On Sample-Efficient Code Generation

Hojae Han♠◇, Yu Jin Kim▷, Byoungjip Kim▷, Youngwon Lee♠◇, Kyungjae Lee▷,
Kyungmin Lee▷, Moontae Lee▷[*], Kyunghoon Bae▷, Seung-won Hwang♠◇[†]
♠Seoul National University, ◇SNU-LG AI Research Center,
{stovecat, ludaya, seungwonh}@snu.ac.kr
▷LG AI Research, {yujin.kim, bjkim, kyungjae.lee,
kyungmin.lee, moontae.lee, k.bae}@lgresearch.ai

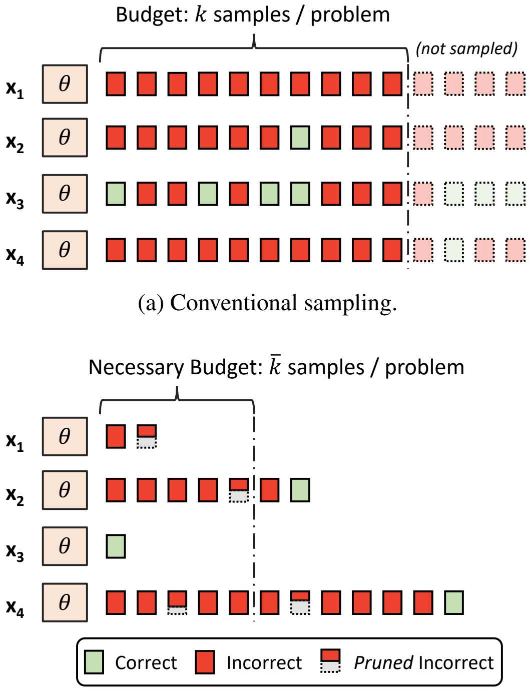
Abstract

Large language models often struggle to predict runtime behavior in code generation tasks, leading to a reliance on rejection sampling (best-of-n) to generate multiple code snippets then select the best. Our distinction is reducing sampling costs, without compromising generation quality. We introduce EFFICODE, a novel framework that prioritizes sampling on test problems that models can solve. We show how EFFICODE estimates solvability to optimize computational costs during multiple sampling. Based on empirical evidence, EFFICODE consistently demonstrates reduced sampling budgets while maintaining comparable code generation performance, especially when problems are challenging. In addition, utilizing EFFICODE to rank sampled code snippets also shows its effectiveness in answer code selection for reducing temporal costs, by not requiring any execution or test case generation.

1 Introduction

Recently, large language models (LLMs) have achieved success in code generation, aiming at synthesizing a functionally correct program based on a natural language problem description (Chen et al., 2021; Li et al., 2022). Ensuring functional correctness is a rigorous objective, as a single token error during generation can render the entire output incorrect, while some grammatical and semantic errors in natural language are tolerable to human readers.

To achieve rigor despite noise during generation, existing approaches utilize rejection sampling (multiple sampling then selecting the best) to increase the likelihood of finding a correct code among the candidates (Li et al., 2022; Shi et al., 2022; Inala et al., 2022; Chen et al., 2023). In this context, the widely used metric is Pass@k (Chen et al., 2021), which assigns a score of 1 if at least one of the k sampled candidates is correct, and 0 if all candidates are incorrect.


Budget: k samples / problem
(not sampled)
x_1
x_2
x_3
x_4
θ
Necessary Budget: $\bar{k}$ samples / problem
Correct
Incorrect
Pruned Incorrect

(a) Conventional sampling.

(b) EFFICODE (this work).

Figure 1: Comparison of EFFICODE to conventional multiple sampling. The solid and the dashed line boxes indicate the sampled code and code to be sampled for each problem x_i by the code generation model θ.

However, the use of multiple sampling in code generation incurs high computational costs. While considerable efforts have been made to optimize the computational expense of pre-training, including addressing its environmental impact (Strubell et al., 2019), resource-intensive inference from excessive sampling have been largely overlooked. To motivate, AlphaCode (Li et al., 2022) generates 1 million code samples for each competition-level problem, resulting in hundreds of petaFLOPS days of computation—equivalent to the cost of pre-training the model.

[*] is also affiliated with the University of Illinois Chicago.
[†] Corresponding author.

Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track, pages 783–791
December 6-10, 2023 ©2023 Association for Computational Linguistics

In addition, recent approaches (Chen et al., 2023; Shinn et al., 2023; Zhang et al., 2023) self-validate the sampled code, by executing them with (generated) test cases. This results in a significant response time overhead, as the samples are often inefficiently implemented and may cause time-outs that take several seconds (Zhang et al., 2023). Thus, there is a critical need to reduce the expense of multiple sampling and refining, for deploying an industry-scale code generation with efficiency and sustainability.

In our research, we aim to minimize the computational and temporal costs in code generation by reducing the sample size and avoiding execution in validation without compromising accuracy. Figure 1 illustrates the contrast between conventional sampling with a uniform sampling cost of k and our proposed approach, referred to as EFFICODE, which prioritizes the necessary $\overline{k}$ samples on average ($\overline{k} < k$) with the following characteristics:

First, **necessary**: we prioritize investing the sampling budget in solving simple problems that terminate early (e.g., x_3 in Figure 1a), or avoiding wasting resources on hard problems that never terminate (e.g., x_1 in Figure 1a), unlike conventional sampling investing equally to all problems. To achieve this, we propose a solvability estimator, which determines if a problem is likely to be solvable based on either 1) producing fewer errors, or 2) close to the problems successfully solved in the past.

Second, **adaptive**: we can assess the correctness of a partially decoded sample even before its completion. In contrast, conventional sampling continues decoding until their completion without verification. This adaptability allows us to make more informed decisions during the decoding process to potentially save computational resources by terminating the decoding early.

In our main experiment, we validate the effectiveness of EFFICODE in improving the sample efficiency of GPT-3.5 (OpenAI, 2022) on CodeContests (Li et al., 2022), HumanEval (Chen et al., 2021), and MBPP (Austin et al., 2021) benchmarks. In addition, we empirically confirm the effectiveness of using EFFICODE as a ranker to select correct code for reducing temporal costs, without requiring any code execution.

In summary, the key contributions of this study are as follows:

- We propose a novel framework, called EFFICODE, which significantly enhances the sample efficiency of code generation models by leveraging solvability estimation, allowing for more effective allocation of sampling budget.
- Our method dynamically adapts to the correctness of partially decoded samples, early terminating wasteful computation on completing unnecessary decoding.
- We empirically validate the improved sample efficiency on various benchmarks. The experimental results provide evidence of the benefits of our approach in practical scenarios.

2 Preliminaries

In this section, we define code generation task and its characteristic of requiring multiple sampling. Next, we explain our research goal, sample efficiency.

Code Generation. Given a set of problems X and a code generation model θ, code generation is the task of synthesizing a correct solution code for each problem $x_i \in X$:

$$c^* = \arg\max_{c \in \mathcal{C}} f(c, x_i), \tag{1}$$

where $\mathcal{C}$ is the set of every possible code that θ can generate. Ideally, $f(c, x_i)$ is calculated by executing the generated code c with test cases for the problem x_i, returning 1 if it passes all test runs and 0 for else. Generally, test cases are unavailable during inference time (Chen et al., 2023; Shinn et al., 2023).

Sampling Multiple Candidates. One of the distinctions of code generation from natural language generation is its rigor; even a single mistakenly generated token can cause the entire code to be incorrect. To compensate this, code generation usually samples a set of multiple candidates C_i to solve each problem $x_i \in X$ (Chen et al., 2021; Li et al., 2022). Code generation passes, when there exists $c \in C_i$ such that $f(c, x_i) = 1$, denote as $F(C_i) = 1$, and fails otherwise, or, $F(C_i) = 0$.

Sample Efficiency. Our objective is to maximize the pass rate of code generation:

$$\frac{\sum_{i=1}^{|X|} F(C_i)}{|X|}, \tag{2}$$

while ensuring sample efficiency by constraining that the total cost of sampling (e.g. the number of generated code samples) should not exceed a total sampling budget B:

$$\sum_{i=1}^{|X|} |C_i| \leqslant B. \tag{3}$$

3 Related Work

3.1 Code Generation with LLMs

Recent work has shown that LLMs trained on source code corpus can synthesize correct code by given natural language descriptions. Early approaches like GPT-NEO (Black et al., 2021) and GPT-J (Wang and Komatsuzaki, 2021) add code data into pre-training corpus. Later, CODEX (Chen et al., 2021), which has Code-davinci-002 as its variation, targets to code generation solely, by first pre-trained on text then further pre-trained on code only corpus. AlphaCode (Li et al., 2022) shows an average human programmer performance in competition-level code generation. Several approaches like CODEGEN (Nijkamp et al., 2023), CODET5 (Wang et al., 2021), CODET5+ (Wang et al., 2023), SANTACODER (Allal et al., 2023), and STARCODER (Li et al., 2023) reveal publicly available LLMs for code generation. CODERL (Le et al., 2022) further improves CODET5 by applying reinforcement learning and critic sampling. Recently, GPT-3.5 (Ouyang et al., 2022) and GPT-4 (OpenAI, 2023) show remarkable performance improvement by reinforcement learning from human feedback (RLHF), and phi-1 (Gunasekar et al., 2023) organizes high quality dataset to significantly improve the performance while keeping the model size as 1.3B.

Our distinction. EFFICODE is a model-agnostic framework that can be employed to improve sample efficiency across LLMs.

3.2 Sample Efficiency on Code Generation

To ensure the correctness of generation, existing approaches aim to over-generate then filter incorrect ones. CODERANKER (Inala et al., 2022) proposes a ranker, trained to distinguish between correct and incorrect code, as well as classify the error types in the incorrect code. Alternatively to a trained ranker, later approaches filter out incorrect code through code execution. AlphaCode (Li et al., 2022) generates test inputs for each problem, clusters the code samples by the outputs from generated inputs, and randomly selects code samples from the biggest cluster to smaller ones. CODET (Chen et al., 2023) synthesizes test cases, and mutually verifies the code candidates and the generated test cases, to filter incorrect ones out. Lastly, one may consider generating then fixing towards correctness. ALGO (Zhang et al., 2023) uses exhaustively searched reference oracle code to verify and refine code candidates. REFLEXION (Shinn et al., 2023) generates test cases then conducts iterative self-verification and refinement over the generated code, regarding the final version as the most correct one.

As an alternative to execution-based correctness evaluation, the similarity of generated code to human annotated reference can be used. CodeBLEU (Ren et al., 2020) employs abstract syntax trees (AST) to capture code syntax and data-flow to quantify similarity.

Our distinction. All three categories require additional model inference, and generally need code execution using (annotated or synthetic) test cases. In contrast, EFFICODE tackles sample efficiency without requiring additional inference or code execution, reducing both computational and temporal costs. Specifically, we repurpose CodeBLEU, from its original use of evaluation, to measure code similarity between generated code and past solutions to estimate generation correctness.

4 EFFICODE

EFFICODE is a novel framework that aims to achieve sample-efficient code generation by estimating the solvability for each problem, then prioritizing the problems to allocate the sampling budget.

4.1 Code Sampling as Discrete Search

We want to estimate the sampling priority among problems. We explain the paradigm of sampling multiple code samples per problem as discrete search, analogous to regarding decoding a text sequence as discrete search (Lu et al., 2022). Specifically, we define a state as $s_t = [C_t^1, C_t^2, ..., C_t^{|X|}]$ where C_t^i is the set of sampled code for each problem $x_i \in X$ until a time step t. An action $a_t \in \mathcal{A}(s_t)$ from an action space $\mathcal{A}$ in s_t means to sample more candidates for $x_{m(a_t)}$ where $m(a_t)$ is an indexing function.[1] The transition of each step of sampling consists of 1) selecting a problem,

[1] For example, if $m(a_t) = 3$, then x_3 is selected to sample more code in time step t.

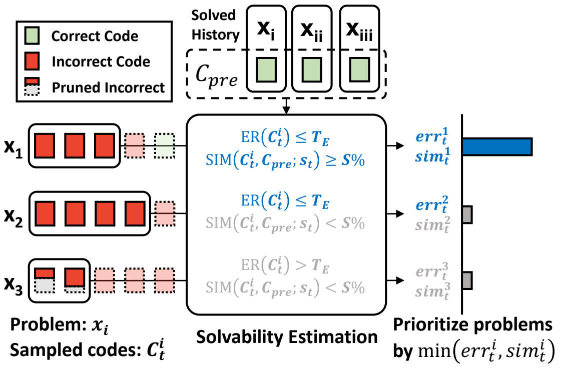

Correct Code
Incorrect Code
Pruned Incorrect
Solved History
x_i
x_{ii}
x_{iii}
C_{pre}
x_1
x_2
x_3
$\mathrm{ER}(C_t^i) \leq T_E$
$\mathrm{SIM}(C_t^i, C_{pre}; s_t) \geq S\%$
$\mathrm{ER}(C_t^i) \leq T_E$
$\mathrm{SIM}(C_t^i, C_{pre}; s_t) < S\%$
$\mathrm{ER}(C_t^i) > T_E$
$\mathrm{SIM}(C_t^i, C_{pre}; s_t) < S\%$
err_t^1 sim_t^1
err_t^2 sim_t^2
err_t^3 sim_t^3
Problem: x_i
Sampled codes: C_t^i
Solvability Estimation
Prioritize problems by $\min(err_t^i, sim_t^i)$

Figure 2: Solvability estimation by EFFICODE. A problem x_i is assigned a high priority if C_t^i, the set of sampled code so far, has little syntax errors ($err_t^i = h$ in Eq (6)) and also exhibits high similarity with C_{pre}, the code for problems that are already solved by the model ($sim_t^i = h$ in Eq (7)).

2) sampling new candidates of the selected problem, and 3) expanding new candidates to the set of sampled code:

$$C_{t+1}^i = \begin{cases} C_t^i \cup C_t', & \text{if } m(a_t) = i, \\ C_t^i, & \text{otherwise}, \end{cases} \tag{4}$$

where C_t' is the set of new code samples of $x_{m(a_t)}$ drawn from $p_\theta(C_t'; x_{m(a_t)})$, i.e., the probability distribution of the code generation model θ. The initial state is $C_i^0 = \{\}$, and the sampling process continues until Eq (3) is violated.

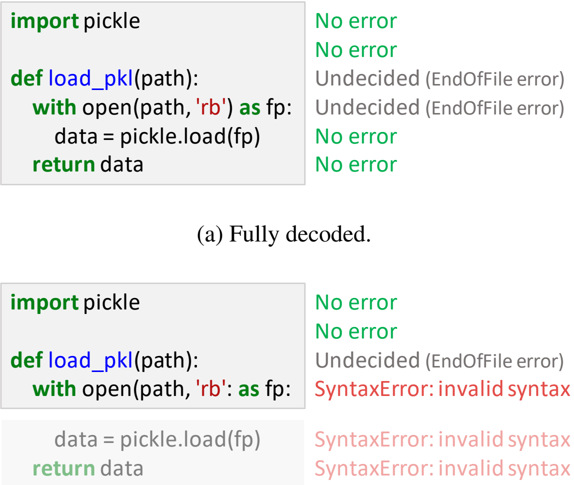
4.2 Solvability Estimation

During the multiple sampling process, EFFICODE prioritizes the problems by estimating the ground truth solvability for each x_i. As the solvability is relative to the capability of θ, we consider 1) the difficulty of x_i by θ, and 2) the similarity of x_i with problems that θ previously solved. Figure 2 shows the overview of solvability estimation by EFFICODE.

Solvability. We define a solvability of the model θ to the problem x_i as the likelihood of sampling a correct code using θ:

$$\mathcal{S}^*(x_i; \theta) = \mathbb{E}_{C_i \sim p_\theta(C_i; x_i)} \left[F(C_i) \right], \tag{5}$$

while satisfying Eq (3). Then, a problem x_i is more solvable than another problem x_j if $\mathcal{S}^*(x_i; \theta) > \mathcal{S}^*(x_j; \theta)$. If both x_i and x_j are not solved yet, we prefer to sample more for x_i over x_j.



```
import pickle                              No error
                                           No error
def load_pkl(path):                        Undecided (EndOfFile error)
    with open(path, 'rb') as fp:           Undecided (EndOfFile error)
        data = pickle.load(fp)             No error
    return data                            No error
```

(a) Fully decoded.

```
import pickle                              No error
                                           No error
def load_pkl(path):                        Undecided (EndOfFile error)
    with open(path, 'rb': as fp:           SyntaxError: invalid syntax

        data = pickle.load(fp)             SyntaxError: invalid syntax
    return data                            SyntaxError: invalid syntax
```

(b) Pruned before fully decoded due to syntax error.

Figure 3: Sample-prunable decoding periodically checks whether the current partially generated code contains syntax errors that will remain after completion.

Error Ratio. Unlike execution-based approaches such as CODET (Chen et al., 2023) and REFLEXION (Shinn et al., 2023), EFFICODE approximates the likelihood of errorneous execution at time step t as syntax errors in C_t^i, following the convention in (Hendrycks et al., 2021).

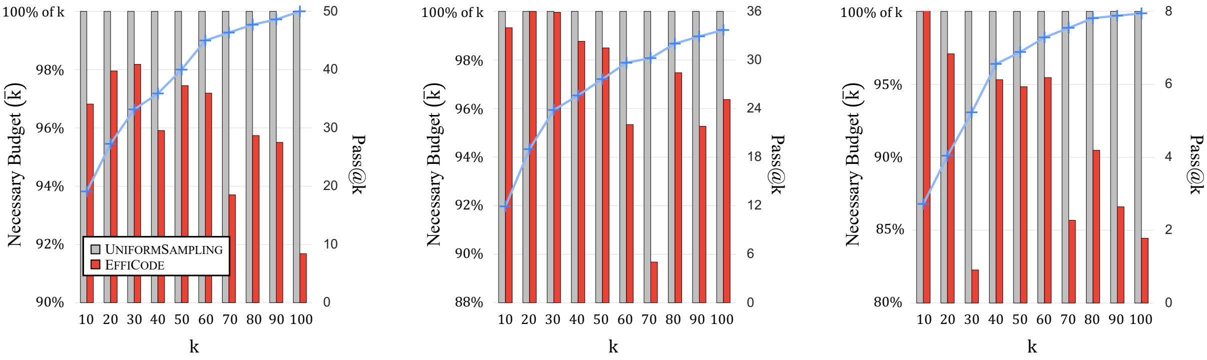
For the representativeness, we skip the prioritization when $|C_t^i|$ is smaller than a hyperparameter N. Formally, we assign err_t^i based on the error ratio in C_t^i as,

$$err_t^i = \begin{cases} h & \text{if } t < N \text{ or } \mathsf{ER}(C_t^i) \leqslant T_E, \\ l, & \text{otherwise}, \end{cases} \tag{6}$$

where h and l ($h > l$) are hyperparameters for priority scores, $\mathsf{ER}(C_t^i)$ is the ratio of code samples with syntax errors in C_t^i, and T_E is the threshold hyperparameter.

Similarity w/ Solved Problems. We prioritize sampling for the problem instance x_i if it is similar to previously solved problems by θ. To investigate the similarity of the problems from the perspective of θ, we compare C_t^i to C_{pre}– a set of correct code for previously solved problems by θ– using CodeBLEU (Ren et al., 2020). This approach is particularly advantageous in industrial contexts, utilizing readily available accumulated logs as C_{pre}.

Formally, we assign sim_t^i the priority of x_i by



(a) HumanEval-Extreme

(b) MBPP-Extreme

(c) CodeContests-Extreme

Figure 4: Results on various benchmarks with extreme settings. Bar charts, which belong to the left side of y-axis, denote the reduced amount of sampling budget to reach Pass@k performance. Line charts belong to the right side of y-axis, indicating Pass@k scores. Throughout the benchmarks, code samples were generated by GPT-3.5 with Self-planning.

the similarity between C_t^i and C_{pre} as,

$$sim_t^i = \begin{cases} h & \text{if } t < N \text{ or} \\ & \quad \texttt{SIM}(C_t^i, C_{pre}; s_t) \geqslant S\%, \\ l, & \text{otherwise,} \end{cases} \tag{7}$$

where $\texttt{SIM}(C_t^i, C_{pre}; s_t)$ ranks the similarity between C_t^i and C_{pre} within the state $s_t = [C_t^1, ..., C_t^{|X|}]$, and returns this rank as a percentage. To enhance the understanding of θ's capability, we use Self-planning (Jiang et al., 2023), which synthesizes commented high-level blueprints then generates code. The similarity is then measured after concatenating the blueprints and code.

For the robustness over errors in estimated solvabilities, we use weighted sampling to select the next action $a_t \in \mathcal{A}(s_t)$. The weight value p_t^i for a_t where $m(a_t) = i$ is the (normalized) min score between err_t^i and sim_t^i:

$$P(m(a_t) = i) = p_t^i \quad \text{for} \quad i \in \{1, 2, \ldots, |X|\},$$
$$p_t^i = \frac{\texttt{min}(err_t^i, sim_t^i)}{\sum_j \texttt{min}(err_t^j, sim_t^j)}. \tag{8}$$

4.3 Adaptive Decoding

EffiCode dynamically adapts to partial decoding by periodically inspecting for early termination. EffiCode specifically targets the subset of incorrect code that exhibits syntax errors, which are relatively common in code generated from LLMs. For example, approximately 11% of Python code generated by AlphaCode contains syntax errors (Li et al., 2022).

EffiCode halts the decoding procedure when syntax errors are detected, while excluding undecidable ones like EndOfFile which can be rectified with proper subsequent code lines.[2] Figure 3 demonstrates EffiCode detecting syntax errors after each line of code is decoded. We leverage the accurate and low-overhead compiler of the designated programming language, such as Python's built-in compiler, for syntax verification. This approach effectively prunes incorrect code segments before their completion, lowering the total decoding expense.

5 Experimental Setup

We evaluate the effectiveness of EffiCode by assessing its impact on the sample efficiency of GPT-3.5-turbo-0301 (OpenAI, 2022), a sibling model of InstructGPT (Ouyang et al., 2022). Throughout the experiments, we use nucleus sampling (Holtzman et al., 2020) with the *top p* = 0.95 and the temperature $T = 0.8$ (Chen et al., 2021; Nijkamp et al., 2023; Chen et al., 2023). The implementation details for EffiCode is explained in Appendix A.

5.1 Evaluation Metrics

We use a popular metric Pass@k (Chen et al., 2021) that equally samples k code samples for each problem, and plot the average number of samples $\bar{k}$ to reach the same performance with Pass@k (i.e. necessary budget), where the reduced number of samples per problem can vary. For correct code

[2] For other languages like C, C++, and Java, we can consider additional undecidable cases such as unfinished parentheses.

Code Selection Method	Execution	Model		$n@k$	
			k	n=1	n=2
		HumanEval-Hard50			
None		GPT-4	30	60.0†	-
Reflexion (w/o test run)		GPT-4	30	52.0†	-
Reflexion	*required*	GPT-4	30	68.0†	-
None		GPT-3.5	30	40.9	48.4
None		GPT-3.5‡	30	47.8	57.4
EffiCode		GPT-3.5‡	30	49.0	57.7
		HumanEval			
CodeRanker		Codex	100	32.3	-
AlphaCode-C	*required*	Code-davinci-002	100	55.1	64.1
CodeT	*required*	Code-davinci-002	100	65.8	75.1
Reflexion	*required*	GPT-4	30	91.0	-
None		GPT-3.5	100	63.0	69.4
None		GPT-3.5‡	100	68.5	77.1
EffiCode		GPT-3.5‡	100	69.9	77.3
		MBPP			
AlphaCode-C	*required*	Code-davinci-002	100	62.0	70.7
CodeT	*required*	Code-davinci-002	100	67.7	74.6
None		GPT-4	30	80.1	-
Reflexion	*required*	GPT-4	30	77.1	-
None		GPT-3.5	100	59.7	66.4
None		GPT-3.5‡	100	66.1	72.3
EffiCode		GPT-3.5‡	100	66.1	72.1
		CodeContests			
CodeT	*required*	Code-davinci-002	1000	2.1	2.3
Algo	*required*	Code-davinci-002	1000	5.6	5.6
None		GPT-3.5	100	2.6	4.1
None		GPT-3.5‡	100	3.9	5.6
EffiCode		GPT-3.5‡	100	6.7	7.9

Table 1: Results for $n@k$ code sample selection are shown above, with values above the dashed line directly sourced from original works. Red and blue colored scores are the results without code execution that are higher or lower than the scores when the code selection method is not applied. Generated code is written in Python language, except for the daggered results (†) written in Rust language. The double dagger (‡) signifies that Self-planning (Jiang et al., 2023) is applied for code generation. For Reflexion, we regard max 30 iterations of refinement and use the final version as selecting one from 30 samples.

selection, we use $n@k$ (Li et al., 2022), which samples k candidates, then ranks or filters to select n samples.

5.2 Benchmarks

We conduct experiments on below three code generation benchmarks: CodeContests (Li et al., 2022) consists of 13K / 113 / 165 of training / valid / test problems from various code competition websites. HumanEval (Chen et al., 2021) is a handcrafted test dataset containing 164 Python problems. MBPP (sanitized; Austin et al., 2021) contains 427 crowd-sourced Python problems.

In extreme settings, we first sample 100 code samples per problem by GPT-3.5 with Self-planning (Jiang et al., 2023), then select problems that the solved ratio (i.e. the ratio of correct code samples to all the generated samples) is below 10%. The dataset size of CodeContests-Extreme, HumanEval-Extreme, and MBPP-Extreme is 151, 22, and 89, respectively.

To compare EffiCode with Reflexion (Shinn et al., 2023) in correct code selection, we also report EffiCode in another HumanEval subset consists of 50 problems, namely HumanEval-Hard50.

6 Experimental Results

6.1 Sample Efficiency

In our main experiment, we validated the effectiveness of EffiCode in improving the sample efficiency, comparing with conventional sampling. When gauging the effectiveness of solvability estimation, it becomes challenging especially when dealing with problems of high solvability. In such cases, even if we were to randomly select them, the Pass@k score would effortlessly increase, potentially masking the true performance of the estimation process. Therefore, we evaluate EffiCode on

Benchmark	Method	n@100	
		n=1	n=2
HumanEval-Extreme	*None*	2.0	3.9
	EFFICODE	**4.6**	**9.1**
MBPP-Extreme	*None*	1.4	2.7
	EFFICODE	**4.5**	**4.5**
CodeContests-Extreme	*None*	0.3	0.6
	EFFICODE	**1.3**	**2.0**

Table 2: Results of selecting n code samples from 100 for each problem (n@100; n@k where k=100).

extreme-level subsets of the benchmarks where each problem has the solve ratio below 10%.

As shown in Figure 4, EFFICODE consistently requires the reduced number of necessary budget $\overline{k}$ to reach the Pass@k performance. This is a novel contribution, as previous research has not addressed the sample efficiency of code generation models during inference. Note that EFFICODE is especially effective when the test set is hard– in CodeContests-Extreme, EFFICODE only requires 16% less budget to reach Pass@100 performance.

6.2 Functional Correctness

Recent approaches focus on specifying which candidate is functionally correct (Li et al., 2022; Inala et al., 2022; Chen et al., 2023). We validate the correctness of EFFICODE, which can reduce temporal costs by alleviating code execution.

The results are shown in Table 1 and Table 2. It is noteworthy that REFLEXION, a popular code refinement method, significantly drops n@k performance when applied without code execution in HumanEval-Hard50, and even with code execution in MBPP. In contrast, EFFICODE consistently improves n@k performance except for MBPP, but still shows comparable performance to that of EFFICODE is not applied.

7 Conclusion

This paper studies sample efficiency in code generation, which significantly affects the computational/temporal costs and environmental consequences yet has been neglected. Our proposed approach EFFICODE prioritizes sampling on test problems by estimating solvability. We conduct extensive experiments on the CodeContests, HumanEval, and MBPP benchmarks, consistently showing the improved sample efficiency. Additionally, EFFICODE can be used as correct code selection while reducing temporal costs by alleviating code execution.

Acknowledgement

This work was partially supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by ICT R&D program of MSIT/IITP (2022-0-00995, Automated reliable source code generation from natural language descriptions).

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

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. *CoRR*, abs/2108.07732.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks are all you need.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *International Conference on Learning Representations*.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu K Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems*.

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*.

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

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):1092–1097.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, Noah A. Smith, and Yejin Choi. 2022. NeuroLogic a*esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 780–799, Seattle, United States. Association for Computational Linguistics.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *International Conference on Learning Representations*.

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue. openai.

OpenAI. 2023. Gpt-4 technical report.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

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

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 3645–3650. Association for Computational Linguistics.

Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. `https://github.com/kingoflolz/mesh-transformer-jax`.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *arXiv preprint arXiv:2305.14591*.

A Implementation Detail

Solvability Estimation. To check syntax errors, we use the built-in compiler function `compile` in Python 3.9.12. The error ratio threshold T_E is set to 0.7, and the skip parameter for representativeness N is set to 10. We set the high/low priority value h and l as 1 and 0.1. For CodeContests, we generate 10 code samples per problem in the validation set, then use the solved problems and the corresponding correct code samples as X_{pre} and C_{pre}. As HumanEval and MBPP[3] have only test data, we use each other as the log to build X_{pre} and C_{pre}. To avoid mistakenly giving a low priority, we conservatively set the top S as 80%.

Adaptivity. To check syntax errors in partial code written in Python language, we use the same built-in `compile` function as in solvability estimation. Specifically, we validate partial code when its current code line is finished. We determine whether a line has been finished or not by checking if the last character is a newline character (`'\n'`)[4]. If the partial code contains any syntax errors except for EndOfFile, we immediately stop decoding and discard the partial sample. If the decoding is successfully done, we conduct a final validation. This time, as there is no further decoding, we discard all the syntactically erroneous code including EndOfFile errors.

[3] To compare with CodeT (Chen et al., 2023), we use the entire MBPP sanitized set as the test set.

[4] We do not check '`\\n`' as the compiler regards the current code line is not finished and is extended to the following line.