         }
12 }
13 # Summary: Reads the requested resource set description or queries
14             all the client's (resource server's) resource sets.
15 ###                                                    Code Demonstration 1
16
17                                  • • •                  Code Demonstrations
18
19 # Code
20 protected boolean isIrreflexive(NodeSet set) throws SiteException {
21         SiteNode[] resources = set.getNodes();
22         boolean isIrreflexive = true;
23         for (int i = 0; i < resources.length; i++) {
24             if (requires(resources[i], resources[i])) {
25                 isIrreflexive = false;
26             }
27         }
28         return isIrreflexive;
29 }
30 # Summary:                                             Test Code Query
31 ---------- end of prompt ----------
32 # Groundtruth: Checks if the dependence relation is irreflexive.
```

Figure 1: Multi-intent code summarization prompt template.

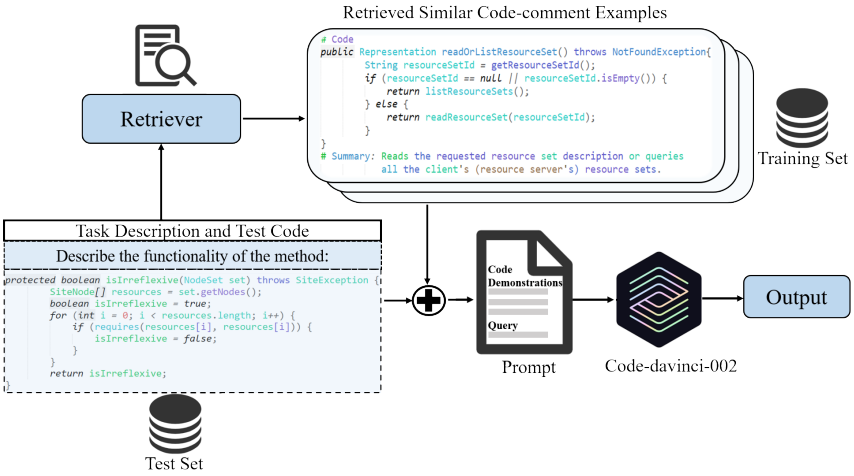

Figure 2: Overview of our in-context learning-based code summarization.

comments to construct a prompt whose template is shown in Figure 1. The prompt is used to query the model and obtain the results. We next introduce the two retrieval strategies in detail.

- **Token-based:** The most commonly-used strategy to identify similar code is focusing on the overlap with respect to the code tokens [23, 33, 76]. Inspired by these studies, our first retrieval strategy is also based on the token level information, i.e., to rank the code snippets from the code corpus based on their token similarities with the target code. In concrete, we first pre-process the target code snippet and the code snippets in the retrieved code corpus by removing the keywords defined in the programming language (i.e., Java in our study). The behind intuition is that such frequently-used tokens may bring side effects to the similarity calculation because a large number of code snippets would contain them, inspired by the recent study [17]. Then, we further split identifiers into sub-tokens to adequately leverage the semantic information hidden in the identifier names [53]. Specifically, such a process is achieved by utilizing the camel cases and the underscore naming convention of Java language. Finally, we convert all the sub-tokens to lower case. As for the token-based similarity between a candidate code snippet and the target code (s_{token}), we exploit the Jaccard Coefficient [50] for the calculation, which is defined as follows: $s_{\text{token}} = \frac{|\text{tokens}_{\text{target}} \cap \text{tokens}_{\text{candidate}}|}{|\text{tokens}_{\text{target}} \cup \text{tokens}_{\text{candidate}}|}$ where $tokens_{target}$ denotes the sub-token list of the target code and $tokens_{candidate}$ denotes the sub-token list of the candidate code. The value of s_{token} ranges from 0 to 1. A larger value of s_{token} indicates a higher similarity between the target code and the candidate code from the retrieved set.
- **Semantic-based:** Recent studies in the clone detection domain have also revealed that beyond the lexical level code token similarity, understanding the code semantics is also important for finding similar code [64, 74]. Therefore, our second strategy relies

on the code semantics to retrieve similar code snippets. Specifically, we exploit the pre-trained sentence transformer model [57], which has been demonstrated to be capable of accurately capturing the semantics of code snippets by a recent study [48], to encode the code snippets as vectors which contain the corresponding semantic information. [2] The cosine similarity is exploited to retrieve the similar candidate code snippets whose vectors are close to that of the target code snippet in the vector space.

3.4 Reranking Strategy

To rerank the generated comments, our intuition is that similar code snippets usually share similar comments, which is a common sense in the literature [37, 69–71]. Therefore, our strategy is to rerank the generated comments based on their similarities to the comment of the code snippet in the retrieval corpus that is similar to the target code. Specifically, we use the comment of the code snippet that is the most similar to the target code as the reference and also calculate comment similarities from two perspectives, i.e., the token-based and the semantic-based. For the **token-based** strategy, we focus on the token level information, since tokens in the comments are usually natural language words that have clear semantics. For the **semantic-based**, we exploit again the pre-trained sentence transformer model [57], embed the whole comment into a semantic vector, and calculate the cosine similarities.

3.5 Datasets

In this study, we use the multi-intent comment generation datasets released by the previous study [47] as our evaluation datasets. In concrete, we use two datasets of Java programming language, i.e., the Funcom [36] and TLC [30] datasets, both of which are the most widely-used datasets for the code comment generation task [3, 13, 20, 35, 77]. Funcom contains 2.1M code-comment pairs from 29K Java projects, which were collected by Lopes *et al.* [1] and further cleaned by LeClair *et al.* [36]. TLC contains 87,136 code-comment pairs collected from more than 9K Java projects created from 2015 to 2016 with at least 20 stars. The intent categories of each comment in these two datasets are labelled by Mu *et al.* [47]: they first invited five domain experts to manually label the intents of 8K code snippets and then fine-tuned the CodeBERT model [19] on the labelled data, which was served as a classifier. Results show that the fine-tuned model can achieve an F1-score of around 90%, which is a relatively high value. Finally, the authors applied the fine-tuned model to predict the intent category of each comment in the datasets and used the prediction results as the ground-truth labels. Since manual labelling of such large-scale datasets would be infeasible, we reuse their provided results in our study. Also, the training/validation/test partition of the datasets is fixed and the statistics of these two datasets are shown in Table 2. Note that in the table, we do not show the statistics of the validation sets of the two datasets. This is because our approach does not need to train a model. In contrast, we only retrieve code examples from the training sets (by following Mu *et al.* [47]) with or without customized strategies and evaluate the effectiveness on the test sets. Therefore, the validation sets are not used in this study. Following existing studies [12, 47], we also exclude comments from the *others* intent category in our evaluation because these comments are considered as unspecified or ambiguous.

Table 2: The statistics of our evaluation datasets.

Dataset	Funcom		TLC	
	Train	Test	Train	Test
What	685,992	44,330	28,991	2,724
Why	152,026	8,402	5,935	381
How-to-use	24,648	1,233	838	48
How-it-is-done	146,571	6,466	11,478	687
Property	166,459	8,326	5,016	396
Total	1,175,696	68,757	52,258	4,236

3.6 Evaluation Metrics

To evaluate the performance of the Codex model on code summarization, we exploit the common metrics including BLEU [51], ROUGE-L [42] and METEOR [6]. BLEU (Bilingual Evaluation Understudy) [51] is a commonly-used evaluation metric in the code comment generation studies [28, 32, 47, 62], which measures the similarity between one sentence to a set of reference sentences using constituent n-grams precision scores. ROUGE denotes the Recall-oriented Understudy for Gisting Evaluation [42]. It computes the count of several overlapping units such as n-grams, word pairs, and sequences. ROUGE has several different variants from which we consider the most popular one ROUGE-L [7, 41, 47], which is calculated based on the longest common subsequence (LCS). METEOR [6], which denotes the Metric for Evaluation of Translation with Explicit ORdering, is another widely used metric to evaluate the quality of generated code summaries [29, 47, 65]. METEOR evaluates the generated summary by aligning it to the reference summary and calculating the similarity scores based on the unigram matching.

3.7 Experiment Settings

In our experiments, beyond the zero-shot and one-shot settings, we choose to use five and ten code demonstrations for the few-shot setting. We cannot use too many code demonstrations since the input length is restricted by the context window limit. Therefore, we decide to provide the model with ten examples at most. The baseline for comparison is DOME [47] since it is so far the only approach that can address the multi-intent comment generation task. For running our experiments, we use the latest Codex model `code-davinci-002`. [3] We set the temperature as the default value, 0.5, to get a well-defined answer from Codex. We run all the experiments on an Hygon C86 7385 32-core CPU 2.50GHz machine with 2TB RAM. The running OS platform is Ubuntu 18.04.

It is important to note that both the results of RQ1 and RQ2 are subject to randomness. RQ2 is affected by the sampling process, while RQ1 is further influenced by the selection of demonstrations. To address this issue, we repeated each setting one hundred times and reported the average values in the paper. Therefore, the results of RQ1 and RQ2 can be regarded as the expected average effectiveness of Codex under specific settings. In contrast, RQ3 investigates

[2] We employ the st-codesearch-distilroberta-base model released at https://huggingface.co/flax-sentence-embeddings/st-codesearch-distilroberta-base, which was pre-trained on the CodeSearchNet dataset [31]

[3] https://platform.openai.com/docs/models/codex

Table 3: The results of Codex on multi-intent comment generation using zero-shot, one-shot, and few-shot learning (in %).

Intent	Method	Funcom			TLC		
		BLEU	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR
What	DOME	33.3	41.7	20.5	25.4	39.6	18.2
	Codex-0-shot	19.3	23.5	10.8	17.8	16.4	15.5
	Codex-1-shot	23.8	27.6	21.5	22.5	20.6	17.4
	Codex-5-shot	27.3	41.8	24.9	25.7	37.4	19.9
	Codex-10-shot	**34.5**	**58.6**	**26.8**	**32.4**	**45.6**	**23.1**
Why	DOME	33.0	42.3	20.5	21.9	35.3	15.7
	Codex-0-shot	21.7	20.3	11.4	19.6	17.8	9.6
	Codex-1-shot	22.9	28.8	12.9	20.8	23.2	11.9
	Codex-5-shot	27.5	45.8	16.9	24.1	40.6	13.5
	Codex-10-shot	**34.8**	**76.1**	**22.6**	**26.2**	**64.6**	**15.8**
How-to-use	DOME	31.6	39.3	19.3	17.1	26.1	12.3
	Codex-0-shot	22.3	11.1	16.8	21.2	10.9	12.2
	Codex-1-shot	23.1	18.9	17.5	21.8	16.6	14.4
	Codex-5-shot	27.9	48.6	19.8	24.4	40.5	15.7
	Codex-10-shot	**33.3**	**84.6**	**22.3**	**26.9**	**76.4**	**17.3**
How-it-is-done	DOME	26.9	39.5	17.6	20.4	36.6	14.7
	Codex-0-shot	18.9	37.9	9.8	16.8	32.1	9.6
	Codex-1-shot	21.0	39.6	13.5	19.1	36.4	12.1
	Codex-5-shot	24.8	49.2	16.2	21.1	52.7	12.8
	Codex-10-shot	**28.4**	**79.3**	**19.5**	**21.9**	**66.7**	**14.9**
Property	DOME	34.1	49.4	24.3	26.0	45.7	21.2
	Codex-0-shot	23.7	33.3	13.2	18.8	28.8	9.5
	Codex-1-shot	24.7	38.4	15.8	21.3	33.6	12.4
	Codex-5-shot	29.7	79.2	25.2	26.5	78.4	22.3
	Codex-10-shot	**36.2**	**81.9**	**29.4**	**28.7**	**80.3**	**24.7**
Average	DOME	31.8	42.5	20.5	22.2	36.7	16.5
	Codex-0-shot	21.2	25.2	12.4	18.8	21.2	11.3
	Codex-1-shot	23.1	30.7	16.2	21.1	26.1	13.6
	Codex-5-shot	27.4	52.9	20.6	24.4	49.9	16.8
	Codex-10-shot	**33.4**	**76.1**	**24.1**	**27.2**	**66.7**	**19.2**

whether better results can be achieved by leveraging the diversity of sampling results. To accomplish this, we repeated the experiments one hundred times and applied our reranking strategy based on the obtained results. The results of this RQ can thus be considered as the optimal achievable effectiveness of Codex.

4 STUDY RESULTS

4.1 RQ1: the Effectiveness of Vanilla In-Context Learning

Table 3 lists the results of DOME and Codex on the multi-intent comment generation task. For Codex, the results of using 0, 1, 5, and 10 demonstration examples are respectively illustrated. Generally, we observe that the effectiveness of in-context learning will be better with the number of code demonstrations increases. For instance, for the "what" intent, the BLEU value of Codex is 19.3% when no code demonstration is used while this values increases to 34.5% when using ten examples, on the Funcom dataset. This is within our expectation because more examples will provide more guidance for the model about the on-going task. When compared with the state-of-the-art DOME, we note that the effectiveness of **zero-shot and one-shot learning** is lower than that of DOME. For instance, the average BLEU values of zero-shot learning on the two datasets are 21.2% and 18.8%, respectively, while the corresponding values of DOME are 31.8% and 22.2%. This indicates that without enough code demonstrations, the potential of LLMs on the multi-intent comment generation task may not be fully leveraged.

> **Finding-1.** *Zero-shot and one-shot learning may not fully exploit the potential of the LLMs and their effectiveness is sub-optimal compared with that of the DOME approach.*

When the number of code demonstrations comes up to five, we observe the effectiveness of Codex is competitive to DOME: the values with respect to the ROUGE-L and METEOR metrics are higher than those of DOME while the BLEU values are sightly lower. A potential reason is that the BLEU metric excessively focuses on measuring n-gram overlapping. In concrete, it requires strict consistency (i.e., the n-grams must be identical), which is difficult for models that have not been fine-tuned to achieve perfect alignment with the references. In contrast, the ROUGE-L and METEOR metrics release this requirement by focusing on the longest common subsequence and considering other features such as the word order in addition to n-grams, respectively. Nonetheless, when the number of code demonstrations reaches ten, Codex outperforms DOME consistently with respect to all the three metrics and two datasets. Specifically, the average values of Codex with respect to the three metrics are 33.4%/76.1%/24.1% and 27.2%/66.7%/19.2% on the Funcom and TLC datasets, respectively. Such performances outperform the state-of-the-art DOME by 5.0%/79.1%/17.6% and 22.5%/81.8%/16.4%, respectively, on the two datasets. We also find that the performance of different approaches varies across the intent categories: generally, all the approaches have relatively low performances on the "how-it-is-done" category. Such a finding is consistent with the results from the existing study [12].

> **Finding-2.** *When the LLM is adequately prompted, its performance will exceed that of the state-of-the-art supervised learning approach. For instance, when the number of demonstrations is ten, the average ROUGE-L values of Codex on the two datasets are 76.1%/66.7%, respectively, outperforming DOME by 79.1%/81.8%.*

4.2 RQ2: the Effectiveness of Demonstration Selection

The results of different retrieval-based demonstration selection strategies are shown in Table 4. The zero-shot setting is excluded from this table since it does not use any code demonstration. We observe that the demonstration selections based on both token and semantic similarities significantly improve the performances compared with the vanilla random selection. For instance, when the number of selected examples is ten, the BLEU values of Codex on the Funcom and TLC datasets are 33.4% and 27.2%, respectively; while such values increase to 64.5% (65.9%) and 60.7% (62.8%) when the examples are selected based on token (semantic) similarities, with the relative improvements being 93% (97%) and 123% (131%). We also note that such performance improvements are universal (i.e., can be observed on each dataset no matter how many code examples are used). Moreover, we note that if similar examples are provided, the performance of 1-shot learning is even better than that of the vanilla 10-shot learning (e.g., the BLEU values on the Funcom dataset are 39.2% and 33.4%, respectively). Such results indicate the importance of the demonstration quality in the in-context learning: the model's performance could be improved if the given prompt is similar to the on-going task.

Case analysis. For qualitative analysis, we present one case to show how the similar code helps to rectify the generated comment of Codex, which is shown in Figure 3. Given the test code whose oracle comment is "Plays previous video in playlist", Codex with random selection generates a semantically-irrelevant comment "Plays the next song or video". This comment is inappropriate since the attributive "next" is wrong (the oracle is "previous") and

Table 4: The results of different retrieval-based demonstration selection strategies (in %).

Intent	Method	Funcom			TLC		
		BLEU	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR
What	Codex-1-shot	23.8	27.6	21.5	22.5	20.6	17.4
	Codex-1-shot ($Selection_{token}$)	**39.5**	**84.6**	35.0	**35.6**	**79.9**	31.4
	Codex-1-shot ($Selection_{semantic}$)	36.7	74.5	**36.1**	33.9	71.6	**32.8**
	Codex-5-shot	27.3	41.8	24.9	25.7	37.4	19.9
	Codex-5-shot ($Selection_{token}$)	41.0	82.3	**41.3**	38.6	76.8	37.7
	Codex-5-shot ($Selection_{semantic}$)	**41.1**	**82.9**	39.3	**39.1**	**78.9**	**38.3**
	Codex-10-shot	34.5	58.6	26.8	32.4	45.6	23.1
	Codex-10-shot ($Selection_{token}$)	**50.5**	**90.0**	**48.4**	**44.8**	**82.6**	**43.9**
	Codex-10-shot ($Selection_{semantic}$)	40.4	84.1	38.7	40.2	79.5	38.2
Why	Codex-1-shot	22.9	28.8	12.9	20.8	23.2	11.9
	Codex-1-shot ($Selection_{token}$)	32.8	**72.8**	27.7	30.7	**68.4**	25.5
	Codex-1-shot ($Selection_{semantic}$)	**33.2**	70.9	**28.0**	**31.6**	66.8	**26.2**
	Codex-5-shot	24.2	45.5	14.7	24.1	40.6	13.5
	Codex-5-shot ($Selection_{token}$)	**37.8**	**85.0**	**32.9**	34.5	78.7	29.8
	Codex-5-shot ($Selection_{semantic}$)	37.7	82.1	32.5	**35.1**	**79.3**	**30.2**
	Codex-10-shot	34.8	76.1	22.6	26.2	64.6	15.8
	Codex-10-shot ($Selection_{token}$)	74.9	**90.0**	**75.1**	72.1	81.4	68.9
	Codex-10-shot ($Selection_{semantic}$)	**75.0**	89.4	74.7	**72.4**	**81.9**	**73.0**
How-to-use	Codex-1-shot	23.1	18.9	17.5	21.8	16.6	14.4
	Codex-1-shot ($Selection_{token}$)	**56.3**	**88.3**	**53.7**	**52.2**	**81.6**	**42.8**
	Codex-1-shot ($Selection_{semantic}$)	52.4	74.4	47.1	46.8	71.5	42.3
	Codex-5-shot	24.2	48.1	18.9	24.4	40.5	15.7
	Codex-5-shot ($Selection_{token}$)	48.0	**86.4**	45.9	43.6	80.3	37.2
	Codex-5-shot ($Selection_{semantic}$)	**68.7**	86.2	**63.6**	**66.4**	**84.5**	**58.4**
	Codex-10-shot	33.3	84.6	22.3	26.9	76.4	17.3
	Codex-10-shot ($Selection_{token}$)	69.6	91.2	70.7	66.4	84.3	68.2
	Codex-10-shot ($Selection_{semantic}$)	**76.3**	**91.2**	**77.4**	**71.6**	**85.4**	**73.6**
How-it-is-done	Codex-1-shot	21.0	39.6	13.5	19.1	36.4	12.1
	Codex-1-shot ($Selection_{token}$)	**31.9**	**72.9**	25.8	**28.6**	**69.4**	24.7
	Codex-1-shot ($Selection_{semantic}$)	30.5	69.6	**27.6**	28.2	68.7	**25.9**
	Codex-5-shot	22.5	48.9	13.7	21.1	52.7	12.8
	Codex-5-shot ($Selection_{token}$)	**33.7**	**85.7**	**30.8**	**29.7**	**78.4**	**26.8**
	Codex-5-shot ($Selection_{semantic}$)	32.9	80.0	27.5	28.3	73.9	25.1
	Codex-10-shot	28.4	79.3	19.5	21.9	66.7	14.9
	Codex-10-shot ($Selection_{token}$)	47.9	84.6	49.6	45.2	80.8	47.7
	Codex-10-shot ($Selection_{semantic}$)	**51.6**	**86.4**	**50.8**	**48.9**	**82.9**	**47.9**
Property	Codex-1-shot	24.7	38.4	15.8	21.3	33.6	12.4
	Codex-1-shot ($Selection_{token}$)	35.7	67.7	33.1	33.2	**64.9**	30.8
	Codex-1-shot ($Selection_{semantic}$)	**36.4**	**80.0**	**35.9**	**34.9**	62.8	**32.4**
	Codex-5-shot	29.7	79.2	25.2	26.5	78.4	22.3
	Codex-5-shot ($Selection_{token}$)	**45.1**	**89.2**	43.2	**41.5**	**85.4**	**40.6**
	Codex-5-shot ($Selection_{semantic}$)	43.9	82.7	40.3	39.6	82.1	38.1
	Codex-10-shot	36.2	81.9	29.4	28.7	80.3	24.7
	Codex-10-shot ($Selection_{token}$)	79.6	84.2	75.7	74.8	83.9	68.9
	Codex-10-shot ($Selection_{semantic}$)	**86.3**	**95.8**	**87.4**	**81.0**	**86.4**	**80.8**
Average	Codex-1-shot	23.1	30.7	16.2	21.1	26.1	13.6
	Codex-1-shot ($Selection_{token}$)	**39.2**	**77.3**	**35.1**	**36.1**	**72.8**	31.0
	Codex-1-shot ($Selection_{semantic}$)	37.8	73.9	35.0	35.1	68.3	**31.9**
	Codex-5-shot	27.4	52.9	20.6	24.4	49.9	16.8
	Codex-5-shot ($Selection_{token}$)	41.1	**85.7**	38.8	37.6	**79.9**	34.4
	Codex-5-shot ($Selection_{semantic}$)	**44.9**	82.8	**40.6**	**41.7**	79.7	**38.0**
	Codex-10-shot	33.4	76.1	24.1	27.2	66.7	19.2
	Codex-10-shot ($Selection_{token}$)	64.5	88.0	63.9	60.7	82.6	59.5
	Codex-10-shot ($Selection_{semantic}$)	**65.9**	**89.4**	**65.8**	**62.8**	**83.2**	**62.7**

will thus mislead the potential maintainer of the code. Fortunately, after using the semantic-based demonstration selection strategy, Codex generates a comment that is semantically-identical to the oracle, i.e., "Plays the previous video in your playlist". The achieved BLEU score reaches 73.1%, which is a relatively high performance. By investigating the most semantically-similar code in the corpus (listed in the bottom of the figure), we find that one potential reason for the success of Codex is that the example code shows it the attributive could come from the method name. Specifically, the comment for the semantically-similar code is "Play the first item" and "first" is a token from the method name. With this example in mind, Codex generates the correct attributive "previous", which can also be extracted from the method name.

Finding-3. *Both token-based and semantic-based demonstration selection strategies can improve the effectiveness of Codex to a large extent.*

When it comes to the comparison between the two selection strategies, we find that no strategy can consistently outperform the other under all the settings. For instance, when using one-shot learning, the performance of the token-based selection is better than that of the semantic-based selection on average; and vice versa when using few-shot learning (i.e., the number of examples are five or ten). Moreover, even if the semantic-based selection generally has a better performance when the number of examples is ten, it can also be outperformed by the token-based one under certain settings. For instance, on the *what* intent, the BLEU values of the token-based selection are 50.5% and 44.8%, respectively, on the two datasets, exceeding those of the semantic-based selection, which are 40.4% and 40.2%.

Finding-4. *No demonstration selection strategy can consistently outperform its alternative. The effectiveness depends on the detailed settings (e.g., the number of examples and the intents).*

Table 5: The results of different reranking strategies (in %).

Intent	Method	Funcom			TLC		
		BLEU	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR
what	Codex-1-shot	23.8	27.6	21.5	22.5	20.6	17.4
	Codex-1-shot ($Rerank_{token}$)	**32.2**	76.1	**33.3**	**28.9**	**72.7**	**29.3**
	Codex-1-shot ($Rerank_{semantic}$)	29.7	**76.5**	26.7	27.1	71.9	24.8
	Codex-1-shot ($Selection_{token}$)	39.5	84.6	35.0	35.6	79.9	31.4
	Codex-1-shot ($Selection_{token}$ + $Rerank_{token}$)	44.4	84.9	43.4	41.8	**77.6**	38.5
	Codex-1-shot ($Selection_{token}$ + $Rerank_{semantic}$)	**45.8**	**85.2**	**44.9**	**42.6**	75.8	**40.8**
	Codex-10-shot	34.5	58.6	26.8	32.4	45.6	**23.1**
	Codex-10-shot ($Rerank_{token}$)	36.9	84.5	29.3	34.8	76.9	26.6
	Codex-10-shot ($Rerank_{semantic}$)	**39.7**	**85.6**	**36.5**	**37.1**	**81.0**	**31.8**
	Codex-10-shot ($Selection_{semantic}$)	40.4	84.1	38.7	40.2	79.5	38.2
	Codex-10-shot ($Selection_{semantic}$ + $Rerank_{token}$)	58.6	87.2	61.3	56.3	82.9	58.4
	Codex-10-shot ($Selection_{semantic}$ + $Rerank_{semantic}$)	**60.2**	**89.4**	**64.1**	**58.3**	**85.2**	**60.9**
why	Codex002-1-shot	22.9	28.8	12.9	20.8	23.2	11.9
	Codex-1-shot ($Rerank_{token}$)	23.5	67.6	17.7	22.6	62.7	19.4
	Codex-1-shot ($Rerank_{semantic}$)	**29.2**	**68.0**	**25.7**	**26.7**	**63.3**	**20.1**
	Codex-1-shot ($Selection_{token}$)	32.8	72.8	27.7	30.7	68.4	25.5
	Codex-1-shot ($Selection_{token}$ + $Rerank_{token}$)	36.4	81.0	31.6	34.4	77.1	28.9
	Codex-1-shot ($Selection_{token}$ + $Rerank_{semantic}$)	**38.6**	**83.4**	**35.9**	**36.9**	**80.2**	**30.3**
	Codex-10-shot	34.8	76.1	22.6	26.2	64.6	15.8
	Codex-10-shot ($Rerank_{token}$)	**36.8**	**91.0**	**24.8**	**31.2**	**86.1**	**20.9**
	Codex-10-shot ($Rerank_{semantic}$)	35.3	90.9	23.2	30.4	85.2	20.1
	Codex-10-shot ($Selection_{semantic}$)	75.0	89.4	74.7	72.4	81.9	73.0
	Codex-10-shot ($Selection_{semantic}$ + $Rerank_{token}$)	**78.3**	**92.4**	**76.6**	**74.8**	**88.7**	**74.1**
	Codex-10-shot ($Selection_{semantic}$ + $Rerank_{semantic}$)	76.2	90.6	75.3	73.5	86.2	73.6
How-to-use	Codex-1-shot	23.1	18.9	17.5	21.8	16.6	14.4
	Codex-1-shot ($Rerank_{token}$)	25.1	62.0	19.7	24.2	58.8	17.6
	Codex-1-shot ($Rerank_{semantic}$)	**28.5**	**63.6**	**22.9**	**26.1**	**61.3**	**18.8**
	Codex-1-shot ($Selection_{token}$)	56.3	88.3	53.7	52.2	81.6	42.8
	Codex-1-shot ($Selection_{token}$ + $Rerank_{token}$)	**63.8**	**90.7**	**66.3**	**60.6**	**85.3**	**59.7**
	Codex-1-shot ($Selection_{token}$ + $Rerank_{semantic}$)	61.1	85.7	60.6	58.4	83.6	57.2
	Codex-10-shot	33.3	84.6	22.3	26.9	76.4	17.3
	Codex-10-shot ($rerank_{token}$)	32.7	**86.6**	**27.0**	30.9	**82.4**	**23.2**
	Codex-10-shot ($rerank_{semantic}$)	**35.2**	85.6	24.2	**32.8**	81.5	21.6
	Codex-10-shot ($Selection_{semantic}$)	76.3	91.2	77.4	71.6	85.4	73.6
	Codex-10-shot ($Selection_{semantic}$ + $Rerank_{token}$)	78.8	93.5	74.2	71.9	85.1	73.9
	Codex-10-shot ($Selection_{semantic}$ + $Rerank_{semantic}$)	**79.1**	**93.9**	**75.2**	**72.3**	**85.7**	**74.5**
How-it-is-done	Codex-1-shot	21.0	39.6	13.5	19.1	36.4	12.1
	Codex-1-shot ($rerank_{token}$)	**29.8**	**79.3**	**22.2**	**27.5**	**74.8**	**20.9**
	Codex-1-shot ($rerank_{semantic}$)	29.4	77.3	21.7	26.8	73.1	19.8
	Codex-1-shot ($Selection_{token}$)	31.9	72.9	25.8	28.6	69.4	24.7
	Codex-1-shot ($Selection_{token}$ + $Rerank_{token}$)	**33.8**	**79.1**	**28.9**	**32.2**	**77.4**	**26.3**
	Codex-1-shot ($Selection_{token}$ + $Rerank_{semantic}$)	33.0	77.6	27.1	31.4	75.2	25.8
	Codex-10-shot	28.4	79.3	19.5	21.9	66.7	14.9
	Codex-10-shot ($rerank_{token}$)	**30.6**	**95.3**	**24.7**	**28.1**	**90.8**	**23.2**
	Codex-10-shot ($rerank_{semantic}$)	30.0	95.2	22.3	27.6	90.1	20.1
	Codex-10-shot ($Selection_{semantic}$)	51.6	86.4	50.8	48.9	82.9	47.9
	Codex-10-shot ($Selection_{semantic}$ + $Rerank_{token}$)	**59.2**	**89.3**	**57.4**	**56.1**	**85.6**	**53.5**
	Codex-10-shot ($Selection_{semantic}$ + $Rerank_{semantic}$)	57.6	88.1	55.2	55.3	83.1	53.2
Property	Codex-1-shot	24.7	38.4	15.8	21.3	33.6	12.4
	Codex-1-shot ($rerank_{token}$)	**34.8**	**59.8**	**33.2**	**30.6**	**51.2**	**28.7**
	Codex-1-shot ($rerank_{semantic}$)	34.2	59.5	32.1	29.5	49.8	27.6
	Codex-1-shot ($Selection_{token}$)	35.7	67.7	33.1	33.2	64.9	30.8
	Codex-1-shot ($Selection_{token}$ + $Rerank_{token}$)	**41.6**	**74.2**	**39.2**	**38.4**	**69.6**	**35.9**
	Codex-1-shot ($Selection_{token}$ + $Rerank_{semantic}$)	40.1	72.1	36.0	37.7	67.2	34.3
	Codex-10-shot	36.2	81.9	29.4	28.7	80.3	24.7
	Codex-10-shot ($rerank_{token}$)	**38.4**	**84.2**	**31.2**	**35.2**	**82.7**	**28.6**
	Codex-10-shot ($rerank_{semantic}$)	36.2	78.4	30.9	34.1	81.2	27.4
	Codex-10-shot ($Selection_{semantic}$)	86.3	95.8	87.4	81.0	86.4	80.8
	Codex-10-shot ($Selection_{semantic}$ + $Rerank_{token}$)	**87.2**	**96.4**	**88.7**	**84.9**	**89.1**	**83.2**
	Codex-10-shot ($Selection_{semantic}$ + $Rerank_{semantic}$)	86.8	96.1	87.9	82.5	88.2	82.1
Average	Codex-1-shot	23.1	30.7	16.2	21.1	26.1	13.6
	Codex-1-shot ($rerank_{token}$)	29.1	68.9	25.2	26.8	**64.0**	**23.2**
	Codex-1-shot ($rerank_{semantic}$)	**30.2**	**69.0**	**25.8**	**27.2**	63.9	22.2
	Codex-1-shot ($Selection_{token}$)	39.2	77.3	35.1	36.1	72.8	31.0
	Codex-1-shot ($Selection_{token}$ + $Rerank_{token}$)	**44.0**	**82.0**	**41.9**	**41.5**	**77.4**	**37.9**
	Codex-1-shot ($Selection_{token}$ + $Rerank_{semantic}$)	43.7	80.8	40.9	41.4	76.4	37.7
	Codex-10-shot	33.4	76.1	24.1	27.2	66.7	19.2
	Codex-10-shot ($rerank_{token}$)	35.1	**88.3**	**27.4**	32.0	**83.8**	**24.5**
	Codex-10-shot ($rerank_{semantic}$)	**35.3**	87.1	27.4	**32.4**	83.8	24.2
	Codex-10-shot ($Selection_{semantic}$)	65.9	89.4	65.8	62.8	83.2	62.7
	Codex-10-shot ($Selection_{semantic}$ + $Rerank_{token}$)	**72.4**	**91.8**	**71.6**	**68.8**	**86.3**	68.6
	Codex-10-shot ($Selection_{semantic}$ + $Rerank_{semantic}$)	72.0	91.6	71.5	68.4	85.7	**68.9**

4.3 RQ3: the Effectiveness of Reranking

The results of different reranking strategies are shown in Table 5. Due to the space limitation, we list the results of 1-shot and 10-shot learning. For 1-shot, we also combine different reranking strategies with token-based demonstration selection since according to the results from the above section, this selection strategy achieves better results on 1-shot. Similarly, for 10-shot, we combine different reranking strategies with semantic-based demonstration selection.

```
// Test Code:
private void playPrevious() {
      if (mediaType == ItemType.YOUTUBE_MEDIA_TYPE_VIDEO) {
         restartVideo();
         return;
      }
      if (currentSongIndex - 1 >= 0) {
         currentSongIndex--;
      } else {
         currentSongIndex = youTubeVideos.size() - 1;
      }
      videoItem = youTubeVideos.get(youTubeVideos.size() - 1);
      playVideo();
   }
// Ground-truth Comment: Plays previous video in playlist.
// Codex-1-shot: Plays the next song or video.
// Codex-1-shot (Selection): Plays the previous video in your playlist.
--------------------------------------------------------------------------
// Top-1 Semantic-Similar Code
// Comment: Play the first item in the audio queue.
private void playFirstInQueue() {
        AudioQueueItem queueItem = mAudioQueue.poll();
        try {
            mMediaPlayer.setDataSource(this, queueItem.mUri);
        } catch (IOException e) {
            e.printStackTrace();
            endPlayback();
            return;
        }
        mMediaPlayer.setOnCompletionListener(queueItem.mListener);
        try {
            mMediaPlayer.prepare();
        } catch (IOException e) {
            e.printStackTrace();
            endPlayback();
            return;
        }
        mMediaPlayer.start();
    }
```

Figure 3: An illustrative example to show how semantic-based selection helps improve the comment generation compared with the random selection.

Results show that both reranking strategies help boost the performance of Codex slightly. For instance, for 1-shot learning, the token-based reranking strategy increases the BLEU values on the Funcom and TLC datasets from 23.1% and 21.1% to 29.1% and 26.8%, while the semantic-based strategy further achieves 30.2% and 27.2% on the two datasets. We also note that the reranking can enhance the results no matter whether the demonstration selection is used. The best-performing model variant, i.e., the 10-shot learning with semantic-based demonstration selection and token-based reranking, achieves BLEU scores of 72.4% and 68.8% on the two datasets on average, outperforming the state-of-the-art DOME by 128% and 210%, respectively (cf. Table 3).

Case analysis. We present another case to show how the reranking strategy helps select more qualified comments, which is shown in Figure 4. In this figure, we demonstrate the top-5 generated comments from Codex. The first generated comment is semantically vague since it fails to explicitly explain the meaning of the words `DURABLE_EXPLICIT` and `DURABLE_IMPLICIT`. Similarly, the second generated comment may also mislead developers since it is unclear what is an `Endpoint`, which does not occur in the source code. The third and forth generated comments share a similar meaning but are expressed in different ways, and they are both semantically-identical to the oracle comment. After using the token-based similar code selection, a code snippet with the comment "Determines whether or not ..." is utilized to help rerank the original results. Due to a large degree of token overlap with the reference comment, the forth generated comment from Codex is used as the final result according to the token-based reranking strategy. Compared with

```
// Test Code:
public boolean isDurableSubscriber() {
  return !StringsUtils.isEmpty(m_durableSubscriptionName);
}
// Ground-truth Comment: Determines if the subscriber is durable.
--------------------------------------------------------------------------
/*
Codex-1-shot Top-5 Generated Comments:
1.Returns true if this subscription has the subscribtion type
      DURABLE_EXPLICIT or DURABLE_IMPLICIT.
2.Indicates whether or not the Endpoint is a durable subscriber.
3.Returns TRUE if this is a durable subscription and FALSE otherwise.
4.Determines whether or not the subscriber is durable.
5.Can a durable customer install the said customer.
*/
--------------------------------------------------------------------------
// Top-1 Lexical-Similar Code
// Comment: Determines whether or not this subscription is to all stream or
      to a specific stream.
public boolean isSubscribedToAll() {
    return isNullOrEmpty(streamId);
}
--------------------------------------------------------------------------
// Top-1 result after token-based rerank:
// Determines whether or not the subscriber is durable.
```

Figure 4: An illustrative example to show how our re-ranking strategy helps improve the comment generation.

the original top-1 result, the BLEU value is increased from 23.4% to 68.6%.

> **Finding-5.** *Both token-based and semantic-based reranking strategies can further enhance the performance of Codex.*

As for the comparison between the two reranking strategies, we again observe that no one can consistently outperform its alternative. Generally, token-based reranking works better when combined with demonstration selections while semantic-based reranking works better when no demonstration selection is adopted. There are, however, some corner cases. For instance, for the "what" intent category, semantic-based reranking performs better when combined with demonstration selections.

> **Finding-6.** *No reranking strategy can consistently outperform its alternative.*

5 DISCUSSION

5.1 Human Evaluation

While metrics such as BLEU, ROUGE-L, and METEOR can evaluate the lexical disparity between the generated comments and the oracle, they are inadequate in reflecting the semantic differences. Thus, to further evaluate the quality of comments generated by various approaches, we conduct a human evaluation. Specifically, we recruit six participants with at least five years of experience in Java development. The participants include three Ph.D students and three senior researchers who are not co-authors of this paper. We randomly select 100 code snippets (20 from each intent category) to perform this user study. For each code snippet, we show the participants the oracle comment and the results from four approaches, namely, DOME, Codex-10-shot, Codex-10-shot with semantic-based selection, and Codex-10-shot with semantic-based selection and token-based reranking, which results in 400 generated comments as our evaluation subjects. To ensure fairness, the participants are not aware of where the comments are generated from. Each participant is asked to rate all the 400 comments from

Table 6: The statistic results of our user study.

	Approach	Avg.	Std.
Naturalness	DOME	3.9	0.8
	Codex-10-shot	4.2	0.7
	Codex-10-shot ($Selection_{semantic}$)	4.3	0.8
	Codex-10-shot ($Selection_{semantic}$+$Rerank_{token}$)	**4.3**	0.7
Adequacy	DOME	3.3	1.3
	Codex-10-shot	3.5	1.1
	Codex-10-shot ($Selection_{semantic}$)	3.8	1.2
	Codex-10-shot ($Selection_{semantic}$+$Rerank_{token}$)	**4.1**	0.9
Usefulness	DOME	3.0	1.4
	Codex-10-shot	3.1	1.3
	Codex-10-shot ($Selection_{semantic}$)	3.7	1.1
	Codex-10-shot ($Selection_{semantic}$+$Rerank_{token}$)	**3.8**	1.3

three aspects: (1) **Naturalness** which reflects the fluency of generated comments from the perspective of grammar; (2) **Adequacy** which reflects the information richness of generated comments; and (3) **Usefulness** which reflects how can generated comments help developers, on a 5-point Likert scale (1 for poor, 2 for marginal, 3 for acceptable, 4 for good, and 5 for excellent). Such an experiment setting follows existing studies [46, 59].

Results of our user study are listed in Table 6. We observe that higher metric values lead to higher scores rated by the participants. Specifically, the best-performing model variant in our quantitative evaluation, i.e., 10-shot learning with semantic-based demonstration selection and token-based reranking, also achieves the highest scores from participants (i.e., 4.3, 4.1, and 3.8 with respect to the three aspects, respectively). We also note that LLMs are good at generating fluent NL descriptions, since all the model variants achieve scores higher than 4 with respect to the naturalness property. In contrast, all scores achieved on the usefulness property are lower than 4, which indicates there is still a room for improving the usefulness of the generated comments.

5.2 Implications

Large language models are few-shot summarizers. Our empirical investigation shows that LLMs are capable of generating high-quality code comments with diverse intents. Results show that the best-performing model variant, i.e., Codex-10-shot with semantic-based demonstration selection and token-based reranking, outperforms the state-of-the-art DOME approach to a large extent on the two datasets (e.g., outperforms DOME by 128%/210% with respect to the BLEU metric on the Funcom/TLC datasets). This indicates that in practice, developers can refer to the LLMs for helping them automatically generate comments with different intents. LLMs are thus of great potential to facilitate program comprehension activities. For researchers, this also indicates that the comparison with LLMs is necessary when evaluating a newly-proposed code summarization approach.

On the importance of prompt quality. Our results show that the quality of the prompt provided to LLMs can significantly impact the generated results. Specifically, providing LLMs with examples that are similar to the target code may help them generate more qualified results. This calls for more attention to the demonstration selection process. However, as for the selection strategy, our results also indicate that there is no silver bullet: the token-based similar code selection and the semantic-based one complement each other. This means that more research efforts could be devoted to devise a better selection strategy.

More attempts, more gains. Due to the sampling process, LLMs can generate multiple results for a specific input. Our results (e.g., the case in Figure 4) show that sometimes a comment similar to the oracle one may not be generated at the first place. Therefore, in practice, developers may query the LLMs for more times if they feel the generated comments are not good enough. For researchers, how to automatically rerank the results of LLMs also deserves more in-depth explorations and our initial attempt with two simple heuristics achieves promising results.

5.3 Threats to Validity

Internal validity. Codex is trained on open-source projects and thus there may be data leakage, i.e., Codex may have seen the comments for the test cases during its pre-training. However, we observe that Codex does not perform effectively under the zero-shot setting, which indicates that the model's output is not generated due to memorization. Such a threat is also faced by other studies on large language models [48], and to fully address this threat requires to re-train the model from scratch, which would be currently infeasible considering the limitation of the computation resource.

As introduced, our results are affected by the randomness incurred by the model sampling process or the demonstration selection. To mitigate this threat as well as keep the time cost of the experiments in a reasonable scale, we repeat each experiment for one hundred times. However, one hundred may not fully eliminate the randomness and we leave more experiments as future work.

External validity. The first threat to applying our observation in practice is that it is unclear whether developers can find code snippets similar to the target code for constructing better prompts to the LLMs. However, our results also show that under the 10-shot setting, the performance of Codex exceeds that of the state-of-the-art DOME even if the demonstrations are randomly selected.

Another threat is that we only focus on Java programming language. This setting is restricted by the availability of multi-intent comment dataset in the literature. This threat is alleviated considering that the two datasets are large-scale and Java is the most widely-studied language in the comment generation domain [28, 36, 47].

6 CONCLUSION

Our empirical study mainly investigates whether it is feasible to utilize the LLMs for addressing multi-intent comment generation and further how to improve the effectiveness of LLMs on this task. Our results gives positive answer to the first point: by utilizing few-shot in-context learning, the performance of Codex exceeds that of the state-of-the-art supervised learning approach. We also demonstrate that both demonstration selection and result reranking can help boost the performance of Codex. Our study establishes new baselines for the multi-intent comment generation task as well as pointing research directions that deserve more in-depth investigations.

7 DATA AVAILABILITY

All code and data in this study are publicly available at:

https://github.com/gmy2013/LLM_Comment_Generation.

REFERENCES

[1] 2010. Original Funcom Dataset. (2010). http://www.ics.uci.edu/lopes/datasets/.

[2] 2022. Codex model. (2022). https://beta.openai.com/docs/models/codex-seriesprivate-beta.

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[5] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net.

[6] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 65–72.

[7] Aakash Bansal, Sakib Haque, and Collin McMillan. 2021. Project-level encoding for neural source code summarization of subroutines. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 253–264.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[9] Ruichu Cai, Zhihao Liang, Boyan Xu, Zijian Li, Yuexing Hao, and Yao Chen. 2020. TAG: Type auxiliary guiding for code comment generation. *arXiv preprint arXiv:2005.02835* (2020).

[10] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

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

[12] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why my code summarization model does not work: Code comment improvement with category prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–29.

[13] Junyan Cheng, Iordanis Fostiropoulos, and Barry Boehm. 2021. GN-Transformer: Fusing Sequence and Graph Representation for Improved Code Summarization. *arXiv preprint arXiv:2111.08874* (2021).

[14] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, Zhen Ming, et al. 2022. GitHub Copilot AI pair programmer: Asset or Liability? *arXiv preprint arXiv:2206.15331* (2022).

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 4171–4186. https://doi.org/10.18653/v1/n19-1423

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[17] Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: precisely and efficiently measuring the similarity of code. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[18] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Improving automatically generated code from Codex via Automated Program Repair. *arXiv preprint arXiv:2205.10583* (2022).

[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[20] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael Lyu. 2023. Code Structure–Guided Transformer for Source Code Summarization. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–32.

[21] Mingyang Geng, Shangwen Wang, Dezun Dong, Shanzhi Gu, Fang Peng, Weijian Ruan, and Xiangke Liao. 2022. Fine-grained code-comment semantic interaction analysis. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 585–596.

[22] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Shaomeng Cao, Kechi Zhang, and Zhi Jin. 2023. Interpretation-based Code Summarization. In *Proceedings of the 31st IEEE/ACM International Conference on Program Comprehension*.

[23] Yaroslav Golubev, Viktor Poletansky, Nikita Povarov, and Timofey Bryksin. 2021. Multi-threshold token-based code clone detection. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 496–500.

[24] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*.

[25] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*. IEEE, 35–44.

[26] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 300–310.

[27] Emily Hill, Lori Pollock, and K Vijay-Shanker. 2009. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 232–242.

[28] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*. 200–210.

[29] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25 (2020), 2179–2217.

[30] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. (2018).

[31] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[32] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[33] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering* 28, 7 (2002), 654–670.

[34] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch Generation with Language Models: Feasibility and Scaling Behavior. In *Deep Learning for Code Workshop*.

[35] Alexander LeClair, Aakash Bansal, and Collin McMillan. 2021. Ensemble models for neural source code summarization of subroutines. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 286–297.

[36] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 795–806.

[37] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. Editsum: A retrieve-and-edit framework for source code summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 155–166.

[38] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. 2022. AUGER: automatically generating review comments with pre-training models. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1009–1021.

[39] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F. Bissyandé. 2021. Automated Comment Update: How Far are We?. In *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*. 36–46. https://doi.org/10.1109/ICPC52881.2021.00013

[40] Bo Lin, Shangwen Wang, Zhongxin Liu, Xin Xia, and Xiaoguang Mao. 2023. Predictive Comment Updating With Heuristics and AST-Path-Based Neural Learning: A Two-Phase Approach. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1640–1660. https://doi.org/10.1109/TSE.2022.3185458

[41] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving code summarization with block-wise abstract syntax tree splitting. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 184–195.

[42] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[43] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. 2016. Recurrent neural network for text classification with multi-task learning. *arXiv preprint arXiv:1605.05101* (2016).

[44] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2022. Using Transfer Learning for Code-Related Tasks. *IEEE Transactions on Software Engineering* (2022).

[45] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work? *arXiv preprint arXiv:2202.12837* (2022).

[46] Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2022. Automatic Comment Generation via Multi-Pass Deliberation. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[47] Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2023. Developer-Intent Driven Code Comment Generation. *arXiv preprint arXiv:2302.07055* (2023).
[48] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE.
[49] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. 2023. LEVER: Learning to Verify Language-to-Code Generation with Execution. *arXiv preprint arXiv:2302.08468* (2023).
[50] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, Vol. 1. 380–384.
[51] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.
[52] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.
[53] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.
[54] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair*. 69–75.
[55] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).
[56] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.
[57] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).
[58] Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*. 390–401.
[59] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1105–1116.
[60] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2021. Learning to retrieve prompts for in-context learning. *arXiv preprint arXiv:2112.08633* (2021).
[61] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454* (2022).
[62] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 397–407.
[63] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. 2022. No More Fine-Tuning? An Experimental Evaluation of Prompt Tuning in Code Intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 382–394. https://doi.org/10.1145/3540250.3549113
[64] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.
[65] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, S Yu Philip, and Guandong Xu. 2020. Reinforcement-learning-guided source code summarization using hierarchical attention. *IEEE Transactions on software Engineering* 48, 1 (2020), 102–119.
[66] Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Shi Han, Hongyu Zhang, and Dongmei Zhang. 2021. Cocosum: Contextual code summarization with multi-relational graph neural network. *arXiv preprint arXiv:2107.01933* (2021).
[67] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
[68] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems* 32 (2019).
[69] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: exemplar-based neural comment generation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 349–360.
[70] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 380–389.
[71] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 562–567.
[72] Rui Xie, Wei Ye, Jinan Sun, and Shikun Zhang. 2021. Exploiting method names to improve code summarization: A deliberation multi-task learning approach. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 138–148.
[73] Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. 2020. Leveraging code generation to improve code retrieval and summarization via dual learning. In *Proceedings of The Web Conference 2020*. 2309–2319.
[74] Chen Zeng, Yue Yu, Shanshan Li, Xin Xia, Zhiming Wang, Mingyang Geng, Linxiao Bai, Wei Dong, and Xiangke Liao. 2022. deGraphCS: Embedding Variable-based Flow Graph for Neural Code Search. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2022).
[75] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1359–1371.
[76] Aiping Zhang, Liming Fang, Chunpeng Ge, Piji Li, and Zhe Liu. 2023. Efficient transformer with code token learner for code clone detection. *Journal of Systems and Software* 197 (2023), 111557.
[77] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1385–1397.
[78] Tianyi Zhang, Tao Yu, Tatsunori B Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I Wang. 2022. Coder Reviewer Reranking for Code Generation. *arXiv preprint arXiv:2211.16490* (2022).