

ACECODER: Utilizing Existing Code to Enhance Code Generation

Jia Li 
Peking University
Beijing, China
lijia@stu.pku.edu.cn

Yunfei Zhao
Peking University
Beijing, China
zhaoyunfei@pku.edu.cn

Yongmin Li
Peking University
Beijing, China
liyongmin@pku.edu.cn

Ge Li
Peking University
Beijing, China
lige@pku.edu.cn


Zhi Jin
Peking University
Beijing, China
zhijin@pku.edu.cn

ABSTRACT

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

In this paper, 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. (1) Guided code generation asks LLMs first to analyze requirements and output an intermediate preliminary (*e.g.*, test cases). The preliminary is used to clarify requirements and tell LLMs “*what to write*”. (2) 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 three LLMs (*e.g.*, Codex) 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. **(1) In terms of Pass@1, ACECODER outperforms the state-of-the-art baseline by up to 56.4% in MBPP, 70.7% in MBJP, and 88.4% in MBJSP.** (2) ACECODER is effective in LLMs with different sizes (*i.e.*, 6B to 13B) and different languages (*i.e.*, Python, Java, and JavaScript). (3) Human evaluation shows human developers prefer programs from ACECODER.

ACM Reference Format:

Jia Li , Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2023. ACECODER: Utilizing Existing Code to Enhance Code Generation. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Code generation aims to automatically generate the source code based on a natural language requirement [22, 23, 25]. Recently,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA

© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Large Language Models (LLMs) have achieved state-of-the-art (SOTA) results on code generation [1, 14, 15, 27, 32]. 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 LLMs strongly relies on the prompt surface [51]. Thus, how to design prompts (*i.e.*, prompting techniques) is a popular topic. Existing prompting techniques (*e.g.*, few-shot prompting [11] and chain-of-thought prompting [49]) 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 on a real-world benchmark - HumanEval [14]. Thus, it is necessary to explore more advanced prompting techniques for code generation.

In this paper, 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 - MBPP [9] 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 [10], 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 requirement, how to implement the source code using a programming language is challenging. It requires LLMs to master related 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 new requirement, developers often search for similar programs. They learn programming skills (e.g., APIs) or directly reuse relevant content from similar programs [31].

Specifically, we use a *retriever* to search for programs with similar requirements (e.g., Top-20). Considering the maximum input length of LLMs is limited (e.g., 1024 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 find similar programs as examples, i.e., <requirement, code> pairs.
- **Prompt construction.** It uses an *analyzer* to convert retrieved examples into <requirement, preliminary, code> triples. Then, it concatenates triple examples with the input requirement together to construct a prompt.
- **Code generation.** It feeds the prompt into LLMs. By learning from examples, LLMs first output an intermediate preliminary and then generate code for the input requirement.

We apply ACECODER to three representative LLMs, i.e., CodeGeeX [1], CodeGen [32], and InCoder [15]. We conduct extensive experiments on three popular code generation benchmarks, i.e., MBPP (Python) [9], MBJP (Java) [8], and MBJSP (JavaScript) [8]. We employ Pass@ k ($k = 1, 3, 5$) to measure the performance of different approaches. We obtain some findings from experimental results. **(1) 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. **(2) 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. **(3) 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%. **(4) 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. **(5) We explore the contributions of different**

```

(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") -> "heolo word"
-----
(c) Few-shot Prompting:
def remove_0cc(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_0cc(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

```

Figure 1: A motivating example of guided code generation.

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., chain-of-thought prompting, few-shot prompting).

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 [9] 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 requirement is crucial to write correct programs.

```

(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

```

Figure 2: A motivating example of example retrieval.

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

Code Implementation → Example Retrieval. After understanding the input requirement, how to implement 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 [9] and its unit test for evaluation, respectively. We select Codex as the base model. Figure 2 (c) shows a program generated by few-shot prompting. The program contains a wrong condition statement highlighted in yellow. This is because 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 search for similar programs and learn programming skills from them. Figure 2 (d) shows some retrieved programs based on the similarity of requirements. The retrieval metric is the BM25 score. We sort the results in descending order of BM25 score. We can see that the retrieved programs contain lots of relevant content (e.g., `re.search`), which benefits code implementation. Thus, we design a retriever to search

for 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., 1024 tokens), the number of examples in a prompt is limited. Thus, we need to further select a set of programs from retrieved results as examples. A straightforward idea is to pick top similar programs as examples. However, as the programs are retrieved independently, we find that retrieved results may contain redundant programs. In Figure 2 (d), Program-1, Program-2, and Program-3 are redundant, as all of them provide an `API re.search`, which 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&2) and miss more informative Program-4.

Thus, we design a selector for selecting examples, which can 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 METHODOLOGY

In this section, we propose a novel prompting technique for code generation, named ACECODER. In the subsections, we first 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 large language models (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 $\langle \text{requirement}, \text{code} \rangle$ pairs $(\{x_i, y_i\}_{i=1}^k)$ from a retrieval corpus as examples.
- **Prompt Construction.** It employs an *analyzer* to convert examples into $\langle \text{requirement}, \text{preliminary}, \text{code} \rangle$ 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 first 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 i -th example, respectively.

3.2 Example Retrieval

As shown in Figure 3, the first step has two goals: (i) retrieve similar programs and (ii) 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 [17, 25]. There, we take the input requirement

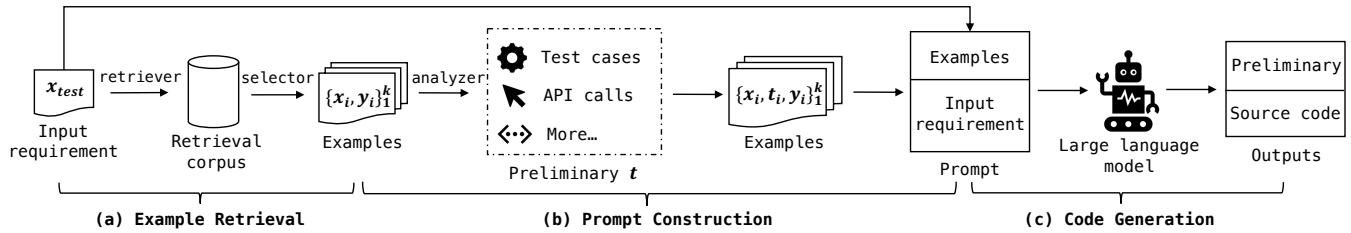


Figure 3: An overview of ACECODER. Given a requirement, it selects examples from similar programs and constructs a prompt. LLMs first output an intermediate preliminary and then generate the source code. x , y , and t denote requirements, programs, and intermediate preliminaries, respectively.

```

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)

```

Figure 4: A requirement and its similar programs.

as a query to search for similar requirements in a retrieval corpus. Then, we extract the corresponding programs as similar programs.

Specifically, we leverage an open-source search engine named Lucene [5] to build our retriever and use the training data as a retrieval corpus. We employ the BM25 score [40] as the retrieval metric, which is widely used in previous studies [24, 47]. 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 paper, 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-RQ5, 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., 1024 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.

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 , $k \leq m$, decay factor λ .

Outputs:

Selected examples T , $\{(x_i, y_i)\}_{i=1}^k$.

```

1:  $T \leftarrow$  Empty Ordered List
2:  $S \leftarrow$  Extract_ngrams_with_count( $x_{test}$ )
3: for  $i$  in  $\{1, \dots, m\}$  do
4:    $Q[i] \leftarrow$  Extract_ngrams_with_count( $x_i$ )
5: end for
6: while  $len(T) < k$  do
7:   for  $i$  in  $\{1, \dots, m\}$  do
8:      $Score[i] \leftarrow$  Ngram_overlap_score( $S, Q[i]$ )
9:   end for
10:   $j \leftarrow$  argmax( $Score$ )
11:   $T.append((x_j, y_j))$ 
12:   $matched\_ngrams \leftarrow S \cap Q[j]$ 
13:   $Q[j] \leftarrow \emptyset$ 
14:  for  $ngram \in matched\_ngrams$  do
15:     $S[ngram] \times \lambda$ 
16:  end for
17: end while
18: return  $T$ 

```

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 sequences of a specific pattern. Program-4 contains a relevant regex expression. However, as Program-4 has fewer overlapping n -grams with the input requirement, it has a relatively small 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 paper, 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 paper, n is set to 4 by default. Then, we calculate a recall-based ROUGE- n score

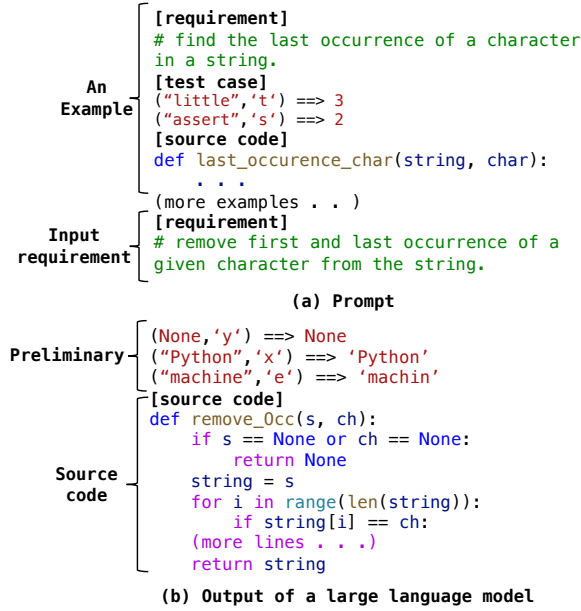


Figure 5: Examples of our prompt and an LLM’s output.

between the input requirement and each similar requirement using the following equations (lines 7-9).

$$R_n = \frac{\sum_{n_gram \in S \cap Q} S(n_gram)}{\sum_{n_gram \in S} S(n_gram)} \quad (1)$$

$$Score = \exp\left(\frac{1}{n} \sum_n \log(R_n)\right) \quad (2)$$

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 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 first 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 [10], this paper considers test cases as the preliminary by default. We also explore other choices (*e.g.*, APIs, method signature) in our experiments (Section 5-RQ5). 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 [9] (474 samples), APPS [18] (5,000 samples), and CodeContest [27] (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 [4]) 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 [1, 14, 15, 32], 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 the code based on the preliminary and input requirement.

Figure 5 (b) shows an output of an LLM - CodeGeeX [1]. 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.

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 large language models (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 [21, 36] 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.

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

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 manually review the code generated by ACECODER and baselines. 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** [9] contains 974 real-world programming problems that are constructed by crowd-sourcing. Each problem contains a natural language requirement, a single Python function, and three test cases.
- **MBJP** [8] and **MBJSP** [8] both contain 966 crowd-sourced programming problems in Java and JavaScript, respectively. Each problem consists of a natural language requirement, an individual function, and 3 test cases.

4.2.2 Metrics. Following previous code generation studies [1, 14, 15, 32], 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 paper, k is set to 1, 3, and 5.

We notice that previous studies [17, 46] also use some match-based metrics (*e.g.*, BLEU [35]). These metrics are initially designed for natural language generation and are poor in measuring the functionality of programs [14]. Thus, we omit them in experiments.

4.3 Comparison Baselines

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

- **Zero-shot prompting** [14, 32] directly feeds the input requirement into LLMs. Then, it extracts the code from LLMs' outputs.
- **Few-shot prompting** [14] randomly selects several <requirement, code> pairs as examples and constructs a prompt, which is fed into an LLM. Then, it extracts the code from LLMs' outputs.
- **Chain-of-Thought (CoT) prompting** [49] is a variant of few-shot prompting. CoT prompting asks LLMs first to generate a series of intermediate natural language reasoning steps and then output the code.

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** [36] retrieves similar programs and fine-tunes a pre-trained model - PLBART [7] to generate code based on the requirement and similar programs.
- **Jigsaw** [21] searches for similar programs from API documentation and insert them into the prompts.

4.4 Base Large Language Models

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

- **CodeGeeX** [1] 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, JavaScript). We download the model weight from the official website [2] and run CodeGeeX according to official instructions.
- **CodeGen** [32] 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** [15] 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.

The reason why we do not choose the GPT series of models (*e.g.*, ChatGPT [33]) as the base models is that they are closed source. Although we can access GPT models through the OpenAI's APIs, these models are likely to be updated dynamically, affecting the fairness and reproducibility of experiments. Thus, we leave them to future work.

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. For ensuring 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 [8, 9]. Thus, the analyzer utilizes pre-defined rules to extract test cases and transform retrieved programs into <requirement, test cases, code> triples.

Table 2: The results of ACECODER and prompting baselines on three datasets. The values in parentheses are the relative improvements compared to the SOTA baseline - few-shot prompting.

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
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 (↑ 31.1%)	36.43 (↑ 19%)	41.13 (↑ 14.2%)	28.38 (↑ 70.7%)	36.79 (↑ 40.6%)	41.54 (↑ 20.5%)	21.03 (↑ 88.4%)	31.44 (↑ 58.2%)	36.04 (↑ 41%)
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 (↑ 56.4%)	34.58 (↑ 44.1%)	40.16 (↑ 33%)	22.45 (↑ 23%)	34.27 (↑ 14.2%)	40.96 (↑ 18.1%)	16.45 (↑ 65.5%)	27.31 (↑ 37.4%)	32.16 (↑ 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 (↑ 57.5%)	31.44 (↑ 37.9%)	34.10 (↑ 20.92%)	16.37 (↑ 49.5%)	29.89 (↑ 27%)	34.74 (↑ 32.7%)	15.97 (↑ 25%)	27.13 (↑ 20.5%)	30.65 (↑ 10.3%)

Code Generation. Following previous studies [14, 15, 32], we use nucleus sampling [19] 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).

Results. The results on three benchmarks are shown in Table 2. The values in parentheses are relative improvements compared to the SOTA baseline - few-shot prompting.

Analyses. (1) 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 56.4% in MBPP, 70.7% in MBJP, and 88.4% in MBJSP. Pass@1 is a very strict metric and it 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”. (2) ACECODER is effective in LLMs with different sizes and different programming languages. 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. Besides, ACECODER is 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 [36] and Jigsaw [21]. 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 values in parentheses are relative improvements compared to the SOTA baseline - Jigsaw.

Analyses. 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, 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 settings of human evaluation in previous studies [16, 25]. We have carefully checked the evaluation settings and think our

Table 3: The comparison of retrieval-based baselines and ACECODER. The values in parentheses are relative improvements compared to the SOTA baseline - Jigsaw.

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 (↑ 13.1%)	36.43 (↑ 7.2%)	41.13 (↑ 8.9%)	28.38 (↑ 23.44%)	36.79 (↑ 10.61%)	41.54 (↑ 12.42%)	21.03 (↑ 15.8%)	31.44 (↑ 9.2%)	36.04 (↑ 5.8%)

Table 4: The results of ablation study. The values in parentheses are relative improvements compared to few-shot prompting.

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 (↑ 13.1%)	38.20 (↑ 6.1%)	23.35 (↑ 40.4%)	33.67 (↑ 28.7%)	37.22 (↑ 7.9%)	18.66 (↑ 67.2%)	29.18 (↑ 46.8%)	34.89 (↑ 36.5%)
✓	✓	✗	24.89 (↑ 22%)	35.02 (↑ 14.4%)	39.14 (↑ 8.7%)	25.03 (↑ 50.5%)	34.47 (↑ 31.7%)	39.24 (↑ 13.8%)	19.73 (↑ 76.8%)	30.16 (↑ 51.7%)	35.34 (↑ 38.3%)
✓	✓	✓	26.74 (↑ 31.1%)	36.43 (↑ 19%)	41.13 (↑ 14.2%)	28.38 (↑ 70.7%)	36.79 (↑ 40.6%)	41.54 (↑ 20.5%)	21.03 (↑ 88.4%)	31.44 (↑ 58.2%)	36.04 (↑ 41%)

Table 5: The results of human evaluation. The values in parentheses are the relative improvements compared to the SOTA baseline - few-shot prompting.

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%)

settings are reliable. 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). For ACECODER and baselines, we select a fixed base model (*i.e.*, CodeGen-2B) and collect 200 generated programs per approach. Finally, we obtain 1,000 programs for evaluation. We invite 10 developers with 3-5 years of development experience to evaluate the generated programs in the form of a questionnaire. The 1,000 code snippets are divided into 5 groups, with each questionnaire containing one group. The programs are randomly shuffled and anonymously reviewed by evaluators. Each group is evaluated by two evaluators, and the

final score is the average of two evaluators' scores. Evaluators are allowed to search the Internet for unfamiliar concepts.

Results. The results of the human evaluation are shown in Table 5. The values in parentheses are the relative improvements compared to the SOTA baseline - few-shot prompting.

Analyses. 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 5. ✓ 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. 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 improved 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 [39] 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 [35]. 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 [28] (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 [6] 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 6. “w” is the abbreviation of with. (1) A dense retriever is comparable to our retriever, but has a lower efficiency. In Table 6, compared to

ACECODER, ACECODER with dense retriever has a slight drop in performances. It indicates that code generation prefers lexically similar programs, which contain lots of reusable content. Similar findings can be found in code completion work [29]. Besides, the dense retriever has a higher complexity and is hard to be applied to a large-scale retrieval corpus. (2) 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 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. (3) Test cases are more suitable to the preliminary than APIs and method signatures. We carefully inspect some cases. First, many requirements in benchmarks 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 think about the input-output format, which benefits code generation. But 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 ACECODER vs. CoT prompting

Our guided code generation is similar to Chain-of-Thought (CoT) prompting. Both approaches ask LLMs to first 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 paper of CoT prompting [49]. 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

Table 6: The performance of ACECODER with different designs. “w/” is the abbreviation of with.

Approach	MBPP			MBJP			MBSJP		
	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

different from CoT prompting and is more promising than CoT prompting in code generation.

6.2 ACECODER vs. Rank Techniques

Some recent studies [13, 20] 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 paper, 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.

6.3 Threats to Validity

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

The generalizability of experimental results. To mitigate this threat, we carefully select the experimental datasets, metrics, and baselines. Following previous studies [8, 13], 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). For evaluation metrics, we select a widely used metric - 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 [1, 14, 15, 32], which scale from 6B to 13B. We apply ACECODER and baselines to base models and evaluate their performance on three datasets using Pass@ k . To ensure fairness, we run each approach three times and report the average results.

The impact of retrieved programs. The retrieved programs are important elements in ACECODER. Intuitively, when retrieved programs are less relevant to input requirements, the performance of our approach may suffer. To address this threat, we have two thoughts. (1) A large-scale study on 13.2 million real code files found the proportion of reused code is up to 80% [31]. Thus, we believe that it is quite possible to retrieve similar programs in real development scenarios. (2) Even if retrieved programs are less relevant to input requirements, ACECODER degrades to few-shot prompting at worst. In most cases, ACECODER is superior to few-shot prompting.

7 RELATED WORK

Large Language Models (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 [11, 37, 38] in NLP, OpenAI adapts similar idea into the domain of source code, and fine-tunes GPT models on code to produce closed-source Codex [14]. There are multiple open-source attempts to replicate its success, *e.g.*, CodeParrot [3], CodeGen [32], CodeGeeX [1], InCoder [15], StarCoder [26] and CodeT5+ [45].

Instruction-tuned models are models fine-tuned using instruction tuning [48]. Instruction tuning helps models to follow users’ instructions. OpenAI’s ChatGPT [33] is trained by Reinforcement Learning with Human Feedback (RLHF) [34], making it capable of both natural language tasks and programming tasks. Due to its enormous influence and closed-sourceness, many researchers try to create open-source ChatGPT alternatives using instruction tuning and its variants. Alpaca [41] is LLaMA [42] fine-tuned using self-instruct [44] and ChatGPT feedback. Code Alpaca [12] is LLaMA fine-tuned using self-instruct and ChatGPT feedback with more programming-focused instructions. WizardCoder [30] is StarCoder [26] fine-tuned using Evol-Instruct [50] and ChatGPT feedback with Code Alpaca’s dataset as seed dataset. InstructCodeT5+ [45] is CodeT5+ [45] 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 on the downstream tasks. *Prompting techniques* are a popular approach to leverage LLMs to generate code by inputting a special prompt.

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 \langle requirement, code \rangle pairs to the prompts, so that LLMs can learn code generation from given examples. Chain-of-Thought (CoT) prompting [49] 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 first 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 [43], least-to-most prompting [52]. But these prompting techniques are designed for natural language generation and bring slight improvements in 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.

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. CodeGeeX. <https://models.aminer.cn/codegeex/zh-CN>.
- [2] 2022. CodeGeeX. <https://models.aminer.cn/codegeex/blog/index.html>.
- [3] 2022. CodeParrot. <https://huggingface.co/codeparrot/codeparrot>.
- [4] 2022. GitHub. <https://github.com/>.
- [5] 2022. Lucene. <https://lucene.apache.org/>.
- [6] 2022. tree-sitter. <https://tree-sitter.github.io/tree-sitter/>.
- [7] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668. <https://doi.org/10.18653/v1/2021.naacl-main.211>
- [8] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual Evaluation of Code Generation Models. *arXiv preprint arXiv:2210.14868* (2022).
- [9] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
- [10] Kent Beck. 2003. *Test-driven development: by example*. Addison-Wesley Professional.
- [11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [12] Sahil Chaudhary. 2023. Code Alpaca: An Instruction-following LLaMA model for code generation. <https://github.com/sahil280114/codealpaca>.
- [13] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).
- [14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [15] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).
- [16] 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 preprint arXiv:2206.13179* (2022).
- [17] Tatsunori B Hashimoto, Kelvin Guu, Yonatan Oren, and Percy S Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. *Advances in Neural Information Processing Systems* 31 (2018).
- [18] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.
- [19] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. <https://openreview.net/forum?id=rygGQyrFvH>
- [20] Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andrés Codaş, Mark Encarnación, Shuvendu K. Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-Aware Neural Code Rankers. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2022/hash/5762c579d09811b7639be2389b3d07be-Abstract-Conference.html
- [21] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.
- [22] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Enabling Programming Thinking in Large Language Models Toward Code Generation. *CoRR abs/2305.06599* (2023). <https://doi.org/10.48550/arXiv.2305.06599>
- [23] Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. 2023. CodeEditor: Learning to Edit Source Code with Pre-Trained Models. *ACM Trans. Softw. Eng. Methodol.* (may 2023). <https://doi.org/10.1145/3597207> Just Accepted.
- [24] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. Editsum: A retrieve-and-edit framework for source code summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 155–166.
- [25] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. SkCoder: A Sketch-based Approach for Automatic Code Generation. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2124–2135. <https://doi.org/10.1109/ICSE48619.2023.00179>
- [26] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
- [27] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustín Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.
- [28] CY LIN. 2004. Rouge: A package for automatic evaluation of summaries. In *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop, Barcelona, Spain, 74–81*.
- [29] 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 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 6227–6240. <https://doi.org/10.18653/v1/2022.acl-long.431>
- [30] 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 preprint arXiv:2306.08568* (2023).
- [31] Audris Mockus. 2007. Large-scale code reuse in open source software. In *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE, 7–7.
- [32] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
- [33] OpenAI. 2022. ChatGPT. <https://openai.com/blog/chatgpt>.
- [34] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.
- [35] 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.
- [36] Md. Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 2719–2734. <https://doi.org/10.18653/v1/2021.findings-emnlp.232>
- [37] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).
- [38] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
- [39] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International*

- Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*. Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 3980–3990. <https://doi.org/10.18653/v1/D19-1410>
- [40] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.
- [41] 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. https://github.com/tatsu-lab/stanford_alpaca.
- [42] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971* (2023).
- [43] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. <https://openreview.net/pdf?id=1PL1NIMMrw>
- [44] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. 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, Toronto, Canada, 13484–13508. <https://aclanthology.org/2023.acl-long.754>
- [45] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).
- [46] 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. <https://doi.org/10.18653/v1/2021.emnlp-main.685>
- [47] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 349–360.
- [48] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. 2022. Finetuned Language Models are Zero-Shot Learners. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=gEZrGCozdqR>
- [49] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed H Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*.
- [50] 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 preprint arXiv:2304.12244* (2023).
- [51] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*. PMLR, 12697–12706.
- [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 *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. <https://openreview.net/pdf?id=WZH7099tgfM>