RESEARCH-ARTICLE

Large Language Model-Aware In-Context Learning for Code Generation

JIA LI, Beijing University of Technology, Beijing, China

CHONGYANG TAO, Beihang University, Beijing, China

JIA LI♂, Beijing University of Technology, Beijing, China

GE LI, Beijing University of Technology, Beijing, China

ZHI JIN, Beijing University of Technology, Beijing, China

HUANGZHAO ZHANG, Beijing University of Technology, Beijing, China

View all

Published: 14 August 2025
Online AM: 28 February 2025
Accepted: 02 January 2025
Revised: 23 November 2024
Received: 21 January 2024

Large Language Model-Aware In-Context Learning for Code Generation

JIA LI, Key Lab of High Confidence Software Technology (Peking University), MoE, Beijing, China
CHONGYANG TAO, Beihang University, Beijing, China
JIA LI♂, GE LI, ZHI JIN, HUANGZHAO ZHANG, and ZHENG FANG, Key Lab of High Confidence Software Technology (Peking University), MoE, Beijing, China
FANG LIU, The State Key Laboratory of Software Development Environment (SKLSDE), SEI, School of Computer Science & Engineering, Beihang University, Beijing, China

Large Language Models (LLMs) have shown impressive In-Context Learning (ICL) ability in code generation. LLMs take a prompt context consisting of a few demonstration examples and a new requirement as input, and output new programs without any parameter update. Existing studies have found that the performance of ICL-based code generation heavily depends on the quality of demonstration examples and thus arises research on selecting demonstration examples: given a new requirement, a few demonstration examples are selected from a candidate pool, where LLMs are expected to learn the pattern hidden in these selected demonstration examples. Existing approaches are mostly based on heuristics or randomly selecting examples. However, the distribution of randomly selected examples usually varies greatly, making the performance of LLMs less robust. The heuristics retrieve examples by only considering textual similarities of requirements, leading to sub-optimal performance.

To fill this gap, we propose a *L*arge language model-*A*ware selection approach for *I*n-context-*L*earning-based code generation named LAIL. LAIL uses LLMs themselves to select examples. It requires LLMs themselves to label a candidate example as a positive example or a negative example for a requirement. Positive examples are helpful for LLMs to generate correct programs, while negative examples are trivial and should be ignored. Based on the labeled positive and negative data, LAIL trains a model-aware retriever to learn the preference of LLMs and select demonstration examples that LLMs need. During the inference, given a new requirement, LAIL uses the trained retriever to select a few examples and feed them into LLMs to generate desired programs. We apply LAIL to four widely used LLMs and evaluate it on five code generation datasets. Extensive experiments demonstrate that LAIL outperforms the State-of-the-Art (SOTA) baselines by 11.58%, 3.33%, and 5.07% on CodeGen-Multi-16B, 1.32%, 2.29%, and 1.20% on CodeLlama-34B, and achieves 4.38%, 2.85%, and 2.74%

This research is supported by the National Natural Science Foundation of China (Grant Nos. 62192731, 62192730, 62192733, 62302021, 61925203, U22B2021), the Major Program (JD) of Hubei Province (Grant No. 2023BAA024), and the Fundamental Research Funds for the Central Universities.

Authors' Contact Information: Jia Li, Key Lab of High Confidence Software Technology (Peking University), MoE, Beijing, China; e-mail: lijiaa@pku.edu.cn; Chongyang Tao, Beihang University, Beijing, China; e-mail: chongyangtao@gmail.com; Jia Li, Key Lab of High Confidence Software Technology (Peking University), MoE, Beijing, China; e-mail: lijia@stu.pku.edu.cn; Ge Li (corresponding author), Key Lab of High Confidence Software Technology (Peking University), MoE, Beijing, China; e-mail: lige@pku.edu.cn; Zhi Jin (corresponding author), Key Lab of High Confidence Software Technology (Peking University), MoE, Beijing, China; e-mail: zhijin@pku.edu.cn; Huangzhao Zhang, Key Lab of High Confidence Software Technology (Peking University), MoE, Beijing, China; e-mail: zhang_hz@pku.edu.cn; Zheng Fang, Key Lab of High Confidence Software Technology (Peking University), MoE, Beijing, China; e-mail: fangz@pku.edu.cn; Fang Liu, The State Key Laboratory of Software Development Environment (SKLSDE), SEI, School of Computer Science & Engineering, Beihang University, Beijing, China; e-mail: fangliu@buaa.edu.cn.

ACM 1557-7392/2025/8-ART190
https://doi.org/10.1145/3715908

improvements on Text-davinci-003 in terms of Pass@1 at MBJP, MBPP, and MBCPP, respectively. In addition to function-level code generation, LAIL improves the performance of LLMs on DevEval, a repository-level code generation dataset, which achieves 10.04%, 8.12%, and 4.63% improvements compared to the SOTA baselines at Pass@1, 3, and 5 on CodeLlama-7B. Human evaluation further verifies that the generated programs of LAIL are superior in correctness, code quality, and maintainability. Besides, LAIL has satisfactory transferability across different LLMs and datasets, where the retriever learned on one LLM (dataset) can be transferred to other LLMs (datasets).

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

Additional Key Words and Phrases: Code generation, in-context-learning, large language model

ACM Reference format:
Jia Li, Chongyang Tao, Jia Li♂, Ge Li, Zhi Jin, Huangzhao Zhang, Zheng Fang, and Fang Liu. 2025. Large Language Model-Aware In-Context Learning for Code Generation. *ACM Trans. Softw. Eng. Methodol.* 34, 7, Article 190 (August 2025), 33 pages.
https://doi.org/10.1145/3715908

1 Introduction

During software development, there is growing interest in automatic code generation due to the substantial labor involved in manually writing source code. Code generation aims to automatically generate programs that satisfy the natural language requirements. Recently, many **Large Language Models (LLMs)** with **In-Context Learning (ICL)** have achieved impressive performances in code generation [13, 23, 38, 40].

Different from fine-tuning, ICL is a new paradigm for transferring LLMs into code generation and does not require updating parameters. ICL concatenates a few demonstration examples (i.e., requirement–code pairs) and a new requirement together, and then feeds them into LLMs. LLMs are expected to learn the pattern hidden in these demonstration examples and generate programs for the new requirement. However, ICL is sensitive to the selected demonstration examples, resulting in the performance usually varying from almost random to near **State-of-the-Art (SOTA)** performance [23, 44]. For example, Pass@1 of the MBPP dataset ranges from 14.60% to 22.83% with different demonstration examples on CodeGen-6B [38].

Despite its importance, how to select effective examples in code generation is still an open question. Early, researchers randomly select examples from a candidate pool (e.g., the training data) [11, 33]. The selected examples are typically irrelevant to the test requirement. Besides, the semantic distributions of these examples vary a lot. As shown in Figure 1(a), we report the selected top-3 examples by random selection for the test requirement "write a function to check whether the entered number is greater than the elements of the given array" in the MBPP dataset [4]. We can find that selected examples are uncorrelated to the test requirement. Thus, the random selection approach usually results in low performance. Recently, researchers [38] designed the rule-based heuristic named AceCoder, which considers the textual similarity between a test requirement and the requirement of candidate examples with BM25 [59], then selects examples with high similarities. Although achieving improvements, AceCoder only considers lexical matching among requirements and thus is easily misled by lexicon features, leading to irrelevant examples. Figure 1(b) presents the selected top-3 examples by AceCoder. They have a large amount of textual overlap with the test requirement as labeled by underlines. However, the overlapping words are trivial for accomplishing the requirement such as "write a function to check whether." Meanwhile, the operations of non-overlapping words as labeled by the red color, such as "the lcm" and "parallel or

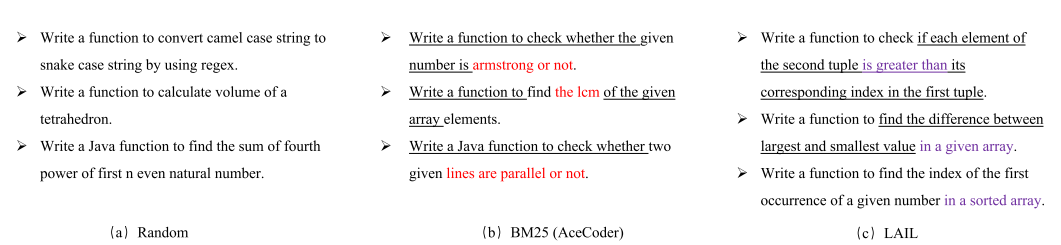

Fig. 1. Exhibition of the selected top-3 demonstration examples for the test requirement "Write a function to check whether the entered number is greater than the elements of the given array" by different example selection approaches, including random selection, AceCode, and LAIL. Concretely, subfigure (a) shows the selected top-3 examples for the test requirement with random selection approach, subfigure (b) demonstrates the selected top-3 results of AceCoder (BM25) approach for the test requirement, and subfigure (c) exhibits the retrieved top-3 examples for the test requirement by LAIL. Note that in order to save space, we only provide NL requirements of selected demonstration examples, not providing corresponding programs of these selected examples.

not," do not support implementing the test requirement. Although one may remedy it by inputting more examples in the prompt, it is highly impractical due to the limited input length of LLMs.

To address the above problem, we propose a *L*arge language model-*A*ware example selection approach for *I*n-context *L*earning-based code generation, named LAIL. Different from existing selection approaches [23, 38], LAIL considers the needs of LLMs and leverages LLMs themselves (LLM-aware) to select different external materials (i.e., different candidate examples) from a codebase. The idea is inspired by how humans solve problems: for a problem, they refer to different external materials from the same external knowledge base according to their own needs and background to figure out the problem, since they are equipped with disparate knowledge. Similar to humans, LLMs are diverse in model structure, model size, and training sources. They typically need different external knowledge to generate desired programs for a specific requirement. Given a requirement, a good selection approach should consider the needs (referred to as preferences) of LLMs.

The pipeline of LAIL consists of three stages. (A) *Estimating Examples with LLMs.* We first require LLMs themselves to label a candidate example as a positive example or a negative example for a requirement. Positive examples are helpful for LLMs to implement the requirement, while negative examples are trivial and should be ignored. Specifically, for each requirement, we calculate LLMs' generation probabilities of ground-truth programs based on a candidate example. We then design a new metric to quantify the probability feedback. The candidate examples with higher metric scores are positive examples, meanwhile examples with lower scores are negative examples. (B) *Learning a Model-Aware Retriever.* Depending on labeled positive and negative data, we train a model-aware retriever to learn the preferences of LLMs for ICL-based code generation. (C) *Generating Programs.* During the inference, given a test requirement, LAIL uses the trained retriever to select a few examples and feed them into LLMs for generating desired programs. Compared to existing approaches [23, 38], LAIL takes into account the needs of LLMs by making LLMs themselves evaluate candidate examples, thus the selected examples can effectively prompt LLMs to predict correct programs. Figure 1(c) illustrates the selected top-3 demonstration examples by LAIL; we can observe that the selected examples provide efficient information to LLMs for resolving the test requirement including operating on the array and comparing the value of two numbers. Besides, although the textual similarities between the test requirement and the selected examples are low, these examples still contain comparison operations for numbers as shown by underlines.

We evaluate LAIL on four advanced LLMs including CodeGen-Multi-16B [74], CodeLlama-7B and -34B [60], Text-davinci-003 [24], and GPT-3.5-turbo [9]. We conduct extensive experiments on five datasets (i.e., MBJP (Java) [3], MBPP (Python) [4], MBCPP (C++) [3], HumanEval (Python) [11], and DevEval (Python) [36]). We use a widely used evaluation metric Pass@k (k = 1, 3, 5) to measure the performance of different approaches. We obtain some findings from experimental results. ❶ In terms of Pass@1, LAIL significantly outperforms the SOTA baselines (e.g., AceCoder [38] and TOP-k-GraphCodeBERT [28] described in Section 4.4) by 11.58%, 3.33%, and 5.07% on CodeGen-Multi-16B, 1.32%, 2.29%, and 1.20% on CodeLlama-34B, and achieves 4.38%, 2.85%, and 2.74% improvements on Text-davinci-003 at MBJP, MBPP, and MBCPP, respectively. ❷ Besides function-level code generation, LAIL acquires 10.04%, 8.12%, and 4.63% improvements on DevEval, a repository-level code generation dataset, compared to the SOTA baseline (i.e., TOP-k-SBERT) at CodeLlama-7B in terms of Pass@1, 3, and 5. ❸ We conduct a human evaluation to measure generated programs in three aspects (e.g., code correctness, quality, and maintainability). The programs of LAIL are more in line with the preferences of humans. ❹ LAIL has satisfactory transferability across LLMs and datasets where the retriever learned on one LLM/dataset can be transferred to other LLMs/datasets. ❺ We investigate the effectiveness of LLMs' probability feedback in estimating examples and compare it to the other three estimation designs.

We summarize our contributions in this article as follows:

- We investigate example selection for ICL-based code generation and argue that a good approach should take into account what knowledge LLMs themselves need.
- We propose a model-aware selection approach for ICL-based code generation dubbed LAIL. LAIL uses LLMs themselves to estimate examples. LLMs depending on their needs label a candidate example as a positive example or a negative example for a requirement. Based on labeled data, it optimizes a model-aware retriever to learn the preference of LLMs.
- We evaluate LAIL on four advanced LLMs and conduct extensive experiments on five datasets. Qualitative and quantitative experiments reveal that LAIL significantly outperforms the SOTA baselines and generates more correct programs.
  We open-source our replication package[1], including the datasets and the source code of LAIL, to facilitate other researchers to repeat our work and evolve the code generation community.

2 Background

2.1 LLMs

In this section, we focus on LLMs that have code generation ability. LLMs are large-scale networks that aim to learn the statistical patterns in programming languages [2]. They are usually pre-trained on a large amount of unlabeled code corpus with the next token prediction objective. Formally, given a program with the token sequence $C = \{c_1, c_2, \cdots, c_n\}$, LLMs are trained to predict the next token based on some previous tokens:

$$\mathcal{L}_{NTP}(C) = -\sum_{i} \log P(c_i | c_{i-j}, \cdots, c_{i-1}; \Theta) \tag{1}$$

where j is the window length of previous tokens, and Θ means parameters of the LLM.

After being pre-trained, LLMs are adapted to a specific downstream task. At the beginning stage, LLMs are used in a fine-tuning manner, which are continually optimized on specific code generation datasets. With the size of LLMs growing rapidly such as CodeLlama [60] and CodeGen [74], fine-tuning is neither economical nor practical, in contrast, a convenient solution ICL arises.

[1] https://figshare.com/s/e31a72833191ead015ad.

2.2 ICL

ICL refers to an emerging ability of LLMs, which allows LLMs to learn tasks given only a few demonstration examples in the form of prompt context [8]. Formally, given a few requirement-code examples $T = \{x_k, y_k\}_{k=1}^{m}$, a test requirement x_t, and an LLM with frozen parameters Θ, ICL defines the generation of a program y_t as follows:

$$y_t \sim P(y_t | \underbrace{x_1, y_1, \cdots, x_m, y_m}_{context}, x_t, \Theta) \tag{2}$$

where $\sim$ represents decoding strategies such as greedy decoding and nuclear sampling [29] in code generation. The generation procedure is attractive because the parameters of LLMs do not need to be updated when executing a new task, which is efficient and practical.

According to the number of demonstration examples in the prompt context, ICL generally contains different scenarios: (a) *Few-shot learning* allows a few demonstration examples as the prompt context where the token length of all examples fits into the model's context window. LAIL is a kind of few-shot learning, which designs a model-aware selection approach to retrieve several examples from a codebase for ICL-based code generation. (b) *One-shot learning* contains only one demonstration example as the prompt context. LLMs concatenate a new requirement and the prompt context as the input, and then output programs for the new requirement. (c) *Zero-shot learning* allows no demonstration examples to be input LLMs. Only a new requirement and an instruction in natural language are fed into LLMs, where the instruction is not necessary.

3 Method: LAIL

Considering that LLMs are diverse in model structure, model size, and training sources, they typically need different external knowledge to generate desired programs for a requirement. Given a requirement, a good selection approach should consider the needs (referred to as preferences) of LLMs. In this section, we propose an LLM-Aware example selection approach for ICL-based code generation named LAIL. In ICL-based code generation, a candidate example pool is necessary where a few demonstration examples are selected from it. Similar to existing work [38], we use the training set of a dataset as the candidate example pool instead of collecting candidates from scratch. Given a test requirement, LAIL selects a few demonstration examples from the candidate pool (i.e., the training set) and then inputs these selected examples and the test requirement into LLMs to generate desired programs.

Different from existing example selection approaches [38], LAIL uses LLMs themselves (LLM-aware) to select examples based on their needs. The pipeline of LAIL contains three stages. Concretely, LAIL requires LLMs themselves to estimate a candidate example in the candidate pool by calculating the prediction probability of the ground-truth program based on the test requirement and the example, then labels the candidate example being positive or negative depending on the probability feedback. Following the above process, we traverse the entire candidate pool and acquire the final labeled data (Section 3.1). Based on the labeled positive and negative data, LAIL trains a model-aware retriever to align with the preference of LLMs (Section 3.2). During the inference, given a requirement of the test set, LAIL uses the trained retriever to select a few examples from the candidate pool (i.e., the training set) and concatenates them together as the prompt context for ICL-based code generation (Section 3.3). The overview of LAIL is shown in Figure 2.

3.1 Estimating Examples with LLMs

In this section, we make LLMs themselves (LLM-aware) estimate candidate examples based on their needs. Considering that the candidate examples that LLMs need can facilitate LLMs to generate

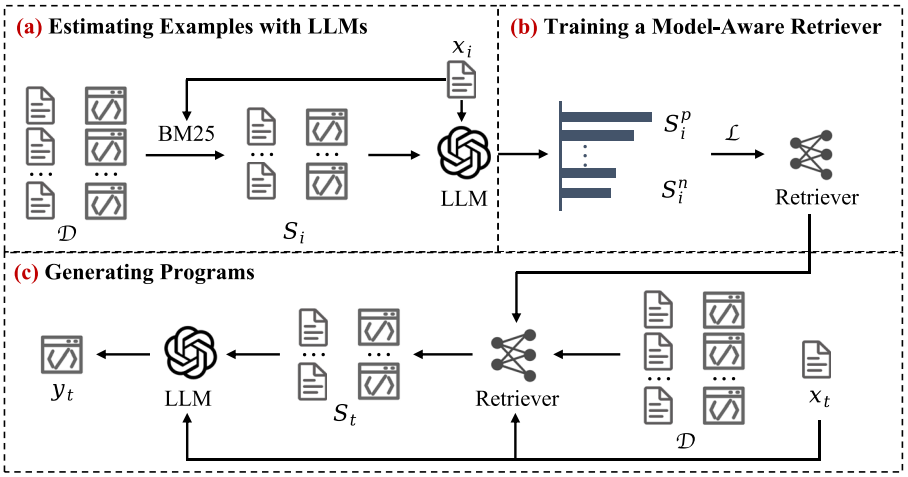

Fig. 2. The overview of LAIL. (a) LAIL uses LLMs themselves to estimate candidate examples and label them as positive or negative. (b) Based on the label date, LAIL then trains a model-aware retriever to align with the preference of LLMs with a contrastive loss. (c) Given a test requirement, the optimized retriever selects several examples that are inputted to LLMs for code generation.

correct programs, we use the prediction probability of the ground-truth program as the feedback of LLMs. Like existing studies [38], we treat the training set of a dataset as the candidate example pool. To estimate the candidate examples, for a requirement in the candidate pool, LAIL requires LLMs themselves to estimate the remaining examples of the candidate pool by calculating the prediction probability of generating ground-truth programs based on the requirement and each remaining example. The higher the probability is, the better the example can meet the needs of LLMs for solving the task. According to the probability feedback, we then label the remaining candidate examples into positive and negative examples. Positive examples are helpful for LLMs to generate correct programs, while negative examples are trivial and should be ignored. Following the above labeling process, we traverse the entire candidate pool and acquire the final labeled data. Based on the labeled data, LAIL trains a model-aware retriever to learn the needs (referring to preferences) of LLMs in Section 3.2.

Since the size of the candidate pool commonly is large, the cost to traverse all examples is high. We introduce a two-stage process to estimate candidate examples in the training set, where we first filter out a few candidate examples for a specific requirement, and then estimate the remaining examples with LLMs themselves.

In the first stage, for each programming task in the training set, to acquire its demonstration examples, we first use BM25 [59] to filter out a portion of the remaining examples. The reason we use BM25 is that it is easy to implement and is effective in filtering hard negative examples, which calculates the n-gram matching between texts. Concretely, the training set is $\mathcal{R} = \{e_i\}_1^N$, where e_i is the ith requirement-code example (x_i, y_i) in $\mathcal{R}$. For an example $e_i = (x_i, y_i)$, we use BM25 to calculate the textual similarities between x_i and x_j, where $i \neq j$, and then acquire its score set $B_i = \{b_j\}_{j=1}^{N-1}$. According to the score set B_i, we sort b_j from high to low and only keep the first t examples as the set $S_i = \{(x_q^i, y_q^i)\}_{q=1}^t$, where $t \ll N$. It is worth noting that LAIL is different from the existing BM25 selection approach [38] that uses BM25 to sort all examples. LAIL only uses BM25 to filter out the hard negative examples, meanwhile, retains the soft positive examples. Then,

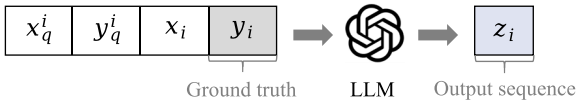

Fig. 3. The illustration of the input and output of LLMs in estimating a candidate example with LLMs themselves.

LAIL makes LLMs themselves estimate these soft positive examples and selects a few candidate examples they need for generating correct programs.

In the second stage, LAIL uses LLMs themselves to estimate the soft positive examples. Specifically, to accomplish the requirement x_i, we first use LLMs to measure the needs of LLMs for each example (x_q^i, y_q^i) in S_i. We concatenate the candidate example (x_q^i, y_q^i) and the requirement x_i and input them into LLMs. Meanwhile, we set the length of the output sequence z_i to 0 since the prediction probability of y_i is determined by LLMs, having nothing to do with the output. The input and output of LLMs are shown in Figure 3. Next, we calculate the prediction probability of the ground-truth program y_i. We design an estimation metric $\mathcal{G}$ to quantify the probability feedback, which is defined as follows:

$$\mathcal{G}_q = \frac{1}{|y_i|} \times P_{LLM}(y_i | x_q^i, y_q^i, x_i) \tag{3}$$

$$P_{LLM}(y_i | x_q^i, y_q^i, x_i) = \sum_{u=1}^{|y_i|} log(p(t_{i,u} | x_q^i, y_q^i, x_i, t_{i,<u})) \tag{4}$$

where $y_i = \{t_1, \cdots, t_b\}$ and t_u is the uth token in y_i. The higher the metric is, the more LLMs need the example for completing the requirement x_i. Finally, we obtain the estimation set $\{\mathcal{G}_q\}_{q=1}^t$ for the requirement x_i.

Based on this, we rank all examples in the set S_i according to their metric scores in $\{\mathcal{G}_q\}_{q=1}^t$. The *top-z* examples with higher scores are inputted into the positive example set S_i^p, meanwhile, the *bottom-v* examples constitute the soft negative example set S_i^n since good examples should be beneficial for LLMs to generate correct programs. We apply this two-stage procedure to the entire training set and finally acquire the labeled data $\mathcal{D}$:

$$\mathcal{D} = \{(e_i, \{S_i^p, S_i^n\})\}_{i=1}^N \tag{5}$$

where $e_i = (x_i, y_i)$ is the ith examples in the training set. S_i^p and S_i^n are the positive and negative example set of e_i, respectively. N is the number of examples in the training set $\mathcal{R}$.

Our labeled data are consistent with the preference of LLMs, where the needful examples with the high estimation metric are treated as positive examples and the candidates helpful less are labeled as negative examples.

3.2 Training a Model-Aware Retriever

As described in Section 3.1, the labeled data can reflect the preference of LLMs. In this section, we use the labeled positive and negative data to train a model-aware retriever, aiming to align with the preference of LLMs. After being trained, given a test requirement, the retriever can select a set of demonstration examples that are beneficial for LLMs to generate correct programs.

We use the multi-layer bidirectional Transformer [70] as the backbone of the retriever. The retriever consists of a 12-layer Transformer with 768 hidden size and 12 attention heads in each layer. To learn the preferences of LLMs, we introduce a contrastive learning objective, because it

can extract similar items and distinguish the dissimilar items on a specific dimension. In our article, the objective target is to acquire the candidate examples that LLMs need. We initialize the retriever with GraphCodeBERT [28] and further train it with contrastive learning. For an example e_i in the labeled data $\mathcal{D}$, we randomly select a positive example e_i^p from S_i^p. Meanwhile, we randomly choose a candidate example e_i^n from the soft negative set S_i^n and select h hard negative examples from the set $\mathcal{H}$ (e.g., $\mathcal{H} = \mathcal{R} \setminus S_i$) as its negative example set $\mathbb{N}_i$. Next, we apply the retriever to encode the requirements of these examples and acquire their representations, individually. Specifically, for the example e_i, we utilize Byte-Pair Encoding algorithm [63] to tokenize the requirement of e_i and acquire a sequence of tokens $\{c_{i,1}, c_{i,2}, ..., c_{i,n_c}\}$, where n_c denotes the sequence length. Then, a classification symbol [CLS] and a segment separation symbol [SEP] are concatenated to the sequence, forming the input as $\{[\texttt{CLS}], c_{i,1}, c_{i,2}, ..., c_{i,n_c}, [\texttt{SEP}]\}$. The output of the retriever is the vector representation of the requirement, i.e., $\{E^i_{[\texttt{CLS}]}, E^i_1, ..., E^i_{n_c}, E^i_{[\texttt{SEP}]}\}$. We perform the same pre-processing procedure for the requirements of positive and negative examples, respectively, and acquire the vector representation of the requirement for each positive or negative example, denoting as $\left\{E^{i,p}_{[\texttt{CLS}]}, E^{i,p}_1, ..., E^{i,p}_{n_q}, E^{i,p}_{[\texttt{SEP}]}\right\}$ and $\left\{E^{i,n}_{[\texttt{CLS}]}, E^{i,n}_1, ..., E^{i,n}_{n_l}, E^{i,n}_{[\texttt{SEP}]}\right\}$, respectively. Following previous works [21, 28], we utilize $E^i_{[\texttt{CLS}]}$, $E^{i,p}_{[\texttt{CLS}]}$, and $E^{i,n}_{[\texttt{CLS}]}$ as the entity representation of the requirement of e_i, e_i^p, and e_i^n since they are the aggregated representations. Finally, we model the relations of these representatives with contrastive loss. Following SimCLR [12], the learning objective $\mathcal{L}$ is formulated as:

$$\mathcal{L} = -\left[log \frac{e^{s(E^i_{[\texttt{CLS}]}, E^{i,p}_{[\texttt{CLS}]})/\tau}}{\sum_{E^{i,n}_{[\texttt{CLS}]} \in \mathbb{N}_i} e^{s(E^i_{[\texttt{CLS}]}, E^{i,n}_{[\texttt{CLS}]})/\tau}} \right] \tag{6}$$

where τ is a temperature parameter. $E_{[\texttt{CLS}]}$ means the representation of a requirement. s(·) calculates the cosine similarity of two vectors.

Based on the objective, the retriever learns the preference of LLMs. For a test requirement, the model-aware retriever is able to select examples that LLMs need from the training set, helping LLMs generate correct programs.

3.3 Generating Programs

During the inference, instead of using heuristic approaches, we apply the trained retriever to select a few examples from the training set, which can retrieve examples with high metric scores for a test requirement and then provide useful demonstration examples to LLMs for ICL-based code generation. Specifically, we first feed requirements in the training set $\mathcal{R}$ to the trained retriever individually and acquire their representational vectors $\left\{E^i_{\texttt{CLS}}\right\}_{i=1}^N$. Given a test requirement x_t, we obtain its representation $E^t_{\texttt{CLS}}$ by using the retriever. Then, we match $E^t_{\texttt{CLS}}$ and $E^i_{\texttt{CLS}}$ and thus lead to N pairs of representations $\left\{(E^t_{\texttt{CLS}}, E^i_{\texttt{CLS}})\right\}_{i=1}^N$. We calculate their cosine similarities $\{c_i\}_{i=1}^N$ of all pairs and rank candidate examples according to their similarity scores from high to low. The *top-r* examples are selected for ICL-based code generation.

After acquiring demonstration examples, we concatenate the top-r examples as sequence $\{e_1; \cdots; e_i; \cdots; e_r\}$ where their similarity scores gradually decrease (e.g., $c_1 < c_i < c_r$). The sequence and the test requirement x_t are then concatenated as a sequence. Finally, we feed the sequence into LLMs and make LLMs generate programs with nuclear sampling [29] as described in Equation (2). The generation process is presented in Figure 4.

Note that LAIL only needs to encode examples in the training set one time. Given a test requirement, LAIL just calculates cosine similarities between it and all candidates.

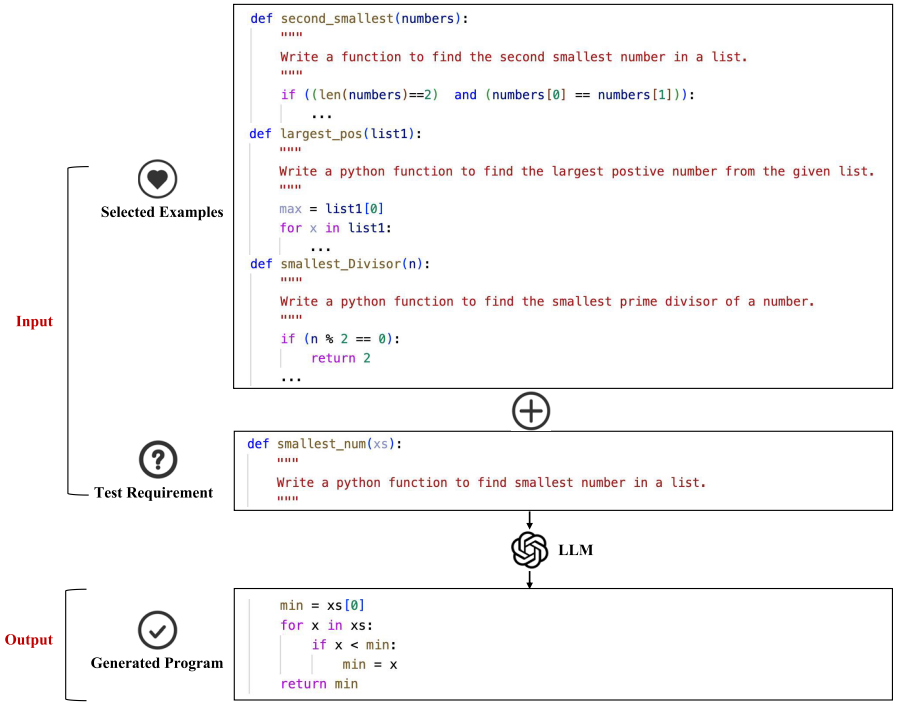

Fig. 4. The illustration of ICL-based code generation.

4 Experimental Setup

To investigate the effectiveness of LAIL, we perform a large-scale study to answer four research questions. In this section, we describe the details of our study, including datasets, evaluation metrics, baselines, base LLMs, and experimental details.

4.1 Research Questions

Our study aims to answer the following research questions.

RQ1: How does LAIL perform compared to the SOTA baselines? This RQ aims to verify that LAIL can generate more correct programs than SOTA baselines. We apply four widely used LLMs to evaluate our approach. We compare LAIL to seven baselines in four datasets. These datasets cover multiple mainstream languages, including Java, Python, and C++. Then, we measure the correctness of generated programs with the Pass@k metric. For the Pass@k metric, higher scores indicate the better performance of approaches.

RQ2: Do developers prefer programs generated by LAIL? The ultimate goal of a code generation model is to assist developers in writing programs. In this RQ, we hire 10 developers to manually estimate the programs generated by LAIL and baselines. We evaluate generated programs in three aspects, including correctness, code quality, and maintainability.

RQ3: How effective is the probability feedback of LLMs to estimate examples? As described in Section 3.1, we leverage the prediction probability of ground-truth programs to estimate candidate examples. In this RQ, we explore the effectiveness of LLMs' probability feedback in estimating candidate examples and compare it with other estimation designs, aiming to provide in-depth insights to researchers in selecting demonstration examples for ICL-based code generation.

Table 1. Statistics of Three Code Generation Datasets on the Different Split Sets

	MBJP	MBPP	MBCPP	HumanEval	DevEval
Language	Java	Python	C++	Python	Python
#Train	383	384	413	–	–
#Valid	90	90	–	–	–
#Test	493	500	435	164	1,874
Avg. test cases of examples	3	3	3	7.7	–
Avg. tokens of requirements	16.71	16.50	17.38	37.2	91.5
Avg. tokens of programs	247.79	92.68	113.94	24.4	–

Avg. represents "Average."

RQ4: What is the performance of LAIL on the repository-level code generation? In addition to function-level code generation, we also apply LAIL to the repository-level code generation dataset, in order to evaluate the effectiveness of our approach on the complex real-world programming scenario. We treat functions contained in the repository as the candidate pool. Then, LAIL retrieves a few functions from the candidate pool as demonstration examples. To avoid the impact of context programs on results, in this article, LLMs generate programs based on the requirement, corresponding signatures, and a few demonstration examples. We then input the generated programs into the needed position of the repository and use test cases to evaluate their correctness.

4.2 Datasets

We evaluate LAIL on three mainstream languages, e.g., Java, Python, and C++. Concretely, we select five representative datasets, including MBJP [3], MBPP [4], MBCPP [3], HumanEval [11], and DevEval [36]. The statistics of these datasets are presented in Table 1.

MBPP [4] contains 974 Python programming problems constructed by crowd-sourcing. Each example consists of a brief description, a single self-contained function solving the problem specified, and three test cases to evaluate the correctness of the generated programs. The problems range from simple numeric operations or tasks that require the basic usage of standard library functions to tasks that demand nontrivial external knowledge.

MBJP [3] and *MBCPP* [3] have 966 and 848 crowd-sourced programming problems in Java and C++, respectively. Each problem consists of a single self-contained function, three test cases, and a text description that has typically one sentence each. These problems mainly consist of mathematical manipulations, list processing, string processing, and other data structures. We follow previous studies [3, 4] to split the above three datasets into the training set, the valid set, and the test set, respectively. We measure the performance of different ICL approaches on the test set.

HumanEval [11] is a Python code generation dataset, which contains 164 hand-written programming problems. Each programming problem consists of a natural language requirement, a function signature, and several test cases, with an average of 7.7 test cases per problem.

DevEval [36] is a repository-level code generation dataset, which is collected from real-word code repositories. The dataset aligns with real-world code repositories in multiple dimensions, e.g., real code distributions, sufficient dependencies, and real-scale repositories. It comprises 1,874 testing samples from 117 repositories. Each repository contains 243 files, 45,941 program lines, and 4,672 code dependencies on average. In this article, we randomly select two domains from this dataset to evaluate LAIL and baselines, including the scientific engineering domain and text

processing domain. We randomly split the tasks of the two domains into the training set and the test set. Finally, we acquire 101 examples in the training set and 49 examples in the test set.

4.3 Evaluation Metrics

Following previous code generation studies [3, 11], we use the pass rate (e.g., Pass@k) to evaluate LAIL. Pass@k evaluates the functional correctness of the generated programs by executing test cases, which is a strict metric. Precisely, given a test requirement, we generate k programs with the sampling strategy. If any of the generated k programs passes all test cases, we think the requirement is solved. Finally, the percentage of solved requirements in all test requirements is treated as Pass@k. In this article, we set k to 1, 3, and 5.

4.4 Baselines

There are a few studies to investigate example selection for ICL-based code generation, including zero-shot learning [8], random selection [11], and AceCoder [38]. Among these approaches, AceCoder [38] is the SOTA baseline. Note that Gao et al. [23] empirically explore the impact of demonstration examples on ICL in code intelligence tasks and provide some important findings. However, this study does not aim to provide a systematical selection approach, thus we do not employ it as the baseline in this article.

Zero-shot learning [8] directly inputs a requirement into LLMs without any examples as the prompt. LLMs directly generate programs for the given requirement.

Random [11] selection randomly retrieves a few examples from the training set. Then, the selected examples and the test requirement are fed into LLMs. Finally, LLMs predict the source code based on the prompt without any parameter update.

AceCoder [38] uses BM25 [59] to calculate the textual similarities between a test requirement and the requirements of candidates. Then, it retrieves a set of examples equipped with high similarities from the training set.

To extensively evaluate the effectiveness of LAIL, we also transfer some advanced ICL approaches in natural language processing to the source code.

TOP-k-SBERT [47] leverages Sentence-BERT [57], a representative sentence encode, to encode all requirements in the training set. Given a test requirement, we first encode it and compute semantic similarities between it and the training requirements. Next, we select the top-k similar examples to prompt LLMs in code generation.

TOP-k-GraphCodeBERT [28] is a variant of TOP-k-SBERT. It applies GraphCodeBERT [28] to encode requirements and retrieve a few examples from the training set based on semantic similarity.

TOP-k-VOTE [64] is a graph-based method [64] to vote examples. It first encodes each example by GraphCodeBERT [28], and each example is a vertice in the graph. Each vertice connects with its k nearest vertices based on their semantic similarities. Finally, the approach treats the k nearest candidates as demonstration examples.

Uncertainty-Target [15] assumes that examples with higher uncertainty have a greater impact on LLMs. It defines uncertainty as the perplexity when LLMs generate ground truths. The approach computes the uncertainty of each candidate and selects k items with high perplexity as a prompt.

4.5 Base LLMs

This article focuses on code generation with LLMs. Thus, we select four popular LLMs for code generation as the base models, including CodeGen-Multi-16B [74], CodeLlama-7B and -34B [60], Text-davinci-003 [24], and GPT-3.5-turbo [9]. The details of the base models are shown as follows:

CodeGen-Multi-16B [74] is a language models for code generation. CodeGen is trained with a 635GB code corpus and 1,159GB English text data. In this article, we leverage the largest version with 16 billion parameters as the base model.

CodeLlama-7B and -34B [60] is for source code based on Llama 2 [45], providing impressive performance among open models by applying the training and fine-tune procedures. It is trained on the sequence of 16k tokens and achieves improvements on inputs with up to 100k tokens.

Text-davinci-003 [24] is a closed-source LLM. It is trained on a large unlabeled multimodel corpus with 175 billion parameters, supporting the natural language and programming language. In this article, we use OpenAI's APIs to access it.

GPT-3.5-turbo [9] is a powerful language model for code generation. It is trained on much natural language text and programming data. Then, it is continually trained with reinforcement learning and learns to align with human instructions. In this article, instead of selecting GPT-4 [25], we import OpenAI's APIs to access GPT-3.5-turbo because we have limited expenses.

4.6 Implementation Details

Estimating Examples with LLMs. When collecting the predicted probabilities of ground-truth programs, we feed a requirement and a candidate example into LLMs and set the length of generated programs as 0 since the predicted probabilities of input are determined by LLMs and are independent of output. For efficiency, the number of examples in the set S_i is 50 since it is efficient enough meanwhile has a high probability of containing positive samples. We set the number of the positive example set S_i^p and the soft negative example set S_i^n as 5, respectively. Besides, we analyze their effects on code generation performance in Section 6.3.

Learning a Model-Aware Retriever. In this procedure, we set the number of the negative example set $\mathbb{N}_i$ as 64, that is, the number of hard negative examples h is set to 63. For each epoch, we randomly select 63 hard negative examples from the set $\mathcal{H}$ and choose 1 soft negative example from S_i^n. We also attempt to set the size of $\mathbb{N}_i$ to 32 and 128. The analysis results are shown in Section 6.3. The learning rate is 5e-5 and the batch size is 32. We train the neural retriever for about 1 hour on 4 NVIDIA A100.

Generating Programs. We treat the LLM as a black-box generator and sample programs from it. The input of LLMs only contains demonstration examples and a test requirement without any natural language instructions. During generation, we use nuclear sampling [29] to decode, where the temperature is set to 0.8 and the top-p is set to 0.95. The maximum generated length is 500 tokens. For each test requirement, we generate five programs. For a fair comparison, we set the same parameters to generate programs for all baselines and LAIL.

5 Results and Analysis

RQ1: How Does LAIL Perform Compared to the State-of-the-Art Baselines?

In RQ1, we apply LAIL and baselines to three LLMs including CodeGen-Multi-16B [74], CodeLlama-7B and -34B [60], and Text-davinci-003 [24]. Because the closed-source GPT-3.5-turbo [9] cannot provide the prediction probability of the ground-truth program, we do not present the results of LAIL on this base model. In Section 6, we further analyze the performance of GPT-3.5-turbo [9] based on the feedback of other LLMs.

Results. Tables 2–4 and 6 report the Pass@k (k ∈ [1, 3, 5]) of different approaches on MBJP, MBPP, MBCPP, and HumanEval datasets, respectively. The human evaluation results for all approaches are presented in Appendix A. Numbers in bold mean the best performances among LAIL and baselines. The percentages represent the improvements from the best performances of baselines

Table 2. Evaluation Results of LAIL and Baselines on CodeGen-Multi-16B at MBJP, MBPP, and MBCPP

	MBJP			MBPP			MBCPP		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Zero-shot learning [8]	14.27	23.69	27.83	8.80	20.60	25.60	15.39	25.97	30.94
Random [11]	15.74	24.25	28.14	15.20	25.40	28.40	15.70	28.72	32.25
AceCoder [38]	17.65	27.18	30.63	16.80	26.40	29.80	17.47	30.11	36.47
TOP-k-SBERT [47]	16.63	26.77	29.21	15.40	25.00	29.80	18.09	29.43	33.33
TOP-k-GraphCodeBERT [28]	17.44	24.34	28.40	18.00	25.80	28.40	18.16	30.68	36.35
TOP-k-VOTE [64]	15.42	21.70	23.73	17.40	26.00	29.40	17.01	30.03	35.63
Uncertainty-Target [15]	17.09	23.06	27.93	14.60	24.20	28.80	17.70	30.35	35.78
LAIL	**21.30** (↑ **11.58%**)	**28.49** (↑ **4.82%**)	**32.05** (↑ **4.64%**)	**18.60** (↑ **3.33%**)	**27.80** (↑ **6.92%**)	**30.60** (↑ **2.68%**)	**19.08** (↑ **5.07%**)	**31.36** (↑ **2.21%**)	**37.94** (↑ **4.03%**)

Numbers in bold indicate the best performances among LAIL and all baselines in terms of Pass@k (i.e., k = 1, 3, and 5). "↑" means the relative improvements from the best results among baselines to LAIL.

Table 3. Evaluation Results of LAIL and Baselines on CodeLlama-34B at MBJP, MBPP, and MBCPP in Terms of Pass@k (i.e., k = 1, 3, and 5)

	MBJP			MBPP			MBCPP		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Zero-shot learning [8]	10.95	24.14	33.06	5.40	14.20	19.20	0.23	0.23	0.92
Random [11]	37.93	53.35	57.81	32.00	47.80	53.20	37.93	54.25	61.15
AceCoder [38]	44.85	57.26	60.40	34.60	45.40	50.80	42.76	55.86	59.54
TOP-k-SBERT [47]	42.72	56.58	59.97	34.80	48.20	52.20	43.14	56.08	60.27
TOP-k-GraphCodeBERT [28]	42.19	56.39	59.43	35.00	48.20	52.00	43.45	56.55	61.60
TOP-k-VOTE [64]	43.20	56.99	60.26	34.20	45.00	49.20	38.39	53.33	58.62
Uncertainty-Target [15]	37.73	50.91	56.39	29.40	44.20	49.20	2.30	5.29	8.05
LAIL	**45.44** (↑ **1.32%**)	**57.81** (↑ **0.96%**)	**61.26** (↑ **1.42%**)	**35.80** (↑ **2.29%**)	**48.80** (↑ **1.24%**)	**52.80** (↑ **1.15%**)	**43.97** (↑ **1.20%**)	**57.04** (↑ **0.87%**)	**62.03** (↑ **0.70%**)

Numbers in bold indicate the best results among all approaches and "↑" means the relative improvements compared to the best performances of baselines.

to the counterpart of LAIL. We use T-tests as the statistical tests and calculate Cohen's d as effect size to measure the validity of our experimental results. *P*-value is smaller than 0.05 on the four datasets in terms of Pass@k (i.e., k = 1,3,5). Cohen's d of each dataset on Pass@k is larger than 0.7.

Analyses. (1) *LAIL achieves the best performance among all approaches.* In all datasets, LAIL generates more correct programs than baselines. Compared to the best results of baselines, LAIL outperforms them by 11.58%, 3.33%, and 5.07% on CodeGen-Multi-16B, 1.32%, 2.29%, and 1.20% on CodeLlama-34B, and acquires 4.38%, 2.85%, and 2.74% improvements on Text-davinci-003 in terms of Pass@1 on MBJP, MBPP, and MBCPP, respectively. LAIL achieves 1.53% improvements on HumanEval in Pass@1 when CodeLlama-7B is the base model. Note that Pass@1 is a very strict metric and is difficult to improve. The significant improvements prove that LAIL can select proper examples and help LLMs generate more satisfying programs. (2) *Selecting proper demonstration examples is critical to the performance of ICL-based code generation.* For instance, compared to random selection, LAIL acquires 35.32%, 22.37%, and 21.53% improvements in Pass@1 on CodeGen-Multi-16B in the MBJP, MBPP, and MBCPP datasets. Subsequently, AceCoder and other heuristic approaches further improve code generation performance by selecting textually or semantically similar examples. Considering the needs of LLMs, LAIL uses LLMs themselves to estimate examples and trains a model-aware retriever to learn the preference of LLMs, achieving the best results

Table 4. The Performance of Our LAIL and Baselines on Text-davinci-003 at MBJP, MBPP, and MBCPP in Terms of the Pass@k (i.e., k = 1, 3, and 5) Metric

	MBJP			MBPP			MBCPP		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Zero-shot learning [8]	44.83	53.05	59.72	20.00	27.00	29.60	21.85	37.94	49.90
Random [11]	47.87	59.83	63.69	43.00	56.40	60.80	50.02	61.43	65.28
AceCoder [38]	50.91	61.25	65.15	47.40	61.00	64.20	53.25	63.22	66.21
TOP-k-SBERT [47]	50.30	60.46	64.70	49.00	61.40	64.00	52.87	63.21	67.13
TOP-k-GraphCodeBERT [28]	50.30	60.85	64.50	49.20	58.20	64.20	52.64	63.19	67.28
TOP-k-VOTE [64]	50.52	60.45	63.49	47.80	59.20	63.40	51.03	62.98	65.51
Uncertainty-Target [15]	47.67	59.23	63.69	37.80	51.40	57.80	42.87	52.60	57.18
LAIL	**53.14** (↑ **4.38%**)	**62.87** (↑ **2.64%**)	**66.72** (↑ **2.41%**)	**50.60** (↑ **2.85%**)	**62.40** (↑ **1.63%**)	**65.20** (↑ **1.53%**)	**54.71** (↑ **2.74%**)	**65.98** (↑ **4.37%**)	**69.67** (↑ **3.55%**)

"↑" represents the relative improvements compared to the SOTA baseline. Numbers in bold mean the best performances among all approaches.

among all approaches. That demonstrates the importance of selecting proper examples in ICL and verifies the reasonability of using LLMs themselves to estimate examples. (3) *LAIL is effective in LLMs with different sizes and different programming languages.* As described above, our approach achieves impressive performance on all base LLMs. Besides, Tables 2–4 and 6 also show that LAIL can generate more correct programs in different mainstream languages including Java (MBJP), Python (MBPP and HumanEval), and C++ (MBCPP). This reveals that LAIL has good generalization ability and can be applied to different LLMs and languages.

In Table 3, we find that the Uncertainty-Target approach performs poorly on the MBCPP dataset, only resulting in 2.30% in terms of Pass@1. The approach selects a few examples with high uncertainty, and then, the selected examples are applied to all test requirements. We carefully analyze the demonstration examples and generated programs in the approach. The programs in demonstration examples have two common features: being very short and only containing simple operations such as "return null." We observe that the generated programs are biased by the two features. They are similar to the programs in the demonstration examples and thus cannot implement the test requirements.

Answer to RQ1: LAIL achieves the best results compared to all baselines and has good generalization ability. In MBJP, MBPP, and MBCPP datasets, LAIL acquires 11.58%, 3.33%, and 5.07% improvements on CodeGen-Multi-16B, 1.32%, 2.29%, and 1.20% on CodeLlama-34B, and achieves 4.38%, 2.85%, and 2.74% improvements on Text-davinci-003 at Pass@1. In HumanEval, LAIL achieves 1.53% improvements on CodeLlama-7B in terms of Pass@1. The impressive improvements prove that our approach can effectively learn and align with the needs of LLMs. Thus, LAIL is able to select suitable examples for ICL-based code generation.

RQ2: How Effective Is the Probability Feedback of LLMs to Estimate Examples?

In this RQ, we aim to evaluate the effectiveness of LLMs' probability feedback in estimating candidate examples. In LAIL, we require LLMs themselves to estimate a candidate example by calculating the prediction probability of the ground-truth program based on a requirement and the candidate example. Besides the probability feedback, there are other perspectives to estimate the

ability of candidate examples in promoting LLMs' coding ability, including the similarity between the generated code and the ground-truth code, and the functional correctness of the generated programs. In the code generation task, BLEU and CodeBLEU are commonly used to evaluate the similarity between the generated program and the ground-truth program, meanwhile, Pass@k is usually applied to calculate the functional correctness of the generated code by using test cases. Therefore, we introduce BLEU, CodeBLEU, and Pass@k to estimate the ability of candidate examples in promoting LLMs' coding ability from different perspectives, aiming to inspire developers to select the appropriate approach to evaluate candidate examples in ICL-based code generation. We conduct extensive experiments to explore and analyze the effectiveness of these designs in measuring candidate examples with LLMs themselves.

Results. We explore three other designs to estimate examples in this RQ, including the Match-BLEU, Match-CodeBLEU, and Match-Pass@k designs based on BLEU, CodeBLEU, and Pass@k. We first present the definitions of BLEU, CodeBLEU, and Pass@k and then describe Match-BLEU, Match-CodeBLEU, and Match-Pass@k designs based on them. Specifically, BLEU [53] computes the *n-gram* overlapping between the generated program and the ground truth, which can measure the token-level similarity of two sequences. It is formulated as $\text{BP} \cdot \exp(\sum_{n=1}^{N} w_n \log p_n)$, where N is set to 4, BP is a brevity penalty to prevent generating very short programs, and p_n represents the *n-gram* matching precision score. CodeBLEU [58] is a version of BLEU and is designed for source codes. This metric measures the *n-gram* match, the semantic match, and the syntactic match between the generated code tokens and the reference code tokens. Pass@k computes the pass rate of the generated codes that pass the unit tests. Concretely, models automatically generate n programs for each requirement. A requirement is considered solved if any of the first $k \leq n$ generated programs can pass all the test cases. The percentage of solved requirements in total requirements is treated as Pass@k. The Match-BLEU uses BLEU score of the generated program to measure candidate examples. Specifically, in the second stage of Section 3.2, we input a candidate example (x_q^i, y_q^i) and a requirement x_i into LLMs and acquire the generated program $\hat{y}_i$. Then, we calculate the BLEU score $\mathcal{B}_q$ of the predicted program $\hat{y}_i$. The higher the BLEU score is, the more LLMs need the candidate example for completing the requirement x_i. We apply this process to the candidate set S_i and obtain the BLEU score set $\{\mathcal{B}_q\}_{q=1}^{t}$. Finally, we label positive and negative candidates for the requirement x_i based on the BLEU score set $\{\mathcal{B}_q\}_{q=1}^{t}$. Similar to Match-BLEU, the Match-CodeBLEU approach applies the CodeBLEU score of the predicted program to label examples. The Match-Pass@k design uses the Pass@1 score of the generated program to assess candidates. In this RQ, we use Text-davinci-003 and CodeGen-Multi-16B as the base LLMs and evaluate their performance on the three datasets. The results of different estimation designs are represented in Table 5. For comparison, Table 5 also presents the results of random selection. We do not provide the performance of the Match-Pass@k in Text-davinci-003 since the model is not available at the time we revise the article.

Analyses. (1) *The probability-based approach of LAIL is more effective than Match-BLEU, Match-CodeBLEU, and Match-Pass@k in estimating candidate examples.* In all datasets, LAIL outperforms them by 21.29%, 8.31%, and 15.49% on the three datasets at Pass@1 when CodeGen-Multi-16B is the base model, respectively. The reason might be that the predicted probability of ground truth in LAIL can accurately reflect how certain an LLM is for generating correct programs. The BLEU-based and CodeBLEU-based methods can only reflect the literal accuracy of programs generated by LLMs, but cannot provide how certain LLMs are when predicting programs. Similarly, the Pass@k-based method can only reflect whether the generated program of LLMs is functionally correct or not by using a few test cases, which still does not represent the certainty of LLMs when generating programs. (2) *Match-CodeBLEU approach is more effective than Match-BLEU method.*

Table 5. The Comparison of Different Designs to Estimate Examples on MJBP, MBPP, and MBCPP at CodeGen-Multi-16B and Text-davinci-003 in Terms of Pass@k (i.e., k = 1, 3, and 5)

	MBJP			MBPP			MBCPP		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
	Text-davinci-003								
Random [8]	47.87	59.83	63.69	43.00	56.40	60.80	50.04	61.45	65.28
Match-BLEU	50.19	59.87	63.92	46.20	60.20	62.40	50.26	61.47	65.73
Match-CodeBLEU	51.07	60.04	64.58	47.80	59.20	63.80	52.72	62.03	66.80
Match-Pass@k	–	–	–	–	–	–	–	–	–
Probability-Based (LAIL)	**53.14** (↑ **4.05%**)	**62.87** (↑ **4.71%**)	**66.72** (↑ **3.31%**)	**50.60** (↑ **5.86%**)	**62.40** (↑ **3.65%**)	**65.20** (↑ **2.19%**)	**54.71** (↑ **3.77%**)	**65.98** (↑ **6.37%**)	**69.67** (↑ **4.30%**)
	CodeGen-Multi-16B								
Random [8]	15.74	24.25	28.14	15.20	25.40	28.40	15.70	28.72	32.25
Match-BLEU	16.35	25.41	29.06	15.40	25.40	28.60	16.43	30.10	34.02
Match-CodeBLEU	17.56	26.39	29.59	16.20	25.80	29.40	16.52	29.65	35.36
Match-Pass@k	15.83	24.71	28.68	14.60	25.20	28.00	16.08	29.48	34.21
Probability-Based (LAIL)	**21.30** (↑ **21.29%**)	**28.49** (↑ **7.96%**)	**32.05** (↑ **3.31%**)	**18.60** (↑ **8.31%**)	**27.80** (↑ **14.81%**)	**30.60** (↑ **7.75%**)	**19.08** (↑ **15.49%**)	**31.36** (↑ **4.18%**)	**37.94** (↑ **7.29%**)

The values in parentheses mean the improvements achieved by our probability-based design. "Probability-Based" represents LAIL. Numbers in bold mean the best performances among all designs to estimate examples.

Match-CodeBLEU outperforms Match-BLEU on all datasets. We argue that BLEU can only measure the n-gram similarity between the generated programs and the ground-truth source codes, while CodeBLEU can measure the n-gram match, the semantic match, and the syntactic match between the generated programs and the reference programs, which is more comprehensive for evaluating the generated programs. (3) *The performance of Match-Pass@k is worse than other designs including Match-BLEU, Match-CodeBLEU, and the probability-based approach (LAIL).* Because the Pass@1 results only have two values (i.e., 0 or 1) for a program. For each requirement, there are many cases of generated correct programs (Pass@1 = 1) when different candidate examples are input into LLMs, which leads to these candidate examples not being effectively ranked in terms of helping LLMs generate correct programs according to Pass@1 scores.

Answer to RQ2: Probability-based approach is better than Match-CodeBLEU, Match-BLEU, and Match-Pass@k designs. It can effectively reflect the LLMs' preference for candidate examples given a test requirement.

RQ3: What Is the Performance of LAIL on the Repository-Level Code Generation?

In this RQ, we access the performance of LAIL on the repository-level code generation to evaluate whether our approach can generalize to the complex real-world programming scenario.

Results. We evaluate LAIL and baselines on the DevEval dataset [36]. DevEval contains a lot of real-world repositories collected from Github, where each repository has 243 files, 45,941 code lines, and 4,672 dependencies on average. Given a requirement from a repository, we use tree-sitter[2] to parse the repository and acquire all functions of the repository. We treat functions contained in the repository as the candidate pool. Then, LAIL and baselines retrieve a few functions from the candidate pool as demonstration examples. On average, each repository contains 1,957 functions. We verify LAIL and baselines on CodeLlama-7B and use Pass@k to measure the correctness of generated programs. The results of LAIL and baselines are shown in Table 6. The percentages in

[2]https://github.com/tree-sitter/tree-sitter.

Table 6. The Performance of LAIL and Baselines on the HumanEval Dataset at CodeLlama-34B and the DevEval Dataset at CodeLlama-7B in Terms of Pass@k (i.e., k = 1, 3, and 5)

	HumanEval			DevEval		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Zero-shot learning [8]	4.02	10.67	15.85	10.83	18.37	22.45
Random [11]	33.78	51.89	59.75	4.49	11.22	16.32
AceCoder [38]	35.43	53.46	60.97	9.79	16.53	18.37
TOP-k-SBERT [47]	33.78	52.87	60.98	12.24	22.65	26.53
TOP-k-GraphCodeBERT [28]	35.43	53.54	62.07	9.79	18.57	22.45
TOP-k-VOTE [64]	33.90	51.34	59.76	8.57	15.92	18.37
Uncertainty-Target [15]	34.27	52.31	61.58	8.57	16.71	20.56
LAIL	**35.97** (↑ **1.53%**)	**53.86** (↑ **1.52%**)	**63.29** (↑ **0.59%**)	**13.47** (↑ **10.04%**)	**24.49** (↑ **8.12%**)	**27.76** (↑ **4.63%**)

"↑" represents the relative improvements compared to the SOTA baseline. Numbers in bold mean the best performances among all approaches.

parentheses indicate the improvements achieved by LAIL compared to the SOTA baseline-TOP-k-SBERT.

Analyses. (1) *LAIL performs better than baselines on the DevEval dataset.* Compared to the SOTA baseline-TOP-k-SBERT, LAIL outperforms it by 10.04%, 8.12%, and 4.63% in terms of Pass@1, 3, and 5. Pass@k is a very strict metric, and it is difficult to improve. The impressive improvements prove that LAIL is not only suitable for function-level code generation, but also effective in complex repository-level code generation. Given a requirement, LAIL can effectively select demonstration examples that LLMs need from the function pool of a repository. (2) *The performances of some baselines are lower than the results of the zero-shot learning setting.* Specifically, zero-shot learning achieves 10.83%; however, the generated results of some baselines, such as TOP-k-GraphCodeBERT and TOP-k-VOTE, are less than 10% in terms of Pass@1. We analyze the selected functions of baselines and find that baselines sometimes retrieve trivial examples. These functions sometimes might be noises for LLMs in generating programs.

Answer to RQ3: LAIL outperforms the SOTA baseline-TOP-k-SBERT by 10.04%, 8.12%, and 4.63% on the repository-level code generation in terms of Pass@1, 3, and 5. Our approach can effectively retrieve demonstration examples that LLMs need from a function pool of the repository.

6 Discussion

6.1 Transferability

We explore whether the retriever based on one LLM's feedback and specific dataset can be transferred to other LLMs or code generation datasets without further tuning. This is a significant research question since the retriever for each LLM and dataset needs to be trained in real applications. In the next, we verify the transferability of our approach from two aspects.

6.1.1 Transfer across LLMs. We consider transferring the retriever based on one LLM's feedback to another LLM. Specifically, we use a source LLM (e.g., CodeGen-Multi-16B or Text-davinci-003) to

Table 7. The Performances of LAIL's Transfer Abilities across Different LLMs on MBJP, MBPP, and MBCPP in Terms of Pass@k (i.e., k = 1, 3, and 5)

♣ GPT-3.5-turbo	MBJP			MBPP			MBCPP		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Zero-shot learning [8]	16.63	34.48	44.21	26.60	32.00	34.60	30.34	57.01	63.68
Random [11]	53.34	62.27	65.72	50.80	60.60	63.40	40.15	60.06	66.28
AceCoder [38]	54.46	64.01	66.75	54.40	62.40	65.20	42.53	62.29	68.32
TOP-k-SBERT [47]	54.26	63.89	66.53	54.80	63.00	65.40	43.37	61.45	67.11
TOP-k-GraphCodeBERT [28]	53.95	62.67	66.32	54.20	62.60	65.20	44.08	61.61	68.27
TOP-k-VOTE [64]	51.93	62.88	65.72	42.00	56.00	61.00	42.74	62.68	67.73
Uncertainty-Target [15]	49.69	60.45	54.50	36.40	51.80	56.80	42.46	61.84	66.67
	♠ CodeGen								
LAIL	**54.79 (↑ 0.61%)**	**64.70 (↑ 1.08%)**	**67.43 (↑ 1.02%)**	**55.60 (↑ 1.44%)**	**63.80 (↑ 1.27%)**	**66.20 (↑ 1.22%)**	**45.21 (↑ 2.56%)**	**63.45 (↑ 1.23%)**	**68.93 (↑ 0.89%)**
	♠ Text-davinci-003								
LAIL	**55.97 (↑ 2.77%)**	**64.97 (↑ 1.50%)**	**68.27 (↑ 2.28%)**	**56.20 (↑ 2.56%)**	**64.60 (↑ 2.54%)**	**66.80 (↑ 2.14%)**	**45.98 (↑ 4.31%)**	**63.84 (↑ 1.85%)**	**70.35 (↑ 2.97%)**
♣ Text-davinci-003	♠ CodeGen								
Random [11]	47.87	59.83	63.69	43.00	56.40	60.80	50.02	61.43	65.28
LAIL	**51.82 (↑ 8.25%)**	**61.75 (↑ 3.21%)**	**64.89 (↑ 1.88%)**	**48.20 (↑ 12.09%)**	**59.00 (↑ 4.61%)**	**64.20 (↑ 5.59%)**	**52.18 (↑ 4.32%)**	**64.54 (↑ 5.06%)**	**67.90 (↑ 4.01%)**
♣ CodeGen	♠ Text-davinci-003								
Random [11]	15.74	24.25	28.14	15.20	25.40	28.80	15.70	28.72	32.25
LAIL	**19.25 (↑ 22.30%)**	**27.53 (↑ 13.53%)**	**30.98 (↑ 10.09%)**	**17.40 (↑ 14.47%)**	**26.00 (↑ 2.36%)**	**30.00 (↑ 4.17%)**	**18.36 (↑ 16.94%)**	**30.41 (↑ 5.88%)**	**35.54 (↑ 10.20%)**

In this case, LAIL transfers a retriever learned on one LLM (the source LLM) to the other LLM (the target LLM). ♣ means the source LLM. ♠ indicates the target LLM. "↑" represents the relative improvements compared to the SOTA baseline. Numbers in bold mean the best performances among all approaches.

estimate examples for training a retriever and then apply the retriever to another target LLM (e.g., GPT-3.5-turbo) in generating programs. Table 7 shows the performance of GPT-3.5-turbo in the three datasets. We find that the retriever based on CodeGen-Multi-16B and Text-davinci-003 can bring obvious improvements to GPT-3.5-turbo. In particular, in terms of Pass@1, GPT-3.5-turbo achieves 2.56% improvements from CodeGen-Multi-16B's feedback and 4.31% enhancements from Text-davinci-003's feedback compared to the SOTA baseline. The phenomenons demonstrate that LAIL has satisfying transfer ability across different LLMs. Note that GPT-3.5-turbo cannot provide the prediction probability of ground truths in practice; thus, LAIL is a quite meaningful approach, especially for LLMs whose parameters are unavailable. Besides, the performance of GPT-3.5-turbo from Text-davinci-003's feedback is higher than the counterpart from CodeGen-Multi-16B's feedback. The reason might be that Text-davinci-003 and GPT-3.5-turbo have comparable abilities in code generation; thus, their preferences are similar and Text-davinci-003 can provide more proper examples as prompts to GPT-3.5-turbo.

To verify the transfer ability among LLMs with different sizes, we further evaluate the performance of CodeGen-Multi-16B based on the feedback of Text-davinci-003 and the results of Text-davinci-003 from CodeGen-Multi-16B's feedback, where the parameter size of CodeGen is much smaller than the counterpart of Text-davinci-003. As shown in Table 7, compared to the random approach, the retriever learned from CodeGen-Multi-16B achieves 8.25%, 12.09%, and 4.32% improvements to Text-davinci-003 on Pass@1, meanwhile, the retriever under Text-davinci-003's feedback brings 22.30%, 14.47%, and 16.94% improvements to CodeGen-Multi-16B, respectively. This further indicates that the retriever trained on the source LLMs can bring improvements to target LLMs, particularly for LLMs that have similar code generation abilities.

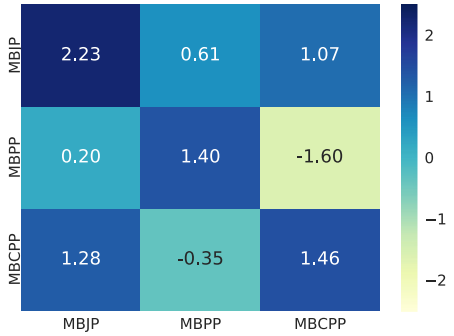

Fig. 5. Improvements of transferring the retriever trained on one dataset (row) to others (column) on Text-davinci-003 in the MBPP dataset. The number represents the absolute improvements from the best results of baselines to the transferring performance of LAIL.

6.1.2 Transfer across Datasets. Considering that the compositional features of natural language are general, the retriever trained on one dataset may apply to other datasets and exploit similar knowledge in different datasets. In this section, we further investigate whether a retriever trained on one dataset can transfer to others. We transfer the retriever among the three datasets (e.g., MBJP, MBPP, and MBCPP). Figure 5 demonstrates the transferring results on Text-davinci-003 in terms of Pass@1. The number in Figure 5 represents the absolute improvement from the best results among baselines to the transferring performance. We find that most retrievers can successfully transfer to other datasets and bring improvements compared to their SOTA baselines. Concretely, the retriever trained on MBJP (MBCPP) achieves 1.28% (1.07%) absolute improvements when it migrates to MBCPP (MBJP). On the contrary, the retriever optimized on MBCPP hardly transfers to MBPP, meanwhile the MBPP-based retriever suffers the generation performance on MBCPP. The reason might be that Java and C++ are object-oriented programming languages, and their syntax and code morphology are similar. Exploring a retriever that is suitable for many datasets is a challenging but meaningful research question, and we leave this topic as our future work.

6.2 Impacts of Demonstration Example Numbers

Most of the LLMs are trained with a limited input length, which restricts the number of examples in the prompt. Gao et al. [23] find that LLMs are affected by the number of demonstration examples in code-related tasks. Here, we explore the impacts of the number of examples in the prompt on baselines and LAIL. Figure 6 reports how the performance of LAIL and TOP-k-GraphCodeBERT change with respect to different demonstration example numbers on Text-davinci-003 at the MBPP dataset. We choose TOP-k-GraphCodeBERT to compare with our approach for two reasons. First, TOP-k-GraphCodeBERT is the best baseline in the MBPP dataset on the strictest metric Pass@1. Second, TOP-k-GraphCodeBERT uses GraphCodeBERT to select a few candidate examples. On the contrary, LAIL uses GraphCodeBERT to initialize the model-aware retriever and then trains the retriever with positive and negative examples estimated by LLMs themselves. Comparing LAIL to TOP-k-GraphCodeBERT, we can directly observe the improvements of LLMs caused by LAIL in code generation. From Figure 6, we observe that the performances of TOP-k-GraphCodeBERT and LAIL monotonically increase in rough with the increase of demonstration example numbers. In addition, LAIL always outperforms GraphCodeBERT in all cases, which further proves the superiority of LAIL even with different demonstration example numbers.

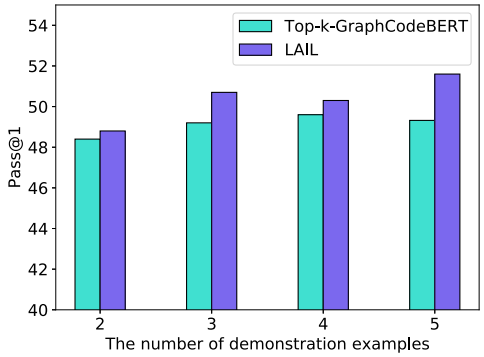

Fig. 6. The impacts of the numbers of demonstration examples on TOP-k-GraphCodeBERE and LAIL in the MBPP datasets. The base model is Text-davinci-003.

Table 8. Effects of the Number of Hard Negative Examples and Soft Negative Examples in Learning the Preference of LLMs on the MBPP Dataset

	Pass@1	Pass@3	Pass@5
Nums. of hard negative examples			
31	50.20	61.80	65.00
63	50.60	62.40	65.20
127	50.80	62.40	65.40
Nums. of soft negative examples			
1	50.60	62.40	65.20
5	50.20	62.20	65.00
10	49.40	61.80	64.40

"Num." Represents "Number."

6.3 Effects of Contrastive Learning Parameters

The number of negative examples is an important element in contrastive learning for training a neural retriever. As described in Section 3.2, the negative example set contains the soft negative example and the hard negative examples. We investigate how they affect the performance of our approach in the MBPP dataset, respectively. We use Text-davinci-003 as the base model. The results are shown in Table 8.

To investigate the effect of hard negative example numbers, we keep the number of soft negative examples unchanged. We can find that with the number of hard negative examples increasing, LAIL achieves consistent improvements and can generate more correct programs. For soft negative examples, we randomly select examples from the soft negative example set. We can find that the more soft negative examples, the worse our approach performs. We argue that the examples in the soft negative example set achieve high BM25 scores compared to the hard negative examples. Thus, the soft negative example set may contain some false-negative examples. Selecting more examples from the set will confuse the retriever in selecting the suitable demonstration examples that LLMs need. Between the two factors, the number of soft negative examples has a greater influence on the performance of LAIL.

6.4 Execution Time and Deployment of LAIL and Baselines

Different from existing works [38], LAIL first uses LLMs themselves to estimate candidate examples, then trains a model-aware retriever based on the labeled data, and applies the trained retriever to select a few demonstration examples. In this section, we analyze the execution time and ease of deployment of LAIL and compare them to the counterpart of baselines on the MBPP dataset based on CodeGen-Multi-16B.

Execution Time of LAIL and Baselines. For LAIL, the execution time contains three parts: (1) The process of estimating examples (Section 3.1). The process contains two stages. In the first stage, for an example $e_i = (x_i, y_i)$, LAIL uses BM25 to calculate the textual similarities between x_i and x_j, where $i \neq j$, and then filters hard negative examples through its score set $B_i = \{b_j\}_{j=1}^{N-1}$. The process costs 0.072 s to filter out hard negative candidates and acquires the set S_i. In the second stage, LAIL utilizes CodeGen-Multi-16B to measure the needs of LLMs for each example (x_q^i, y_q^i) in S_i, which takes 52.147 s. Thus, to label positive and negative candidates for a requirement, the total execution time is 52.219 s, including 0.072 s for filtering hard negative examples and 52.147 s for estimating examples with LLMs. (2) The process of training a model-aware retriever (Section 3.2). LAIL uses a small neural network as the retriever, which only has 125 M parameters and is initialized with GraphCodeBERT, instead of training the model from scratch. As described in Section 4.6, we train the retriever for about 1 hour. (3) The process of selecting a few demonstration examples (Section 3.3). In the process, the trained retriever encodes a new requirement and the requirement of candidate examples, then retrieves a few demonstration examples according to the cosine similarity of their representational vectors. The time of encoding requirements is minimal since the retriever only has 125 M parameters. Given a requirement, the retriever costs 0.023 s to encode the requirement and requirements of all candidate examples.

In LAIL, the processes of estimating examples and training a model-aware retriever only are executed one time, which can be done offline. Given a new requirement, LAIL only needs to complete the third process, i.e., applying the trained retriever to retrieve a few demonstration examples, which costs 0.023 s as described above. When it comes to baselines, the neural model-based approaches (i.e., TOP-k-GraphCodeBERT, TOP-k-SBERT, and TOP-k-VOTE) use neural networks to encode requirements and calculate the cosine similarity of requirements' vectors, which cost similar execution time to LAIL in retrieving demonstration examples. For the baseline AceCoder, it applies BM25 to calculate the textual similarities between a test requirement and the requirements of candidates and then retrieves a set of examples equipped with high similarities from the training set, whose execution time is 0.071 s. Uncertainty-Target estimates candidate examples with perplexity when LLMs generate ground truths. Given a new requirement, it needs to invoke LLMs N times where N is the number of examples in the candidate pool. The time cost of Uncertainty-Target is larger than LAIL because LAIL only inferences a smaller retriever N times to retrieve a few examples for a new requirement. For a fair comparison, the approaches that use neural networks to select candidate examples are executed on 4 NVIDIA A6000 GPUs with 48GB memory, including LAIL, TOP-k-SBERT, TOP-k-GraphCodeBERT, TOP-k-VOTE, and Uncertainty-Target. It is worth noting that the execution time of selecting demonstration examples in Uncertainty-Target is related to the base LLMs since the approach selects candidate examples based on the uncertainty of base LLMs in generating target programs. We do not consider the execution times of Uncertainty-Target on closed-source base LLMs, such as Text-davinci-003 and GPT-3.5-turbo, because we invoke them through APIs provided by OpenAI[3] and thus do not know the hardware for running them. In practice, LAIL does not require 4 NVIDIA A6000 GPUs since its retriever only has 125 M parameters. Considering that Uncertainty-Target needs 4 NVIDIA A6000 GPUs to select candidate examples,

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

Table 9. Execution Times of LAIL and All Baselines

Approach	Execution times (seconds)
Zero-shot [8]	–
Random [11]	–
TOP-k-SBERT [44]	0.018s
TOP-k-GraphCodeBERT [26]	0.023s
TOP-k-VOTE [61]	0.023s
LAIL	0.023 s
AceCoder [36]	0.071s
Uncertainty-Target [15]	52.147s

for a fair comparison, we also use the above hardware to analyze the execution time of LAIL and baselines. Although AceCoder uses BM25 to select examples, instead of neural networks, we also employ the same hardware to execute this approach. In addition, we run all experiments on the deep learning development framework—Pytorch[4] and a python package—transformers.[5] The used CPU is Intel(R) Xeon(R) Gold 6326 CPU with 2.90 GHz. To measure the execution time, we use the "times" package to calculate the time of different approaches. The reported execution time is the average over the test set of MBPP. In Table 9, we show the execution time of LAIL and baselines, which demonstrates the time of retrieving a few demonstration examples for a new requirement since estimating examples and training a retriever only need to be executed once in LAIL. We can find that for a requirement, the execution time of selecting demonstration examples in LAIL is less than the counterpart of Uncertainty-Target and AceCoder, meanwhile, its execution time is comparable to the time of other baselines.

Ease of Deployment of LAIL and Baselines. For a new requirement, LAIL uses a neural retriever to select a few demonstration examples from a candidate pool. In this article, the retriever wields GraphCodeBERT [28] as the backbone which only has 125 M parameters, so the demand for computing resources is acceptable for most developers. To apply LAIL, users only need to configure the relevant environment for running the retriever. Similarly, the baselines, such as TOP-k-SBERT, TOP-k-GraphCodeBERT, and TOP-k-VOTE, also use a neural network to retrieve candidate examples, where their parameter scale is similar to the retriever of LAIL. Developers are also required to install the environment to run the retriever. The heuristic baselines AceCoder uses BM25 to retrieve a few candidates, which is easy to employ. For Uncertainty-Target, users need to prepare more computing resources to deploy LLMs or invoke the API interface so deploying this baseline is harder than LAIL.

6.5 Effect of Different Filters in Filtering out Hard Negative Examples

As described in Section 3.1, for each programming task in the training set, before estimating candidate examples by LLMs, LAIL first uses BM25 to filter out hard negative examples. Although BM25 is easy to implement and is effective, it is based on the n-gram similarity, which is usually biased by the lexical features of requirements. Thus, it is necessary to investigate whether other filters can lead to better performance.

In this section, we consider two embedding-based filters to figure out hard negative examples, including GraphCodeBERT [28] and SBERT [47]. Concretely, the training set is $\mathcal{R} = \{e_i\}_{i=1}^{N}$,

[4]https://pytorch.org/.
[5]https://huggingface.co/docs/transformers/index.

Table 10. Effect of Different Filters in Filtering out Hard Negative Examples in MBJP on CodeGen-Multi-16B

MBJP	Pass@1	Pass@3	Pass@5
LAIL	21.30	28.49	32.05
$\text{LAIL}_{\text{GraphCodeBERT}}$	21.07	27.93	31.81
$\text{LAIL}_{\text{SBERT}}$	21.43	28.45	32.14

where e_i is the ith requirement-code example (x_i, y_i) in $\mathcal{R}$. For an example $e_i = (x_i, y_i)$, we use GraphCodeBERT to encode the requirement x_i and the requirement x_j of remaining candidates in $\mathcal{R}$, and then acquire their representational vectors $E^i_{[\text{CLS}]}$ and $E^j_{[\text{CLS}]}$, where $i \neq j$. Then, we calculate the cosine similarities between $E^i_{[\text{CLS}]}$ and $E^j_{[\text{CLS}]}$. Finally, we acquire the score set $B_i = \left\{b_j\right\}_{j=1}^{N-1}$ for e_i and filter out hard negative examples which have lower cosine similarities. Similarly, we utilize CodeBERT to encode requirements and apply the same process as GraphCodeBERT to filter out hard negative examples. Table 10 reports the performances of different filters in filtering out hard negative examples at MBJP when CodeGen-Multi-16B is the base model. $\text{LAIL}_{\textbf{GraphCodeBERT}}$ demonstrates that LAIL uses GraphCodeBERT as the filter, $\text{LAIL}_{\textbf{SBERT}}$ means SBERT is applied to filter out hard negative examples. We can find that when using the embedding-based filters to label hard negative examples, the performances of LAIL are comparable to the results of LAIL where BM25 is a filter. Considering that the simplicity of implementation, we use BM25 to filter out hard negative examples in Section 3.1.

6.6 Benefits of LAIL for Software Engineering (SE) Practitioners and Researchers

LAIL proposes an LLM-aware selection approach for ICL-based code generation. LAIL uses LLMs themselves to label demonstration examples as positive and negative examples. Based on the labeled positive and negative data, LAIL trains a model-aware retriever to select examples that are more helpful for LLMs to generate correct programs. During the inference, given a new requirement, LAIL directly uses the trained retriever to select a few demonstration examples as the prompt context for generating desired programs. As described above, LAIL proposes an example selection approach for ICL-based code generation, which is meaningful to SE practitioners and researchers.

Benefits of LAIL for SE Practitioners. First, LAIL can assist SE practitioners in improving the functional correctness of code snippets. Because LAIL aims to select a few demonstration examples that are more helpful for LLMs to generate correct programs, which has been demonstrated by our experiments using test cases to evaluate the functional correctness of generated programs. Second, LAIL can enhance the programming productivity of SE practitioners in the ICL-based code generation scenario. Because LAIL automates the process of selecting suitable demonstration examples, SE practitioners can save a lot of time and effort in manually selecting examples from a candidate pool. Third, LAIL can support different software development scenarios, which is flexible for SE practitioners to use. On the one hand, LAIL is effective in both function-level and repository-level code generation. On the other hand, as shown in experimental results, LAIL has a satisfying generalization ability. LAIL can be applied to various programming languages (e.g., Python, Java, and C++) and a series of LLMs (e.g., CodeGen-Multi-16B, CodeLlama-34B, Text-davinci-003, and GPT-3.5-turbo) that have different parameter sizes. LAIL is meaningful for SE practitioners in improving programming quality and productivity in their daily work.

Benefits of LAIL for Researchers. LAIL verifies the effectiveness of the model-aware example selection approach for ICL-based code generation, which can provide insights to researchers on

ICL. Researchers can further optimize this approach, including the manner to estimate candidate examples and the way to train the retriever except for using contrastive learning. In addition, LAIL further supports the opinion as demonstrated in existing works [16, 32], that is, the performance of ICL-based code generation heavily depends on the quality of selected examples. LAIL provides detailed experimental results and conducts extensive analyses, which can help researchers do more in-depth studies based on these results and analyses. Besides, LAIL can be applied to other tasks in SE, such as code translation and code summarization. Researchers can design suitable metrics to estimate candidate examples by LLMs' feedback and apply our model-aware example selection approach to different ICL-based tasks.

6.7 Threats to Validity

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

The Generalizability of Our Experimental Results. For the datasets, we select the datasets that contain mainstream programming languages. Following previous studies [3, 4, 11], we use three widely used datasets including MBPP [4], MBJP [3], and MBCPP [3]. The three datasets are collected from real-world software communities and cover Java, Python, and C++ languages. To verify the superiority of LAIL, we consider seven existing ICL approaches in both the code generation task and many maintain natural language tasks. In addition, to effectively evaluate our approach, we select a series of advanced pre-trained LLMs (e.g., CodeGen-Multi-16B [74], CodeLlama-34B [60], Text-davinci-003 [24], and GPT-3.5-turbo [9]) as base models. We apply our approach and baselines to base models and evaluate their performance in code generation. For the metric, following existing studies [10, 74], we select a widely used Pass@k metric to evaluate all approaches. It is an execution-based metric that utilizes test cases to check the correctness of generated programs. Besides, we manually measure generated programs in terms of their correctness, quality, and maintainability. To ensure fairness, we execute each method three times and report the average experimental results.

The Implementation of Models and Prompts. It is widely known that deep neural models are sensitive to the implementation details. In this article, we need to execute all baselines and our approach on the four base LLMs. For baselines, we apply the source code and parameters published by their original papers [28, 57, 74]. For base LLMs, the hyper-parameters of sampling will impact their outputs. In our experiments, we keep hyper-parameters the same for all approaches such as the temperature and the generation length. In addition, the performance of LLMs heavily depends on the prompts, including the instruction and the number of examples. To alleviate this threat, we leverage the same number of examples for all approaches and directly construct prompts without any natural language instructions. Besides, it is worth noting that the candidate pool for example selection also influences the ICL performance. Following previous studies [38], we use the training set of each dataset as the candidate set for all approaches. A large-scale study on 13.2 million real code files showed that the proportion of reused code is up to 80% [50], thus we believe even in practical applications, our approach can still suitable candidate examples to support LLMs in code generation. We do not tune the prompt and hyper-parameters experimentally and set them empirically. Thus, there might be room to tune the hyper-parameter settings of our approach for more improvements.

The Predicted Probability of Ground Truths Produced by LLMs. As described in Section 3.1, LAIL uses LLMs themselves to estimate demonstration examples. Because LLMs are not perfect, the prediction probability of the ground-truth program provided by LLMs may be biased. When terms to LLMs with relatively weak coding ability, such as CodeGen-Multi-16B used in this article, they may introduce a relatively larger bias when calculating the prediction probability of the ground-truth program in principle. For more powerful LLMs, when LLMs calculate the prediction probability of

the ground-truth program, the bias introduced by LLMs will be relatively small since these powerful LLMs have impressive coding ability. For example, the Pass@1 performance of CodeLlama-34B and Text-davinci-003 used in this article surpasses 45% in terms of Pass@1 on the MBJP dataset. Despite existing bias, the performance of LAIL still outperforms the SOTA baselines and LAIL effectively improves LLMs' coding ability. For example, even CodeGen-Multi-16B with relatively weak coding ability, LAIL achieves 11.58% improvements on Pass@1 in the MBJP dataset, which indicates that the bias introduced by LLMs themselves is acceptable to a certain extent. In the future, we will explore approaches to mitigate the bias introduced by LLMs' probability feedback.

7 Related Work

7.1 Code Generation

Code generation can automatically generate source codes given a requirement, which attracts more and more attention in industry and academia. Nowadays, a lot of approaches are proposed for code generation. According to the modeling architecture, existing code generation approaches can be mainly divided into the sequential modeling, tree modeling, graph modeling, and pre-trained model approaches.

Sequential Modeling. To generate programs, one straightforward solution is to employ the seq2seq framework, which directly maps a requirement to the code token sequence in an end-to-end manner. Researchers [51] first verify the plausibility of generating character-level code by the sequential model LSTM through empirical case studies. Later, latent prediction network [42] introduces a copy mechanism into the character-level RNN model. Iyer et al. [31] proposes an LSTM encoder–decoder model to generate Java token sequence from the requirement along with member variables and member method signatures under the same class. These simple but effective solutions prove the feasibility of code generation with sequential modeling. Furthermore, thanks to the powerful architecture and the abundant corpus, the transformer model, especially the pre-trained model, enables the outstanding proficiency of code generation [67]. Besides the end-to-end manner, some studies [19, 52] decompose the decoding process into multiple stages and decode from coarse (e.g., an abstracted sketch) to fine (e.g., the concrete code token sequence). The pre-trained model is one of the greatest inventions, therefore we introduce it separately in another paragraph, despite of its sequential modeling nature.

Tree Modeling. It is known that the source code snippet can be parsed into an **Abstract Syntax Tree (AST)**, and the two counterparts are equivalent. Therefore, there is a natural thought to generate the syntax tree first and then convert it back to code [49]. Most studies [32, 66, 71] employ top-down generation. Yin and Neubig [71] present TRANX, a transition-based neural semantic parser that maps a requirement into formal **Meaning Representations (MRs)**. TRANX uses a transition system based on the abstract syntax description language for the target MR and acquires desired programs. The study [32] proposes a context-based branch selector, which is able to dynamically determine optimal expansion orders of branches for multi-branch nodes. It optimizes the selector through reinforcement learning and formulates the reward function as the difference of model losses obtained through different expansion orders. Lately, TreeGen [66] uses the attention mechanism of Transformers to alleviate the long dependency problem and introduces a novel AST encoder to incorporate grammar rules and AST structures into the network. Different from the top-down generation, researchers [61] propose an alternative approach, a semi-autoregressive bottom-up parser, which allows to decoding of all sub-trees of a certain height in parallel, leading to logarithmic runtime complexity rather than linear. Compared with the token sequence, the syntax tree contains rich structural information, which may benefit the process of code generation.

Graph Modeling. The **Graph Neural Network (GNN)** [62] is a specialized type of deep learning model designed to process graph-structured data. GNN usually models the AST of the program with additional edges such as dataflow or control flow. One representative study [7] employs the gated GNN [41] to generate the code graph by sequentially adding new nodes and edges in a pre-determined order. Based on the generated graph, the ExprGen task is introduced to fill the statement in a given hollowed code snippet. However, due to the inherent challenges in graph modeling and the computational demands of GNNs, there are currently very few code generation methods utilizing graph modeling. The graph modeling approach presents significant challenges for code generation, and further research in this area is necessary.

Pre-Trained Model. Recently, numerous pre-trained models have been developed to automate the code generation process. These models are typically pre-trained on extensive corpora consisting of both natural language and programming language and are made available through parameters or APIs. According to the structure of models, they can be divided into encoder-only models, encoder–decoder models, and decoder-only models. ❶ Encoder-only models contain an encoder, which are usually trained with language comprehension tasks, e.g., masked language modeling or replaced token detection. CodeBERT [21] and CuBERT [34] are two pioneer studies. They apply BERT-style pre-training techniques to source codes. Later, researchers [28] further introduce the dataflow graph and propose GraphCodeBERT. GraphCodeBERT introduces two new pre-training objectives to learn the code structure. To support code generation, researchers usually add a randomly initialized transformer decoder along with encoder-only models. ❷ Encoder–decoder models are composed of an encoder and a decoder. An encoder takes a requirement as the input, and a decoder outputs a program. Many popular encoder–decoder architectures in natural language processing have been applied to the source code. For example, T5 model [56] is applied to support source code processing tasks and produces code-related models such as CodeT5 [10]. Similarly, researchers [1] apply a BART model to the source code and propose the PLBART. PLBART is pre-trained on an extensive collection of Java and Python functions and associated natural language text via denoising auto-encoding. These models on code corpus have been widely used and achieved impressive performances in code generation. ❸ Decoder-only models consist of a decoder, which are trained to predict the next token based on the input context. Inspired by GPTs' success in natural language generation [55], researchers also migrate this structure to programs. Researchers [46] first adapt GPT-2 trained on a large amount of code corpus, obtaining CodeGPT. Developers [11] continually fine-tune GPT-3 [8] on public code from GitHub and acquire Codex. Subsequently, a lot of LLMs are proposed for coding programs, which are introduced in Section 7.2.

7.2 LLMs

LLMs have shown impressive capabilities in code generation. They have billions of parameters and are trained on a large amount of corpus with different training objectives and then transferred to code generation. Inspired by the success of GPT series [54] in natural language processing, researchers attempt to adapt similar models to source code. Early, GPT-2 [55] model is adapted to the source code resulting in CodeGPT [46] that supports the code completion and the code generation tasks. CodeGPT is pre-trained on Python and Java corpora from the CodeSearchNet dataset [30], which includes 1.1 M Python functions and 1.6 M Java methods. Lately, GPT-3 [8] model is fine-tuned on code corpus to produce CodeX [11]. It is fine-tuned on publicly available 159GB code from GitHub and has code-writing capabilities. A distinct production version of Codex powers GitHub Copilot [16]. Meanwhile, two similar works in spirit to CodeX are GPT-Neo [6] and GPT-J [27]. They are trained on the Pile dataset [22] that contains text from a variety of sources as well as 8% Github code. Since the parameters of CodeX are not available, many researchers try to replicate them and bring CodeParrot [14], GPT-CC [26], CodeGen [74], and PyCodeGPT [72]. PyCodeGPT [72] is a

110 M parameters model based on GPT-Neo, which is pre-trained on 13 M high-quality Python files with a size of 96GB. It has the ability to generate good standalone Python code. CodeGen [74] is a family of LLMs up to 16B parameters. It is in the form of autoregressive transformers with the next-token prediction objective trained on natural language and programming language data including the Pile [22], BigQuery, and BigPython datasets. Recently, OpenAI[6] proposes GPT-3.5 [24] and GPT-4 [25] series models (e.g., ChatGPT [9]), which have shown strong generation capabilities in natural language and programming languages. Since neither GPT-3.5 [24] nor GPT-4 [25] is open sourced, some researches StarCoder [39], WizardCoder [48], and CodeLLaMA [60]. WizardCoder [48] empowers Code LLMs with complex instruction fine-tuning, by adapting the Evol-Instruct method to the domain of code. CodeLlama [60] is a family of LLMs for code based on Llama 2 [69]. It is trained on sequences of 16k tokens and shows improvements on inputs with up to 100k tokens, supporting large input contexts. These LLMs effectively evolve the code generation community. In this article, we evaluate LAIL on diverse LLMs.

7.3 ICL

ICL is an emerging approach to using LLMs. By providing limited examples as a prompt, ICL empowers LLMs to learn a specific downstream task. ICL [8] is first proposed in natural language processing and achieves impressive performance in many tasks [17, 18, 20, 35, 68], such as text generation and time series forecasting.

Inspired by the success of ICL in natural language processing, researchers attempt to adapt ICL to source code [5, 11, 13]. They design task-specific instructions with a set of examples to prompt LLMs and improve the performance on many tasks (e.g., code generation [13, 43, 65, 73] and code clone detection [37]). Meanwhile, ICL introduces instability in performance: given different sets of examples as prompts, LLMs' performance usually varies from random to near SOTA [23, 44]. Some studies [13, 38] make efforts to alleviate this issue. Gao et al. [23] study the effects of in-context examples for code-related tasks, including the number and the sequence of selected examples. Chen et al. [13] randomly select limited examples for ICL and verify the results in code generation. Lately, AceCoder [38] uses BM25 to select n-gram matching examples and further generates more correct programs. However, these approaches are based on lexical features, which require developers to design heuristics and ignore the needs of LLMs. Instead of considering heuristic approaches, LAIL uses LLMs themselves to measure candidate examples and trains a neural retriever to learn the preferences of LLMs in code generation.

8 Conclusion

Due to the impact of demonstration examples, there is an increasing demand for selecting proper demonstration examples in ICL-based code generation. This article proposes an LLM-aware selection approach for ICL-based code generation named LAIL. It requires LLMs themselves to select demonstration examples. LLMs depending on their needs label a candidate example as a positive example or a negative example for a new requirement. Based on labeled data, it optimizes a model-aware retriever to learn the preference of LLMs. During the inference, LAIL uses the retriever to select examples for ICL-based code generation. Experimental results on five code generation datasets demonstrate that LAIL achieves the best performance among all approaches. LAIL also shows satisfactory transferability across LLMs and datasets, showing that using LLMs themselves to select examples is an efficient and practical approach to code generation.

[6] https://openai.com/.

References

[1] 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*, 2655–2668.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys* 51, 4 (2018), 1–37.

[3] 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:2210.14868. Retrieved from https://arxiv.org/abs/2210.14868

[4] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang,Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. arXiv:2108.07732. Retrieved from https://arxiv.org/abs/2108.07732

[5] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code generation tools (almost) for free? A study of few-shot, pre-trained language models on code. arXiV:2206.01335. Retrieved from https://arxiv.org/abs/2206.01335

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards,Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

[7] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2018. Generative code modeling with graphs. arXiv:1805.08490. Retrieved from https://arxiv.org/abs/1805.08490

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

[9] ChatGPT. 2022. Retrieved from https://platform.openai.com/docs/models/gpt-3-5

[10] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code generation with generated tests. arXiv:2207.10397. Retrieved from https://arxiv.org/abs/2207.10397

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

[12] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *Proceedings of the International Conference on Machine Learning*. PMLR, 1597–1607.

[13] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. arXiv:2304.05128. Retrieved from https://arxiv.org/abs/2304.05128

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

[15] Cody Coleman, Christopher Yeh, Stephen Mussmann, Baharan Mirzasoleiman, Peter Bailis, Percy Liang, Jure Leskovec, and Matei Zaharia. 2019. Selection via proxy: Efficient data selection for deep learning. arXiv:1906.11829. Retrieved from https://arxiv.org/abs/1906.11829

[16] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software* 203 (2023), 111734.

[17] Jinliang Deng, Xiusi Chen, Renhe Jiang, Xuan Song, and Ivor W. Tsang. 2022. A multi-view multi-task learning framework for multi-variate time series forecasting. *IEEE Transactions on Knowledge and Data Engineering* 35, 8 (2022), 7665–7680.

[18] Jinliang Deng, Xiusi Chen, Renhe Jiang, Du Yin, Yi Yang, Xuan Song, and Ivor W. Tsang. 2024. Disentangling structured components: Towards adaptive, interpretable and scalable time series forecasting. *IEEE Transactions on Knowledge and Data Engineering* 36, 8 (2024), 3783–3800.

[19] Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. arXiv:1805.04793. Retrieved from http://arxiv.org/abs/1805.04793

[20] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. 2023. A survey for in-context learning. arXiv:2301.00234. Retrieved from https://arxiv.org/abs/2301.00234

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. arXiv:2002.08155. Retrieved from https://arxiv.org/abs/2002.08155

[22] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2021. The Pile: An 800GB dataset of diverse text for language modeling. arXiv:2101.00027. Retrieved from https://arxiv.org/abs/2101.00027

[23] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, and Michael R. Lyu. 2023. Constructing effective in-context demonstration for code intelligence tasks: An empirical study. arXiv:2304.07575. Retrieved from https://arxiv.org/abs/2304.07575
[24] GPT-3.5. 2022. Retrieved from https://platform.openai.com/docs/deprecations
[25] GPT-4. 2023. Retrieved from https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo
[26] GPT-CC. 2022. Retrieved from https://github.com/CodedotAl/gpt-code-clippy
[27] GPT-J. 2023. Retrieved from https://github.com/kingoflolz/mesh-transformer-jax
[28] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training code representations with data flow. arXiv:2009.08366. Retrieved from https://arxiv.org/abs/2009.08366
[29] Ari Holtzman, Jan Buys, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. arXiv:1904.09751. Retrieved from http://arxiv.org/abs/1904.09751
[30] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNetChallenge: Evaluating the state of semantic code search. arXiv:1909.09436. Retrieved from http://arxiv.org/abs/1909.09436
[31] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. arXiv:1808.09588. Retrieved from http://arxiv.org/abs/1808.09588
[32] Hui Jiang, Chulun Zhou, Fandong Meng, Biao Zhang, Jie Zhou, Degen Huang, Qingqiang Wu, and Jinsong Su. 2021. Exploring dynamic selection of branch expansion orders for code generation. arXiv:2106.00261. Retrieved from https://arxiv.org/abs/2106.00261
[33] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language model. arXiv:2303.06689. Retrieved from https://arxiv.org/abs/2303.06689
[34] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the International Conference on Machine Learning*. PMLR, 5110–5121.
[35] Itay Levy, Ben Bogin, and Jonathan Berant. 2022. Diverse demonstrations improve in-context compositional generalization. arXiv:2212.06800. Retrieved from https://arxiv.org/abs/2212.06800
[36] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, et al. 2024. DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories. arXiv:2405.19856. Retrieved from https://arxiv.org/abs/2405.19856
[37] Jia Li, Chongyang Tao, Zhi Jin, Fang Liu, Jia Li, and Ge Li. 2023. ZC^3: Zero-shot cross-language code clone detection. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*. IEEE, 875–887. DOI: https://doi.org/10.1109/ASE56229.2023.00210.
[38] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2024. AceCoder: An effective prompting technique specialized in code generation. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 204:1–204:26. DOI: https://doi.org/10.1145/3675395
[39] 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:2305.06161. Retrieved from https://arxiv.org/abs/.2305.06161
[40] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.
[41] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated graph sequence neural networks. In *Proceedings of the 4th International Conference on Learning Representations (ICLR '16)*, ConferenceTrack Proceedings. Yoshua Bengio and Yann LeCun (Eds.). Retrieved from http://arxiv.org/abs/1511.05493
[42] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Andrew W. Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. arXiv:1603.06744. Retrieved from http://arxiv.org/abs/1603.06744
[43] Fang Liu, Jia Li, and Li Zhang. 2023. Syntax and domain aware model for unsupervised program translation. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE '23)*. IEEE, 755–767. DOI: https://doi.org/10.1109/ICSE48619.2023.00072
[44] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2021. What makes good in-context examples for GPT-3? arXiv:2101.06804. Retrieved from https://arxiv.org/abs/2101.06804
[45] Llama. 2023. https://huggingface.co/meta-llama/Llama-2-70b-chat
[46] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664. Retrieved from https://arxiv.org/abs/2102.04664
[47] Man Luo, Xin Xu, Zhuyun Dai, Panupong Pasupat, Seyed Mehran Kazemi, Chitta Baral, Vaiva Imbrasaite, and Vincent Y. Zhao. 2023. Dr.ICL: Demonstration-retrieved in-context learning. arXiv:2305.14128. Retrieved from https://arxiv.org/abs/2305.14128

[48] 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/abs/2306.08568

[49] Chris Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *Proceedings of the International Conference on Machine Learning*. PMLR, 649–657.

[50] 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.

[51] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. 2015. On end-to-end program generation from user intention by deep neural networks. arXiv:1510.07211. Retrieved from http://arxiv.org/abs/1510.07211

[52] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural sketch learning for conditional program generation. In *Proceedings of 6th International Conference on Learning Representations (ICLR '18)*, Conference Track Proceedings. Retrieved from https://openreview.net/forum?id=HkfXMz-Ab

[53] 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.

[54] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever. 2018. Improving Language Understanding by Generative Pre-Training. Retrieved from https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[55] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019), 9.

[56] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[57] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-Networks. arXiv:1908.10084. Retrieved from https://arxiv.org/abs/1908.10084

[58] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco,and Shuai Ma. 2020. CodeBLEU: A method for automatic evaluation of code synthesis. arXiv:2009.10297. Retrieved from https://arxiv.org/abs/2009.10297

[59] Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval* 3, 4 (2009), 333–389.

[60] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open foundation models for code. arXiv:2308.12950. Retrieved from https://arxiv.org/abs/2308.12950

[61] Ohad Rubin and Jonathan Berant. 2020. SmBoP: Semi-autoregressive bottom-up semantic parsing. arXiv:2010.12412 Retrieved from https://arxiv.org/abs/2010.12412

[62] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE Transactions on Neural Networks* 20, 1 (2008), 61–80.

[63] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. arXiv:1508.07909. Retrieved from https://arxiv.org/abs/1508.07909

[64] Hongjin Su, Jungo Kasai, Chen Henry Wu, Weijia Shi, Tianlu Wang, Jiayi Xin, Rui Zhang, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, et al. 2022. Selective annotation makes language models better few-shot learners. arXiv:2209.01975. Retrieved from https://arxiv.org/abs/2209.01975

[65] Zhihong Sun, Yao Wan, Jia Li, Hongyu Zhang, Zhi Jin, Ge Li, and Chen Lyu. 2024. Sifting through the chaff: On utilizing execution feedback for ranking the generated code candidates. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. ACM, 229–241. DOI: https://doi.org/10.1145/3691620.3695000.

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

[67] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1433–1443.

[68] Zhengwei Tao, Zhi Jin, Yifan Zhang, Xiancai Chen, Xiaoying Bai, Yue Fang, Haiyan Zhao, Jia Li, and Chongyang Tao. 2024. A comprehensive evaluation on event reasoning of large language models. arXiv:2404.17513. Retrieved from https://arxiv.org/abs/2404.17513

[69] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288. Retrieved from https://arxiv.org/abs/2307.09288

[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* 30 (2017).

[71] Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. arXiv:1810.02720. Retrieved from http://arxiv.org/abs/1810.02720

[72] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: Continual pre-training on sketches for library-oriented code generation. arXiv:2206.06888. Retrieved from https://arxiv.org/abs/2206.06888

[73] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL '24)*. Association for Computational Linguistics, 13643–13658. DOI: https://doi.org/10.18653/V1/2024.ACL-LONG.737

[74] Maosheng Zhong, Gen Liu, Hongwei Li, Jiangling Kuang, Jinshan Zeng, and Mingwen Wang. 2022. CodeGen-Test: An automatic code generation model integrating program test information. arXiv:2202.07612. Retrieved from https://arxiv.org/abs/2202.07612

A Appendix

A.1 Human Evaluation

The goal of a code generation approach is to assist developers in writing programs. Thus, a good program not only satisfies the requirement but also is easy to read and maintain. Although the Pass@k results are reported in RQ1 of Section 5, the test cases contained in datasets cannot achieve 100% coverage, such as whether test cases consider special input values. It is necessary to manually assess generated programs. In this section, we manually measure generated programs in three aspects (e.g., correctness, code quality, and maintainability) following previous works [38].

Setup. We randomly select 50 test examples from MBJP, MBPP, and MBCPP, respectively. Then, we use Text-davinci-003 to generate programs with different ICL approaches. 1,200 ($50 \times 3 \times 8$) programs for human evaluation, where 3 is the number of datasets and 8 means the number of all selection approaches. Finally, we obtain 1,200 ($50 \times 3 \times 8$) programs for human evaluation, where 3 is the number of datasets and 8 means the number of all selection approaches. We randomly divide test examples into 5 groups and each group contains 240 ($10 \times 3 \times 8$) programs, where 10 denotes the number of problems for each group. For each aspect, the score is an integer and ranges from 0 to 2 considering the balance of evaluation quality and scoring time. The higher the score is, the better the program is. ❶ Correctness measures whether the generated program satisfies the requirement. If the generated program completely meets the requirement, the score is 2. The score is 1 if a generated program only satisfies a portion of a requirement. The score is set to 0 in other situations. ❷ Code quality verifies whether predicted programs contain bad code smell. The bad code smell usually indicates potential issues or poor design choices in the code. These issues might not cause errors but can lead to difficulties in reading and understanding the code. If the generated program is clear, concise, and easy to understand, it is scored to 2. Scorers give 1 point for the program where some parts of the code snippet are better organized and minor redundancies are present. If the code is hard to understand or follow due to complexity or lack of clarity, it is scored as 0. ❸ Maintainability refers to the ease with which a program can be modified, extended, and maintained over its lifecycle. The score is 2 if a program is well organized and consistent in programming criterion and style, facilitating easy maintenance and modification. If the program lacks maintainability and consistency, scorers give it 1 point. If the generated program is challenging to maintain or modify, it is scored to 0.

We publicly recruit 10 participants to verify these programs, which contain 5 computer science students and 5 industrial practitioners. The participants must be experts in computer science and have at least 3 years of programming experience in Java, Python, and C++ used in our article. To recruit students, we post posters in the building of the computer science college at our school and invite the first five students who meet the above requirements based on the order of registration. Finally, the student participants include four PhDs and one master. On average, they write more

than 200 line programs every day, containing their coursework, projects, and research experiments. To recruit industrial practitioners, we make a public announcement in WeChat groups of an internet company. The first five people meeting the requirements are invited. In the recruitment process, the requirements for participants and scoring criteria are always available. Finally, the hired industrial practitioners have rich experience in project development. Both students and industrial practitioners have more than 3 years of programming experience in Java, Python, and C++. These participants are randomly split into five groups, where each group contains a student and an industrial practitioner. Before scoring the generated programs, we conduct uniform training for participants to ensure that they have a consistent understanding in the scoring criteria as much as possible. Participants must spend no less than 2 minutes reviewing each program and then give the corresponding score based on the scoring criterion. Meanwhile, we ask participants not to consider their programming styles, doing our best to ensure the fairness of human scores.

To assess generated programs, we put the generated programs of each approach into a file with the *json* format. In the file, each example is stored as a dictionary, containing a requirement and the generated program for the requirement. To ensure fairness, participants do not know the corresponding approach of each file. Each group receives a compressed package that includes eight *json* files corresponding to the generated results of eight different approaches and documentation about scoring standards. Each file contains $10 \times 8 \times 3$ examples to be scored and is measured by two participants of a group, where 3 is the number of datasets. Considering that two participants in each group may have conflicts, we design different strategies to calculate the final score of a generated program. Concretely, for a generated program, when the absolute difference of two scores provided by two participants is greater than 1 (i.e., one person scores 0 and the other person scores 2), we randomly invite one participant from other groups to further assess the program, aiming to reduce the negative effects of scoring conflicts. If the third participant scores 1, the final score of the program is the average score of the three people. If the third participant scores 2 (or 0), we abandon the existing score 0 (or 2) provided by the original scorers and calculate the average of the remaining two scores. If the absolute difference between two scores provided by two participants is no more than 1, we calculate the average of the two participants' scores directly.

Results. The results of the human evaluation are shown in Table A1. The percentages in parentheses represent the improvements from the best results of baselines to LAIL. We apply T-tests to complete the statistical tests and use Cohen's d as effect size. The p-value of human evaluation on correctness, code quality, and maintainability is smaller than 0.05, and the values of Cohen's d on the three aspects are all larger than 0.65. The detailed statistic test results are shown in Table A2.

Analyses. (1) *Developers prefer programs generated by LAIL over all baselines.* LAIL substantially outperforms all baselines in three aspects, which proves that generated programs by LAIL not only satisfy more requirements but also are more human-preferred. (2) *In addition to satisfying requirements, programs provided by our approach are easier to read and have less bad smell.* Particularly, LAIL surpasses the SOTA approach by 5.862% in code quality and 6.714% in maintainability. (3) *Some heuristic approaches that select the same examples as prompts for all test requirements are not optimal for ICL.* As shown in Table A1, Uncertainty-Target and TOP-k-VOTE perform comparably to the random approach, which indicates that applying the same prompt to all requirements in the test data are not a good choice. For different test requirements, we should use their appropriate prompts to help LLMs for code generation.

Table A1. The Results of Human Evaluation on the Randomly Sampled Examples of the Three Datasets in Aspects of Correctness, Code Quality, and Maintainability When Text-davinci-003 is as the Baseline

Approach	Correctness	Code Quality	Maintainability
Zero-shot learning [8]	0.472	1.098	1.195
Random [11]	0.925	1.325	1.520
AceCoder [38]	1.523	1.657	1.653
TOP-k-SBERT [47]	1.569	1.643	1.668
TOP-k-GraphCodeBERT [28]	1.582	1.740	1.609
TOP-k-VOTE [64]	1.325	1.615	1.624
Uncertainty-Target [15]	0.736	1.306	1.335
LAIL	**1.764 (↑ 11.504%)**	**1.842 (↑ 5.862%)**	**1.780 (↑ 6.714%)**

The numbers in bold represent the best performances among LAIL and baselines.

Table A2. Statistical Test Results of Human Evaluation on Text-davinci-003

	Correctness	Code Quality	Maintainability
p-value	0.033	0.045	0.037
Cohen's d	0.82	0.67	0.79

Cohen's d reflects effect size.

Received 21 January 2024; revised 23 November 2024; accepted 2 January 2025