Natural Language to Code: How Far Are We?

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

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

Bo Lin
linbo19@nudt.edu.cn
College of Computer, National University of Defense Technology
Changsha, China

Zhensu Sun
zhensuuu@gmail.com
School of Computing and Information Systems, Singapore Management University
Singapore

Ming Wen
mwenaa@hust.edu.cn
School of Cyber Science and Engineering, Huazhong University of Science and Technology
Wuhan, China

Yepang Liu
liuyp1@sustech.edu.cn
Department of Computer Science and Engineering, Southern University of Science and Technology
Shenzhen, China

Li Li
lilicoding@ieee.org
School of Software, Beihang University
Beijing, China

Tegawendé F. Bissyandé
tegawende.bissyande@uni.lu
University of Luxembourg
Luxembourg

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

ABSTRACT

A longstanding dream in software engineering research is to devise effective approaches for automating development tasks based on developers' informally-specified intentions. Such intentions are generally in the form of natural language descriptions. In recent literature, a number of approaches have been proposed to automate tasks such as code search and even code generation based on natural language inputs. While these approaches vary in terms of technical designs, their objective is the same: transforming a developer's intention into source code. The literature, however, lacks a comprehensive understanding towards the effectiveness of existing techniques as well as their complementarity to each other. We propose to fill this gap through a large-scale empirical study where we systematically evaluate *natural language to code* techniques. Specifically, we consider six state-of-the-art techniques targeting code search, and four targeting code generation. Through extensive evaluations on a dataset of 22K+ natural language queries, our study reveals the following major findings: (1) code search techniques based on model pre-training are so far the most effective while code generation techniques can also provide promising results; (2) complementarity widely exists among the existing techniques; and (3) combining the ten techniques together can enhance the performance for 35% compared with the most effective standalone technique. Finally, we propose a post-processing strategy to automatically integrate different techniques based on their generated code. Experimental results show that our devised strategy is both effective and extensible.

CCS CONCEPTS

• **Software and its engineering** → **Reusability**; **Automatic programming**.

KEYWORDS

Code Search, Code Generation, Pre-Training Technique

ACM Reference Format:
Shangwen Wang, Mingyang Geng, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Li Li, Tegawendé F. Bissyandé, and Xiaoguang Mao. 2023. Natural Language to Code: How Far Are We? . In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616323

1 INTRODUCTION

Recommender systems are widely studied in software engineering research as they are concrete building blocks for improving developers' productivity [3]. A highly-sought achievement in this domain is to effectively transform developers' intentions, which are generally specified in natural language, into pieces of code [39]. Addressing such a challenge will alleviate some software development burdens, and facilitate critical designs and implementation choices such as selecting the appropriate programming interfaces to use [25]. Indeed, developers in all levels of programming proficiency frequently ask questions of varying complexity about how to

† Ming Wen and Yepang Liu are the corresponding authors.
This work was done when the first author was a visiting scholar at Southern University of Science and Technology.
Shangwen Wang, Bo Lin, and Xiaoguang Mao are also with the Key Laboratory of Software Engineering for Complex Systems, Yepang Liu is also with the Research Institute of Trustworthy Autonoumous Systems.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616323

implement specific functionalities [62]. For example, it is typical to see developers having the intention to "remove a specific item from an array" look for related code in Q&A forums.[1] Recent advances in deep learning have enabled the development of promising techniques in two lines of research towards transforming developers' informally-specified intentions (a.k.a *queries* that are generally in the form of natural language descriptions) into source code: (1) *code search* aims at retrieving a relevant piece of code within a large codebase [8, 51, 64], while (2) *code generation* aims to synthesize code from scratch [32, 65, 66]. In practice, to obtain the desired code, a developer may leverage code generation techniques to generate code directly, or retrieve relevant code snippets from a large-scale codebase such as StackOverflow. As a result, the application scenarios of these two types of techniques could overlap with each other to a certain degree [58, 62]. We refer to all such relevant literature techniques as **Natural Language to Code (NL2Code)** techniques.

NL2Code techniques differ in terms of various design aspects. First, at a high level, the theoretical working mechanisms of code search and code generation are different: code search focuses on mapping the semantic relevance between the query and an existing code snippet and directly returns the code with the highest relevance score; code generation, in contrast, constructs a piece of code from scratch. Second, they can be differentiated according to whether a pre-trained model is used. Those that build on pre-training techniques, such as GraphCodeBERT [18] and CodeT5 [59], adopt a pre-training and fine-tuning pipeline where deep learning models are first pre-trained on a large-scale corpus aiming at capturing the semantic relation between natural language and programming language, and then fine-tuned on specific downstream tasks. In contrast, those that do not rely on pre-trained models (referred to as *non-pre-training techniques*), such as MMAN [57] and Tranx [66], train their models from scratch on relatively small-scale labelled datasets. These training methods can significantly affect the effectiveness of NL2Code techniques. For instance, Zeng *et al.* observed that pre-training techniques outperform non-pre-training techniques on a number of code intelligence tasks such as code clone detection [69]. Furthermore, according to the intrinsic difference between code search and code generation, the former may produce high-quality results if there exists a code snippet that is similar to the intended functionality; on the contrary, the latter may generate more reasonable results if there is no code snippet for reference whose semantic is similar to the intention.

Although enormous efforts have been made towards advancing the NL2Code techniques [3, 12, 31, 48], the effectiveness of existing techniques has not been systematically studied and compared. Particularly, in the literature, code generation and code search techniques are often evaluated separately [17, 57, 66, 69], which means that code generation techniques are compared to each other without considering code search techniques, and vice versa. As a result, little is known about their complementarity with each other, i.e., can the query ineffectively handled by one technique be addressed well by another? There is thus an urgent need for a comprehensive empirical study comparing and analyzing the effectiveness of the state-of-the-art NL2Code techniques based on a large number of NL descriptions. Such a study is necessary and essential, which can help us find the answers to important questions when designing NL2Code techniques in the future. For instance, which is the most effective NL2Code technique so far and what is the common weakness of existing NL2Code techniques? Moreover, to what extent do existing NL2Code techniques complement each other and whether the integration of them can enhance the performance? Answering such questions can provide practical guidance for studies within this field. Driven by this, Xu *et al.* [62] took the first step in this direction, but their user study is limited in its scale due to the extensive human intervention. Specifically, only 14 functionalities and two NL2Code techniques were investigated.

In this paper, we aim at fill the gap by performing the first large-scale empirical study that evaluates the effectiveness of both code generation and code search techniques collectively under a controlled experiment setting. Specifically, our study covers ten state-of-the-art NL2Code techniques, including six code search techniques and four code generation techniques, on a comprehensive benchmark containing 22K+ natural language queries. Our experiment setting is user-oriented. First, we aim to evaluate how similar the code returned by an NL2Code technique is to the oracle and we use the CodeBLEU metric [46] to compare the similarity between the returned result and the oracle code with respect to the tokens, syntactic structures, and data flows. Second, to mimic real-world scenarios, we remove the oracle code (i.e., the code snippet that corresponds to each query) from the search space of code search techniques. The rationale behind this is that in practice, developers can hardly find exactly what they want from the codebase [6, 7, 15, 62]. Third, we evaluate the effectiveness of current techniques to generate method-level code snippets, which is the most desirable granularity for developers compared with other granularities (e.g., variables and statements) [37, 50].

Our study makes the following important findings:

- F1: The effectiveness of code generation techniques is promising and exceeds that of the non-pre-training code search techniques. However, the state-of-the-art pre-training based code search technique is still the most effective one among the NL2Code techniques.
- F2: Accurately generating program identifiers is a universal challenge for both code search and code generation techniques since they generally achieve relatively low token similarities to the oracle.
- F3: Existing NL2Code techniques complement each other well: if we combine all the ten selected techniques, we can enhance the performance of over 35% with respect to Top-1, compared to the most effective standalone technique.
- F4: Combining code search with code generation or different code search techniques demonstrates promising results.

Moreover, we design a post-processing strategy that re-ranks the results from different techniques based on the number of overlapped tokens with the query. Our results show that such a combination strategy is effective: by combining the most effective code search and code generation techniques, we can gain an effectiveness improvement of 16% and 35% with respect to the top-1 results, compared with each standalone technique. Furthermore, it is also extensible: further effectiveness enhancement could be achieved by involving more techniques for combination.

[1] A related question: https://stackoverflow.com/questions/5767325

2 BACKGROUND AND RELATED WORKS

2.1 Code Search

Code search (CS) aims at helping developers retrieve some implementations that can serve as references for their development activities [9, 38, 47, 61]. Given a natural language (NL) query from the developer, CS searches for the relevant code snippets from a large-scale code corpus. Traditionally, this process is mainly done by the information retrieval technique such as keyword matching [34, 37]. However, these techniques are known to be suboptimal at capturing the semantic relations between code and natural language queries [17]. Later on, researchers have proposed various deep-learning-based approaches to bridge the semantic gaps. For instance, Gu *et al.* [17] propose DeepCS which jointly embeds code snippets and natural language descriptions into a high-dimensional vector space, where code snippets and queries can be matched according to their similarities. Wan *et al.* [57] design a multi-modal attention network that aggregates information from the token sequence, abstract syntax tree (AST), and control flow graph (CFG) for representing programs.

2.2 Code Generation

Code generation aims at directly generating source code according to software requirements [32]. Traditional approaches leverage formal methods to automatically generate source code [19, 60], but the formal specifications are hard to create [32]. With the advances in deep learning, researchers propose to automatically learn the transformations from the requirements to source code. Specifically, Ling *et al.* [29] treat code generation as sequence-to-sequence translation and build a neural network that targets general-purpose programming languages like Python. Dong *et al.* [13] explore the idea of using two decoders in the code generation task, where the first one aims at predicting a rough program *sketch* and the second one fills in the details.

2.3 Pre-Training Techniques

Training a deep learning model from scratch usually needs a large amount of task-specific annotated data, which is hard to collect in practice. To overcome this limitation, pre-training techniques have been proposed in recent years. The core idea is to pre-train a model on one or more self-supervised tasks where large amounts of training data are readily available so that the network weights can encode some commonsense knowledge compared with randomly initialized. After that, with a small amount of task-specific data, the pre-trained models can be fine-tuned in the traditional supervised manner. Recently, researchers have build several pre-trained models for programming language (PL) by using the large amount of bimodal instances of NL-PL pairs (i.e., the source code and its corresponding comments) [18, 36, 59]. A recent study [69] investigated the existing pre-trained models for PL on standalone downstream tasks (i.e., code search and code generation are separately evaluated). In contrast, our study includes both pre-training and non-pre-training techniques and investigates their strengths and weaknesses in a controlled *Natural Language to Code* experiment setting (i.e., using the identical queries and the oracle code is removed from the search space for code search techniques).

Table 1: Selected techniques in this study.

	Code Search	Code Generation
w/o pre-training	`Self-attention` [23], `Tree-LSTM` [55], `GGNN` [68], `Multi-modal` [57]	`Tranx` [66]
w/ pre-training	`CodeBERT` [14], `GraphCodeBERT` [18]	`CodeT5` [59], `NatGen` [10], `SPT-Code` [42]

3 STUDY DESIGN

3.1 Selected Techniques

Over the years, a large number of code search and code generation techniques have been proposed [26, 31]. Therefore, it requires tremendous engineering efforts to evaluate all of them. In this study, we select representative techniques and we leave the exploration of more techniques as our future work. Totally, we use ten NL2Code techniques, including six code search techniques and four code generation techniques. Those techniques can be classified into two types according to whether they use pre-training and Table 1 lists the categorization. The selected techniques have served as the baselines for a number of studies [49, 53, 54, 72] and achieved promising results in recent replication studies [9, 32, 52, 69], and thus they can represent the state of the art well. For instance, through a comprehensive comparison among existing pre-training techniques (e.g., PLBART [2] and CodeGPT [33]), Zeng *et al.* [69] found that GraphCodeBERT and CodeT5 achieve the best performance on code search and code generation, respectively. The following briefly introduces each of the selected techniques.

3.1.1 Code Search. Typically, a code search technique should embed both the code snippet and the query into vectors, after which the relevance between the code and the query can be calculated. For the selected four non-pre-training techniques, the approach used to embed queries is identical: we use an encoder with six Transformer blocks [56] to deal with the natural language token sequence plus with byte-pair encoding (BPE) [16] to split tokens. In the following four paragraphs, we introduce how to embed the code snippets.

Self-attention. This is a baseline proposed by Husain *et al.* [23]. It treats code as token sequences and uses an encoder of the Transformer architecture to embed such sequences. This approach mimics the workflow of the well-known DeepCS [17] (i.e., both approaches treat programs as code tokens) but is expected to establish a more advanced effectiveness baseline, as the Transformer architecture is known to perform well on the long-term dependency problem faced by the Recurrent Neural Networks (RNN) [56], which is used by DeepCS.

Tree-LSTM. Tree-LSTM is an approach that generalizes the Long Short-Term Memory (LSTM) network to tree-structured topologies. Initially, it targeted at capturing the syntactic properties of natural languages [55] and it was firstly applied to the ASTs of programs by Wan *et al.* [57].

GGNN. Zeng *et al.* [68] propose to construct the variable-based flow graph that depicts data and control flows in the program. Such graphs are constructed by transforming the programs into their Intermediate Representations (IRs) [1], extracting the identifiers in each IR instruction as nodes, and building dependencies among nodes. After that, a Gated Graph Neural Network (GGNN) is used to generate the embedding for the graph, which is also the representation of the code.

Multi-modal. With the intuition that aggregating information from multiple modalities of source code can enrich its representation, Wan *et al.* [57] propose MMAN that utilizes the token sequence, the AST, as well as the graph information of a program. In this paper, we rebuild the multi-modal learning model via fusing the three aforementioned approaches (Self-attention + Tree-LSTM + GGNN). It should be noted that the rebuilt multi-modal learning model is supposed to perform better than the vanilla MMAN, since MMAN only involves control flow information while the GGNN approach involves both data and control flow information.

CodeBERT. CodeBERT [14] is a Transformer-based pre-trained model for programming languages like Python and Java. It has two tasks in the pre-training stage which are masked language modeling and replaced token detection. To apply the pre-trained model on the code search task, the representation of a special token `[CLS]` (the beginning token of the input) is used to measure the semantic relevance between the code snippet and query.

GraphCodeBERT. Guo *et al.* [18] take data flow information into consideration in the pre-training stage. In addition to masked language modeling, there are two newly-proposed structure-aware tasks in the pre-training, i.e., edge prediction and node alignment. Then, the workflow of applying the pre-trained model to code search is identical to that of CodeBERT.

3.1.2 Code Generation. ***Tranx.*** Tranx [66] predicts a sequence of actions to construct an AST, based on which the source code is generated. It first defines an abstract syntax description language framework, which is a grammatical formalism of ASTs. Based on that, three types of actions are predicted at each time step to expand the tree until the whole tree is constructed. Note that a number of follow-up studies rely on the grammar rules introduced by Tranx to construct ASTs [24, 53, 54]. Therefore, we select Tranx as the representative technique.

CodeT5. CodeT5 [59] follows the T5 architecture [45] with the input being the sequence of code and text tokens and the output also in a sequential format. One specially-designed pre-training task is NL-PL dual generation in which the model learns to generate code from texts and generate texts for code simultaneously. After pre-training, CodeT5 can be naturally adapted to code generation due to its encoder-decoder framework. A number of follow-up studies rely on the pre-trained parameter values of CodeT5 [27], so we select CodeT5 as the representative pre-training-based code generation approach. The authors of CodeT5 provide two versions of this model, which have different parameter sizes. We use *CodeT5-base* in this study since it is more effective [59].

NatGen. NatGen [10] is designed based on CodeT5 and incorporates an extra pre-training task, "Code Naturalizing", which is designed to teach the model how to transform unnatural code into a more natural, human-written form. This additional task is intended to encourage the model to better understand the underlying semantics of the code, and thus enhance the model's capability on generating code that closely resembles human-written ones.

SPT-Code. SPT-Code [42] is another state-of-the-art pre-trained model with the encoder-decoder framework. The input to the SPT-Code model during the pre-training stage differs from that of CodeT5 in two ways. First, its input includes the Abstract Syntax Tree (AST) of the code, which enables it to leverage syntactic information. Second, to eliminate the need for a bilingual corpus (i.e., a code snippet paired with a corresponding comment), SPT-Code leverages the method name and the names of the methods that are called within that method as a natural language description of the source code being analyzed.

Exclusion. A branch of study focuses on utilizing the retrieval results to guide code generation [20, 21]. We exclude them from this study since (1) the retrieval-and-edit approach [20] assumes that the input and output of the method is already known and the method is partially written, which is unfair to our study subjects (i.e., we do not require prior knowledge); and (2) ReCode [21] is built on top of a set of suboptimal grammar rules, which is not as general as Tranx. We also exclude a recently-proposed code generation approach, PyCodeGPT [67], since it only supports generating code that reuses the third-party libraries *Pandas* and *NumPy*, which is not general enough.

3.2 Dataset

We select the widely-used CodeSearchNet dataset [23] as our evaluation benchmark, which is mined from popular GitHub projects (in terms of the number of stars and forks). In our study, we focus on the bimodal data (i.e., the code snippet and its associated documentation) by treating the documentation as the natural language query and the code snippet as the ground truth. Code in this dataset are all method-level snippets, and our study thus focuses on the effectiveness of existing techniques at the method level. To keep in high-quality, this dataset has already been pre-processed by several steps. For instance, any documentation shorter than three tokens is removed since it might not be informative, and any code snippet shorter than three lines is also removed since it is likely to be getters and setters. We explicitly focus on the Python language in this study since Python is the most widely targeted general-purpose programming language in the code generation domain [32, 65, 66], and our selected Tranx only supports the Python language so far. The dataset has already been split into the training/validation/test sets by the authors of CodeSearchNet, and the training set has been used for pre-training, which means the cross-validation on the whole dataset is inappropriate (doing so will favor pre-training techniques due to the data leakage). Therefore, we evaluate NL2Code techniques on the fixed test set, following existing studies [18, 59]. In the end, our dataset contains 412,178/23,107/22,176 code-query pairs for training/validation/testing, respectively.

3.3 Research Questions

RQ1: How effective are existing techniques on transforming natural language descriptions into source code? We first systematically investigate the effectiveness of each individual technique as summarized in Table 1 on generating method-level code snippets based on natural languages. Beyond the traditional setting where the oracle code snippets are within the search space of code search techniques, we design a new experiment setting in this RQ where we assume the oracle does not exist in the search space and perform the search on the left 22,175 code snippets in the test set. Previous studies have shown that developers usually need to modify the retrieved code snippets to adapt them to the local contexts [6, 7, 15, 62], which means that, in a realistic scenario, the desired code snippets can rarely be directly retrieved. Specifically, Gabel

and Su [15] investigated the syntactic uniqueness of source code and found that redundancy usually exists only at the line level, while at the method level, which is our investigated granularity, near-total uniqueness was observed. More recently, in the user study performed by Xu *et al.* [62], users modified 18 tokens of the retrieved code chunks (several lines of code) on average, and such a number is expected to increase when it comes to the method level. Consequently, our oracle-excluded setting mimics the real application scenario where the users search for the results from a large-scale corpus that does not contain the exactly desired code snippet and see how useful the retrieved results could be. Evaluations under such a user-oriented setting can help us better understand the usefulness of code search techniques in real-world scenarios. This is thus the basic setting of this study and our follow-up investigations are based on the results obtained from this setting.

RQ2: Do different techniques complement each other? In this RQ, we aim to investigate if different techniques tend to perform similarly on the same queries or if they exhibit performance differences on certain queries. As we have introduced, existing techniques can be characterized in different aspects, such as using either search or generation strategy, with or without pre-training. This question investigates whether such differences in the design spaces lead to certain complementarities with respect to their effectiveness. The answer is essential to our further investigations: we could be able to design a combinational strategy to integrate different techniques only if they demonstrate certain complementarities.

RQ3. Can we go beyond the state of the art by combining existing techniques? Based on the experimental outputs obtained from the previous RQs, we further seek to investigate whether combining different techniques can achieve better performance. We propose to investigate two sub-questions:

- **RQ3.1** *What is the best performance achievable by combining different techniques?* We first aim to investigate the best performance achievable via combining different techniques, whose results will pave the way for the following question:
- **RQ3.2** *Can we automatically combine different techniques?* Reaching the best performance requires manually inspecting a number of results, which would be time-consuming. We further seek to design a novel strategy that is able to combine different techniques automatically.

3.4 Effectiveness Assessment

To jointly evaluate code generation and code search, we focus on assessing the similarity between the predicted result and the oracle code. Specifically, we decide to use four metrics following the existing study [10], including **Token match (TM)** which is calculated by the standard BLEU [43] and aims to reflect the similarity between the token sequences of the predicted and oracle code; **Syntax match (SM)** which aims to evaluate code quality from the natural tree structure of programming language (i.e., the AST); **Dataflow match (DM)** which aims to evaluate the semantic information of code through the dependency relations among variables; and **CodeBLEU (CB)** which is a combination of the above three metrics and provides a holistic perspective to the quality of generated code. Readers can refer to [46] for more details about these metrics.

Table 2: Effectiveness of Each Selected Technique (in %).

Techniques	Top-1				Top-5			
	TM	SM	DM	CB	TM	SM	DM	CB
Tranx	2.5	19.2	25.5	12.7	2.7	21.3	28.2	14.0
CodeT5	**9.2**	**27.3**	**39.5**	**22.2**	**10.2**	**29.2**	**42.3**	**23.9**
NatGen	9.5	24.2	36.2	21.0	9.6	25.0	38.1	21.7
SPT-Code	3.4	16.9	37.2	16.3	3.8	18.8	40.8	17.9
Self-attention	32.1	47.6	58.7	42.8	55.8	69.2	79.8	65.4
Tree-LSTM	21.4	39.3	52.1	33.8	43.1	60.4	73.9	55.5
GGNN	21.6	39.5	52.2	34.0	43.8	60.9	74.3	56.0
Multi-modal	34.0	49.0	60.0	44.5	56.7	69.8	80.2	66.1
CodeBERT	60.2	70.4	74.9	66.8	82.0	87.8	90.7	85.9
GraphCodeBERT	**61.6**	**71.5**	**76.0**	**68.1**	**83.1**	**88.6**	**91.4**	**86.8**
Self-attention w/o oracle	1.4	23.8	40.0	16.9	1.7	31.6	54.9	23.0
Tree-LSTM w/o oracle	1.4	23.9	40.1	17.0	1.7	31.7	54.9	23.0
GGNN w/o oracle	1.4	23.9	40.0	17.0	1.7	31.6	54.9	23.0
Multi-modal w/o oracle	1.4	23.8	40.2	17.0	1.7	31.6	54.8	23.0
CodeBERT w/o oracle	9.9	33.5	44.3	25.4	15.9	44.0	56.8	34.7
GraphCodeBERT w/o oracle	**10.2**	**33.9**	**44.7**	**25.8**	**16.4**	**44.5**	**57.1**	**35.1**

The bold name means the technique requires pre-training. The green cell denotes the oracle is excluded from the search space of code search techniques. The optimum performances of generation/search techniques are in bold.

In this study, we calculate the similarity for the top-1 results of each technique as well as the maximum values from the top-5 results. The rationale is that as suggested by the prior study, developers only inspect a few results returned by recommendation tools [44].

3.5 Experiment Setting

All our experiments were performed on a server which possesses 8 NVIDIA Tesla V100 with 32GB memory. Note that to alleviate potential reproducible bias [69], the selected non-pre-training techniques are trained from scratch and the pre-training ones are fine-tuned by ourselves. Since all of our selected techniques open sourced their artifacts on GitHub, we reused the original implementations as well as the values of the hyper-parameters selected for fine-tuning, to avoid potential bugs in our implementation as well as enhance the reliability of our results. Note that initially GGNN [68] targeted C language. To apply it to Python, we use the `Dis` module[2] to generate the IRs, after which the graph can be generated based on the scripts released by the authors. SPT-Code was not evaluated on the code generation task in the original study [42]. To fine-tune it on this task, we reuse the hyperparameters used to fine-tune this model on the code summarization task, which can be considered as another generation task.

4 STUDY RESULTS

4.1 RQ1: Effectiveness of Existing Techniques

For each technique, we calculate the metrics of the code produced by them for each query and the mean values on the whole test set are shown in Table 2. The mean value is one of the most representative statistics and it has been widely used by existing studies to assess the performances of different techniques [10, 53, 59, 69]. From the results, we first note that compared with non-pre-training techniques, pre-training techniques generally achieve better performances. For instance, the CB of *CodeT5* with respect to the top-1 result is 22.2%, which exceeds that of *Tranx* (i.e., 12.7%) by 75%. Similarly, the CB of *CodeBERT* with respect to the top-1 result, when the oracle is excluded from the search space, is 25.4%, which exceeds that of *Multi-modal* (i.e., 17.0%) by around 50%. Furthermore, we

[2] https://pypi.org/project/dis/

note from the results that the most effective NL2Code technique so far is *GraphCodeBERT*, with the CB score of 25.8%. This is within our expectation considering that pre-training techniques require much more data (e.g., *GraphCodeBERT* and *CodeT5* are pre-trained on data from six programming languages) and thus can preserve more domain knowledge [69]. We also note that compared with the other two pre-training-based code generator (i.e., CodeT5 and NatGen), SPT-Code achieves comparatively low performances. One possible explanation, as introduced in Section 3.1, is that the pre-training phase of SPT-Code relies on method names (which may be syntactically incomplete) to approximate the natural language description of the code. This setting could potentially reduce its ability to align natural language and programming language and generate code from natural language inputs.

Finding-1 ☞ *The pre-training techniques achieve better performances than non-pre-training techniques for both code generation and code search. The most effective NL2Code technique so far is GraphCodeBERT with the CodeBLEU of 25.8%.*

We also find that with removing the oracle from the search space, the effectiveness of code search techniques decreases significantly. Specifically, the CB of the state-of-the-art code search technique, i.e., *GraphCodeBERT*, with respect to the top-1 results when the oracle code is involved/excluded is 68.1%/25.8%, respectively, and the other five search techniques undergo the similar decreases when the oracle is excluded. By further investigating its search results, we find that with oracle involved, it can rank the oracle code at top-1 for 12,672 queries, which account for nearly 60% of the total queries (12,672/22,176). That is why it can achieve a high CB with the oracle involved. Among the non-pre-training code search techniques, *Multi-modal* achieves the best performance when the oracle is involved with the CB of 44.5%. However, these four techniques achieve nearly identical effectiveness when the oracle is excluded, with respect to both top-1 and top-5 results. Specifically, their CBs with respect to the top-1 and top-5 results are all around 17.0% and 23.0% respectively. Such results may indicate that there is a gap between the current evaluation of code search techniques and their real usefulness in practice. Indeed, the current evaluation always assumes the existence of the exactly matched code in the search space [17, 52, 57], which amplifies the usefulness of code search techniques. We thus call for a user-oriented evaluation for future studies, that is, to investigate to what extent the retrieved results could help developers when what they exactly need may not be retrieved.

Finding-2 ☞ *If the oracle does not exist in the search space, the effectiveness of code search techniques decreases significantly.*

Another phenomenon we observe is that comparing with focusing only on the top-1 results, the effectiveness of the code search techniques increases significantly if the top-5 results are considered, while that of the code generation techniques nearly remains unchanged. Specifically, without the oracle, the CBs of *GraphCodeBERT* when considering top-1/top-5 results are 26.1%/35.3% respectively, an increase of 35% when all the top-5 results are considered. In contrast, those of *CodeT5* are 22.2% and 23.9% respectively, with only slight enhancement. Such results suggest that the most qualified candidate code snippets are sometimes not ranked at the top-1 positions for code search techniques while the top-1 generated code snippets usually reach the optimum. Therefore, recommending more results from the returned lists to the users could be useful for code search, but such usefulness would not be significant for code generation. Despite that, we find the top-1 results from code generation techniques are already promising: they can be more similar to the oracle code compared with the retrieved results of certain code search techniques. For instance, the CB of *CodeT5* with respect to the top-1 results is 22.2% while those of the four non-pre-training code search techniques are around 17.0%.

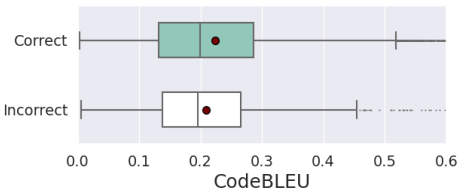


Figure 1: The CBs of the syntactically correct (incorrect) top-1 code snippets generated by *CodeT5*.

Finding-3 ☞ *Unlike code search techniques that sometimes do not rank the best candidates at the top-1 positions, code generation techniques usually predict the optimum results at the top-1 positions, and their effectiveness can exceed that of non-pre-training code search techniques.*

We further investigate the promising results achieved by *CodeT5*, the best performing code generation technique. Since it directly generates the token sequence without any grammatical guideline (unlike *Tranx*), one critical concern is that the generated code might be syntactically incorrect. Our investigation shows that the generated top-1 code snippets are syntactically correct for 19,340 queries, accounting for 87% of the queries in the test set. We also dissect the CBs of these syntactically correct/incorrect code snippets and demonstrate the results in Figure 1. We find that the medians of the CBs of syntactically correct/incorrect code are very similar (22.3% vs. 21.0%) and the differences between the two groups are not statistically significant (i.e., with the p-value > 0.05 in a one-sided Mann-Whitney U-Test [35]). This indicates that being syntactically correct or not does not necessarily affect the metric value of the generated code. It also indicates that more metrics are needed to better reflect the syntax differences of different code. We perform further analysis towards such incorrect cases and find that *CodeT5* often generates a block of code recurrently. The incorrectness happens when the token sequence exceeds a pre-defined length determined by the hyper-parameter (which means *CodeT5* stops generating more tokens) while the current line is not finished. Therefore, such incorrect code still fulfills certain functionalities and thus can have high CBs. We give an example in our online repository.

Finding-4 ☞ *Being syntactically correct or not does not necessarily affect the CBs of code generated by CodeT5.*

We also carefully check our experiment results. We find that our results are generally consistent with those reported in previous

studies. For instance, for four non-pre-training code search techniques, our results show that if the oracle is involved, combining information from multiple modalities can achieve the best performance and the token sequence information is the most rewarding single modality (i.e., *Self-attention* achieves higher CB than *Tree-LSTM* and *GGNN*). This is identical to the phenomenon reported by Wan *et al.* [57]. We note the TM of *Tranx* (2.5%) is significantly lower than the value reported in the previous study [32], which is 18.4%. After further investigation, we find that their dataset is from contest programs in which the identifiers are usually simple but meaningless like `i`, `j`, and `k`. On the contrary, our dataset is from real-world open-source projects in which each identifier is expected to express rich semantic information and thus may be more complex (e.g., camel cases and underscore naming conventions [5]). Since TM, the standard BLEU, focuses on the identifier matching relations, we consider our result as reasonable: it reflects that currently semantic-meaningful identifiers in real-world projects are difficult to predict.

Indeed, our results illustrate that for all the involved techniques, their TMs are significantly lower than their SMs and DMs, indicating that the inability to accurately generate identifiers is a universal weakness of the existing code generation/search techniques (please note the identifier name is ignored when calculating SM and DM). This can be explained by the fact that program identifiers usually demonstrate uniqueness. For instance, Nguyen *et al.* [41] found that more than 60% of the method names occur only once among 14K+ projects. Suppose an identifier in the oracle code is unique, the retrieved results will not match with it, and similarly, the generation techniques seem unlikely to generate it since it may not be involved in the vocabulary from which the output is predicted. A concrete example is shown in Listing 1. In this case, the query expresses the intention to remove quotes from a string. We find that the semantics of the code generated by *CodeT5* is nearly identical to that of the oracle code, except that it fails to check if the string starts and ends with double quotes (in Python, a string can be wrapped with either single or double quotes). Therefore, the SM and DM of *CodeT5* are extremely high, both exceeding 70%. However, *CodeT5* fails to accurately predict the names of the identifiers. For instance, it uses `s` to represent the input string parameter while in the oracle, this identifier is named as `istr`. Since this identifier occurs for many times in the code, the TM of *CodeT5* is thus only 13.1%, a relatively low value. This case also reveals that relying solely on the BLEU value to evaluate the generated code is potentially biased, demonstrating the rationale of a more comprehensive metric like CodeBLEU.

> **Finding-5** ☞ *Producing accurate program identifiers is a universal challenge for both generation and search techniques.*

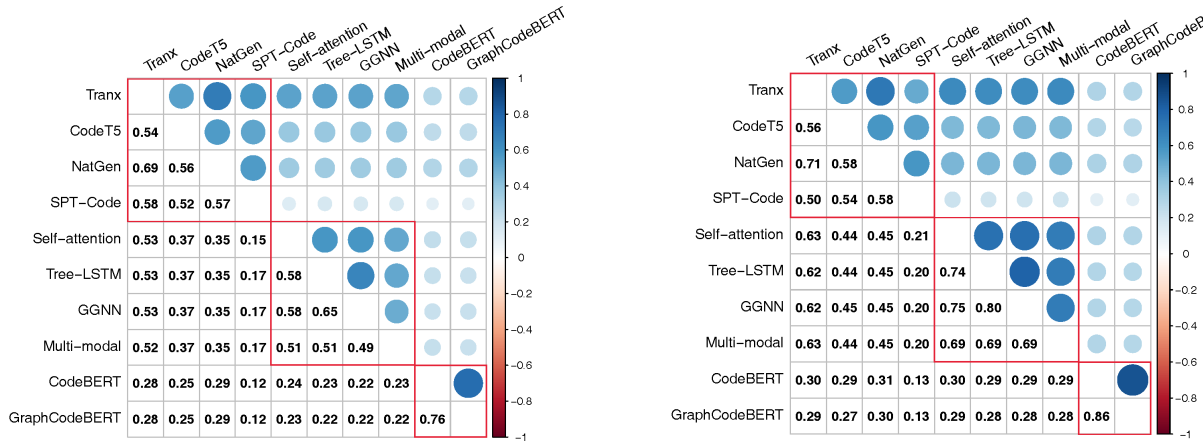
4.2 RQ2: Complementarity of Existing Techniques

To investigate the complementarity of existing NL2Code techniques, for each technique pair, we compute the Pearson correlation (r) [4] with respect to their CBs achieved on each query (we focus on CB here since it represents the overall effectiveness). Pearson correlation is a widely used metric to assess the correlation degree between two sets of data [11, 22]. Theoretically, a high Pearson correlation coefficient suggests that the two sets of data follow a similar trend. In our context, it means two techniques may have similar CBs for a specific query. In contrast, if two techniques have a relatively low Pearson value, it suggests that there is little or no correlation between their CBs. This indicates the potential existence of queries on which the two techniques achieve rather different CBs. In such cases, they could be considered as complementary to each other. For instance, if two techniques exhibit identical performances on each query, their Pearson value would reach the maximum value of 1. However, they may not complement each other well because they share similar effectiveness towards the same inputs. Our interpretation of r is based on the previous study [22]: negligible correlation ($|r| < 0.3$), low correlation ($0.3 \leq |r| < 0.5$), moderate correlation ($0.5 \leq |r| < 0.7$), high correlation ($0.7 \leq |r| < 0.9$), and very high correlation ($0.9 \leq |r| < 1$).

```
# Code generated by CodeT5
def _remove_quotes(s):
    if s[0] == "'" and s[-1] == "'":
        return s[1:-1]
    else:
        return s

# Oracle code
def unquote_ends(istr):
    if not istr:
        return istr
    if (istr[0]=="'" and istr[-1]=="'") or \
       (istr[0]=='"' and istr[-1]=='"'):
        return istr[1:-1]
    else:
        return istr

# Code retrieved by GraphCodeBERT
def strip_email_quotes(text):
    lines = text.splitlines()
    matches = set()
    for line in lines:
        prefix = re.match(r'^(\\s*>[ >]*)', line)
        if prefix:
            matches.add(prefix.group(1))
```

Listing 1: The code generated by *CodeT5*, the oracle code, and the code retrieved by *GraphCodeBERT* for the query "Remove a single pair of quotes from the endpoints of a string".

Results are shown in Figure 2. We observe that according to the Pearson correlation values, the selected techniques can generally be classified into three clusters as highlighted: the code generation techniques, the non-pre-training code search techniques, and the pre-training code search techniques. For techniques in each cluster, they have a relatively high correlation with each other, and a relatively low correlation with those from other clusters. Specifically, in Figure 2a, *Tranx* and *CodeT5* have moderate Pearson correlation between them (i.e., 0.54). Similarly, *CodeBERT* and *GraphCodeBERT* have high Pearson correlation (i.e., 0.76). As for the four non-pre-training code search techniques, they all have moderate Pearson correlation between each other (e.g., the value between *Tree-LSTM* and *GGNN* is 0.65). In contrast, the Pearson correlation between cross-cluster techniques is usually low or negligible (e.g., the value between *Multi-modal* and *GraphCodeBERT* is 0.22), demonstrating the effectiveness of such techniques is weakly correlated. We also note that the highest Pearson value (i.e., 0.76 between *GraphCodeBERT* and *CodeBERT*) is still lower than 0.9 (the threshold of the very high correlation degree). This indicates that the effectiveness



(a) Pearson correlation for CBs of the top-1 results

(b) Pearson correlation for CBs of the top-5 results

Figure 2: Pearson correlation results for the selected techniques.

of two highly-correlated techniques may still differ to a certain degree on specific queries. We also observe the similar trend in Figure 2b. Such results indicate that complementarities widely exist among existing techniques.

Figure 3 illustrates the relationship among three representative techniques of each cluster (the ones with the highest effectiveness in each cluster, i.e., *GraphCodeBERT*, *CodeT5*, and *Multi-modal*) via a Scatter plot. The x and y values of each scatter denote the CodeBLEU values of two different techniques achieved on a specific query. We investigate both top-1 and top-5 results and find similar trends, so we only show the top-1 results here. For ease of comparison, we also draw the line `y=x` in the figure. Scatters above this line represent that the technique denoted by the vertical axis outperforms the technique denoted by the horizontal axis on those queries and vice versa. We observe that no technique can consistently outperform the other competitor: even the most effective one, *GraphCodeBERT*, can still perform worse on specific queries, compared with *CodeT5* or *Multi-modal*. This further shows the complementarity of the existing techniques.

> **Finding-6** ☞ *Existing NL2Code techniques are complementary since (1) they can generally be classified into three clusters with high intra-cluster Pearson correlations and low inter-cluster Pearson correlations; and (2) no technique can consistently outperform the others on all the queries.*

Case analysis. To demonstrate the complementarity of existing techniques, we analyze two cases here. The first is shown in Listing 1 where we also list the retrieved result from *GraphCodeBERT* for the same query. Due to space limitation, we only show the first several lines. We recall that *CodeT5* achieves a high CB on this query (i.e., higher than 50%). We note that both the syntactic structure and the tokens of the code returned by *GraphCodeBERT* are very dissimilar to the oracle. For instance, the oracle code uses an if-else structure while the code returned by *GraphCodeBERT* contains a loop structure. Therefore, the CB of *GraphCodeBERT* on this query is only 15%, which is much lower than that of *CodeT5*.

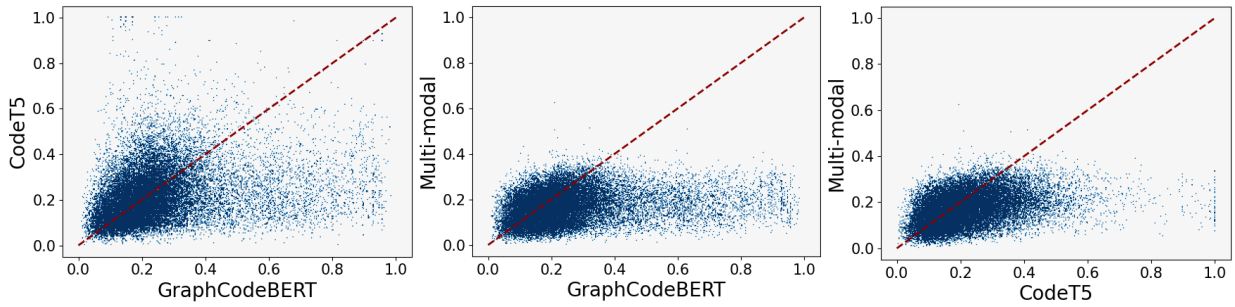
Another example is shown in Listing 2. The intended functionality is to migrate data from one dataset to another. The oracle code fulfills this by checking if the ID of the source dataset is provided,

```
# Code generated by CodeT5
def migrate(self, target, **kwargs):
    if 'commit_mode' not in kwargs:
        kwargs['commit_mode'] = self.commit_mode
    if 'commit_mode' not in kwargs:
        kwargs['commit_mode'] = self.commit_mode
    return self._migrate(target, **kwargs)

# Oracle code
def migrate(self, target, follow=True, **kwargs):
    if 'id' not in self or not self['id']:
        raise Exception('No source dataset ID found.')
    if isinstance(target, Dataset):
        target_id = target.id
    else:
        target_id = target
    migration = DatasetMigration.create(source_id=self['id'],
        target_id=target_id, **kwargs)
    return migration

# Code retrieved by GraphCodeBERT
def migrate(self, target, follow=True, **kwargs):
    if isinstance(target, Dataset):
        target_id = target.id
    else:
        target_id = target
    limit = kwargs.pop('limit', None)
    params = self._build_query(limit=limit)
    migration = DatasetMigration.create(source_id=self._dataset_id,
        target_id=target_id, source_params=params, **kwargs)
    return migration
```

Listing 2: The code generated by *CodeT5*, the oracle code, and the code retrieved by *GraphCodeBERT* for the query "Migrate the data from this dataset to a target dataset".

obtaining the ID of the target dataset, and finally performing the migration. The code retrieved by *GraphCodeBERT* is only slightly different from the oracle code since it initializes a variable which is not used by the oracle code during migration (i.e., `params`). The code generated by *CodeT5* differs significantly to the oracle code since (1) it does not perform the sanity check, (2) it generates a block of code recurrently as we have mentioned before, and (3) it does not rely on the *DatasetMigration* package to perform the migration. Consequently, the CB of *GraphCodeBERT* on this query is much higher than that of *CodeT5* (57.1% vs. 29.2%).

These two cases demonstrate that different techniques perform well on different queries and thus complement each other.



(a) GraphCodeBERT vs. CodeT5 (b) GraphCodeBERT vs. Multi-modal (c) CodeT5 vs. Multi-modal

Figure 3: Scatter plots of CodeBLEUs of three representative techniques with respect to the top-1 results. We also draw the line `y=x` for comparison.

Table 3: The highest CodeBLEU values achievable by combining different strategies (in %).

Combinations	CB (Top-1)	CB (Top-5)
CodeT5	22.2	23.9
Multi-modal	17.0	23.0
GraphCodeBERT	25.8	35.1
CodeT5 + Tranx	22.4	24.1
Multi-modal + Self-attention	19.6	24.8
GraphCodeBERT + CodeBERT	28.4	37.0
GraphCodeBERT + CodeT5	30.2	37.4
GraphCodeBERT + Multi-modal	27.5	36.0
CodeT5 + Multi-modal	24.1	27.6
GraphCodeBERT + CodeT5 + Multi-modal	31.0	38.0
GraphCodeBERT + CodeT5 + CodeBERT	32.0	38.9
All (10 techniques)	35.1	40.8

4.3 RQ3: Combination of Existing Techniques

4.3.1 RQ3.1: what is the best performance achievable by combining different techniques? To investigate this RQ, for each query, we suppose all the results from a set of techniques can be inspected by the developer and the most qualified code (the one with the highest CB score) can be identified and used as the final result of such a technique combination. We calculate the overall performance on the whole test set obtained in such a manner and results are shown in Table 3. We first observe that several techniques together can work better than standalone techniques, which shows that combinations of different techniques are promising. Specifically, if we take all the ten techniques into consideration, the CB of the Top-1 results can reach 35.1%, outperforming the best search and generation techniques (i.e., *GraphCodeBERT* and *CodeT5*) by 36% and 58%, respectively.

Finding-7 ☞ *Combining the ten techniques can gain at least 35% effectiveness enhancement compared with standalone techniques.*

Obtaining the results of all the eight techniques, however, requires much computation resource, which may not be affordable in practice. Therefore, we also investigate the effectiveness of combining a pair of techniques. Specifically, we combine techniques from the same clusters (e.g., *CodeT5 + Tranx* shown in the second part) and techniques across different clusters (e.g., *GraphCodeBERT + CodeT5* shown in the third part). Surprisingly, we find that although the latter is more effective than the former in general (e.g., combining *CodeT5* with *Multi-modal* works better than combining it with *Tranx*), combining *GraphCodeBERT* with *CodeBERT* is the most effective way for search-search intra-combinations: such a strategy can achieve higher CBs than combining *GraphCodeBERT* with *Multi-modal* and its CB with respect to the top-5 results nearly equals to that of *GraphCodeBERT + CodeT5* (37.0% vs 37.4%). This could be explained through Figure 3 where we note that for the subfigure comparing *GraphCodeBERT* and *Multi-modal*, the majority of the scatters are below the line `y=x`, which means the latter only outperforms the former on a limited set of queries. As a result, combining these two techniques may not boost the effectiveness to a large extent, although they have relatively low Pearson correlation. Similarly, we also try to combine three representative techniques from different clusters (shown as *GraphCodeBERT + CodeT5 + Multi-modal*) but this is still outperformed by replacing *Multi-modal* with *CodeBERT*. Consequently, if we are able to use only two techniques under a resource-constrained situation, search-generation inter-combination of *GraphCodeBERT* with *CodeT5* and search-search intra-combination of *GraphCodeBERT* with *CodeBERT* can provide promising results.

Finding-8 ☞ *Search-generation inter-combination of GraphCodeBERT with CodeT5 and search-search intra-combination of GraphCodeBERT with CodeBERT show promising results.*

4.3.2 RQ3.2: can we automatically combine different techniques? To achieve an automatic combination, we design a post-processing strategy where we re-rank results obtained from different techniques to generate the final outputs, inspired by a recent study [70].

Intuition. To achieve our target, we need a predictor to assess the quality of each generated code snippet. Recall that one of our observations is that existing techniques usually have relatively poor performance towards TM (cf. Table 2). That is to say, if a generated code snippet has a high TM value, it is likely to achieve good overall performance (i.e., CB). Inspired by previous studies which point out that query tokens may represent key concepts in the requirements [30, 37], we postulate that a code snippet with more overlapped tokens with the query may contain more meaningful identifier names and thus has higher value towards TM (so as CB). For instance, the code to implement the functionality required by the query "convert string to int" needs to include the API `int()` and it thus contains the overlapped token `int`. Given a query, we denote its number of tokens as NUM_t and the number of its tokens contained in a generated code snippet as NUM_o. We propose to rely on the *overlap degree*, which is calculated as $\frac{NUM_o}{NUM_t}$, to help assess the quality of the generated code snippet: a code snippet with a higher overlap degree is considered to be more qualified.

Table 4: The CodeBLEU values achieved by different combinations using our strategy (in %).

Combinations	CB (Top-1)	CB (Top-5)
GraphCodeBERT + CodeBERT	28.3/28.4	36.5/37.0
GraphCodeBERT + CodeT5	30.0/30.2	36.9/37.4
GraphCodeBERT + CodeT5 + CodeBERT	31.8/32.0	37.9/38.9

Specifically, to perform such analysis, the code and query are tokenized by the NLTK package and program identifiers are further split into multiple tokens based on the camel cases and underscore naming conventions.

Hypothesis Validation. To validate our intuition, we split the overlap degree into five different intervals and calculate the CBs of the top-1 code snippets returned by different techniques whose overlap degrees fall in each interval.

Results are shown in Figure 4. We note that code snippets with higher overlap degrees are generally more similar to the oracle code (with higher CBs), for all three representative techniques. Specifically, when the overlap degree is in the [0.8, 1] interval, the median value of the CBs of the top-1 results returned by *GraphCodeBERT* is around 40%, nearly as twice as that of the code snippets whose overlap degree is in the [0, 0.2) interval (which is only around 20%). We also perform the one-sided Mann-Whitney U-Test [35] to analyze the statistical significance of the CB differences for code snippets from adjacent intervals. Our Null hypothesis is that **H0: code with higher overlap degrees to the query will not achieve significantly higher CBs**, and the Alternative hypothesis is **H1: code with higher overlap degrees to the query will achieve significantly higher CBs**. Results reveal that the differences are statistically significant (i.e., p-value < 0.05) under all the cases, indicating that **H0** can be rejected with a confidence level of over 0.95. Such results indicate that the overlap degree with the query could be a competent indicator to re-assess the quality of the code snippets returned by existing techniques.

Strategy. Motivated by our validation, we design a combination strategy to integrate the results from different techniques whose overall process is straightforward. Given a natural language description (i.e., the query), different techniques are executed and their results are stored into a candidate code snippet pool. After that, we use a heuristic that assesses the overlap degree between the query and each candidate code snippet to re-rank those candidates: code snippets possessing high overlaps with the query are ranked at the top positions. Consequently, results from different techniques are re-ranked together and integrated into one list at this step, and the output is the final combination result. In this study, to keep reasonable trade-offs between the effectiveness and efficiency, we combine the top-5 results of each selected technique.

Evaluation Results. To investigate the effectiveness of our proposed combination strategy, we select the three representative techniques (i.e., *GraphCodeBERT*, *CodeBERT*, and *CodeT5*) identified through our analysis in Section 4.3, and evaluate the performances after combining two or all of them. Results are shown in Table 4 where the data in the format "x/y" denotes the effectiveness obtained by our strategy/the best performance achievable by different combinations.

We find that our combination strategy is generally effective: all the combinations can nearly reach their maximum potential. For instance, if for each query, the maximum CodeBLEU value from *GraphCodeBERT* and *CodeT5* is achieved, then the average CodeBLEU value of the top-1 results is 30.2%. By using our strategy, such a combination can have a CodeBLEU of 30.0% with respect to the top-1 results. Moreover, given the data in Table 2, such an automatic combination can outperform each standalone technique by 16% (30.0% vs. 25.8%) and 35% (30.0% vs. 22.2%), respectively. We also note that search-generation inter-combination works more effectively than search-search intra-combination: the combination of *GraphCodeBERT + CodeT5* achieves higher CodeBLEUs than the combination of *GraphCodeBERT + CodeBERT* with respect to both top-1 and top-5 results, especially when we only focus on the top-1 results (30.0% vs. 28.3%). This indicates that in a resource constrained scenario where we can only execute a few techniques (e.g., two), combining code search and code generation techniques is recommended. Furthermore, our strategy is also extensible: the effectiveness of the combination keeps increasing when involving more techniques. Specifically, the CodeBLEU value of the top-1 results increases by nearly two percentage points when all three representative techniques are combined, compared with only considering two of them (31.8% vs. 30.0%). As a result, further effectiveness enhancement is expected when involving more techniques.

> **Finding-9** ☞ *A simple heuristic-based post-processing strategy can lead to significant effectiveness enhancement compared with each standalone technique.*

5 DISCUSSION

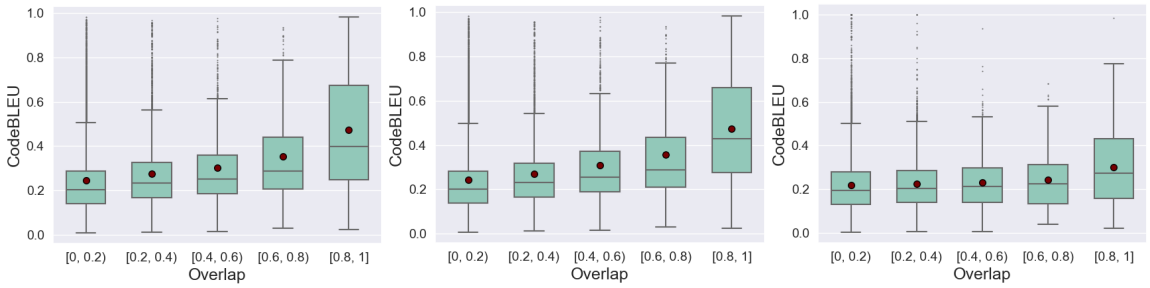
5.1 Implications: It Takes Two to Tango

Our investigation shows that code search and code generation techniques share certain complementarities: a query that is not handled effectively by one technique may be addressed well by the other. Therefore, *developers* may consider using both of them in their development activities to boost their productivity. Our study proposes a post-processing approach for combining these two types of techniques. In fact, we also explore a pre-processing way for combination where we train a model to predict whether a search or generation technique should be used for a given query. Specifically, we use a pre-trained BERT model to embed the query and train a fully-connected layer to predict if *GraphCodeBERT* or *CodeT5* is to be used (as a preliminary exploration, we focus on combining the most effective search and generation techniques), but the accuracy is only 60% on our dataset. Therefore, for *researchers*, efforts could be devoted to devise more effective way for combination in the future.

5.2 Comparison with ChatGPT

ChatGPT is a hot chatbot that can interact with humans in a conversational way.[3] To compare it with the study subjects in this paper, we also investigate its code generation performance on our test set. To perform this experiment, we leverage the ChatGPT API (accessed on May 9, 2023) with the prompts to the model in the form of "Assume that you are a Python programmer. Please write a Python function that ...", followed by the query contents. The

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



(a) GraphCodeBERT **(b) CodeBERT** **(c) CodeT5**

Figure 4: The performances of three representative techniques under different overlap degrees.

first sentence is to prepare ChatGPT for the code generation task while the second one describes the detailed requirement. We set the *temperature* parameter to be 0, which ensures that ChatGPT always return the code with the highest probability. Results show that on average, the CodeBLEU score of the code returned by ChatGPT is 21.1%, which is slightly lower than that of the state-of-the-art code generation techniques such as CodeT5. Such results indicate that although ChatGPT can provide detailed instructions to developers, its performance may not exceed those of the state-of-the-art NL2Code techniques if we only focus on the generated code. One possible explanation could be that the state-of-the-art code generation techniques have been adequately fine-tuned for this task, whereas ChatGPT is optimized for artificial general intelligence (AGI) and not specifically optimized for the task of code generation.

5.3 The Existence of Code Similar to the Oracle

In this study, to mimic the real scenario of applying code search techniques, we remove the oracle code from the search space for each query. One following question is that is there any code snippet in the search space similar to the oracle one? To investigate such a question, we utilize a state-of-the-art code clone detector, NIL [40], to identify code clone pairs among our test set. We recall that NIL is a token-based clone detector since it identifies code clones based on the N-gram representation and the longest common subsequence of code token sequences. That is to say, the detected clones of the oracle code can be transformed to the oracle through minor modifications on their code tokens. Results show that more than 75% (i.e., 17,068/22,176) of the code snippets have the corresponding clones in the search space. This indicates that for most queries, code snippets that can match the query with minor modifications exist in the search space and a qualified code search technique is supposed to rank such code snippets at top positions. By analyzing the search logs of developers, the previous study [47] concludes that developers sometimes get nothing from their searches. This observation suggests that in a realistic setting, it is not always possible for all the queries to have code snippets that are similar to the oracle, as otherwise developers could always obtain a solution by making minor modifications to the oracle's clones. Our setting is aligned with this assumption and is well-suited for practical scenarios.

5.4 Threats to Validity

External Threats. Code search and code generation are active research fields with a number of approaches being proposed during the last years. It is thus quite hard to involve all of them in this study. The selected approaches in this paper are state-of-the-art and have served as baselines for many studies [32, 52], and thus can be considered as representative ones safely.

Internal Threats. In our study, we use the code comment as the query, which is widely adopted by existing studies [28, 49, 57, 59, 63]. The rationale is that the comment usually summarises the main functionality of the code, making the code-comment pair close to actual use scenarios. Existing studies have shown that common queries from developers are similar to the comments (i.e., either being identical to the comment or by slightly prepending the comment with "how to"/"how do I") [17, 34].

We rely on the CodeBLEU score to serve as a proxy of code quality, following existing studies [10, 33, 69, 71]. The previous study [46] has demonstrated that CodeBLEU is strongly related with human evaluations, which means code with higher CodeBLEU scores is more qualified to fulfill the intended functionality, as judged by humans. Our case analysis also shows that code with higher CodeBLEU scores is more semantically similar to the oracle code. As a result, we leave assessing the usefulness of the generated code from the developers' perspective as our future work.

6 CONCLUSION

In this paper, we evaluate the effectiveness of ten representative NL2Code techniques on a large-scale dataset. Through in-depth analysis of their correlation degrees and case analysis, we show that existing NL2Code techniques complement each other well. We also investigate the theoretical upper-bound effectiveness which can be achieved by combining different techniques and find that it outperforms those of standalone techniques to a large extent. Therefore, future studies could be undertaken to further utilize the complementarity of NL2Code techniques. Moreover, we design a strategy to automatically combine results from different techniques and achieve promising results. All code and data in this study are publicly available at: **https://doi.org/10.5281/zenodo.7546358**.

ACKNOWLEDGMENTS

The authors greatly thank the anonymous reviewers for their constructive comments. This work is supported by the National Natural Science Foundation of China No.61932021 and No.62002125, the Young Elite Scientists Sponsorship Program by CAST (Grant No.2021QNRC001), and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 949014).

REFERENCES

[1] [n.d.]. Intermediate representation. https://en.wikipedia.org/wiki/Intermediate_representation.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[3] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *Comput. Surveys* 51, 4 (2018), 81:1–81:37. https://doi.org/10.1145/3212695

[4] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. In *Noise reduction in speech processing*. Springer, 1–4.

[5] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. 2009. To camelcase or under_score. In *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 158–167.

[6] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 513–522.

[7] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1589–1598.

[8] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 511–521.

[9] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.

[10] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar Devanbu, and Baishakhi Ray. 2022. NatGen: Generative pre-training by "Naturalizing" source code. In *Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM.

[11] Deborah Coughlin. 2003. Correlating automated and human assessments of machine translation quality. In *Proceedings of Machine Translation Summit IX: Papers*.

[12] Luca Di Grazia and Michael Pradel. 2022. Code Search: A Survey of Techniques for Finding Code. *arXiv preprint arXiv:2204.02765* (2022).

[13] Li Dong and Mirella Lapata. 2018. Coarse-to-Fine Decoding for Neural Semantic Parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 731–742.

[14] 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. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[15] Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 147–156.

[16] Philip Gage. 1994. A new algorithm for data compression. *C Users Journal* 12, 2 (1994), 23–38.

[17] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[18] 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*.

[19] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. 1990. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on software engineering* 16, 4 (1990), 403–414.

[20] Tatsunori B Hashimoto, Kelvin Guu, Yonatan Oren, and Percy S Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. *Advances in Neural Information Processing Systems* 31 (2018).

[21] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-Based Neural Code Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.

[22] Xing Hu, Qiuyuan Chen, Haoye Wang, Xin Xia, David Lo, and Thomas Zimmermann. 2021. Correlating Automated and Human Evaluation of Code Documentation Generation Quality. *ACM Transactions on Software Engineering and Methodology* (2021).

[23] 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).

[24] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping Language to Code in Programmatic Context. In *EMNLP*.

[25] Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.

[26] Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–38.

[27] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating Code Review Activities by Large-Scale Pre-Training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. 1035–1047. https://doi.org/10.1145/3540250.3549081

[28] Chunyang Ling, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2020. Adaptive deep code search. In *Proceedings of the 28th International Conference on Program Comprehension*. 48–59.

[29] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Fumin Wang, and Andrew Senior. 2016. Latent Predictor Networks for Code Generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 599–609.

[30] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery* 18, 2 (2009), 300–336.

[31] Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. 2021. Opportunities and challenges in code search tools. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–40.

[32] Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. 2020. Deep Learning Based Program Generation from Requirements Text: Are We There Yet? *IEEE Transactions on Software Engineering* 01 (2020), 1–1.

[33] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[34] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.

[35] Henry B Mann and Donald R. Whitney. 1947. On a Test of Whether One of Two Random Variables Is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[36] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[37] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. 111–120.

[38] Collin McMillan, Negar Hariri, Denys Poshyvanyk, Jane Cleland-Huang, and Bamshad Mobasher. 2012. Recommending source code for use in rapid software prototypes. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 848–858.

[39] Parastoo Mohagheghi and Reidar Conradi. 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering* 12, 5 (2007), 471–516.

[40] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. 2021. NIL: large-scale detection of large-variance clones. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–841.

[41] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting Natural Method Names to Check Name Consistencies. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1372–1384.

[42] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2006–2018. https://doi.org/10.1145/3510003.3510096

[43] 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.

[44] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.

[45] 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. *Journal of Machine*

Learning Research 21 (2020), 1–67.

[46] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[47] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 191–201.

[48] Jiho Shin and Jaechang Nam. 2021. A Survey of Automatic Code Generation from Natural Language. *Journal of Information Processing Systems* 17, 3 (2021), 537–555.

[49] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving code search with co-attentive representation learning. In *Proceedings of the 28th International Conference on Program Comprehension*. 196–207.

[50] Susan Elliott Sim, Charles LA Clarke, and Richard C Holt. 1998. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*. IEEE, 180–187.

[51] Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code Search based on Context-aware Code Translation. In *Proceedings of the 44th International Conference on Software Engineering*.

[52] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the Importance of Building High-quality Training Datasets for Neural Code Search. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. ACM.

[53] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 7055–7062.

[54] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.

[55] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 1556–1566.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[57] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 13–25.

[58] Shangwen Wang, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Yan Lei, and Xiaoguang Mao. 2023. Two Birds with One Stone: Boosting Code Generation and Code Search via a Generative Adversarial Network. *Proceedings of the ACM on Programming Languages* OOPSLA2 (2023).

[59] 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. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[60] Michael W Whalen. 2000. High-integrity code generation for state-based formalisms. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*. IEEE, 725–727.

[61] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* 22, 6 (2017), 3149–3185.

[62] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47.

[63] Ling Xu, Huanhuan Yang, Chao Liu, Jianhang Shuai, Meng Yan, Yan Lei, and Zhou Xu. 2021. Two-Stage Attention-Based Model for Code Search with Textual and Structural Features. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 342–353.

[64] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. Coacor: Code annotation for code retrieval with reinforcement learning. In *The world wide web conference*. 2203–2214.

[65] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 440–450.

[66] Pengcheng Yin and Graham Neubig. 2018. TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 7–12.

[67] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: Continual Pre-Training on Sketches for Library-Oriented Code Generation. *arXiv preprint arXiv:2206.06888* (2022).

[68] 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).

[69] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An Extensive Study on Pre-trained Models for Program Understanding and Generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM.

[70] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. CoditT5: Pretraining for Source Code and Natural Language Editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM.

[71] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022. XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence. arXiv:2206.08474 https://arxiv.org/abs/2206.08474

[72] Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. 2020. OCoR: an overlapping-aware code retriever. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 883–894.

Received 2023-03-02; accepted 2023-07-27