AceCoder: An Effective Prompting Technique Specialized in Code Generation

JIA LI (he/him/his), YUNFEI ZHAO, YONGMIN LI, GE LI, and ZHI JIN, Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University, Beijing, China and School of Computer Science, Peking University, Beijing, China

Large language models (LLMs) have shown great success in code generation. LLMs take as the input a prompt and output the code. How to make prompts (i.e., *Prompting Techniques*) is a key question. Existing prompting techniques are designed for natural language generation and have low accuracy in code generation.

In this article, we propose a new prompting technique named AceCoder. Our motivation is that code generation meets two unique challenges (i.e., requirement understanding and code implementation). AceCoder contains two novel mechanisms (i.e., guided code generation and example retrieval) to solve these challenges. ❶ Guided code generation asks LLMs first to analyze requirements and output an intermediate preliminary (e.g., test cases). The preliminary clarifies requirements and tells LLMs *"what to write."* ❷ Example retrieval selects similar programs as examples in prompts, which provide lots of relevant content (e.g., algorithms, APIs) and teach LLMs *"how to write."* We apply AceCoder to four LLMs (e.g., GPT-3.5, CodeGeeX) and evaluate it on three public benchmarks using the Pass@k. Results show that AceCoder can significantly improve the performance of LLMs on code generation. *In terms of Pass@1, AceCoder outperforms the SOTA baseline by up to 56.4% in MBPP, 70.7% in MBJP, and 88.4% in MBJSP.* AceCoder is effective in LLMs with different sizes (i.e., 6B–13B) and different languages (i.e., Python, Java, and JavaScript). Human evaluation shows human developers prefer programs from AceCoder.

CCS Concepts: • **Computing methodologies** → *Neural networks*; *Natural language processing*; • **Software and its engineering** → **Automatic programming**;

Additional Key Words and Phrases: Code generation, large language models, prompting engineering

This research is supported by the National Natural Science Foundation of China (Nos. 62192731, 62152730), the National Key R & D Program (No. 2023YFB4503801), the National Natural Science Foundation of China (Nos. 62072007, 62192733, 61832009, 62192730), and the Major Program (JD) of Hubei Province (No.2023BAA024).

Authors' Contact Information: Jia Li, Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: lijia@stu.pku.edu.cn; Yunfei Zhao, Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: zhaoyunfei@pku.edu.cn; Yongmin Li, Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: liyongmin@pku.edu.cn; Ge Li (Corresponding author), Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: lige@pku.edu.cn; Zhi Jin, Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University, Beijing, China and School of Computer Science, Peking University, Beijing, China; e-mail: zhijin@pku.edu.cn.

ACM 1557-7392/2024/11-ART204
https://doi.org/10.1145/3675395

ACM Reference format:
Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2024. AceCoder: An Effective Prompting Technique Specialized in Code Generation. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 204 (November 2024), 26 pages. https://doi.org/10.1145/3675395

1 Introduction

Code generation aims to automatically generate the source code based on a natural language requirement. Recently, **large language models (LLMs)** have achieved **state-of-the-art (SOTA)** results on code generation [12, 13, 25, 30, 51]. LLMs do not require fine-tuning and take a prompt as input. A prompt consists of several examples (e.g., <requirement, code pairs>) and a new requirement. LLMs learn code generation from examples and analogously generate code for the new requirement.

The performance of In-Context Learning strongly relies on the prompt surface [50]. How to design prompts (i.e., prompting techniques) is still an open question. Existing prompting techniques (e.g., few-shot prompting [9] and **chain-of-thought (CoT)** prompting [48]) are designed for natural language generation and have low accuracy in code generation. For example, Codex with few-shot prompting only achieves 37.2% Pass@1[1] on a real-world benchmark—HumanEval [12]. Thus, exploring more advanced prompting techniques for code generation is necessary.

In this article, we propose a novel prompting technique specialized in code generation, named AceCoder**.** It significantly improves the performance of LLMs in code generation. Our motivation is that code generation aims to build a mapping from natural language requirements to source code. There are two unique challenges in this mapping, i.e., requirement understanding and code implementation. AceCoder proposes two novel mechanisms to alleviate two challenges. The details of AceCoder are shown as follows.

Challenge 1: Requirement Understanding. Understanding requirements is the starting point of code generation. In real-world programming problems, the requirement may be a brief purpose without specific details. For example, a requirement from a real-world benchmark—**Mostly Basic Programming Problems (MBPP)** [7] is `write a function to check if the triangle is isosceles or not`. Before writing code, we need to analyze the requirement and determine specific details, e.g., input–output formats, and possible exceptions.

Novelty 1: Guided Code Generation. To alleviate this challenge, we propose *guided code generation*. Our motivation is that human developers often use some software artifacts to assist in analyzing requirements. For example, in test-driven development [8], developers clarify requirements by designing test cases. It forces developers to think about details of requirements, e.g., input–output formats and boundary values. These test cases exactly define the requirement and tell developers *what to write*.

To implement the above process, we design a special prompt consisting of triple examples (i.e.,<requirement, preliminary, code>). A preliminary is a specific software artifact (e.g., test cases, APIs) for clarifying the requirement. Given a new requirement, based on the prompt, LLMs first output a preliminary and then generate code based on the preliminary. We illustrate the guided code generation in Section 2 and describe the details in Section 3.3.

Challenge 2: Code Implementation. After understanding the requirements, implementing the source code using a programming language is challenging. It requires LLMs to master related

[1]Pass@k (e.g., $k = 1$) is a widely used evaluation metric in code generation. It denotes the percentage of programs passing all test cases within models' outputs. The detailed definition of Pass@k can be found in Section 4.2.

grammar, algorithms, and libraries. Even for human developers, it is difficult to write an exactly correct program from scratch.

Novelty 2: Example Retrieval. To solve the above challenge, we propose *example retrieval*. It is inspired by the human developers' code reuse. In real-world scenarios, given a requirement, developers often refer to programs with similar requirements. They learn programming skills (e.g., APIs) or directly reuse relevant content from similar programs [23, 29].

Specifically, we use a *retriever* to search for programs with similar requirements. Considering the maximum input length of LLMs is limited (e.g., 1,024 tokens), the number of examples in a prompt is also limited, such as three examples. Thus, we further design a *selector* to select a set of programs from retrieved results as examples. The selector will filter out redundant programs and pick informative examples. Then, examples are inserted into prompts and teach LLMs how to implement code. We illustrate the example retrieval in Section 2 and describe the details in Section 3.2.

In conclusion, given a requirement, AceCoder generates a program in three steps:

- —*Example retrieval.* It uses a *retriever* and a *selector* to search for examples, i.e., <requirement, code> pairs.
- —*Prompt construction.* It uses an *analyzer* to convert example into <requirement, preliminary, code> triples. Then, it concatenates examples with the input requirement together to construct a prompt.
- —*Code generation.* It feeds the prompt into LLMs. By learning from examples, LLMs output an intermediate preliminary and then generate the source code.

We apply AceCoder to four representative LLMs, i.e., GPT-3.5 [32], CodeGeeX [51], CodeGen [30], and InCoder [13]. We conduct extensive experiments on three popular code generation benchmarks, i.e., MBPP [7], **Mostly Basic Java Problems (MBJP)** [6], and **Mostly Basic JavaScript Problems (MBJSP)** [6]. We employ Pass@k ($k = 1, 3, 5$) to measure the performance of different approaches. We obtain some findings from experimental results. ❶ AceCoder significantly outperforms existing prompting techniques. In terms of Pass@1, AceCoder outperforms the SOTA baseline—few-shot prompting by up to 56.4% in MBPP, 70.7% in MBJP, and 88.4% in MBJSP. The improvements prove the superiority of AceCoder in code generation. ❷ AceCoder substantially outperforms retrieval-based models. In terms of Pass@1, AceCoder outperforms the SOTA retrieval-based baseline by up to 13.1% in MBPP, 23.44% in MBJP, and 15.8% in MBJSP. ❸ AceCoder is effective in LLMs of different sizes. We apply AceCoder to three LLMs, which scale from 6B to 13B. In terms of Pass@1, AceCoder improves CodeGeeX-13B by up to 88.4%, CodeGen-6B by up to 65.5%, and InCoder-6B by up to 57.5%. ❹ Human evaluation shows that human developers prefer programs generated by AceCoder. Results show that AceCoder outperforms the SOTA baseline in multiple aspects, including correctness, code smell, and maintainability. ❺ We explore the contributions of different modules and discuss different designs for AceCoder. Results show that three modules are all necessary and our designs for three modules are superior to multiple alternates.

We summarize our contributions in this paper as follows:

- —We propose a novel prompting technique named AceCoder, for improving the performance of LLMs in code generation.
- —AceCoder contains two novel techniques (i.e., guided code generation and example retrieval) to alleviate two challenges (i.e., requirement understanding and code implementation) in code generation, respectively.
- —We apply AceCoder in three LLMs and conduct extensive experiments on three public benchmarks. Qualitative and quantitative experiments show that AceCoder significantly outperforms the SOTA baselines (e.g., CoT prompting, few-shot prompting).

(a) Input requirement:

```
# Write a python function to remove first and last
occurrence of a given character from the string
```

(b) Unit tests:

```
(1) ("hello world", "x") -> "hello world"
(2) ("hello world", "l") -> "helo word"
```

(c) Few-shot Prompting:

```
def remove_Occ(s, ch):
  return s.replace(ch, '', 1).replace(ch, '', -1)
```

Evaluation: (1) pass (2) fail

(d) AceCoder:

```
# ("test", "x") -> "test"
# ("test", "e") -> "tst"
# ("test", "t") -> "es"
def remove_Occ(s, ch):
  first_idx, last_idx = -1, -1
  for i in range(len(s)):
    if s[i] == ch:
      if first_idx == -1:
        first_idx = i
      last_idx = I
  # case 1: ch not in s
  if first_idx == -1 and last_idx == -1:
    return s
  # case 2: ch appears once in s
  elif first_idx == last_idx:
    return s[:first_idx] + s[first_idx+1:]
  # case 3: ch appears multi-times in s
  else:
    return s[:first_idx] + \
s[first_idx+1:last_idx] + s[last_idx+1:]
```

Evaluation: (1)(2) pass

Fig. 1. A motivating example of guided code generation.

Article Organization. Section 2 shows some examples to illustrate the motivations of AceCoder. Section 3 presents the details of our AceCoder. In Section 4 and Section 5, we conduct a large-scale study and evaluate the effectiveness of AceCoder. Section 6 discusses the differences between AceCoder and related works. Section 7 surveys related works about prompting techniques in code generation. Finally, Section 8 summarizes this article and points out some future directions.

2 Motivating Examples

In this section, we explain our motivations by some real cases.

Requirement Understanding → Guided Code Generation. Figure 1(a) and (b) show a requirement from a real-world benchmark [7] and its unit test for evaluation, respectively. We select Codex as the base model. Figure 1(c) shows a program generated by few-shot prompting. The program fails, as it ignores some essential scenarios in the requirement, such as `ch` appearing multiple times in `s`. It shows that comprehensively understanding the requirements is crucial to writing correct programs.

Thus, we propose guided code generation, which asks LLMs first to analyze the requirement and then generate code. Figure 1(d) shows a program generated by AceCoder. We consider test cases to be the intermediate preliminary. We can see that the generated test cases cover multiple

(a) Input requirement:

```
# Write a function to find sequences of lowercase
letters joined with an underscore.
```

(b) Unit tests:

```
(1) ('a_b_c') -> True
(2) ('a_ c_') -> False
```

(c) Few-shot Prompting:

```
def text_lowercase_underscore(text):
  words = text.split()
  for word in words:
    if word.islower() and '_' in word:
      return True
  return False
```

Evaluation: (1) pass (2) fail

(d) Retrieved Programs:

```
Program-1: find sequences of literals in a string.
def find_literals(text, pattern):
  match = re.search(pattern, text)
  (more lines...)
Programs-2&3: re.search(…)
Program-4: split a string at lowercase letters.
def split_upperstring(text):
  return re.findall("[a-z][^a-z]+", text)
```

(e) AceCoder:

```
def text_lowercase_underscore(text):
  import re
  patterns = '^[a-z]+_[a-z]+$'
  if re.search(patterns, text):
    return True
  else:
    return False
```

Fig. 2. A motivating example of example retrieval.

scenarios, e.g., boundary inputs (“test,” “e”). They further clarify the requirement and benefit the following code implementation. Based on the test cases, AceCoder generates a correct program, which considers three scenarios and gives solutions, respectively. The example shows that our guided code generation can help LLMs to analyze requirements and improve the functional correctness of code.

Code Implementation → Example Retrieval. After understanding the input requirement, implementing the code is challenging. It requires LLMs to use various algorithms or libraries. Figure 2(a) and (b) show a requirement from a real-world benchmark [7] and its unit test for evaluation, respectively. Figure 2(c) shows a failed program generated by few-shot prompting. The requirement is to find sequences of lowercase letters joined with an underscore (e.g., a_b_c). The failed program simply checks whether the sequence contains lowercase letters and underscores rather than joining. We suspect that the model does not know how to judge a string containing lowercase letters joined with an underscore.

To alleviate the above problem, we propose *example retrieval*. Our motivation is that human developers often learn programming skills from similar programs. Figure 2(d) shows a few programs with similar requirements. The retrieval metric is the BM25 score. We sort these programs in the descending order of BM25 scores. We can see that similar programs contain lots of relevant content (e.g., re.search), which benefits code implementation. Thus, we design a retriever to search for

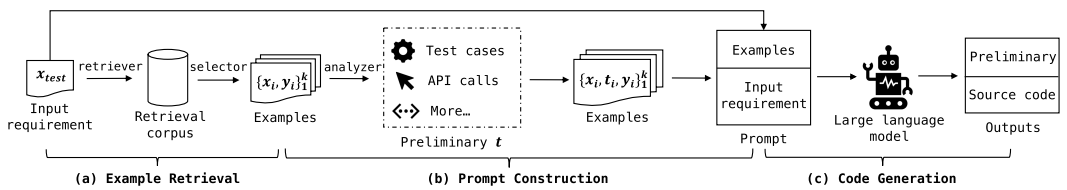

Fig. 3. An overview of AceCoder. Given a requirement, it selects examples from similar programs and constructs a prompt. LLMs first output an intermediate preliminary and then generate the source code. x, y, and t denote requirements, programs, and intermediate preliminaries, respectively.

similar programs as examples in prompts. We expect LLMs can learn from similar programs how to implement new programs.

Since the maximum input length of LLMs is usually limited (e.g., 1,024 tokens), the number of examples in a prompt is limited. Thus, we need to select a few programs from retrieved results as examples. A straightforward idea is to pick programs with the highest similarities. However, since each retrieval is independent, we find that retrieved results may contain redundant programs. For example, Program-1$2$3 in Figure 2(d) are redundant because they all present an API `re.search` that teaches how to search a pattern in the text. Program-4 contains a relevant regular expression, which tells how to design a pattern. Suppose the number of examples is 2. The examples will contain redundant programs (i.e., Program-1 and 2) and miss more informative Program-4.

Thus, we design a selector to filter out redundant programs in retrieved results. Suppose the number of examples is 2. In Figure 2(d), our selector will select Program-1 and Program-4 as examples. Figure 2(e) shows a program generated by AceCoder. It successfully learns how to write regular expressions from Program-4 and learns how to use `re.search` to find patterns from Program-1.

3 AceCoder

In this section, we propose a novel prompting technique for code generation, named AceCoder. We present an overview of AceCoder and then describe its details.

3.1 An Overview

Code generation aims to generate the source code y based on a natural language requirement x. AceCoder leverages LLMs to generate programs via prompting. Figure 3 shows an overview of AceCoder during inference. Given an input requirement x_{test}, AceCoder generates code in three steps.

- *Example Retrieval.* It uses a *retriever* and a *selector* to select k similar <requirement, code> pairs ($\{x_i, y_i\}_{i=1}^k$) from a retrieval corpus as examples.
- *Prompt Construction.* It employs an *analyzer* to convert examples into <requirement, preliminary, code> triples ($\{x_i, a_i, y_i\}_{i=1}^k$). A preliminary is a software artifact for clarifying the requirement, such as test cases. The examples are concatenated with the input requirement to construct a prompt.
- *Code Generation.* The prompt is fed into LLMs. By learning from examples, LLMs output an intermediate preliminary and then generate the code.

where x_i, y_i, a_i denote the requirement, the code, and the preliminary in ith example, respectively.

Input requirement:
```
# Write a function to find sequences of lowercase
letters joined with an underscore in a string.
```
Similar program-1:
```
# find sequences of literals in a string.
def find_literals(text, pattern):
  re.search(…)
```
Similar program-2:
```
# find sequences of an a followed by zero or more b's.
def text_match(text):
  re.search(…)
```
Similar program-3:
```
# find sequences of numbers containing a decreasing
trend or not.
def decreasing_trend(nums):
  re.search(…)
```
Similar program-4:
```
# split a text at lowercase letters.
def split_upperstring(text):
  return re.findall("[a-z][^a-z]*", text)
```

Fig. 4. A requirement and its similar programs.

3.2 Example Retrieval

As shown in Figure 3, the first step has two goals: (1) retrieve similar programs and (2) select a few examples from retrieved programs. We design a *retriever* and a *selector* to achieve these goals, respectively. The details of the two modules are shown as follows.

3.2.1 Retriever. Similar programs often have similar natural language requirements [AceCoder, 15, 23]. There, we take the input requirement as a query to search for similar requirements in a retrieval corpus. In this article, we consider the training data in experimental datasets as the retrieval corpus. Then, we extract the corresponding programs as similar programs.

Specifically, we leverage an open-source search engine named Lucene [3] to build our retriever and use the training data as a retrieval corpus. We employ the BM25 score [39] as the retrieval metric, which is widely used in previous studies [22, 46]. The BM25 score is a bag-of-words retrieval function and is used to estimate the lexical-level similarity of two sentences. The more similar the two sentences are, the higher the value of BM25 scores. In this article, the retriever outputs top-m similar programs based on the BM25 score.

The reason for choosing BM25+Lucene is that they can achieve good retrieval accuracy and have low complexity. Considering that the retrieval corpus is often large-scale, a lightweight retriever is closer to practical applications. In Section 5, we also explore other designs for the retriever and compare them to our design.

3.2.2 Selector. We can obtain top-m similar programs from the retriever. However, the maximum input length of LLMs (e.g., 1,024 tokens) and the inference budget are often limited. It leads that the number of examples (i.e., k) in a prompt is also limited (e.g., three examples). It is necessary to further select k programs from retrieved results as examples.

A straightforward idea is to pick top-k similar programs as examples. However, as the programs are scored independently, we find that retrieved results may contain redundant programs. Figure 4 shows a requirement and its similar programs. Similar programs are ranked by the BM25 score. We can see that top-3 programs are redundant, as all of them use an API (i.e., `re.search`) to find

Algorithm 1: The Algorithm of Our Selector

Inputs:
Input requirement x_{test}, similar programs $\{(x_i, y_i)\}_{i=1}^{m}$;
The number of examples $k, k <= m$, decay factor λ.
Outputs:
Selected examples T, $\{(x_i, y_i)\}_{i=1}^{k}$.

```
1:  T ← Empty Ordered List
2:  S ← Extract_ngrams_with_count(x_test)
3:  for i in {1, ··· , m} do
4:      Q[i] ← Extract_ngrams_with_count(x_i)
5:  end for
6:  while len(T) < k do
7:      for i in {1, ··· , m} do
8:          Score[i] ← Ngram_overlap_score(S, Q[i])
9:      end for
10:     j ← argmax(Score)
11:     T.append((x_j, y_j))
12:     matched_ngrams ← S ∩ Q[j]
13:     Q[j] ← ∅
14:     for i in {1, ··· , m} do
15:         for ngram ∈ match_ngrams do
16:             S[i][ngram]× = λ
17:         end for
18:     end for
19: end while
20: return T
```

sequences of a specific pattern. The Program-4's requirement contains a relevant regex expression. However, as Program-4 has fewer overlapping n-grams with the input requirement, it has a relatively low BM25 score. Obviously, directly selecting top-k (e.g., top-3) retrieved programs is unreasonable, as it will introduce redundant programs and ignore more informative Program-4.

In this article, we design a selector, which can filter out redundant programs in retrieved results. The algorithm of the selector is shown in Algorithm 1. We first extract all n-grams of the input requirement and all similar requirements (lines 2–5). In this article, n is set to 4 by default. Then, we calculate a recall-based ROUGE-n score between the input requirement and each similar requirement using the following equations (lines 7–9).

$$R_n = \frac{\sum_{n\_gram \in S \cap Q} S(n\_gram)}{\sum_{n\_gram \in S} S(n\_gram)} \tag{1}$$

$$Score = \exp\left(\frac{1}{n}\sum_{n} \log(R_n)\right). \tag{2}$$

We get a similar requirement with the maximum score and add its corresponding program to examples (lines 10–11). Then, the matched n-grams between the similar requirement and the input requirement are decayed by a factor λ. This process (lines 6–17) is repeated until the number of examples reaches the upper bound. The motivation for the decay is to filter out redundant programs, i.e., programs with the same matched n-grams. For example, in Figure 4, we first add Program-1 to

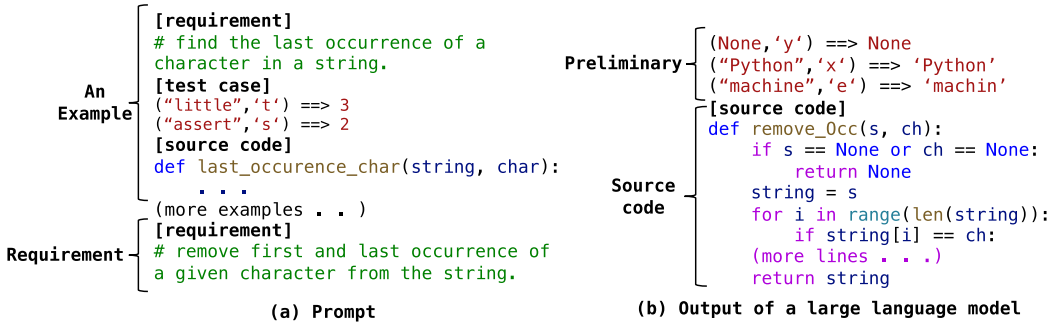

Fig. 5. Examples of our prompt and an LLM's output.

examples and then decay its matched n-grams (e.g., `find sequences of`). Subsequent programs with the same matched n-grams (i.e., Program-2 and Program-3) are considered redundant and will be ignored. Program-4 contains new matched n-grams (e.g., `lowercase letters`) and probably contains new information. Thus, Program-4 will obtain a higher score and is added to the examples.

By the above process, our selector filters out redundant programs and selects k similar programs as examples. In practice, m and k are small numbers, such as $m = 50$, $n = 3$. Thus, the time complexity of our selector is acceptable.

3.3 Prompt Construction

The goal of this step is to construct a prompt. As stated in Section 1, our guided code generation expects that LLMs can output an intermediate preliminary and then generate the final code. To achieve this goal, we design a special prompt consisting of triple examples (i.e., <requirement, preliminary, code>).

Specifically, we first use an analyzer to introduce preliminaries $\{t_i\}_{i=1}^{k}$ into selected examples $\{x_i, y_i\}_{i=1}^{k}$, obtaining triple examples $\{x_i, t_i, y_i\}_{i=1}^{k}$. The preliminary is a software artifact for clarifying requirements. Inspired by test-driven development [8], this article considers test cases as the preliminary by default. We also explore other choices (e.g., APIs, method signature) in Section 5. Then, we concatenate these triple examples with the input requirement to construct a prompt.

Figure 5(a) shows an example of our prompt. The prompt begins with several examples and ends with a new requirement. `[requirement]`, `[test case]`, and `[source code]` are special tags that mark different parts in a triple.

We assume that test cases of examples are available. We think this assumption is acceptable. The reasons are two-fold. First, there are many public code generation datasets containing test cases, e.g., MBPP [7] (474 samples), APPS [16] (5,000 samples), and CodeContest [25] (13,328 samples). We can extract training data from these datasets and construct a retrieval corpus. Second, test-driven software development is popular in real-world scenarios. We can mine software repositories from open-source communities (e.g., GitHub [2]) and extract code snippets equipped with test cases.

3.4 Code Generation

In this step, we leverage an LLM to generate code based on the prompt. Following previous studies [12, 13, 30, 51], we view the LLM as a black-box generator and use it to complete the prompt. By learning from examples in the prompt, LLMs will first output a preliminary (e.g., test cases) and then generate code based on the preliminary and the requirement.

Figure 5(b) shows an output of an LLM—CodeGeeX [51]. We can see that CodeGeeX first generates some test cases and then implements a Python function. The test cases provide lots of valuable information (e.g., input–output formats, invalid inputs) and guide the subsequent code generation.

Table 1. Statistics of the Datasets in Our Experiments

Statistics	MBPP	MBJP	MBJSP
Language	Python	Java	JavaScript
# Train	384	383	383
# Dev	90	90	90
# Test	500	493	493
Avg. tokens in requirement	16.50	16.71	16.53
Avg. tokens in code	92.68	247.79	100.75

4 Study Design

To assess AceCoder, we perform a large-scale study to answer six research questions. In this section, we describe the details of our study, including datasets, evaluation metrics, baselines, and base LLMs.

4.1 Research Questions

Our study aims to answer the following research questions (RQs).

RQ1: How does AceCoder perform compared to existing prompting techniques? This RQ aims to validate that AceCoder has higher accuracy than existing prompting techniques in code generation. We apply AceCoder and baselines to three LLMs and measure their accuracy on three code generation benchmarks. The evaluation metric is Pass@K.

RQ2: How does AceCoder perform compared to retrieval-based models? AceCoder retrieves similar programs as examples in prompts. Some existing studies [19, 35] also introduce information retrieval to augment code generation. In this RQ, we compare AceCoder to these retrieval-based models. The evaluation metric is Pass@K.

RQ3: Do human developers prefer code generated by AceCoder? The ultimate goal of code generation is to assist human developers in writing code. In this RQ, we hire 10 developers (including industry employees and academic researchers) to review the code generated by AceCoder and baselines manually. We measure the quality of code in three aspects, including correctness, code smell, and maintainability.

RQ4: What are the contributions of different modules in AceCoder? AceCoder contains three modules, i.e., a retriever, a selector, and an analyzer. This RQ is designed to analyze the contributions of three modules to the performance. We select a base model, gradually introduce three modules, and observe the fluctuations in accuracy.

RQ5: What are the better designs for three modules? This RQ aims to validate the superiority of our designs for three modules in AceCoder. Specifically, we explore multiple designs for three modules and compare them to our designs.

4.2 Evaluation Datasets and Metrics

4.2.1 Datasets. We conduct experiments on three public code generation benchmarks, including the MBPP in Python, MBJP in Java, and MBJSP in JavaScript. The statistics of the datasets are shown in Table 1. The details of the datasets are described as follows.

— *MBPP* [7] contains 974 real-world programming problems that are constructed by crowdsourcing. Each problem contains a natural language requirement, a single Python function, and three test cases.

—*MBJP* [6] and *MBJSP* [6] both contain 966 crowd-sourced programming problems in Java and JavaScript, respectively. Each problem consists of a natural language requirement, an individual function, and three test cases.

4.2.2 Metrics. Following previous code generation studies [12, 13, 30, 51], we employ Pass@k as our evaluation metric. Specifically, we generate k programs for each requirement. A requirement is considered solved if any generated programs pass all test cases. We compute the percentage of solved requirements in total requirements as Pass@k. In this article, k is set to 1, 3, and 5.

We notice that previous studies [15, 45] also use some match-based metrics (e.g., **Bilingual Evaluation Understudy (BLEU)** [34]). These metrics are initially designed for natural language generation and are poor in measuring the functionality of programs [12]. Thus, we omit them in experiments.

4.3 Comparison Baselines

This article is to propose a new prompting technique for code generation. Thus, we select three existing prompting techniques as baselines.

—*Zero-shot prompting* [12, 30] directly feeds the input requirement into LLMs. Then, it extracts the code from LLMs' outputs.
—*Few-shot prompting* [12] randomly selects several (3 in this article) <requirement, code> pairs from the training data as examples and constructs a prompt, which is fed into an LLM. Then, it extracts the code from LLMs' outputs. The prompt of few-shot prompting is available in our replication package [AceCoder].
—*CoT prompting* [48] is a variant of few-shot prompting. CoT prompting uses several (3 in this article) <requirement, intermediate steps, code> triples as examples and constructs a prompt. Based on the prompt, LLMs first generate a series of intermediate steps and then output the code. The prompt of CoT prompting is available in our replication package [AceCoder].

AceCoder retrieves similar programs to assist LLMs in generating code. Some studies also introduce information retrieval to augment code generation. We compare AceCoder to these retrieval-based models.

—REDCODER [35] retrieves similar programs and fine-tunes a pre-trained model—PLBART [5] to generate code based on the requirement and similar programs.
—*Jigsaw* [19] searches for similar programs from API documentation and insert them into the prompts.

4.4 Base LLMs

We select three open-source LLMs as base models. The details of the base models are shown as follows.

—*GPT-3.5* [32] is a variant of gpt-3 [9] through the **reinforcement learning with human feedback (RLHF)**. The RLHF can improve models' instruction-following capabilities and avoid the generation of harmful or toxic content. In this article, we utilize the `gpt-3.5-turbo-0301` version.
—*CodeGeeX* [51] is a multilingual LLM for source code with 13 billion parameters. CodeGeeX is pre-trained with a large corpus of more than 20 programming languages (e.g., Python, Java, and JavaScript). We download the model weight and run CodeGeeX following official instructions.

Table 2. The Results of AceCoder and Prompting Baselines on Three Datasets

Base model	Prompting Technique	MBPP			MBJP			MBJSP		
		Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
GPT-3.5	Zero-shot prompting	50.47	58.20	61.40	51.32	62.07	65.52	48.28	55.81	57.24
	CoT prompting	52.80	61.00	63.00	55.78	63.29	65.52	51.36	57.45	60.88
	Few-shot prompting	52.20	59.24	61.92	53.81	62.71	64.33	50.11	56.74	59.12
	AceCoder	**57.82**	**64.74**	**66.83**	**57.36**	**68.50**	**70.73**	**54.38**	**61.75**	**63.42**
Relative Improvement		**8.3%**	**6.9%**	**7.6%**	**2.8%**	**8.2%**	**8%**	**5.9%**	**7.5%**	**4.2%**
CodeGeeX-13B	Zero-shot prompting	5.20	13.80	19.40	4.46	11.97	18.26	0.20	0.20	0.41
	CoT prompting	12.60	23.40	30.20	14.40	28.19	33.67	11.35	21.10	25.96
	Few-shot prompting	20.40	30.60	36.00	16.63	26.17	34.48	11.16	19.88	25.56
	AceCoder	**26.74**	**36.43**	**41.13**	**28.38**	**36.79**	**41.54**	**21.03**	**31.44**	**36.04**
Relative Improvement		**31.1%**	**19%**	**14.2%**	**70.7%**	**40.6%**	**20.5%**	**88.4%**	**58.2%**	**31%**
CodeGen-6B	Zero-shot prompting	10.40	19.40	24.40	14.81	25.76	31.44	8.72	19.67	22.92
	CoT prompting	13.00	21.00	26.00	13.59	25.35	31.24	11.56	20.08	24.54
	Few-shot prompting	14.60	24.00	30.20	18.25	30.02	34.68	9.94	19.88	23.12
	AceCoder	**22.83**	**34.58**	**40.16**	**22.45**	**34.27**	**40.96**	**16.45**	**27.31**	**32.16**
Relative Improvement		**56.14%**	**44.1%**	**33%**	**23%**	**14.2%**	**18.1%**	**65.5%**	**37.4%**	**39.1%**
InCoder-6B	Zero-shot prompting	4.20	11.40	16.20	2.23	5.88	9.13	3.65	5.88	8.11
	CoT prompting	3.99	10.65	15.31	1.83	4.46	7.10	1.22	2.03	4.67
	Few-shot prompting	12.80	22.80	28.20	10.95	23.53	26.17	12.78	22.52	27.79
	AceCoder	**20.16**	**31.44**	**34.10**	**16.37**	**29.89**	**34.74**	**15.97**	**27.13**	**30.65**
Relative Improvement		**57.5%**	**37.9%**	**20.9%**	**49.5%**	**27%**	**32.7%**	**25%**	**20.5%**	**10.3%**

The bold indicates important experimental results. The numbers in red denote AceCoder's relative improvements compared to the SOTA baseline—few-shot prompting.

—*CodeGen* [30] is a family of LLMs for source code that is pre-trained with extensive natural language and code data. We select CodeGen-Multi-6.1B (CodeGen-6B) as a base model.
—*InCoder* [13] is a multilingual LLM for code generation. It is pre-trained with 216 GB of code data. We use a version with 6.7 billion parameters (InCoder-6B) as a base model.

4.5 Implementation Details

Example Retrieval. For each dataset, the retrieval corpus is its training data. We exclude the ground truths from the outputs of our retriever. We first retrieve top 20 similar programs and then use the selector to select three examples. The hyper-parameters—n and λ are set to 4 and 0.1, respectively. Both default values are determined based on an initial hyper-parameter search on the development data. To ensure fairness, the number of examples in AceCoder and baselines is the same.

Prompt Construction. In experimental datasets, the retrieval corpus (i.e., training data) has been equipped with test cases by data collector [6, 7]. Thus, the analyzer utilizes pre-defined rules to extract test cases and transform retrieved programs into <requirement, test cases, code> triples.

Code Generation. Following previous studies [12, 13, 30], we use nucleus sampling [17] to decode programs from LLMs. The temperature is 0.8 and the top-p is 0.95. The maximum generated lengths are 400, 500, and 500, respectively. The sampling settings of baselines are the same as the ones of AceCoder.

5 Results and Analyses

In the first research question, we evaluate the performance of AceCoder with respect to existing prompting techniques.

RQ1: How does AceCoder perform compared to existing prompting techniques?

Setup. We apply AceCoder and three prompting baselines to three base models (Section 4.4). Then, we use Pass@k to measure their performance on three benchmarks (Section 4.2).

Table 3. The Comparison of Retrieval-Based Baselines and AceCoder

Approach	MBPP			MBJP			MBJSP		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
REDCODER	3.37	6.21	9.74	4.46	7.51	9.94	4.87	10.34	12.78
Jigsaw	23.65	33.97	37.78	22.99	33.26	36.95	18.16	28.79	34.08
AceCoder	**26.74**	**36.43**	**41.13**	**28.38**	**36.79**	**41.54**	**21.03**	**31.44**	**36.04**
Relative Improvement	**13.1%**	**7.2%**	**8.9%**	**23.4%**	**10.6%**	**12.4%**	**15.8%**	**9.2%**	**5.8%**

The bold indicates important experimental results. The numbers in red denote AceCoder's relative improvements compared to the SOTA baseline—Jigsaw.

Results. The results on three benchmarks are shown in Table 2. The numbers in red denote AceCoder's relative improvements compared to the SOTA baseline—few-shot prompting.

Analyses. ❶ AceCoder performs better than baselines on three benchmarks. Compared to the SOTA baseline—few-shot prompting, in terms of Pass@1, AceCoder outperforms it by up to 56.14% in MBPP, 70.7% in MBJP, and 88.4% in MBJSP. Pass@1 is a rigorous metric that is difficult to improve. The significant improvements prove the superiority of AceCoder in code generation. We attribute the improvements to our novel techniques, i.e., example retrieval and guided code generation. The retrieved examples contain many relevant code elements teaching LLMs "how to write." Guided code generation asks LLMs to analyze requirements that tell LLMs "what to write." ❷ AceCoder is effective in different LLMs and programming languages. AceCoder achieves substantial improvements on general LLMs (e.g., GPT-3.5) and code LLMs (e.g., CodeGeeX). Besides, AceCoder works well on LLMs with different sizes. Compared to few-shot prompting, in terms of Pass@1, AceCoder improves CodeGeeX-13B by up to 88.4%, CodeGen-6B by up to 65.5%, and InCoder-6B by up to 57.5%. In particular, we find that an LLM with AceCoder even outperforms larger LLMs. For example, in the MBJSP, InCoder-6B with AceCoder outperforms CodeGeeX-13B with few-shot prompting. It proves the potential of AceCoder. AceCoder is also language-agnostic and is effective in multilingual code generation (i.e., Python, Java, and JavaScript).

> *Answer to RQ1:* AceCoder outperforms existing prompting techniques on three benchmarks. In terms of Pass@1, AceCoder outperforms the SOTA baseline by up to 56.4% in MBPP, 70.7% in MBJP, and 88.4% in MBJSP. Besides, AceCoder is effective in LLMs with different sizes. It improves CodeGeeX-13B by up to 88.4%, CodeGen-6B by up to 65.5%, and InCoder-6B by up to 57.5%. The significant improvements prove the effectiveness of AceCoder in code generation.

RQ2: How does AceCoder perform compared to retrieval-based models?

Setup. In this RQ, we compare AceCoder to two retrieval-based baselines, including REDCODER [35] and Jigsaw [19]. Baselines and AceCoder use the same retrieval corpus. Because REDCODER requires fine-tuning, we follow the official instructions and use the training data to train REDCODER.

Results. The results on three benchmarks are shown in Table 3. The numbers in red denote AceCoder's relative improvements compared to the SOTA baseline—Jiagsaw.

Analyses. ❶ AceCoder outperforms retrieval-based baselines in three benchmarks. Compared to the SOTA baseline—Jigsaw, in terms of Pass@1, AceCoder outperforms it by up to 13.1% in MBPP, 23.44% in MBJP, and 15.8% in MBJSP. Jigsaw also retrieves similar programs for making prompts. The improvements show the effectiveness of our selector and analyzer. The selector filters out redundant similar programs and further improves the quality of examples. The analyzer constraints LLMs to first analyze requirements and then generate code. Besides, we notice that REDCODER has poor accuracy in three benchmarks. This is because the training data is limited,

Table 4. The Results of Human Evaluation

Approach	Correctness	Code smell	Maintainability
Zero-shot prompting	0.3167	1.1033	1.2749
CoT prompting	0.6671	1.1405	1.4479
Few-shot prompting	0.9769	1.2148	1.5420
AceCoder	**1.5802 (↑ 61.8%)**	**1.6241 (↑ 33.7%)**	**1.7544 (↑ 13.8%)**

The bold indicates important experimental results. The values in parentheses are the relative improvements compared to the SOTA baseline—few-shot prompting.

and fine-tuning easily leads to overfitting. It validates our motivation that introducing similar programs by prompting is a more suitable approach to LLMs.

Answer to RQ2: AceCoder outperforms retrieval-based baselines. Specifically, it outperforms the SOTA baseline—Jigsaw by up to 13.1% in MBPP, 23.44% in MBJP, and 15.8% in MBJSP.

RQ3: Do human developers prefer code generated by AceCoder?

Setup. The ultimate goal of code generation is to assist human developers in writing code. Thus, we conduct a human evaluation to measure programs generated by AceCoder and baselines. We follow the settings of human evaluation in previous studies [14, 23].

Metrics. We manually evaluate programs in three aspects:

- —Correctness (whether the program satisfies the given requirement). 0 point: the program is totally inconsistent with the requirement. 1 point: the program is implemented, but misses some details. 2 points: the program is correctly implemented.
- —Code Smell (whether the program contains bad code smell). 0 point: There are better solutions in terms of performance. Or there is serious code smell. 1 point: Some details are not in place. There is code smell of low severity. 2 points: No obviously better code in terms of performance exists. If possible, resources are released accordingly. No obvious code smell.
- —Maintainability (whether the implementation is standardized and has good readability). 0 point: The program does not follow a consistent specification, or there are many meaningless names in variable naming, or there are certain repetitions and redundant codes. 1 point: The program implementation meets certain specifications. But some variable names can be further refined. 2 points: The program implementation is relatively standardized, the variable naming is basically semantically straightforward, and the readability is better.

We explain the above aspects to evaluators through some examples. After discussing with evaluators, we set the score of each aspect to an integer, ranging from 0 to 2 (from bad to good).

Sampling Strategy. We randomly select 200 testing samples from a benchmark—MBPP. Then, we consider the CodeGen-6B as a base model and leverage four prompting techniques (i.e., three baselines and our approach) to generate programs based on the 200 samples. In this way, we obtain 800 (200*4) generated programs for human evaluation.

We recruit 10 participants with 3–5 years of development experience to evaluate the generated programs in the form of a questionnaire. The participants are developers from IT companies and academics in universities. The 800 programs are divided into five groups, with each questionnaire containing one group. The programs are randomly shuffled and anonymously reviewed by evaluators. Two evaluators evaluate each group, and the final score is the average of the two evaluators' scores. Evaluators are allowed to search the Internet for unfamiliar concepts. All evaluators obtain adequate payments given their countries of residence.

Table 5. Inter-Rater and Intra-Rater Reliability in Human Evaluation

	Inter-Rater Reliability	Intra-Rater Reliability
Correctness	0.89	0.96
Code Small	0.81	0.95
Maintainability	0.79	0.94

We use Cohen's Kappa coefficients as the metrics. Cohen's Kappa coefficients in all metrics are greater than 0.75.

Table 6. The Results of Ablation Study

Retriever	Selector	Analyzer	MBPP			MBJP			MBJSP		
			Pass@1 (%)	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
✗	✗	✗	20.40	30.60	36.00	16.63	26.17	34.48	11.16	19.88	25.56
✓	✗	✗	24.00 (↑ 17.6%)	34.60	38.20	23.35 (↑ 40.4%)	33.67	37.22	18.66 (↑ 67.2%)	29.18	34.89
✓	✓	✗	24.89 (↑ 22%)	35.02	39.14	25.03 (↑ 50.5%)	34.47	39.24	19.73 (↑ 76.8%)	30.16	35.34
✓	✓	✓	**26.74 (↑ 31.1%)**	**36.43**	**41.13**	**28.38 (↑ 70.7%)**	**36.79**	**41.54**	**21.03 (↑ 88.4%)**	**31.44**	**36.04**

The bold indicates important experimental results. The values in parentheses are relative improvements compared to few-shot promopting.

Results. The results of the human evaluation are shown in Table 4. The values in parentheses are the relative improvements compared to the SOTA baseline—few-shot prompting. We use Cohen's Kappa coefficients to measure the inter-rater and intra-rater reliability. The results are shown in Table 5. The Cohen's Kappa coefficients on all metrics are greater than 0.75. The results demonstrate that our human evaluation is valid.

Analyses. ❶ AceCoder is better than all baselines in three aspects. Specifically, our AceCoder outperforms the SOTA baseline—few-shot prompting by 61.8% in correctness, 33.7% in code smell, and 13.8% in maintainability. The improvements show that AceCoder has better usability and is promising in practical applications. Besides, all the p-values are substantially smaller than 0.05, which shows the improvements are statistically significant.

Answer to RQ3: Human evaluation shows that human developers prefer programs generated by AceCoder. It outperforms the SOTA baseline by 61.8% in correctness, 33.7% in code smell, and 13.8% in maintainability.

RQ4: What are the contributions of different modules in AceCoder?

Setup. AceCoder contains three modules, i.e., a retriever, a selector, and an analyzer. This RQ is designed to analyze the contributions of three modules to the performance. We select CodeGeeX as the base model and conduct an ablation study by gradually adding three modules.

Results. The results are shown in Table 6. ✓ and ✗ represent adding and removing corresponding modules, respectively. Without three modules, the base model uses few-shot prompting to generate code. After adding a retriever, the base model selects top-k similar programs as examples and directly generates code. After adding a selector, the base model selects k examples from similar programs and then generates code. After further introducing an analyzer, the base model uses AceCoder to generate code.

Analyses. ❶ All modules are necessary for AceCoder to perform the best. After adding a retriever, the performance of the base models is improved. In terms of Pass@1, the retriever brings a 17.6% improvement in MBPP, a 40.4% improvement in MBJP, and a 67.2% improvement in MBJSP. It validates our motivation that retrieved programs contain lots of useful information that benefits code generation. After adding a selector, the performance of the base model is further improved.

It shows that our selector can effectively filter out redundant programs in retrieved results and improve the quality of examples. After further introducing an analyzer, the base model achieves better results. In terms of Pass@1, the base model is improved by 31.1% in MBPP, 70.7% in MBJP, and 88.4% in MBJSP. It proves the effectiveness of guided code generation in analyzing requirements.

> *Answer to RQ4:* Three modules are essential for the performance of AceCoder. The performance of CodeGeeX on three benchmarks is substantially improvement by gradually adding three modules.

RQ5: What are the better designs for three modules in AceCoder?

Setup. As stated in Section 3.1, AceCoder contains three modules, i.e., a retriever, a selector, and an analyzer. In this RQ, we explore different designs for three modules and validate the superiority of our designs. We select CodeGeeX as the base model. The evaluation settings are shown as follows:

(1) A retriever takes the input requirement as a query and searches for similar programs from a retrieval corpus. We design two choices for the retriever:

- —Dense retriever. It uses a neural encoder to convert the requirements into vector representations. Then, it retrieves similar programs based on the similarity of vector representations. In experiments, we use an off-the-shelf natural language representation model [38] as the encoder.
- —Sparse retriever (AceCoder). As stated in Section 3.2, it uses the BM25 score as the retrieval metric. BM25 score can measure the lexical-level similarity of two requirements.

(2) A selector aims to score similar programs and filter redundant programs. For the score function in the selector (line 8 of Algorithm 1), we design two choices:

- —BLEU [34]. It extracts overlapping n-grams between the input requirement and the similar requirement. Then, it computes the precision of n-grams in the similar requirement.
- —ROUGE-N [26] (AceCoder). It extracts overlapping n-grams between the input requirement and the similar requirement. Then, it computes the recall of n-grams in the input requirement.

(3) An analyzer is to introduce preliminaries into examples. A preliminary is a special software artifact that benefits the requirement understanding. For the preliminary, we design three choices:

- —API sequence. APIs are important elements in code and reflect the functionality of the code. Pre-designing APIs help LLMs to think about how to solve requirements. We use a program analysis tool [4] to extract APIs from examples and view the API sequence as a preliminary (e.g., `open, numpy.array, write`).
- —Method signature. It contains input–output parameters and their types, which clearly indicate the inputs and outputs of requirements. Thus, we consider the method signature as a preliminary (e.g., `def` $floor_{Min}$`(A: int, B: int, N: int)`).
- —Test cases (AceCoder). Test cases exactly define the requirement, including the input–output format, edge cases, and functionality. We consider several test cases as the preliminary, such as `(“Python,”“o”) ---> 1); (“little,”“t”) ---> 2);`.

Results and Analyses. The results are shown in Table 7. “w/” is the abbreviation of with. ❶ A dense retriever is comparable to our retriever but has a lower efficiency. In Table 7, compared to AceCoder, AceCoder with dense retriever has a slight drop in performances. This indicates that code generation prefers lexically similar programs that contain a lot of reusable content. Similar findings can be found in code completion work [27]. Besides, the dense retriever has a higher complexity and is hard to apply to a large-scale retrieval corpus. ❷ The BLEU selector prefers shorter examples and is suboptimal. Compared to AceCoder, AceCoder with BLEU selector has an obvious decrease in accuracy. We inspect some failed samples and find that the BLEU selector

Table 7. The Performance of AceCoder with Different Designs

Approach	MBPP			MBJP			MBJSP		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
AceCoder	**26.74**	**36.43**	**41.13**	**28.38**	**36.79**	**41.54**	**21.03**	**31.44**	**36.04**
w/ Dense retriever	26.63	36.42	41.10	28.16	36.55	41.32	20.88	31.27	35.94
w/ BLEU selector	25.61	35.71	40.74	27.86	35.91	40.77	20.15	30.42	35.47
w/ API analyzer	25.10	35.24	40.38	26.44	35.16	40.12	19.86	30.23	35.41
w/ signature analyzer	26.14	35.96	40.89	27.35	36.11	40.98	20.58	30.89	35.86

The bold indicates important experimental results. w, with.

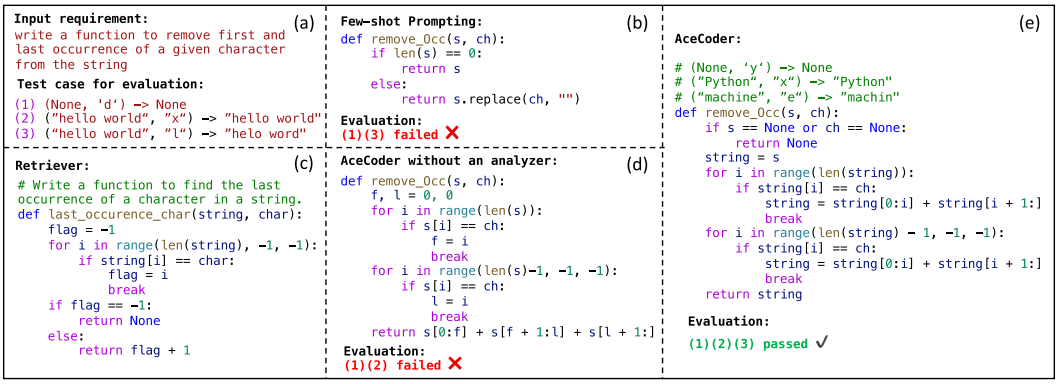

Fig. 6. Some programs generated by AceCoder and baselines.

prefers shorter examples. This is because BLEU is the precision of n-gram in similar requirements. The shorter the similar requirement, the higher the BLEU. It leads that the selector tends to select short programs as examples and ignores some informative but long examples. ❸ Test cases are more suitable for the preliminary than APIs and method signatures. We carefully inspect some cases. First, many requirements do not require APIs or only involve a few trivial APIs (e.g., range, split, and len). It causes that generated APIs bring limited benefits to code generation. Second, by generating method signatures, LLMs are asked to consider the input–output format, which benefits code generation. However, method signatures miss other necessary details, such as edge cases. AceCoder considers test cases as the preliminary. Test cases are common in code files. Thus, it is feasible for LLMs trained with extensive code data to generate plausible test cases. With the guidance of test cases, LLMs can comprehensively understand requirements and determine related details (e.g., input–output formats, boundary inputs, outliers), thus generating more correct programs.

Answer to RQ5: We explore the other four designs for AceCoder and compare them to our designs. Results on three benchmarks show the superiority of our design.

6 Discussion

6.1 Why AceCoder Works

In this section, we discuss why AceCoder works through some real cases. Figure 6 shows some programs generated by our AceCoder and baselines on the MBPP dataset. Based on these cases, we analyze two reasons why AceCoder works.

❶ AceCoder considers similar programs as examples, which contain many relevant elements and teach LLMs "how to write." Figure 6(a) shows a real requirement and the corresponding test cases

for evaluation from the MBPP dataset. Figure 6(b) shows a program generated by the few-shot prompting. Few-shot prompting randomly selects programs as examples and generates a wrong solution, i.e., removing all occurrences of the character. It shows that generating a correct program from scratch is challenging.

AceCoder proposes to retrieve similar programs as examples. Figure 6(b) shows the top-1 similar program. We can see that the similar program provides many reusable code elements (e.g., control flows—for i in range(len(string)-1, -1, -1):). The similar code teache s models how to find a specific character in a string. Figure 6(e) shows the correct program generated by AceCoder. AceCoder learns a well-formed algorithm structure from the similar program and introduces details based on the input requirement.

❷ AceCoder helps LLMs comprehensively understand requirements and know "what to write." AceCoder designs an analyzer to help LLMs understand requirements. Figure 6(d) shows a program generated by AceCoder without an analyzer. The program looks well but fails in test cases (1) and (2). This is because the LLMs do not comprehensively understand the input requirement and ignore important boundary inputs. Figure 6(e) shows test cases and a program generated by our AceCoder. After adding an analyzer, LLMs first reason test cases as a preliminary and then generate programs. The generated test cases contain various boundary inputs and define the requirement exactly. LLMs know "what to write" based on the test cases and further generate correct programs.

6.2 AceCoder vs. CoT Prompting

Our guided code generation is similar to CoT prompting. Both approaches ask LLMs to generate an intermediate result and then output the final code. The intermediate result in CoT prompting is a series of natural language steps describing how to write code step by step. In contrast, AceCoder leverages some software artifacts (e.g., test cases) as the intermediate result.

We argue that our guided code generation is superior to the CoT in code generation. Table 2 shows the comparison results between AceCoder and CoT prompting. CoT prompting achieves slight improvements over few-shot prompting and is even worse than zero-shot prompting. We inspect some failed samples and summarize the main reason. We find that CoTs describe how to write code in a series of steps almost at the same level as code. The LLMs for source code are mainly pre-trained with code data and are relatively weak in natural language generation. The generated CoTs often contain ambiguities or errors and negatively affect the subsequent code generation. Similar findings can be found in the original article of CoT prompting [48]. Compared to CoT prompting, AceCoder uses a software artifact (i.e., test cases) as intermediate preliminaries. Compared to natural languages, test cases are more suitable to clarify requirements and contain fewer ambiguities. Besides, test cases are common in real-world code files, and LLMs have abilities to generate plausible test cases. Thus, AceCoder is different from CoT prompting and is more promising than CoT prompting in code generation.

6.3 AceCoder vs. Rank Techniques

Some recent studies [11, 18] propose *rank techniques* to improve the performance of LLMs on code generation. Given a requirement, they first sample many programs from LLMs and then use test cases or neural networks to rerank sampled programs.

In this article, we do not directly compare our approach to rank techniques. The reason is that AceCoder and rank techniques have different focuses, and they are complementary. Our work is a new prompting technique that improves the accuracy of LLMs in code generation. Rank techniques do not care about LLMs and aim to select the best one from LLMs' multiple outputs. In practice, users can use AceCoder to generate many programs and then use rank techniques to pick a final output. Thus, we omit them in experiments.

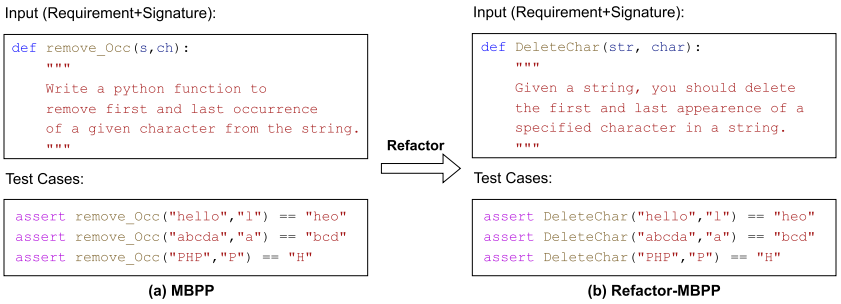

Fig. 7. (a) A testing sample in MBPP and (b) its corresponding refactored version in Refactor-MBPP.

6.4 Threats to Validity

There are three main threats to the validity of our work.

The Impact of Retrieved Programs. AceCoder relies on the quality of retrieved programs. Intuitively, when the retrieved code is less relevant to the target code, the performance of AceCoder may suffer. To address this threat, we have three thoughts.

❶ It is feasible to construct a reliable retrieval corpus. A large-scale study on 13.2 million real-world code files found the proportion of reused code is up to 80% [29]. Besides, many public code generation datasets (i.e., the training data) provide well-formed nl-code pairs, e.g., AixBench-L (190,000 pairs) [23], CodeContest [25] (13,328 pairs), and APPS [16] (5,000 pairs). Therefore, we believe that it is quite possible to retrieve similar programs in real development scenarios.

❷ AceCoder brings substantial improvements with a few nl-code pairs. In our experiments, the retrieval corpus (i.e., training data) only contains 384 nl-code pairs. Experiments show that AceCoder outperforms the SOTA baseline by up to 56.4% in MBPP, 70.7% in MBJP, and 88.4% in MBJSP. The results demonstrate that AceCoder works with only a few nl-code pairs.

❸ The performance of AceCoder may degrade when faced with rare software libraries. When the requirements involve rare third-party libraries, it may be difficult to retrieve relevant programs. In this scenario, AceCoder may degrade to few-shot prompting (i.e., randomly selecting examples) at worst. However, in most cases, AceCoder is better than the existing approaches. We leave this limitation to future work.

Data Leakage. Theoretically, all open-source code projects may be included in the training data for LLMs. Consequently, there is a risk of data leakage where several samples in experimental benchmarks appear in the training data. Because most LLMs' training data is unavailable, we can not determine which samples are leaked. To address this threat, we refactor the experimental benchmarks and evaluate our AceCoder on new benchmarks. The details of refactoring and evaluation are described as follows.

❶ Refactoring Benchmarks. Benchmarks provide natural languages and function signatures as inputs, and test cases for evaluation. First, we hire five annotators to rewrite all requirements without changing their semantics. These annotators have 3–5 years of development experience. Then, we ask annotators to refactor function and argument names in signatures. Finally, we design heuristic rules to update the functions under test in test cases automatically. In this way, we refactor all samples in the MBPP and obtain a new benchmark—Refactor-MBPP. Figure 7 shows an original sample in MBPP and its corresponding refactored version in Refactor-MBPP. Since refactoring is time-consuming and laborious, we leave other benchmarks to future work.

❷ Evaluating AceCoder on Refactor-MBPP. We select GPT-3.5 as the base model and evaluate different approaches on Refactor-MBPP. The results are shown in Table 8. We can see that our

Table 8. The Results of AceCoder and Prompting Baselines on MBPP and Refactor-MBPP Datasets

Base Model	Prompting Technique	MBPP			Refactor-MBPP		
		Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
GPT-3.5	Zero-shot prompting	50.47	58.20	61.40	46.32	55.12	59.13
	Few-shot prompting	52.20	59.24	61.92	47.92	57.67	59.46
	CoT prompting	52.80	61.00	63.00	48.77	60.25	62.34
	AceCoder	**57.82**	**64.74**	**66.83**	**55.73**	**63.42**	**65.14**
Relative Improvement		**8.3%**	**6.9%**	**7.6%**	**14.3%**	**5.3%**	**4.5%**

The bold and red indicate important experimental results.

AceCoder substantially outperforms all baselines in Refactor-MBPP. The results determine that data leakage has a slight impact on our experiments.

The Generalizability of Experimental Results. To mitigate this threat, we carefully select the experimental datasets, metrics, and baselines. Following previous studies [6, 11], we pick three representative code generation benchmarks. They are collected from real-world software projects and cover three popular programming languages (i.e., Python, Java, and JavaScript). We select a widely used metric for evaluation metrics—Pass@k ($k = 1, 3, 5$). Pass@k is an execution-based metric that utilizes test cases to check the correctness of programs. We select existing prompting techniques and retrieval-based models as comparison baselines. We pick three representative LLMs as base models [12, 13, 30, 51], which scale from 6B to 13B. We apply AceCoder and baselines to base models and evaluate their performance on three datasets using Pass@k. We run each approach three times to ensure fairness and report the average results.

7 Related Work

LLMs for code generation are large-scale neural networks pre-trained on a large corpus of natural language and programming language. With the development of LLM research, current Code LLMs can be divided into two categories: standard language models and instruction-tuned models.

Standard Language Models are pre-trained on the raw corpus with the next-token prediction. They can continually complete the given context, which makes them useful in tasks like code completion and code generation. With the success of GPT series [9, 36, 37] in Natural Language Processing, OpenAI adapts similar ideas into the domain of source code and fine-tunes GPT models on code to produce closed-source Codex [12]. There are multiple open-source attempts to replicate its success, e.g., CodeParrot [1], CodeGen [30], CodeGeeX [51], InCoder [13], StarCoder [24] and CodeT5+ [44].

Instruction-Tuned Models are models fine-tuned using instruction tuning [47]. Instruction tuning helps models follow users' instructions. OpenAI's ChatGPT [31] is trained by RLHF [33], making it capable of both natural language tasks and programming tasks. Due to its enormous influence and closed-source, many researchers try to create open-source ChatGPT alternatives using instruction tuning and its variants. Alpaca [40] is LLaMA [41] fine-tuned using self-instruct [43] and ChatGPT feedback. Code Alpaca [10] is LLaMA fine-tuned using self-instruct and ChatGPT feedback with more programming-focused instructions. WizardCoder [28] is StarCoder [24] fine-tuned using Evol-Instruct [49] and ChatGPT feedback with Code Alpaca's dataset as seed dataset. InstructCodeT5+ [44] is CodeT5+ [44] fine-tuned on Code Alpaca's dataset.

Prompting Techniques. LLMs are too large to fine-tune, so researchers need to find a new way to adapt the LLMs to the downstream tasks. *Prompting techniques* are a popular approach to leverage LLMs to generate code by inputting a special prompt.

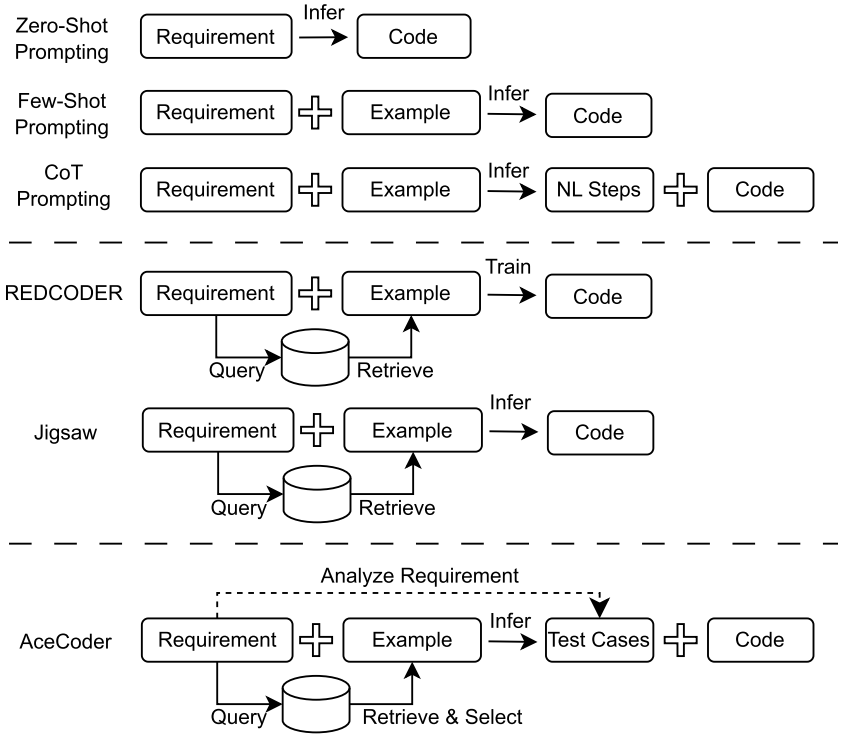

Fig. 8. The comparison between AceCoder and existing prompting techniques and retrieval-augmented approaches.

Early, researchers proposed zero-shot prompting and few-shot prompting. Zero-shot prompting concatenates a task instruction (e.g., `please generate a program based on the requirement`) and a requirement together to make the prompt. Based on the zero-shot prompting, few-shot prompting further adds several ⟨requirement, code⟩ pairs to the prompts so that LLMs can learn code generation from given examples. CoT prompting [48] is a recently proposed prompting technique. CoT asks LLMs first to generate CoTs (i.e., intermediate natural language reasoning steps) and then output the final code. It allows LLMs to design a solving process that leads to the code. CoT has achieved the SOTA results in natural language generation and sparked lots of follow-up research, such as self-consistency prompting [42], least-to-most prompting [52]. However, these prompting techniques are designed for natural language generation and bring slight improvements in code generation.

Differences between AceCoder and existing prompting techniques. As shown in Figure 8, AceCoder proposes two novel designs:

—E*xample Retrieval*. Previous approaches typically randomly select examples from the training data, which probably are irrelevant to current requirements. AceCoder retrieves similar programs as examples. Figure 2(d) shows some similar programs used by AceCoder. Similar programs contain many relevant elements (e.g., APIs—re.search, algorithms), which can be reused and beneficial to code generation.
—*Guided Code Generation*. Zero-shot and few-shot prompting ask LLMs to output the code directly. CoT prompting asks LLMs to generate intermediate natural language reasoning steps and output the code. In comparison, Guided code generation asks LLMs to analyze

the requirement (i.e., generating an intermediate preliminary) and then generate the code. A preliminary is a specific software artifact (e.g., test cases, APIs) to clarify the requirement. Figure 1(d) shows the preliminary (i.e., test cases) generated by AceCoder. The test cases cover multiple requirement scenarios (e.g., boundary inputs) and benefit the following code implementation.

Differences between AceCoder and existing retrieval-augmented approaches. REDCODER [35] and Jigsaw [19] both are retrieval-augmented code generation baselines. Compared to them, AceCoder contains two novel mechanisms:

— *An Example Selector.* REDCODER and Jigsaw simply take top-K retrieved programs as inputs. As stated in Section 2, top-K retrieved programs may contain redundant contents and ignore more informative programs. Thus, we propose a selector to filter out redundant programs and maximize the information of retrieved results.
— *Guided Code Generation.* REDCODER and Jigsaw ask models to output the final code directly. In comparison, our proposed code generation teaches models to generate an intermediate preliminary and then generate the code. As discussed above, guided code generation benefits the requirement understanding and thus improves the accuracy of code generation.

8 Conclusion and Future Work

We propose a new prompting technique named AceCoder to improve the performance of LLMs on code generation. AceCoder designs two novel techniques (i.e., guided code generation and example retrieval) to help LLMs understand requirements and implement programs. Guided code generation asks LLMs to output an intermediate preliminary (e.g., test cases) before generating programs. The preliminary helps LLMs understand requirements and guides the next code generation. Example retrieval selects similar programs as examples, which provide many reusable elements for program implementation. We apply AceCoder to three LLMs and conduct experiments on three benchmarks. Results show that AceCoder significantly outperforms the SOTA baselines.

Based on AceCoder, researchers can explore the following directions in future work:

Semantic Example Retrieval. Our example retrieval uses BM24 as the retrieval metric. Though BM25 is widely used in code search, it focuses on the requirement text and ignores the requirement semantics. Two requirements that are very similar intext may be very different in semantics, such as `Upload an image to a server` and `Download an image from a server`. This limitation causes that retrieved programs contain little information for code generation and bring few improvements. Therefore, it is necessary to explore more advanced retrieval techniques that consider both text and semantics.

Non-Standalone Code Generation. Our experimental benchmarks only comprise standalone programs. Recent studies [20, 21] release benchmarks containing non-standalone programs. The performance of AceCoder on non-standalone code is unclear. A straightforward approach to addressing this problem is to extend our example retrieval. Besides similar programs, we also retrieve programs that may be invoked in target functions. Then, both types of programs are inserted into prompts and inputted to LLMs. We will explore this direction in future work.

Long Code Generation. In this article, we focus on function-level code generation and omit the longer code, e.g., a class. Theoretically, AceCoder can be used to generate the longer code. A challenge is how to design the corresponding requirements. For example, a class typically consists of multiple functions and fields. How to clearly express the elements in the target class and their roles is a future direction.

In the future, we will explore how to improve the usability of LLMs in code generation. For example, how to teach LLMs to use unseen frameworks without re-training.

References

[1] 2022. CodeParrot. Retrieved from https://huggingface.co/codeparrot/codeparrot

[2] 2022. GitHub. Retrieved from https://github.com/

[3] 2022. Lucene. Retrieved from https://lucene.apache.org/

[4] 2022. tree-sitter. Retrieved from https://tree-sitter.github.io/tree-sitter/

[5] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-Training for Program Understanding and Generation. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT '21)*. Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tür, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.). Association for Computational Linguistics, 2655–2668. DOI: https://doi.org/10.18653/v1/2021.naacl-main.211

[6] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, and Ramesh Nallapati. 2023. Multi-lingual Evaluation of Code Generation Models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*. OpenReview.net. Retrieved from https://openreview.net/pdf?id=Bo7eeXm6An8

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732. Retrieved from https://arxiv.org/abs/2108.07732

[8] Kent L. Beck. 2003. *Test-Driven Development - By Example*. Addison-Wesley.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 33. 1877–1901.

[10] Sahil Chaudhary. 2023. Code Alpaca: An Instruction-Following LLaMA Model for Code Generation. Retrieved from https://github.com/sahil280114/codealpaca

[11] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*. OpenReview.net. Retrieved from https://openreview.net/pdf?id=ktrw68Cmu9c

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

[13] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In Proceedings of the *11th International Conference on Learning Representations (ICLR '23)*. OpenReview.net. Retrieved from https://openreview.net/pdf?id=hQwb-lbM6EL

[14] Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. AixBench: A Code Generation Benchmark Dataset. arXiv:2206.13179. Retrieved from https://arxiv.org/pdf/2206.13179

[15] Tatsunori B. Hashimoto, Kelvin Guu, Yonatan Oren, and Percy Liang. 2018. A Retrieve-and-Edit Framework for Predicting Structured Outputs. In *Proceedings of the Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018 (NeurIPS '18)*. Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolo Cesa-Bianchi, and Roman Garnett (Eds.). 10073–10083. Retrieved from https://proceedings.neurips.cc/paper/2018/hash/cd17d3ce3b64f227987cd92cd701cc58-Abstract.html

[16] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence with APPS.

In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021*. Joaquin Vanschoren and Sai-Kit Yeung (Eds.). Retrieved from https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html

[17] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *Proceedings of the 8th International Conference on Learning Representations (ICLR '20)*. OpenReview.net. Retrieved from https://openreview.net/forum?id=rygGQyrFvH

[18] Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andrés Codas, Mark Encarnación, Shuvendu K. Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-Aware Neural Code Rankers. In *Proceedings of the Advances in Neural Information Processing Systems* (*NeurIPS*). 13419–13432. Retrieved from http://papers.nips.cc/paper_files/paper/2022/hash/5762c579d09811b7639be2389b3d07be-Abstract-Conference.html

[19] Naman Jain, Skanda Vaidyanath, Arun Shankar Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram K. Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models Meet Program Synthesis. In *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering (ICSE '22)*. ACM, New York, NY, 1219–1231. DOI: https://doi.org/10.1145/3510003.3510203

[20] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024a. EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories. arXiv:2404.00599. Retrieved from https://arxiv.org/pdf/2404.00599

[21] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. 2024b. DevEval: A Manually-Annotated Code Generation Benchmark Aligned with Real-World Code Repositories. arXiv:2405.19856. Retrieved from https://arxiv.org/pdf/2405.19856

[22] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. EditSum: A Retrieve-and-Edit Framework for Source Code Summarization. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE '21)*. IEEE, 155–166. DOI: https://doi.org/10.1109/ASE51524.2021.9678724

[23] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023b. SkCoder: A Sketch-Based Approach for Automatic Code Generation. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE '23)*. IEEE, 2124–2135. DOI: https://doi.org/10.1109/ICSE48619.2023.00179

[24] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023a. StarCoder: May the Source be With You! arXiv:2305.06161. Retrieved from https://arxiv.org/pdf/2305.06161

[25] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. DOI: https://doi.org/10.1126/science.abq1158. Retrieved from https://www.science.org/doi/pdf/10.1126/science.abq1158

[26] C. Y. Lin. 2004. Rouge: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*. 74–81.

[27] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (ACL '22)*. Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 6227–6240. DOI: https://doi.org/10.18653/v1/2022.acl-long.431

[28] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv:2306.08568. Retrieved from https://arxiv.org/pdf/2306.08568

[29] Audris Mockus. 2007. Large-Scale Code Reuse in Open Source Software. In *Proceedings of the 1st International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS '07: ICSE Workshops 2007)*. IEEE, 7–7.

[30] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In

Proceedings of the 11th International Conference on Learning Representations (ICLR '23). OpenReview.net. Retrieved from https://openreview.net/pdf?id=iaYcJKpY2B_

[31] OpenAI. 2022. ChatGPT. Retrieved from https://openai.com/blog/chatgpt

[32] OpenAI. 2023. gpt-3.5-turbo. Retrieved from https://platform.openai.com/docs/models/gpt-3-5

[33] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training Language Models to Follow Instructions with Human Feedback. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 35. 27730–27744.

[34] 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.

[35] Md. Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. In *Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2021*. Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 2719–2734. DOI: https://doi.org/10.18653/v1/2021.findings-emnlp.232

[36] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving Language Understanding by Generative Pre-Training. Retrieved from https://www.mikecaptain.com/resources/pdf/GPT-1.pdf

[37] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models Are Unsupervised Multitask Learners. *OpenAI blog* 1, 8 (2019), 9.

[38] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP '19)*. Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 3980–3990. DOI: https://doi.org/10.18653/v1/D19-1410

[39] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and TrendsÂ® in Information Retrieval* 3, 4 (2009), 333–389.

[40] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-Following LLaMA Model. Retrieved from https://github.com/tatsu-lab/stanford_alpaca

[41] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Roziére, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971. Retrieved from https://arxiv.org/pdf/2302.13971

[42] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023c. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*. OpenReview.net. Retrieved from https://openreview.net/pdf?id=1PL1NIMMrw

[43] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023a. Self-Instruct: Aligning Language Models with Self-Generated Instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics. 13484–13508. Retrieved from https://aclanthology.org/2023.acl-long.754

[44] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023b. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP '23)*. Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 1069–1088. DOI: https://doi.org/10.18653/V1/2023.EMNLP-MAIN.68

[45] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. 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*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. DOI: https://doi.org/10.18653/v1/2021.emnlp-main.685

[46] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and Refine: Exemplar-Based Neural Comment Generation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. IEEE, 349–360.

[47] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022a. Finetuned Language Models are Zero-Shot Learners. In *Proceedings of the International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=gEZrGCozdqR

[48] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022b. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Proceedings of the Advances in Neural Information Processing Systems*. 24824–24837.

[49] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. WizardLM: Empowering Large Language Models to Follow Complex Instructions. arXiv:2304.12244. Retrieved from https://arxiv.org/pdf/2304.12244

[50] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-Shot Performance of Language Models. In *Proceedings of the International Conference on Machine Learning*. PMLR, 12697–12706.

[51] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23)*. Ambuj K. Singh, Yizhou Sun, Leman Akoglu, Dimitrios Gunopulos, Xifeng Yan, Ravi Kumar, Fatma Ozcan, and Jieping Ye (Eds.). ACM, New York, NY, 5673–5684. DOI: https://doi.org/10.1145/3580305.3599790

[52] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. 2023. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*. OpenReview.net. Retrieved from https://openreview.net/pdf?id=WZH7099tgfM

Received 22 October 2023; revised 11 May 2024; accepted 17 June 2024