2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 979-8-3503-3066-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SANER60148.2024.00018

Guiding ChatGPT for Better Code Generation: An Empirical Study

Chao Liu[1], Xuanlin Bao[1], Hongyu Zhang[1], Neng Zhang[2], Haibo Hu[1], Xiaohong Zhang[1], and Meng Yan[1,*]
[1]*School of Big Data and Software Engineering, Chongqing University, China*
{liu.chao, baoxuanlin, hyzhang, haibo.hu, xhongz, mengy}@cqu.edu.cn
[2]*School of Software Engineering, Sun Yat-sen University, China*
zhangn279@mail.sysu.edu.cn

***Abstract*—Automated code generation is a powerful technique for software development, which can significantly reduce developers' effort and time for writing code. Recently, OpenAI's large language model ChatGPT has emerged as a powerful tool for generating human-like responses to a wide range of textual inputs (i.e., prompts), including those related to code generation. However, the effectiveness of ChatGPT in code generation is still not well understood. The code generation performance could also be heavily influenced by the choice of prompts, which should be further explored. In this paper, we report an empirical study on ChatGPT's capabilities for two types of code generation tasks, namely text-to-code and code-to-code generation. We investigate different types of prompts by leveraging the chain-of-thought strategy with multi-step optimizations. Our empirical results show that by carefully designing prompts to guide ChatGPT, the code generation performance can be improved substantially. We also analyze the factors that influence the prompt design and provide insights that could guide future research.**

***Index Terms*—ChatGPT, code generation, prompt engineering**

I. Introduction

Code generation is a technique that aims to automatically generate code based on developers' requirements [1], [2]. It can reduce repetitive coding effort and improve software development productivity [3], [4]. The requirements can be expressed as natural language (NL) descriptions, allowing developers to specify their needs in an intuitive way. For instance, a developer can ask a code generation tool to fulfill the requirement *"convert an integer variable n to a string in Java"*, and the tool will generate an appropriate code snippet such as: *"String s = Integer.toString(n)"*. This process is known as Text-to-Code (T2C) generation [5], [6]. Another type of code generation is Code-to-Code (C2C) generation, which translates an existing code snippet from one programming language to another [5], [7]. For instance, the C# code *"String s = n.ToString()"* can be translated to the above Java code. The C2C generation can be useful when porting existing code to a new programming language [7].

Large language models (LLMs) have emerged as a powerful tool for natural language processing (NLP) tasks, such as sentiment analysis [8], [9] and language translation [10]. These models are pre-trained on massive amounts of unsupervised textual data and fine-tuned on task-specific datasets. This *"pre-train, fine-tune"* paradigm has been applied to many software engineering (SE) tasks with promising results. For instance, Feng et al. [11] developed CodeBERT, an LLM that has a similar architecture to BERT [12] and is pre-trained on six programming languages [12]. CodeBERT can be used for various SE tasks, such as code search and summarization, with good performance. Another notable model is CodeGPT developed by Lu et al. [5], which is pre-trained on Python and Java datasets using the GPT-2 [13] architecture and fine-tuned for various SE tasks such as code generation and translation.

ChatGPT, developed by OpenAI, is a revolutionary LLM based on the GPT-3.5 architecture [14]. It can work on various tasks including code generation. Different from existing LLMs, ChatGPT is able to generate human-like responses through reinforcement learning based on users' textual inputs (i.e., prompts). Owing to its effectiveness on various tasks, ChatGPT has attracted 100 million active users worldwide within just two months after its initial release [15]. However, the performance of ChatGPT is highly dependent on the quality of the prompts used. Designing better prompts, called prompt engineering [16], is under active investigation. In this paper, we report an empirical study that investigate the capability of ChatGPT in code generation with various prompt engineering methods.

We have conducted an evaluation of ChatGPT's code generation capabilities using the widely used CodeXGlue [5] dataset for both T2C and C2C generation tasks. We used the GPT-3.5-Turbo model by invoking ChatGPT API [17]. Initially, we employed basic prompts for the tasks: *"write a Java method that" + NL description* for the T2C task, and *"translate C# code into Java code:" + code* for the C2C task. Experimental results showed that these prompts (ChatGPT-task) achieved CodeBLEU scores of 22.76 and 39.37, respectively, where CodeBLEU is a widely used overall evaluation metric [18]. To improve the generation performance, we leveraged the chain-of-thought strategy with manual construction [19] to augment the prompts for T2C and C2C tasks. This approach conducts multi-step optimizations based on the feedback from ChatGPT. Our results showed that: 1) adding more specific requirements to the prompts improved the CodeBLEU of ChatGPT-task by 73.58% and 3.45% for the two tasks, respectively; 2) directly asking ChatGPT to generate concise code in the prompt led to further improvement in CodeBLEU for the T2C task, reaching to 50.18; 3) sharing a ChatGPT session for different testing also boosted the CodeBLEU of the C2C task to 48.80; and 4) the generation randomness of ChatGPT had little effect on the generation performance due to the specific instructions in the prompt.

2640-7574/24/$31.00 ©2024 IEEE
DOI 10.1109/SANER60148.2024.00018

Therefore, carefully designed prompts can guide ChatGPT to achieve significantly better code generation performance. Furthermore, we compared the performance with state-of-the-art fine-tuned LLMs and analyzed the correctness and quality of the generated code.

In summary, the major contributions of this paper are:

- Conducting an empirical study on how to guide ChatGPT to generate better code with prompt engineering.
- Evaluating ChatGPT on a widely-used dataset CodeXGlue for two types of code generation tasks.
- Releasing a replication package[1] for future exploration in this research direction.

II. Background and Related Work

A. Large Language Model for Code Generation

Many language models (LMs) have been proposed, which are pre-trained with a special objective (e.g., masked language modeling [20]) and applied to downstream tasks by fine-tuning. Generally, there are three types of LMs: *1) Masked LM,* a model is trained to predict a masked word in a sentence given its surrounding contexts, such as BERT [12] and RoBERTa [21]. *2) Encoder-Decoder,* a model works for sentence-to-sentence tasks like translation and summarization, where an encoder encodes the input into a fixed-length vector and a decoder generates output from the encoded vector, such as T5 [13], BART [22], and MASS [23]. *3) Left-to-Right LM,* a model is trained to predict the next word in a sentence given the previous words, such as GPT [24], GPT-2 [13], and GPT-3 [25]. For these LMs, Transformer [26] is used as the base model because its self-attention layers can efficiently process input with long-term memory and effectively adapt itself to various downstream tasks [27].

Researchers have proposed many LM-based models that can be used for code generation tasks. The representatives are: *1) BERT-Based.* CodeBERT [11] trained a BERT-like model with six programming languages. GraphCodeBERT [4] is an improved model that considers the inherent structure of code instead of plain text as CodeBERT. UniXcoder [28] addressed the difficulty in learning code structure by transforming the code into a sequence but retaining the structural information. ContraBERT [29] leveraged contrastive learning [30] to improve the robustness of CodeBERT and GraphCodeBERT. *2) T5-Based.* Mastropaolo et al. [31] showed that fine-tuning T5 is possible to work on SE tasks. CodeT5 [32] is an identifier-aware T5 model that can distinguish which code tokens are identifiers and recover them when they are masked. *3) BART-Based.* PLBART [33] is constructed on BART pre-trained with an extensive collection of Java and Python functions and associated NL text via denoising autoencoding. CommitBART [7] pre-trained BART using data collected from GitHub commits. *4) GPT-Based.* GPT-C [34] is a variant of GPT-2 pre-trained on a large unsupervised multilingual source code dataset. CodeGPT [5] pre-trained GPT-2 on Python and Java corpora from the CodeSearchNet [35]. CodeGen [6] presents a family of architectures similar to GPT-3 designed for multi-turn program synthesis. CodeX [36] fine-tuned GPT-3 on publicly available code from GitHub, whose distinct production version powers the GitHub Copilot [37].

[1]Replication Package: https://github.com/BaoBaoGitHub/guiding-chatgpt-for-code-generation

B. ChatGPT and Prompt Engineering

ChatGPT is an LM developed by OpenAI and it is designed for conversational tasks (e.g., question-answering and code generation) [14]. ChatGPT is built on the GPT-3.5 series with 175 billion parameters and optimized by using reinforcement learning from human feedback [38]. It can generate human-like responses to the user's textual prompt based on its context understanding and conversation history. Besides, OpenAI is improving ChatGPT by keeping optimizing GPT-4 [39].

As LM (e.g., ChatGPT [14]) with a large number of parameters (>100 million) emerges with advanced textual generation capability, prompt engineering (PE) becomes a new paradigm for NLP [27]. The goal of PE is to design an appropriate prompt for a pre-trained model and conduct prediction as expected with good performance, leading to the *"pre-train, prompt, predict"* paradigm. Specifically, the PE creates a prompt $x' = f_{prompt}(x) \in X$ for a textual input x (e.g., *"write a Java code for converting int to string"*) that describes a downstream task (e.g., code generation). With the prompt x', LM performs prediction $y = f_{LM}(x') \in Y$. Two basic PE tasks are: *1) Prompt Template Engineering,* it designs an appropriate template x' for the LM input (e.g., *"write a Java code for [x]"*, where "[x]" is a variable for NL description), as the performance of LM prediction y is sensitive to sentence(s) designed in the template. *2) Prompt Answer Engineering,* it aims to design an answer space Z in the prompt so that a better answer y could be generated from a limited scope $y \in Z$ (e.g., *"Which code is better? A or B"*). More advanced PE tasks intend to manipulate multi-prompt, such as prompt augmentation [40], composition, etc. [27]

The prompt (x') can be generated in four ways [27], [41]: *1) Manual Construction,* it is suitable for template-based prompts and few-shot prompting where the prompt is uncomplicated [40], [42]. *2) LM Generation,* it leverages LM to generate customized prompt (x') for each textual input (x), which can make up for the shortcomings of the manual construction [43]. *3) Retrieval-Based Prompt,* it relies on well-annotated external resources (e.g., Wikipedia) to alleviate the unstable issue of generation [44]. *4) Prompt Learning,* it builds a supervised model to automatically update the prompt according to the LM's generation and the associated ground-truth [27].

C. Prompt Optimization for Software Engineering

Previously, Reeves et al. [45] indicated that small variations in LMs' prompts will have a noticeable effect on the performance of code generation. Wang et al. [46] showed that tuning prompts with proper templates can help LMs achieve better performance in defect prediction, code summarization, and code translation. Shrivastava et al. [47] showed that enriching the programming context of a target repository into the prompt

leads to better code completion. Liu et al. [48] demonstrated the semantic gap between prompts and code and showed the importance of developers' feedback on the prompt updation. Huang et al. [49] showed that tuning prompt for LMs can help infer the type of a partial code. Existing studies demonstrated the importance of tuning prompts but did not investigate the mechanism behind the prompt design. In this study, we leveraged manual construction to explore the possibility of guiding ChatGPT for code generation tasks with a multi-step optimization strategy, investigate the influential factors in prompt design, and provide researchers with empirical insights for future works.

III. Study Subjects

This section describes the investigated tasks, datasets, and evaluation metrics used in our empirical study.

A. Code Generation Tasks

Code generation is the process of automatically generating code according to a requirement specification. Code generation can save time and effort for developers. The specification can be expressed in different ways. In this study, we investigated two representative tasks: *1) Text-to-Code (T2C) Generation.* It takes a textual description written in natural language (NL) as a specification. A code generation model generates code according to the textual description. *2) Code-to-Code (C2C) Generation.* It takes a code snippet written in a programming language (e.g., C#) as input and an NL model generates code written in another programming language (e.g., Java) with the same functionality. This task is also called a code translation.

B. Datasets

Here we present the datasets for testing the investigated two types of code generation tasks.

T2C Dataset. We chose the widely used dataset CONCODE [50], which is collected in CodeXGlue [5]. This dataset collected about 33k Java repositories from GitHub, consisting of 100k training data, 2k validation data, and 2k test data. Each instance is a tuple of three elements: *1) Code Snippet*, it is the ground-truth for code generation; *2) Natural Language Description,* it is extracted from the Javadoc of the code snippet; *3) Code Environment,* it describes the class file (i.e., the programmatic context) where the code snippet works, including class name, class path, member variables, and signatures of member functions.

C2C Dataset. We used the C2C dataset from CodeXGlue [5], a popular benchmark dataset for code understanding and generation. The C2C dataset collected data from several open-source repositories, including Lucene, POI, JGit, and Antlr. Totally, it contains 10k training data, 0.5k validation data, and 1k test data. Each instance contains a pair of code snippets written in C# and Java, which share the same functionality. In this study, we regarded the Java code snippet as the generation target and used the C# code snippet as the input.

C. Evaluation Metrics

To analyze the effectiveness of code generation, we measured the performance with the widely used metrics for code generation task [5], including BLEU [51] and CodeBLEU [18]. Details are described as follows. Following Lu et al. [5], we used the CodeBLEU as the overall evaluation metric.

BLEU, is a popular metric to measure the generation accuracy for the code snippets with various lengths [5], [51]. Specifically, $BLEU = BP * e^{(logP_1 + ... + logP_n)/n}$ where BP is the brevity penalty value, which equals 1 if the generated code is longer than the ground truth. Otherwise, it equals the ratio between the lengths of two code. P_i is the metric for the overlapping between the bags of i-grams appearing in the generated code and the ground truth.

CodeBLEU, is a variant of BLEU, which also considers the syntactic and semantic data flow correctness of code generation. It is similar to BLEU, but it calculates precision scores based on a set of code tokens, rather than natural language n-grams. Generally, CodeBLEU is a weighted average of the lexical, abstract syntax tree, and data flow match between the generated code and the ground-truth [18].

IV. Methodology

This section describes our empirical study on the manual prompt engineering for two code generation tasks.

A. Overview of the Empirical Study

The performance of ChatGPT is often sensitive to the design of prompts [27]. To augment the prompt, Wei et al. [42] indicated that Chain-of-Thought (CoT) prompting is the key strategy, which enables an LLM to solve problems by guiding them to produce a sequence of intermediate steps before giving the final answer. Due to its effectiveness, the CoT strategy is widely investigated and applied [19], [52].

Generally, to guide ChatGPT for code generation tasks, we investigate the candidate prompts with the CoT strategy in two steps: *1) Prompt Description,* we first analyze the requirement of a code generation task, and form a basic prompt in a natural way. Then, we provide the basic prompt for ChatGPT and ask *"how to improve the prompt?"*, and further improve the prompt according to ChatGPT's suggestions. *2) Multi-Step Optimizations,* we optimize the prompt in the first step on some samples from training data of the related dataset and keep optimizing the prompts based on the performance on the sampled training data. Based on the knowledge of the prompt construction process, we will generate some baseline prompts and conduct an empirical analysis on them in Section V.

Fig. 1 illustrates the overall process of this study. In this study, we worked with ChatGPT by invoking its API [17] using the GPT-3.5-Turbo model. The following two sections elaborate on how we constructed the candidate prompts for two code generation tasks, respectively, where the discussed prompts are listed in Table I.

TABLE I
DIFFERENT TYPES OF PROMPTS CONSTRUCTED FOR TWO CODE GENERATION TASKS. NOTE THAT #{NL}, #{CN}, #{MV}, #{MF}, AND #{CODE} STAND FOR THE VARIABLES OF A CLASS NAME, MEMBER VARIABLE, MEMBER FUNCTION, AND CODE, WHICH WILL BE FILLED IN ACTUAL INPUTS FROM THE DATASET.

No.	Prompt Type	Text-to-Code Generation Task	Code-to-Code Generation Task
P1	Task Prompt	write a Java method that + #{NL}	translate C# code into Java code: #{Code}
P2	Context Prompt	remember you have a Java class named + '#{CN}', member variables + '#{MV}', member functions + '#{MF}'	-
P3	Processing Prompt	remove comments; remove summary; remove throws; remove function modifiers; change method name to "function"; change argument names to "arg0", "arg1"...; change local variable names to "loc0", "loc1"...	do not provide annotation
P4	Updated Task Prompt	-	translate C# code into Java code: ```#{Code}```
P5	Behaviour Prompt	write a Java method #{that calls ...} with[out] exception handling to #{NL}	translate C# code into Java code: ```#{Code}```#{that calls ...} with[out] exception handling

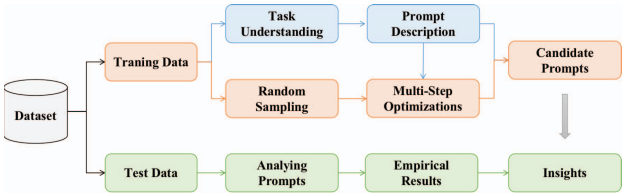

Fig. 1. Overview of the empirical study

TABLE II
EVALUATING COMBINATIONS OF DIFFERENT CANDIDATE PROMPTS IN TABLE I ON 100 SAMPLES RANDOMLY SELECTED FROM TRAINING DATA OF EACH GENERATION TASK. NOTE THAT P5(API) INDICATES THAT WE ONLY USED THE API PART OF THE PROMPT P5.

Task	Model	BLEU	CodeBLEU
T2C	P1	05.29	22.76
	P2+P1	10.42 (+96.98%)	25.05 (+10.06%)
	P2+P1+P3	13.11 (+147.83%)	36.00 (+58.17%)
	P2+P5(API)+P3	22.14 (+318.53%)	44.18 (+94.11%)
	P2+P5+P3	27.48 (+419.47%)	46.78 (+105.54%)
C2C	P1	09.76	39.37
	P1+P3	08.55 (-12.40%)	45.28 (+15.01%)
	P4+P3	15.44 (+58.20%)	45.00 (+14.30%)
	P5(API)+P3	13.37 (+36.99%)	46.17 (+17.27%)
	P5+P3	08.90 (-08.81%)	46.88 (+19.08%)

B. Candidate Prompts for Text-to-Code Generation

Prompt Description. As described by Lu et al. [5], the T2C generation task takes an NL description as a textual input (e.g., *"convert int to String"*) and expects a correct generation of Java code method, which matches the intent of the description. According to the task description, we naturally present a basic task prompt: *"write a Java method that + #{NL}"* (Table I-P1). To assess the effectiveness of the prompts, we randomly sampled 100 instances from training data and asked ChatGPT to generate code given the prompt. We obtain a low generation accuracy of BLEU=5.29 and CodeBLEU=22.76.

Multi-Step Optimizations. With the constructed bask prompt, we asked ChatGPT: *"how to improve the prompt: write a Java method that converts int to string"*. ChatGPT said that by providing more specific details of the method behaviour, programming context, and input/output examples, we can create a more clear and more informative prompt that helps guide the generation of a well-designed Java method. We notice that the programming environment provided in the dataset as described in Section III-B can be used as additional context information. To reach the goal, we added a context prompt before the task prompt: *"remember you have a Java class named + '#{CN}', member variables + '#{MV}', member functions + '#{MF}'"* (Table I-P2). In the prompt, the cloze #{...} will be filled by corresponding information given in the dataset. Note that we tell ChatGPT to remember the class because it will generate the whole class if we do not guide ChatGPT with clear instructions. By adding the context prompt, the generation accuracy of the randomly selected training samples can be improved with BLEU=10.42 and CodeBLEU=25.05.

After analyzing the ground truth, we observe that the ground truth was pre-processed in four aspects: 1) all comments, throws, and method modifiers are removed; 2) the method name is changed to "function"; 3) all the arguments are renamed to "arg0", "arg1", etc.; 4) all the local variables are renamed to "loc0", "loc1", etc. Following these observations, we thus add a processing prompt with a series of instructions after the task prompt: *"remove comments; remove summary; remove throws; remove function modifiers; change method name to "function"; change argument names to "arg0", "arg1"...; change local variable names to "loc0", "loc1"...* (Table I-P3). Note that some summaries generated for part of code snippets cannot be removed by the prompt *"remove comments"* but the *"remove summary"*; we used the ellipsis "..." in the prompt instead of "etc.", because ChatGPT cannot do the renaming actions with the command "etc.". The evaluation of training samples shows a further improvement with BLEU=13.11 and CodeBLEU=36.00.

By comparing the generated code with the ground truth, we notice that ChatGPT may generate code with different APIs and settings of exception handling. It is natural to ask ChatGPT to regenerate the code according to its responses

and users' specific requirements. To extract the requirement of the APIs and exception handling, we input the ChatGPT with the prompts *"list the used API names: #{Code}"* and *"does the code contain exception handling? + #{Code}"* for the ground-truth, respectively. Afterward, we write scripts to analyze the responses for the requirements of APIs (i.e., name list) and exception handling (i.e., true or false). With these two requirements, we replace the task prompt with a behaviour prompt: *"write a Java method #{that calls ...} with[out] exception handling to #{NL}"* (Table I-P5). Note that if the API list is empty, we remove the *"#{that calls ...}"*, otherwise we replace *"..."* with the name list. For the *"with[out]"*, we determine whether it is *"with"* or *"without"* according to the actual demand. We find that considering the API requirement, the generation accuracy of training samples can be enhanced (BLEU=22.14 and CodeBLEU=44.18). Meanwhile, using the whole behaviour prompt (i.e., API + exception handling), the performance on training samples can be further boosted with BLEU=27.48 and CodeBLEU=46.78.

C. Candidate Prompts for Code-to-Code Generation

As the C2C generation has a similar process of prompt design as the T2C generation, we mainly show their key differences in this subsection.

Prompt Description. According to the task description in Section III-B, our C2C generation task aims to generate a Java code method according to a given C# code function. Based on the task requirement, we form the task prompt: *"translate C# code into Java code: #{Code}"* (Table I-P1). For the randomly selected 100 samples from training data, the generation performance is BLEU=9.76 and CodeBLEU=39.37.

Mult-Step Optimizations. Comparing with the T2C generation, we can find many differences in the C2C generation task: the C2C dataset does not involve the related class; ChatGPT does not generate comments and throws in the code; the ground truth is not pre-processed for the method name, modifiers, argument names, and local variable names. However, ChatGPT will generate annotation following the C# code but the ground-truth removed all the annotations. Therefore, we add a simple processing prompt to the task prompt: *"do not provide annotation"* (Table I-P3). Moreover, we find that ChatGPT has the ability to understand the markdown syntax in the prompt. Thus, in the task prompt, we change *#{Code}* to ```` ```#{Code}``` ```` as an updated task prompt (Table I-P4). Evaluating on the training samples with the processing prompt, the generation accuracy is improved on CodeBLEU (45.28) but not on BLEU (8.55). After updating the code format in the task prompt, we achieve further enhancement with BLEU=15.44 and CodeBLEU=45.00.

Similar to T2C generation, we extract the requirement of the API usage and exception handling from the ground-truth code. Subsequently, we added this information to the task prompt as a behaviour prompt: *"translate C# code into Java code: ```#{Code}``` #{that calls ...} with[out] exception handling"* (Table I-P5). Experimental results on the 100 random samples from training data show that by adding the requirements of API usage (BLEU=13.37 and CodeBLEU=46.17), the generation accuracy will be slightly improved in terms of CodeBLEU. Moreover, using the whole behaviour prompt also showed a reduction in BLEU (8.90) and a minor increase in CodeBLEU (46.88). We observed that ChatGPT can understand translation context and generate good results. However, adding more requests may bring uncertainty to the generation. Therefore, this behaviour prompt likely has negative effects on the C2C generation task.

D. Research Questions

In this study, we constructed a series of candidate prompts with a manual prompt engineering method to guide ChatGPT for two code generation tasks. To verify the effectiveness of these prompts and analyze the associated influential factors, this study investigates the following RQs:

RQ1: How effective are different candidate prompts for ChatGPT? As described above, we leveraged the CoT strategy [42] to construct some candidate prompts for two code generation tasks with multi-step optimizations. This RQ intends to verify the possibility of guiding ChatGPT for better code generation.

RQ2: How does the conciseness request affect ChatGPT? In the prompt design, we observed that ChatGPT often generates detailed code, much more complex than the ground truth. Thus, one goal of the multi-step optimizations is to guide ChatGPT to generate concise code with a series of prompts. It is worth investigating whether the generation performance can be further improved by directly requesting ChatGPT for a concise generation.

RQ3: How does the session setting affect ChatGPT? When communicating with ChatGPT, we started one individual session for each prompt. Meanwhile, it is widely known that ChatGPT can learn the session context and generate better responses from the context [42], [53]. Therefore, this RQ intends to answer whether or not ChatGPT can generate better code given a session with a number of prompts.

RQ4: How does the generation randomness affect ChatGPT? It is known that ChatGPT may generate code with slight differences every time for the same prompt [54], [55]. To investigate how randomness affects the generation performance, we re-run the guided ChatGPT multiple times and analyze the stability of the generation performance.

V. Results

This section presents the experimental settings and results for the four RQs described in Section IV-D.

A. Effectiveness of Different Candidate Prompts (RQ1)

Objective. To assess the effectiveness of the candidate prompts, we intend to present some baselines, test them on the testing data of corresponding code generation tasks, and analyze the effectiveness in terms of prediction accuracy.

Method. This study presents three baselines that can generate code by using ChatGPT with three levels of prompts: 1) ChatGPT-task, we used the task prompts in Table I(P1) as the input of ChatGPT, because they can represent the common chatting conditions with a direct request of code generation. 2) ChatGPT-detail, we merged the prompts of task, context, processing, and updated task for two tasks in Table I(P1-P4). Note that the T2C generation has no updated task prompt (P4) while the C2C generation contains no context prompt (P2). 3) ChatGPT-behaviour, based on the baseline ChatGPT-detail, we further updated the task prompt with the behaviour prompt (Table I-P5) to provide more guidance on the code generation. Although Section IV-C showed that P5 has negative effects on the C2C generation task for the sampled data, we still set up this baseline for C2C generation to further confirm the previous observation. The testing data and evaluation metrics were elaborated in Section III.

TABLE III
TESTING THREE CANDIDATE PROMPTS ON T2C AND C2C GENERATION TASKS.

Task	Model	BLEU	CodeBLEU
T2C	ChatGPT-task	05.63	28.05
	ChatGPT-detail	14.09 (+140.27%)	39.90 (+42.25%)
	ChatGPT-behaviour	21.59 (+283.48%)	48.69 (+73.58%)
C2C	ChatGPT-task	10.61	46.12
	ChatGPT-detail	15.79 (+48.82%)	47.71 (+03.45%)
	ChatGPT-behaviour	09.47 (-10.74%)	47.38 (+02.32%)

Result. On the T2C generation task, we compared the performance of ChatGPT with three different prompts in Table III. ChatGPT-task achieves a generation accuracy of BLEU=5.63 and CodeBLEU=28.05. For the ChatGPT-detail with a number of extended prompts, its generation performance is BLEU=14.09 and CodeBLEU=39.90, outperforming ChatGPT-task by 140.27% and 42.25% in terms of BLEU and CodeBLEU respectively. Additionally, the last baseline ChatGPT-behaviour gained a better performance in T2C generation with BLEU=21.59 and CodeBLEU=48.69. We notice that ChatGPT-behaviour improved the performance of the ChatGPT-task by 283.48% and 73.58% in terms of BLEU and CodeBLEU, respectively. These results indicate that the last constructed prompt can substantially improve the T2C generation ability of ChatGPT.

In terms of the C2C generation task, Table III also provided the performance of ChatGPT with three candidate prompts. ChatGPT-tasks can obtain better performance (BLEU=10.61 and CodeBLEU=46.12) compared with the T2C generation task. Furthermore, the ChatGPT-detail shows better generation accuracy with BLEU=15.79 and CodeBLEU=47.71, outperforming those of ChatGPT by 48.82% and 3.45% respectively. However, we can find that ChatGPT-behaviour shows poorer performance with BLEU=9.47 and CodeBLEU=47.38, increased those of ChatGPT-task by -10.74% and 2.32%. Therefore, ChatGPT-detail shows better performance than ChatGPT-behaviour.

Answer to RQ1: Carefully constructing prompts can guide ChatGPT with better performance on the T2C and C2C generation tasks.

B. Impacts of Conciseness Request (RQ2)

Objective. In our prompt construction, we guided ChatGPT to remove irrelevant code components. However, we observed that the generated code is still more complex than the ground truth. Without more information, we cannot design better prompts. Therefore, this RQ intends to investigate whether we can change the task prompt by directly asking ChatGPT to generate a more concise code, instead of manually adding specific instructions.

Method. RQ1 showed that ChatGPT-behaviour and ChatGPT-detail are the best baselines for T2C and C2C generation tasks, respectively. To add conciseness requests to their prompts, we found one viable and simple way to add the word "concise" before the generation targets. Specifically, for the T2C generation, the behaviour prompt of ChatGPT-behaviour is changed to *"write a concise Java method ..."* (Table I-P5). Likewise, for the C2C generation task, the task prompt of ChatGPT-detail is updated: *"translate C# code into concise Java code ..."* (Table I-P4). To evaluate the impacts of the conciseness request, we tested the modified prompts on the testing data.

TABLE IV
TESTING THE BEST BASELINES IN RQ1 WITH (OR WITHOUT) THE CONCISENESS REQUEST (C) ON T2C AND C2C GENERATION TASKS.

Task	Model	BLEU	CodeBLEU
T2C	ChatGPT-behaviour	21.59	48.69
	ChatGPT-behaviour-C	26.86 (+24.41%)	50.18 (+03.06%)
C2C	ChatGPT-detail	15.79	47.71
	ChatGPT-detail-C	16.75 (+06.08%)	46.62 (-02.28%)

Result. For the T2C generation task, Table IV shows that the conciseness request is helpful (ChatGPT-behaviour-C) with BLEU=26.86 and CodeBLEU=50.18. Compared with the best baseline in RQ1 (ChatGPT-behaviour), these two evaluation metrics are further enhanced by 24.41% and 3.06%, respectively. Meanwhile, from the table IV, we can notice that by adding the conciseness request to the C2C generation task, the generation performance of ChatGPT-detail-C shows a slight decrease in terms of the CodeBLEU (46.62) compared with the ChatGPT-detail (best baseline in RQ1), although the BLEU score is improved by 6.08%. These results suggest that the conciseness request is useful for T2C generation but not for C2C generation.

Answer to RQ2: Adding conciseness request to the prompt can further improve the performance of the T2C generation, but show minor negative effects on the C2C generation.

C. Impacts of Session Setting (RQ3)

Objective. By default, we opened an individual session for each prompt and communicated with ChatGPT. In contrast, the communication can be worked with a continuous session that generates responses for a number of prompts. In this way, the ChatGPT can learn from the context and may generate better responses for the code generation tasks. Thus, this RQ aims to analyze the effects of the session setting.

Method. During the communication with ChatGPT, we used one session to generate code for a number of prompts. When the number reaches the maximum limit, a further prompt will receive no response. Then, we started another new session. In this way, we can ensure that each session is fully utilized and ChatGPT can better understand the context. We know that the amount of contextual information would be lower for the prior prompts within a session. It is worth nothing to investigate whether the continuous session is a better choice for code generation compared with the individual session.

TABLE V
TESTING THE BEST BASELINE IN RQ2 WITH (OR WITHOUT) THE SESSION SETTING (S) ON T2C AND C2C GENERATION TASKS.

Task	Model	BLEU	CodeBLEU
T2C	ChatGPT-behaviour-C	26.86	50.18
	ChatGPT-behaviour-CS	29.29 (+09.05%)	49.74 (-00.88%)
C2C	ChatGPT-detail	15.79	47.71
	ChatGPT-detail-S	16.82 (+06.52%)	48.80 (+02.28%)

Result. As shown in Table V, using the continuous session (ChatGPT-behaviour-CS) showed no overall improvement for the T2C generation with BLEU=29.29 and CodeBLEU=49.74. Compared with the best baseline in RQ2 (ChatGPT-behaviour-C), the BLEU score is improved by 9.05% but the CodeBLEU is decreased by 0.88%. On the other hand, the continuous session can present improvement for the C2C generation task (ChatGPT-detail-S) with BLEU=16.82 and CodeBLEU=48.80. Compared with the best baseline ChatGPT-detail in RQ2, the evaluation metrics are further improved by 6.52% and 2.28%, respectively. These experimental results indicate that the continuous session is beneficial for the C2C generation task but has no improvement for the T2C generation task. Therefore, the individual session is more suitable for the T2C generation with the designed prompts.

Answer to RQ3: The continuous session is helpful for the T2C generation task while the individual session is suitable for the C2C generation task.

D. Impacts of Generation Randomness (RQ4)

Objective. Commonly, ChatGPT generates responses with slight differences to balance the accuracy and creativity. Thus, the generation randomness may affect the performance of code generation. This RQ investigates how randomness affects the effectiveness of the designed prompts.

TABLE VI
TESTING THE BEST BASELINES IN RQ3 ON T2C AND C2C GENERATION TASKS IN FIVE ROUNDS (RQ1-R5), WHERE "MIN", "MAX", "AVG", "STD" STAND FOR THE MINIMUM, MAXIMUM, AVERAGE AND STANDARD DEVIATION OF THE GENERATION ACCURACY.

Task	Model	BLEU	CodeBLEU
T2C	R1	26.86	50.18
	R2	26.85	50.07
	R3	27.02	50.18
	R4	26.92	50.20
	R5	27.00	50.17
	MIN	26.86	50.07
	MAX	27.02	50.20
	AVG	26.93	50.16
	STD	00.08	00.05
C2C	R1	16.82	48.80
	R2	17.34	49.17
	R3	17.23	48.38
	R4	17.32	49.15
	R5	17.21	48.80
	MIN	16.82	48.80
	MAX	17.34	49.17
	AVG	17.18	48.86
	STD	00.21	00.33

Method. To reach the goal, we ran the best baselines (i.e., ChatGPT-behaviour-C and ChatGPT-detail-S) in RQ3 five times, respectively. We computed the average (AVG) and standard deviation (STD) of the performance of these multiple generation results. Based on these measurements, we analyzed the generation stability and the effects of the randomness.

Result. From Table VI, we can observe that the five rounds of T2C generation show a stable performance, where BLEU ranges from 26.86 to 27.02 with average=26.93 and STD=0.08; CodeBLEU ranges from 50.07 to 50.20 with average=50.16 and STD=0.05. Meanwhile, the multiple runs of C2C generation also show a stable prediction accuracy, where BLEU ranges from 16.82 to 17.34 with average=17.18 and STD=0.21; CodeBLEU ranges from 48.80 to 49.17 with average=48.86 and STD=0.33. These results indicate that testing ChatGPT with our designed prompts will generate stable responses. We observed that the major reason is that the instructions in the prompts are specific so that the generation randomness is limited. As we observed that the effect of the randomness is negligible, we did not extend this RQ with more rounds of experiments.

Answer to RQ4: The generation randomness shows little effect on the code generation tasks due to the specific instructions described in the designed prompts.

VI. DISCUSSION

This section provides some qualitative analysis of the constructed prompts. Specifically, Section VI-A compares our code generation performance with existing fine-tuned models.

Sections VI-B and VI-C analyze the correctness and quality of the code generated by ChatGPT with the best prompts.

A. Comparison with Fine-Tuned Models

Objective. Section V indicates that carefully constructing prompts can guide ChatGPT to generate substantially better code. This result implies the effectiveness of the *"pre-train, prompt, predict"* paradigm for LLMs as described in Section II-B. However, many LLMs based on the *"pre-train, fine-tune"* paradigm have been successfully applied in the code generation tasks as exemplified in Section II-A. Therefore, we would like to compare the performance of these two paradigms. Specifically, we investigate how effective is the best prompt compared with the related fine-tuned LLMs.

Method. In this study, we investigated the widely used dataset CodeXGlue [5], which provides a number of benchmark models. We used the experimental results reported by Lu et al. [5] as comparisons. Moreover, we went through all the related works presented in Section II-A. Wang et al. [32] also tested their proposed model CodeT5 and baseline model PLBART [33] on CodeXGlue, so we included these models in our comparisons. We excluded the other reports in the related work because they did not use the CodeXGlue dataset.

Result. Table VII illustrated the included fine-tuned LLMs for two code generation tasks, the description and reference of the related LLM, and the reported scores of BLEU and CodeBLEU. We also placed our best prompt settings (ChatGPT-best) in the table. We can find that ChatGPT-best shows the best performance on the T2C generation tasks in terms of CodeBLEU. This result implies that guiding ChatGPT with better prompts outperforms the other state-of-the-art fine-tuned LLMs. However, for the C2C generation task, the ChatGPT-best achieved the fifth rank, only outperforming PBSMT [58]. This poorer performance on the C2C task may result from the limited contextual information, so we cannot extend the designed prompt with more specific instructions as shown in Section V-A. Moreover, these results can demonstrate the potential capabilities of the *"pre-train, prompt, predict"* paradigm, because the performance could be further improved by providing prompts with more specific instructions or fine-tuning ChatGPT with related training data. Additionally, we can notice that ChatGPT-best obtains low values on BLEU compared with other LLMs on two tasks because ChatGPT commonly generate a longer and more complex code. Therefore, fine-tuning ChatGPT with more data will further improve the performance of code generation in terms of BLEU.

> **Finding 1:** *Guiding ChatGPT with the best candidate prompt outperforms the state-of-the-art fine-tuned LLMs for the T2C generation, but it shows a poorer rank on the C2C generation due to the limited contextual information expressed in the prompt.*

B. Correctness of Code Generation

Objective. CodeBLEU (or BLEU) is an effective and widely used metric for automated evaluations. However, a generated code with a high score may not be correct. Therefore, we would like to investigate the correctness of the code generated from our best prompt settings.

Method. We randomly selected 100 samples from the generated code of each code generation task (T2C and C2C). The correctness is measured by functional equivalency between the generated code and the ground truth. The relevancy is voted by three authors (the first, second, and fourth), where relevancy is determined when the number of votes is larger than or equal to two.

Result. We found that among the 100 samples, the T2C task only generated 31 equivalent code as the ground-truth in functional behaviours. We observed that higher CodeBLEU only indicates correct lexical match, syntactic match, and data flow match, but does not necessarily suggest functional equivalency. Meanwhile, the NL descriptions for T2C generation are usually not specific, and ChatGPT is likely to misunderstand the requirement. A simple example is that for the ground-truth code *"String function(){return namespaceURI;}"* which means return the variable namespaceURI, but the NL description provided in the dataset is *"Get the WS-ReliableMessaging namespace to be used for encoding and decoding messages"*. Therefore, for the prompt of T2C generation, the NL description may need to be refined in some ways.

For the C2C generation task, we found 59 generated code snippets that are functionally equivalent to their ground-truth, much better than that for the T2C task. We noticed that the translation from C# code provides many useful contexts so that ChatGPT can generate the corresponding Java code line by line. However, the T2C generation works only for the code with commonly used APIs.

> **Finding 2:** *ChatGPT with the best prompt shows better correctness on the C2C task than on the T2C task, because the NL descriptions are not rigorous.*

C. Quality of Code Generation

Objective. Quality is an important feature of a good generated code, in spite of the correctness of code generation. In this study, we also investigated the quality of the code generated by ChatGPT with the best candidate prompts. Meanwhile, we also checked the quality of the ground-truth code to compare its quality with the generated code.

Method. To measure the quality of the code generation, we utilized the SonarQube (version 9.8) [59], a widely used open-source platform for analyzing and tracking the quality of code [60], [61]. It can check three types of quality issues (bug, vulnerability, and code smell) in five severity levels (i.e., blocker, critical, major, minor, and info) [59]. In this study,

TABLE VII
COMPARISONS BETWEEN THE BEST PROMPTS (CHATGPT-BEST) AND THE STATE-OF-THE-ART FINE-TUNED LLMS REPORTED IN [5] AND [32] ON T2C AND C2C GENERATION TASKS.

Task	Model	Description	BLEU	CodeBLEU
T2C	Seq2Seq	An RNN-based sequence to sequence model [56].	21.31	26.39
	Seq2Action+MAML	A context-aware encoder-decoder mode that leverages model-agnostic meta-learning (MAML) [57].	24.40	29.46
	CodeGPT	A Transformer-based language model that has the same model architecture and training objective of GPT-2 and pre-trained on the programming language (PL) [5].	28.69	32.71
	CodeGPT-adapted	CodeGPT is continually trained on the code corpus [5].	32.79	35.98
	PLBART	A sequence-to-sequence model capable of performing a broad spectrum of program and language understanding and generation tasks [33].	36.69	38.52
	CodeT5	A unified pre-trained encoder-decoder Transformer model that better leverages the code semantics conveyed from the developer-assigned identifiers [32].	41.48	44.10
	ChatGPT-best	ChatGPT-behaviour-C, the ChatGPT works with context prompt, behaviour prompt, processing prompt, and conciseness request.	**26.86**	**50.18**
C2C	PBSMT	A traditional phase-based machine translation method that uses statistical models to translate text from one language to another. [58]	40.06	43.48
	ChatGPT-best	ChatGPT-detail-S, the ChatGPT works with an updated task prompt, processing prompt, and continuous session.	**16.82**	**48.80**
	Transformer	A sequence-to-sequence encoder-decoder model with self-attention mechanism [26].	50.47	61.59
	CodeBERT	A bidirectional encoder representations from transformers (BERT) model with pre-trained with six programming languages [11].	72.14	79.41
	RoBERTa	It is based on the architecture of the BERT model and is pre-trained on a large corpus of text using a masked language modeling objective [57].	71.99	80.18
	PLBART	A sequence-to-sequence model capable of performing a broad spectrum of program and language understanding and generation tasks [33].	65.00	85.27

we measured the quality of code generation by counting the number of code that contain critical or blocker issues. This is because these two severity levels have strong impacts which are commonly considered by developers.

Result. Table VIII shows that for the T2C generation, the code generated by ChatGPT-best shows slightly better quality than the ground truth. Specifically, SonarQube found one critical bug that reminds us to make sure a local variable is not zero before doing the division. And the other 29 issues belong to the critical code smell that requires further attention. In contrast, the associated ground truths contain 35 code smells. For the C2C generation, the generated code possess 62 code smells, much higher than the number of the related ground-truth (17 code smells). In these two tasks, the generated code involve no severe bug or vulnerability.

We found that the identified code smells mainly provide six kinds of suggestions: 1) defining a constant instead of duplicating a string; 2) replacing the call of "replaceAll()" by "replace()"; 3) reducing cognitive complexity of a code; 4) adding default case to a switch, not overriding the Object.finalize() method; 5) renaming method to prevent misunderstanding; and 6) using a copy constructor or copy factory instead of "clone" implementation. We believe that it is worthy of addressing the bug and code smells detected by SonarQube, although the number is not large compared to the total count of the dataset. Therefore, ChatGPT may be not able to ensure the quality of the generated code, which requires further investigations.

Finding 3: *The code generated by ChatGPT contains no severe bug or vulnerability on the experimented datasets but they contain many code smells, which should be addressed by developers.*

VII. IMPLICATION

Based on our experimental results and discussion, this section provides implications for developers and researchers.

Tips of Using ChatGPT Prompts for Developers. Our experimental results showed that guiding ChatGPT with carefully crafted prompts can substantially improve the code generation performance. To make the most of ChatGPT's capabilities, developers are suggested to provide prompts with rich programming context, including relevant classes, member variables, and functions. Additionally, ChatGPT can understand code, so developers can instruct it to preprocess code, such as removing summaries and changing variable names. ChatGTP can better comprehend code blocks with markdown syntax. To generate code with expected APIs or in a concise form, developers can directly request ChatGPT. Chatting with ChatGPT in one continuous session may also be beneficial because it can understand the context and refine its responses. With the chain-of-thoughts feature, developers can optimize their prompts step-by-step by considering ChatGPT's feedback. However, developers should remain cautious about

TABLE VIII
QUALITY ANALYSIS OF THE CODE GENERATED BY CHATGPT-BEST AND THE CORRESPONDING GROUND-TRUTH ON T2C AND C2C GENERATION TASKS.

Task	Data	#Bug		#Vulnerability		#Code Smell		Total
		Blocker	Critical	Blocker	Critical	Blocker	Critical	
T2C	Generated Code	0	1	0	0	0	29	30
	Ground-Truth	0	0	0	0	1	34	35
C2C	Generated Code	0	0	0	0	56	6	62
	Ground-Truth	0	0	0	0	10	7	17

the quality of the generated code, even if the function-level generation appears free of severe bugs or vulnerabilities in our experiments.

Potential Future Research Directions for Researchers. This study demonstrates that providing ChatGPT with improved prompts can enhance code generation performance, outperforming that of state-of-the-art fine-tuned LLMs. Future studies could investigate methods for automatically designing and optimizing prompts for code generation tasks; designing a better evaluation metric to automatically assess the correctness of the generated code; and further assessing and improving the quality of the code generation. Our study also highlights the potential of LLMs like ChatGPT, utilizing the *"pre-train, prompt, predict"* paradigm and prompt engineering, for software engineering research.

VIII. THREATS TO VALIDITY

There are some potential threats affecting the validity of our experimental results and conclusions.

Limited Tasks and Dataset. This study constructed some candidate prompts on two code generation tasks. The construction process may not be suitable for other generation tasks, such as code completion [62] and test case generation [63]. For the investigated T2C and C2C generation tasks, we only chose one dataset CodeXGlue [5] as our study subject, although it is a widely used dataset. On the C2C generation task, we tested the generation from C# to Java. It is uncertain whether our experimental results and findings could be extended to other languages (e.g., Python and Go) and the reverse generation (i.e., from Java to C#). However, the empirical results demonstrated the importance of the prompt design.

Manual Prompt Construction. Same as other prompt engineering studies with manual construction, the prompt design and multi-step optimizations are conducted according to human understanding and observations. The knowledge of the designer may affect the performance of the used prompts. Moreover, our design and combination choices were based on the 100 randomly selected samples from training data. Thus, the size and randomness of the sampling may bring in different choices. Besides, the prompt construction process is coupled with the datasets, which may not be generalizable for other datasets. However, this study aims to explore the viability of prompt design and investigate some related influential factors. In the future, we would like to investigate prompt engineering with automated construction methods, which can adapt to various datasets with few-shot learning [27], [41].

Limited Testing. In our study, we conducted a limited number of tests as presented in Section V, such as the combinations of prompt selection and combinations, conciseness request and session settings, and multiple runs for generation randomness. We suppose that more tests may help us further improve the prompts and strengthen our conclusions and findings. However, invoking ChatGPT API on the whole dataset requires a high cost. Therefore, we only tested a limited number of choices that are likely to improve the generation performance, so that we can demonstrate the importance of prompt design for ChatGPT.

Human Evaluation. To assess the correctness of the generated code, we randomly selected 100 samples to analyze the relevancy between the generated code and ground truth. The relevancy was determined by human evaluations. To mitigate the effects of the subjective bias, the determinations are based on the voting of three authors. Although the number of samples is limited, the correctness analysis of these random samples provided some useful insights for further studies.

IX. CONCLUSION

In this paper, we have designed and improved prompts for guiding ChatGPT on two types of code generation tasks, namely text-to-code generation and code-to-code generation. Our experimental results showed the effectiveness of our constructed prompts when asking ChatGPT to generate code on a widely used dataset CodeXGlue. Moreover, we investigated the influential factors for designing prompts on code generation tasks. Besides, we compared the performance of the best prompts with the state-of-the-art fine-tuned LLMs, and assessed the correctness and quality of the code generated by ChatGPT. Based on our findings, we present the potential future research directions.

ACKNOWLEDGMENT

This research is supported by the National Nature Science Foundation of China (62202074 and 62372071), China Postdoctoral Science Foundation (2022M710519), the Postdoc Foundation of Chongqing (2021LY23), and Chongqing Technology Innovation and Application Development Project (CSTB2023TIAD-STX0015 and CSTB2022TIAD-KPX0067).

REFERENCES

[1] J. Herrington, *Code generation in action*. Manning Publications Co., 2003.

[2] G. Poesia, O. Polozov, V. Le, A. Tiwari, G. Soares, C. Meek, and S. Gulwani, "Synchromesh: Reliable code generation from pre-trained language models," *arXiv preprint arXiv:2201.11227*, 2022.

[3] F. F. Xu, Z. Jiang, P. Yin, B. Vasilescu, and G. Neubig, "Incorporating external knowledge through pre-training for natural language to code generation," *arXiv preprint arXiv:2004.09015*, 2020.

[4] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

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

[6] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[7] S. Liu, Y. Li, and Y. Liu, "Commitbart: A large pre-trained model for github commits," *arXiv preprint arXiv:2208.08100*, 2022.

[8] D. Araci, "Finbert: Financial sentiment analysis with pre-trained language models," *arXiv preprint arXiv:1908.10063*, 2019.

[9] J. Zhou, J. Tian, R. Wang, Y. Wu, W. Xiao, and L. He, "Sentix: A sentiment-aware pre-trained model for cross-domain sentiment analysis," in *Proceedings of the 28th international conference on computational linguistics*, 2020, pp. 568–579.

[10] B. Gunel, J. Du, A. Conneau, and V. Stoyanov, "Supervised contrastive learning for pre-trained language model fine-tuning," *arXiv preprint arXiv:2011.01403*, 2020.

[11] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[13] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[14] OpenAI, "Chatgpt official blog," https://openai.com/blog/chatgpt, 2023.

[15] A. Kothari, "Chatgpt, large language models, and generative ai as future augments of surgical cancer care," *Annals of Surgical Oncology*, pp. 1–3, 2023.

[16] V. Liu and L. B. Chilton, "Design guidelines for prompt engineering text-to-image generative models," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–23.

[17] OpenAI, "Chatgpt api," https://platform.openai.com/docs/api-reference, 2023.

[18] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[19] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *arXiv preprint arXiv:2201.11903*, 2022.

[20] J. Salazar, D. Liang, T. Q. Nguyen, and K. Kirchhoff, "Masked language model scoring," *arXiv preprint arXiv:1910.14659*, 2019.

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

[22] A. Alokla, W. Gad, W. Nazih, M. Aref, and A.-b. Salem, "Pseudocode generation from source code using the bart model," *Mathematics*, vol. 10, no. 21, p. 3967, 2022.

[23] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "Spt-code: sequence-to-sequence pre-training for learning source code representations," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2006–2018.

[24] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

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

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[27] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[28] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[29] S. Liu, B. Wu, X. Xie, G. Meng, and Y. Liu, "Contrabert: Enhancing code pre-trained models via contrastive learning," *arXiv preprint arXiv:2301.09072*, 2023.

[30] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," *arXiv preprint arXiv:2007.04973*, 2020.

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

[32] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[33] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[34] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

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

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

[37] G. Inc., "Copilot," https://github.com/features/copilot, 2023.

[38] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[39] OpenAI, "Gpt-4 technical report," https://doi.org/10.48550/arXiv.2303.08774, 2023.

[40] D. Rajagopal, V. Khetan, B. Sacaleanu, A. Gershman, A. Fano, and E. Hovy, "Template filling for controllable commonsense reasoning," *arXiv preprint arXiv:2111.00539*, 2021.

[41] S. Qiao, Y. Ou, N. Zhang, X. Chen, Y. Yao, S. Deng, C. Tan, F. Huang, and H. Chen, "Reasoning with language model prompting: A survey," *arXiv preprint arXiv:2212.09597*, 2022.

[42] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *arXiv preprint arXiv:2206.07682*, 2022.

[43] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," *arXiv preprint arXiv:2210.03493*, 2022.

[44] S. Mishra, A. Mitra, N. Varshney, B. Sachdeva, P. Clark, C. Baral, and A. Kalyan, "Numglue: A suite of fundamental yet challenging mathematical reasoning tasks," *arXiv preprint arXiv:2204.05660*, 2022.

[45] B. Reeves, S. Sarsa, J. Prather, P. Denny, B. A. Becker, A. Hellas, B. Kimmel, G. Powell, and J. Leinonen, "Evaluating the performance of code generation models for solving parsons problems with small prompt variations," in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, 2023, pp. 299–305.

[46] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "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*, 2022, pp. 382–394.

[47] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 693–31 715.

[48] M. X. Liu, A. Sarkar, C. Negreanu, B. Zorn, J. Williams, N. Toronto, and A. D. Gordon, ""what it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–31.

[49] Q. Huang, Z. Yuan, Z. Xing, X. Xu, L. Zhu, and Q. Lu, "Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[50] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," *arXiv preprint arXiv:1808.09588*, 2018.

[51] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[52] M. Suzgun, N. Scales, N. Schärli, S. Gehrmann, Y. Tay, H. W. Chung, A. Chowdhery, Q. V. Le, E. H. Chi, D. Zhou *et al.*, "Challenging big-bench tasks and whether chain-of-thought can solve them," *arXiv preprint arXiv:2210.09261*, 2022.

[53] P. P. Ray, "Chatgpt: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope," *Internet of Things and Cyber-Physical Systems*, 2023.

[54] Q. Lyu, J. Tan, M. E. Zapadka, J. Ponnatapuram, C. Niu, G. Wang, and C. T. Whitlow, "Translating radiology reports into plain language using chatgpt and gpt-4 with prompt learning: Promising results, limitations, and potential," *arXiv preprint arXiv:2303.09038*, 2023.

[55] M. V. Reiss, "Testing the reliability of chatgpt for text annotation and classification: A cautionary remark," *arXiv preprint arXiv:2304.11085*, 2023.

[56] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[57] D. Guo, D. Tang, N. Duan, M. Zhou, and J. Yin, "Coupling retrieval and meta-learning for context-dependent semantic parsing," *arXiv preprint arXiv:1906.07108*, 2019.

[58] R. Zens, F. J. Och, and H. Ney, "Phrase-based statistical machine translation," in *KI 2002: Advances in Artificial Intelligence: 25th Annual German Conference on AI, KI 2002 Aachen, Germany, September 16–20, 2002 Proceedings 25*. Springer, 2002, pp. 18–32.

[59] Sonar, "Sonarqube 9.8," https://docs.sonarqube.org/9.8/user-guide/rules/overview/, 2023.

[60] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are sonarqube rules inducing bugs?" in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 501–511.

[61] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? a closer look at realistic usage of sonarqube," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 209–219.

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

[63] S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of automated unit test generation for python," *Empirical Software Engineering*, vol. 28, no. 2, p. 36, 2023.