Where Is the Bottleneck of LLM Code Generation? A Study Isolating LLM Performance on Language-Coding from Problem-Solving

Anonymous ACL submission

Abstract

Existing code generation benchmarks for Large Language Models (LLMs) such as HumanEval and MBPP are designed to study LLMs' end-to-end performance, where the benchmarks feed a problem description in nature language as input and examine the generated code in specific programming languages. However, the evaluation scores revealed in this way provide a little hint as to the bottleneck of the code generation – whether LLMs are struggling with their problem-solving capability or language-coding capability. To answer this question, we construct PSEUDOEVAL, a multilingual code generation benchmark that provides a solution written in pseudocode as input. By doing so, the bottleneck of code generation in various programming languages could be isolated and identified. Our study yields several interesting findings. For example, we identify that the bottleneck of LLMs in Python programming is problem-solving, while Rust is struggling relatively more in language-coding. Also, our study indicates that problem-solving capability may transfer across programming languages, while language-coding needs more language-specific effort, especially for undertrained programming languages. Finally, we release the pipeline of constructing PSEUDOEVAL to facilitate the extension to existing benchmarks. PSEUDOEVAL is available at: https://anonymous.4open.science/r/PseudocodeACL25-7B74/.

1 Introduction

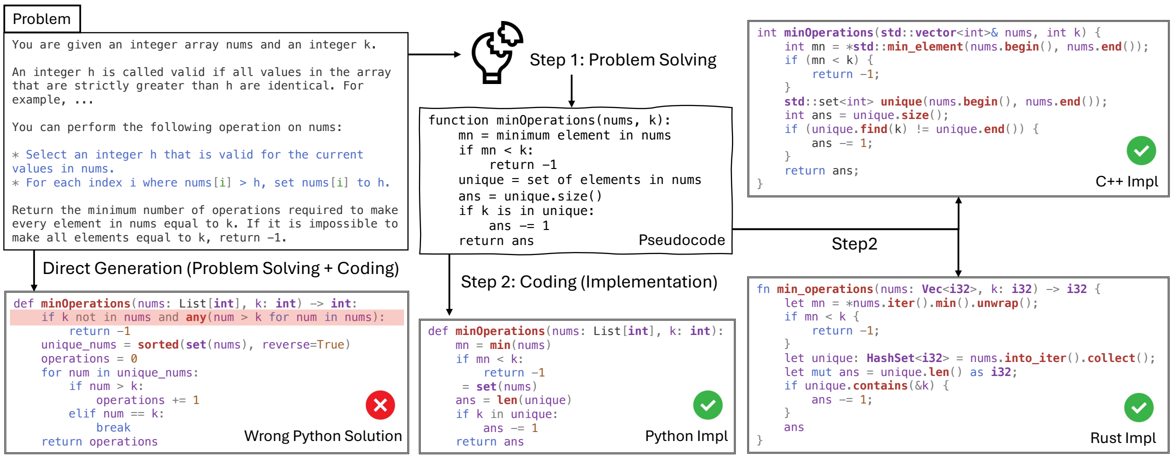
Large Language Models (LLMs) have exhibited impressive proficiency in aiding software development, particularly in the realm of code generation. Existing code generation benchmarks, such as HumanEval (Chen et al., 2021), typically present a natural language description (e.g., "return a list with elements incremented by 1") and require LLMs to generate code that fulfills the described functionality. On the HumanEval leaderboard, various LLMs have achieved scores close to perfection (at most 99.4% [1]). However, on another benchmark known for minimal contamination, LiveCodeBench (Jain et al., 2024a), the highest score recorded is 76.5% [2]. The highest score drops to 52.2% for problems in the hard category.

However, what do these scores truly imply? When scores approach perfection, does it genuinely imply that the LLMs have nearly attained the capability to replace Python developers? The answer seems to be *no*. Numerous studies have revealed significant shortcomings in LLMs' code generation capabilities, such as producing code with syntactic errors, code that does not meet the intended requirements, or code with low-level implementation mistakes. Yet, merely summarizing these phenomena as "hallucinations" (Li et al., 2023; Xu et al., 2024; Zhang et al., 2023b; Dhuliawala et al., 2023; Zhang et al., 2024) is an oversimplification. We seek to understand ***what the bottleneck of code generation is*** – Is it due to a lack of ***problem-solving capability*** or ***language-coding capability***, or both?

To facilitate this study, we constructed a multilingual code generation benchmark, PSEUDOEVAL, with 1,059 subjects with not only problem-solution pairs but also intermediate solutions represented as pseudocode, which serve to isolate the problem-solving capability from the language-coding capability. Take Figure 1 for example. Given a problem description (upper-left corner), the existing end-to-end code generation benchmarks typically examine whether the generated code (lower-left corner) is implemented correctly and report a binary result (pass or fail) as the evaluation output. However, the binary result gives little hint of the bottleneck, i.e., it is still unclear ***whether LLMs are incapable of coming up with solutions for this problem*** or

[1] Result on Feb 13,2024, from https://paperswithcode.com/sota/code-generation-on-humaneval

[2] Result on Feb 13, 2024, from https://livecodebench.github.io/leaderboard.html



```
You are given an integer array nums and an integer k.

An integer h is called valid if all values in the array
that are strictly greater than h are identical. For
example, ...

You can perform the following operation on nums:

* Select an integer h that is valid for the current
values in nums.
* For each index i where nums[i] > h, set nums[i] to h.

Return the minimum number of operations required to make
every element in nums equal to k. If it is impossible to
make all elements equal to k, return -1.
```

```
function minOperations(nums, k):
    mn = minimum element in nums
    if mn < k:
        return -1
    unique = set of elements in nums
    ans = unique.size()
    if k is in unique:
        ans -= 1
    return ans
```

```python
def minOperations(nums: List[int], k: int) -> int:
    if k not in nums and any(num > k for num in nums):
        return -1
    unique_nums = sorted(set(nums), reverse=True)
    operations = 0
    for num in unique_nums:
        if num > k:
            operations += 1
        elif num == k:
            break
    return operations
```

```python
def minOperations(nums: List[int], k: int):
    mn = min(nums)
    if mn < k:
        return -1
     = set(nums)
    ans = len(unique)
    if k in unique:
        ans -= 1
    return ans
```

```cpp
int minOperations(std::vector<int>& nums, int k) {
    int mn = *std::min_element(nums.begin(), nums.end());
    if (mn < k) {
        return -1;
    }
    std::set<int> unique(nums.begin(), nums.end());
    int ans = unique.size();
    if (unique.find(k) != unique.end()) {
        ans -= 1;
    }
    return ans;
}
```

```rust
fn min_operations(nums: Vec<i32>, k: i32) -> i32 {
    let mn = *nums.iter().min().unwrap();
    if mn < k {
        return -1;
    }
    let unique: HashSet<i32> = nums.into_iter().collect();
    let mut ans = unique.len() as i32;
    if unique.contains(&k) {
        ans -= 1;
    }
    ans
}
```

Figure 1: Motivating example

suffering from language-specific implementation such as writing syntactic- or semantic-correct code in certain programming languages such as C++ or Rust. With PseudoEval, the assessment would yield clearer results – by breaking the end-to-end task down into two steps. One could observe when providing the solution (Pseudocode in the middle of Figure 1), LLMs can successfully code it in three languages (Python, C++, and Rust), while all experimental LLMs failed to solve this easy-tagged problem without the provided solution, ***indicating the bottleneck for this problem is more on the problem-solving than language-coding capability***. Furthermore, to expand the usefulness of PseudoEval, we explore four research questions (RQs).

RQ1. To what extent can the provided pseudocode improve the correctness of code generation? This RQ provides an overall profiling of the performance from the question description and the pseudocode. Understanding performance differences with and without pseudocode across different LLMs/programming languages/difficulties of the questions helps identify *the bottleneck of code generation in different programming languages*.

RQ2. To what extent can the solution from one programming language benefit the code generation in another programming language? This RQ extends the study from monolingual to multilingual observation. It explores whether the pseudocode derived from codes in one programming language could benefit the code generation in another programming language. The results could *give hints of the possibility of transferring problem-solving capability across programming languages.*

RQ3. Can different inference strategies yield significantly different observations? Different promptings and attempts may yield different performances. This RQ explores whether the observation of the bottleneck (problem-solving or implementation) would significantly vary under different inference strategies.

RQ4. What is the difference between human-written pseudocode and auto-extracted pseudocode? The pseudocode in PseudoEval is automatically extracted from the solution code. However, no prior study has been made to examine the quality of the pseudocode quantitatively. In this RQ, we compare the difference between the human-written pseudocode and the auto-extracted pseudocode in terms of token lengths, lines of code, and the LLMs' performance with pseudocode generated in both ways. The study provides more evidence to demonstrate the quality of the pseudocode in PseudoEval, and once assured, the auto-extraction we proposed could facilitate the extension to existing benchmarks.

Our study yields interesting observations. First, ***the bottleneck*** in Python code generation is problem-solving, while C++ and Rust struggle relatively more in language-coding. Second, most solutions are ***language-agnostic***, indicating it may be enough for LLMs to learn problem-solving skills in certain programming languages and put more effort into the coding capability in programming languages. Third, the auto-generated pseudocode is ***comparable or even better*** quality than human-written ones. Thus, it is feasible to extend the existing benchmarks with pseudocode with our pipeline. The contribution of this paper includes:

● ***Problem Decomposition***: We break down the end-to-end code generation (from problem description to implementation) into a two-step evaluation (from problem description in natural language or from solutions in pseudocode). By doing so, the bottleneck of code generation in various programming languages could be isolated and identified.

● ***Benchmark* PSEUDOEVAL**: We constructed a multi-lingual (Python, C++, and Rust) code generation benchmark with 1,059 subjects comprising not only problem description in natural language and corresponding tests, but also the intermediate solutions in the form of pseudocode. The benchmark enables exploration of the bottleneck in code generation, provides clear criteria for pseudocode construction, and makes available a pipeline that implements a workflow automating the construction process. With it, one could refurbish existing code generation benchmarks easily.

● ***Insight***: We isolate LLMs' capabilities for code generation into problem-solving and language-coding. Our study finds that the bottleneck of generating code in different programming languages is different. Our study further suggests that problem-solving capability may transfer across programming languages while the coding capability for programming languages beyond the most popular ones remains to be improved.

2 Problem Definition

2.1 Task Definition

As shown in Figure 1, the problem of LLM code generation comprises two tasks.

(1) Problem Solving, which analyzes the problem and reasons a *solution* as output. The granularity of a solution can vary from a one-sentence description of the core algorithm to a pseudocode with a clear control flow and data manipulation.

(2) Language Coding, which transforms the solution into a piece of compilable and executable code that implements the key logic and data manipulation in a target programming language.

The two tasks exercise distinct abilities of LLMs, and previous code generation benchmarks such as LiveCodeBench (Jain et al., 2024a) evaluate them inseparably. This paper studies the coding ability of LLMs by isolating it from their problem-solving ability using pseudocode.

2.2 Pseudocode Criteria

Although there is no universal concrete standard for pseudocode, conventions such as guidebooks have been commonly adopted. To study the coding ability of LLMs, we adopt a set of criteria to prepare the pseudocode for PSEUDOEVAL. The criteria are designed based on the features of pseudocode in textbooks, guidebooks, and research papers.

Completeness. The pseudocode should be mapped to a piece of implementation code *without ambiguity*, e.g., a competent programmer should be able to implement the pseudocode solving the given problem in a specific programming language. If given a piece of implementation code, one can obtain a trivial but complete pseudocode by line-by-line code translation (Kulal et al., 2019a).

Language-agnostic. The pseudocode should describe a language-agnostic solution. It should not be tied to specific language features, such as the `yield` expression in Python and the pointer manipulations in C++. In particular, explicit type information (e.g., `vector` in C++) and type conversion should be absent. The language-agnostic criterion facilitates a fair evaluation of LLMs' coding abilities in different target programming languages with the same pseudocode.

Conciseness. A pseudocode should be concise, which can be measured by the lines of code and the number of tokens. In practice, software developers tend to sketch solutions concisely. Also, verbose pseudocode with implementation details may not help differentiate the abilities of stronger models and weaker models. An interesting case (Appendix C.2) in our study is to simplify a well-known algorithm, Sieve of Eratosthenes, and customize its use in pseudocode. LLMs with higher coding capability can successfully implement the pseudocode, while weaker LLMs have lower success rates and even drop to zero when the target language is Rust.

Following the above criteria, we define more specific rules (Appendix D) to prompt DeepSeek-R1 to convert an implementation code into a pseudocode to construct PSEUDOEVAL (Section 3).

3 Dataset Construction

To build PSEUDOEVAL, we design a pipeline implementing an automated workflow in Figure 2 to collect user-submitted solutions on LeetCode and distill pseudocode solutions from them using a recent reasoning model DeepSeek-R1. The pipeline may also be adapted to refurbish other existing code generation benchmarks.

Data Source. To lessen the data leakage threat, we select user-submitted solutions based

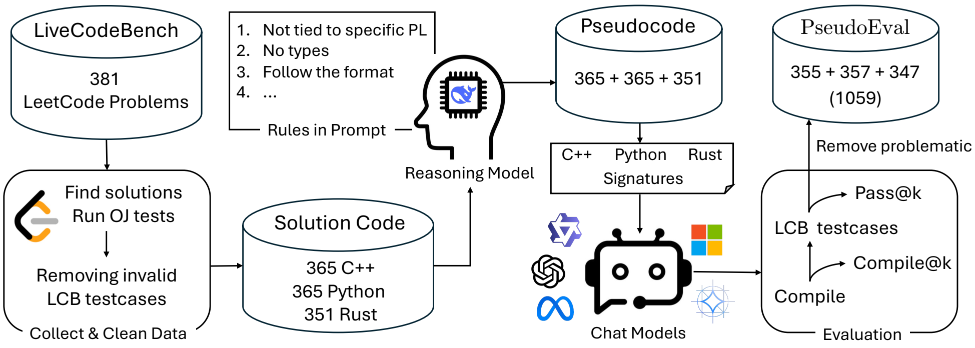


Figure 2: Workflow of constructing the PSEUDOEVAL dataset and empirical study

on the problems most recently collected by LiveCodeBench (Jain et al., 2024a). These are the latest programming problems released after the training cut-off dates of popular LLMs. In other words, we select the most recent subset of problems indexed by LiveCodeBench at LeetCode. We further collect the corresponding user-submitted solutions from LeetCode. For each problem, we manually collect the most popularly voted solutions in Python, C++, and Rust, respectively.

Task Cleaning. To ensure the correctness of the collected user-submitted solutions, we run each solution via the LeetCode online judge to ensure the solution passes all mandated tests. If the most popularly voted solutions fail (usually due to the update of problems/tests), we collect another solution that passes the updated tests. The study of our research questions requires evaluating the correctness of many generated codes. Submitting all of them to the LeetCode online judge for correctness validation is inappropriate. Therefore, we collect the published tests deduced by LiveCodeBench and use them to evaluate the correctness of the generated codes in our study. However, these LiveCodeBench tests are deduced by LLMs and subject to noises. We consider a deduced LiveCodeBench test noisy if it fails the collected solutions. In total, we find 16 noisy instances and exclude them from our study. After cleaning, we collect 365 solutions in C++ and Python and 351 solutions in Rust.

Code to Pseudocode. Each pseudocode used to evaluate the coding capability of LLMs is generated by the reasoning model DeepSeek-R1 (Guo et al., 2025) given a solution code and a detailed list of rules (Appendix D) that the output pseudocode needs to satisfy, i.e., the criteria in Section 2.2. For example, the pseudocode should not contain explicit types like 32-bit or 64-bit integers and language-specific operations like `yield` in Python.

We choose a reasoning model over a chat model like GPT-4o. Our pilot experiments find that chat models often fail to obey the rules in a long context or just write the pseudocode line by line without undergoing a substantial thinking process. The prompt we use consists of only the user query without a system message or few-shot examples, as suggested by the DeepSeek team (Guo et al., 2025). We also follow their experiment setting (`temperature=0.6`, `top_p=0.95`). One pseudocode sample is obtained for each selected user-submitted solution due to the limited access to the R1 service and the incurred time latency.

Pseudocode Quality Assessment. To remove incorrect R1-generated pseudocode, we use LLMs to generate code from the R1-generated pseudocode using our study setup and remove the tasks where *NO LLMs* can pass the task with ten attempts. Finally, we remove 22 subjects where R1 hallucinates a pseudocode with incorrect logic (e.g., adding an incorrect condition), and keep 1,059 subjects. Besides, we compare the lengths and effectiveness of pseudocode annotated by R1 and humans for randomly sampled subjects in RQ4. The results also suggest good quality of the retained pseudocode.

4 Experiment

Data. To facilitate the comparison across programming languages, 999 (333 × 3) experiment subjects are drawn from the intersected programming tasks for C++ (355), Python (357), and Rust (347).

Metrics. The correctness of the generated programs is calculated by their Pass@k rates on the tests published by LiveCodeBench. The conciseness of pseudocode is measured by their lengths regarding the number of Byte-Pair Encoding (BPE) tokens and lines of codes.

	Python						C++						Rust					
	from Problem			from Pseudocode			from Problem			from Pseudocode			from Problem			from Pseudocode		
	Easy	Med	Hard	Easy	Med	Hard	Easy	Med	Hard	Easy	Med	Hard	Easy	Med	Hard	Easy	Med	Hard
GPT-4o-mini	0.82	0.32	0.07	0.95	0.88	0.76	0.78	0.27	0.13	0.91	0.81	0.66	0.69	0.20	0.06	0.83	0.55	0.35
Qwen32B	0.90	0.56	0.20	0.93	0.94	0.79	0.86	0.50	0.20	0.92	0.86	0.66	0.80	0.45	0.16	0.90	0.68	0.57
Qwen32Bq4	0.87	0.56	0.22	0.92	0.94	0.79	0.84	0.51	0.20	0.91	0.86	0.68	0.80	0.42	0.13	0.89	0.68	0.54
Qwen14B	0.80	0.51	0.14	0.97	0.90	0.70	0.78	0.49	0.10	0.94	0.81	0.55	0.72	0.34	0.04	0.86	0.53	0.33
Qwen7B	0.68	0.34	0.11	0.86	0.81	0.53	0.69	0.36	0.09	0.86	0.71	0.37	0.54	0.22	0.00	0.70	0.42	0.13
Gemma9B	0.49	0.09	0.04	0.84	0.68	0.45	0.44	0.10	0.06	0.84	0.55	0.25	0.29	0.03	0.01	0.40	0.22	0.06
Llama3-8B	0.40	0.08	0.01	0.68	0.60	0.47	0.31	0.07	0.03	0.67	0.47	0.24	0.23	0.03	0.01	0.50	0.23	0.07
Llama3-3B	0.27	0.04	0.00	0.54	0.41	0.23	0.20	0.03	0.01	0.45	0.25	0.14	0.14	0.00	0.00	0.30	0.06	0.01
Phi4-14B	0.66	0.28	0.04	0.92	0.83	0.77	0.65	0.26	0.10	0.87	0.73	0.42	0.59	0.16	0.05	0.75	0.46	0.25
Phi3.5-3.8B	0.44	0.08	0.03	0.67	0.46	0.30	0.35	0.05	0.03	0.53	0.28	0.09	0.13	0.00	0.02	0.20	0.03	0.00
Average	0.63	0.29	0.09	0.83 *31%↑*	0.75 *160%↑*	0.58 *573%↑*	0.59	0.26	0.10	0.79 *34%↑*	0.63 *140%↑*	0.41 *327%↑*	0.49	0.19	0.05	0.63 *28%↑*	0.39 *109%↑*	0.23 *381%↑*
Overall	0.38			0.75 *(99%↑)*			0.35			0.65 *(85%↑)*			0.27			0.45 *(66%↑)*		

Table 1: Pass@1 of generations from problem descriptions and pseudocode for easy, medium, and hard tasks

Studied LLMs. We study the code generation performance of ten diverse popular LLMs, including Qwen-series (Qwen-2.5-Coder 7B, 14B, 32B, 32B-Int(q)4) (Hui et al., 2024), Gemma-series (Gemma-2-9b) (DeepMind, 2024), Llama-series (Llama-3.1-8B and -3.2-3B) (Meta, 2024), Phi-series (Phi-4-14B and -3.5-3.8B) (Microsoft, 2024), and GPT-series (GPT-4o-mini) (OpenAI, 2024). Relatively more LLMs evaluated are under 15B parameters. This is to understand the language-coding ability of lighter, more deployable models.

Parameters. We follow the setting suggested by LiveCodeBench to sample ten times of generations for each problem using a temperature of 0.2 and top_p of 0.95. We compare one-shot and zero-shot prompts for pseudocode-based code generation.

Experiment Environment. The experiments are conducted on a Linux server with two NVIDIA RTX 6000Ada GPUs. The commercial GPT-4o-mini and the primary DeepSeek-R1 are accessed via API calls. Other open-weight LLMs are deployed locally on the server with the vLLM engine.

4.1 RQ1: Overall Performance

To understand the language-coding capability of LLMs, we analyze the quality of the codes that LLMs generated from pseudocode. Each column in Table 1 presents the Pass@1 rate of LLMs in generating programs of a specified programming language based on the pseudocode derived from the solution codes in the same language. We also list LLMs' Pass@1 rates when directly generating codes from problem descriptions using the prompt adopted by LiveCodeBench as a reference.

Effect of Pseudocode. All ten LLMs achieve significantly higher Pass@1 rates on all programming languages when generating programs from pseudocode than from problem descriptions. Specifically, the overall Pass@1 rates on all difficulties increase from 0.38, 0.35, 0.27 to 0.75, 0.65, 0.45 on Python, C++, and Rust, respectively. The results suggest that the solutions encoded in pseudocode help LLMs generate more correct programs. As such, we consider that *problem-solving ability is a key bottleneck common to LLMs*. Regarding programming languages, all LLMs exhibit the largest performance gain in Python programming (+99% on average), followed by C++ (+85% on average). Performance improvement on Rust is the least (66% on average) yet is still significant. As Rust coding is lower in resource availability (Zheng et al., 2023; Cao et al., 2025), the result suggests *the correlation between language-coding ability and the prevalence of the language in corpus*.

	P_{Py}	$P_{\text{C++}}$ → Python Code	P_{Rust}	P_{Py}	$P_{\text{C++}}$ → C++ Code	P_{Rust}	P_{Py}	$P_{\text{C++}}$ → Rust Code	P_{Rust}
GPT-4o-mini	0.89	0.90	**0.91**	0.80	**0.82**	0.81	0.59	**0.63**	0.62
Qwen32B	**0.91**	0.90	0.90	0.81	**0.85**	0.84	0.69	0.73	**0.74**
Qwen32Bq4	**0.91**	0.90	0.90	0.80	**0.85**	0.84	0.68	0.72	**0.73**
Qwen14B	**0.89**	0.88	0.88	0.76	**0.82**	0.80	0.57	0.61	**0.62**
Qwen7B	0.78	**0.83**	0.81	0.64	**0.71**	0.67	0.43	**0.48**	0.47
Gemma9B	0.69	**0.73**	0.69	0.55	**0.60**	0.56	0.26	**0.28**	0.26
Llama3-8B	0.61	**0.65**	0.60	0.45	**0.50**	**0.50**	0.26	0.28	**0.30**
Llama3-3B	0.42	**0.44**	**0.44**	0.27	**0.30**	**0.30**	**0.16**	0.13	0.13
Phi4-14B	**0.85**	0.83	**0.85**	0.71	**0.73**	0.72	0.51	**0.54**	0.53
Phi3.5-3.8B	**0.51**	**0.51**	0.47	0.30	**0.33**	0.29	0.09	**0.11**	0.08
Average	0.75	**0.76**	0.75	0.61	**0.65**	0.63	0.42	**0.45**	0.45

Table 2: Pass@1 of code generation with pseudocode derived from different programming languages

Language-Coding Capability. LLMs' language-coding capability varies across programming languages. Given pseudocode, most LLMs can generate correct implementations in Python; while they still cannot generate correct Rust implementations for many tasks. For example, given pseudocode, the Python, C++, and Rust Pass@1 rates are 0.89, 0.82, 0.62 for GPT-4o-mini and 0.85, 0.73, 0.53 for Phi-4, respectively. It suggests that *the bottleneck of LLMs in code generation is problem-solving, while as to Rust and C++, they are struggling relatively more in language coding*.

Model-Wise. Given pseudocode, the Python Pass@1 rates of the best-performing studied LLM, QWen32B (and its quantified variant QWen32Bq4), on easy, medium, and hard tasks significantly increase from around 0.90, 0.56, 0.20 to 0.93, 0.94, 0.79, respectively, followed by QWen14B (0.80, 0.51, 0.14 → 0.97, 0.90, 0.70) and GPT-4o-mini (0.82, 0.32, 0.07 → 0.95, 0.88, 0.76). Similar trends are observed in C++ and Rust. This indicates such powerful LLMs likely have mature language-coding capabilities, particularly in Python. Most of their bottleneck in solving LiveCodeBench tasks may reside in the problem-solving procedure. In comparison, although the smaller models show improvement in Pass@1 rates given pseudocode, their generations based on pseudocode are still error-prone. For example, almost all Pass@1 rates Llama-3.1-8B, Llama-3.2-3B, and Phi-3.5-3.8B for medium and hard tasks are still below 0.50. This suggests that *regarding the ability to implement a given programming logic, smaller LLMs are much inferior to the larger LLMs*.

Difficulty-Wise. LLMs show the most improvement in Pass@1 rates on the hard tasks (573%↑, 327%↑, and 381%↑ in Python, C++, and Rust, respectively), followed by the medium tasks (160%↑, 140%↑, and 109%↑) and then the easy tasks (31%↑, 34%↑, and 28%↑). Since hard tasks require more problem-solving ability, the result echos our conclusion that problem-solving is a key bottleneck.

Worsening Cases. We noticed a few cases where providing pseudocode degrades the models' performance. For example, when writing Python code in 16 problems, GPT-4o-mini shows a lower Pass@1 when referring to pseudocode than solving problems directly. Our analysis of the failure cases suggests that in some cases (e.g., Appendix C.4), LLM fails to understand expressions like "cumulative sums" of an array/list indicated in the pseudocode, which is expected to be implemented with `accumulate()`. Such expressions may conversely mislead LLMs who were able to reason a correct solution from problem descriptions by themselves, particularly for easy problems. In practice, ambiguous natural language expressions are inevitable in pseudocode or instructions, despite the semi-structured format of pseudocode. It is an interesting future work to detect and fix such noise.

In general, these interesting findings help us understand the language-coding ability of LLMs. They echo our motivation to gain a clearer picture of LLMs' coding capabilities by isolating the evaluation of their problem-solving and language-coding abilities by introducing pseudocode.

4.2 RQ2: Cross-Programming Language

Recalling that pseudocode abstracts language-specific details of solutions (Section 2.2), we further investigate if pseudocode manifests generalizable effectiveness such that the pseudocode derived from solution code in a programming language can benefit LLMs in generating codes in another language. Table 2 presents the comparison.

Language-Wise. The pseudocode derived from solution codes written in any programming language (P_{lang}) effectively help LLMs gain much higher Pass@1 rates in comparison to the performance of generating from problems listed in Table 1 (e.g., 0.75~0.76 *v.s.* 0.38 Pass@1 in Python generation). The result suggests that *pseudocode can serve as a language-agnostic representation to hint LLMs about solution logic and guide LLMs generating programs in various programming languages*, which may shed light on cross-language tasks such as code translation and code search. Furthermore, we surprisingly found from the comparison among P_{lang} that *the pseudocode derived from C++ solutions help LLMs gain the highest Pass@1 in generating not only C++ but also Python and Rust programs on average*; meanwhile, pseudocode of Python solutions show inferior effectiveness for C++ and Rust generation. Our manual analysis suggests the reason may be that Python codes often implement logic with various libraries, and thereby, the detailed idea to implement some features cannot be extracted into pseudocode. As a result, when there is no available corresponding library to use in C++ and Rust, the LLMs cannot correctly implement the logic. The analysis of the lengths of pseudocode derived from different programming languages also shows the trend as indicated in the 2nd~4th columns in Table 3.

Model-Wise. The studied LLMs consistently gain improvement in Pass@1 rates with the help of pseudocode derived from any-language solution codes.

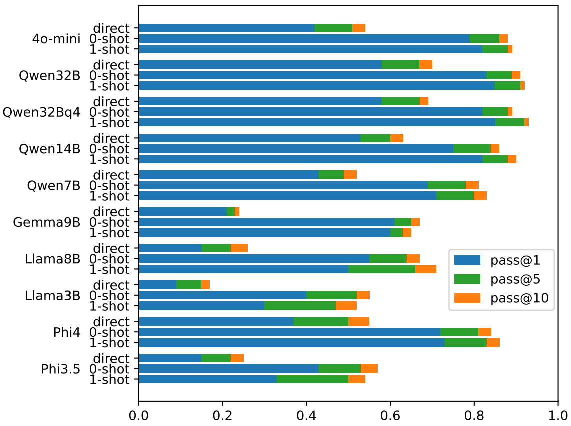


Figure 3: Zero-/one-shot Pass@{1,5,10} rates of C++ programs generated from pseudocode of C++ solutions

Meanwhile, the most helpful programming language varies across LLMs. For example, Qwen14B and QWen32B work best when referring to the pseudocode derived from the solution code written in the target programming language, GPT-4o-mini prefers pseudocode of C++ or Rust solutions, and the others prefer C++. This may indicate distinct LLMs need unique logic information implied in solution codes of specific programming languages.

4.3 RQ3: Effects of Inference Strategies

We investigate if two typical configurations, i.e., whether using in-context learning (zero-shot → one-shot) and increasing the attempts (1→5→10), help improve the performance. Figure 3 presents the Pass@{1, 5, 10} rates of LLMs when using zero-/one-shot prompts on generating C++ codes based on the pseudocode derived from C++ solutions. The results of other languages show consistent conclusions and are available at Appendix E.

Zero-shot *v.s.* One-shot. One-shot prompting benefits most LLMs but may disturb poorer LLMs like Phi-3.5-3.8B and Llama-3.1-3B which may not effectively handle long contexts. The result suggests using one-shot prompting to guide most LLMs better while driving smaller LLMs with more concise prompts. For consistency, in RQ1 and RQ2, we use one-shot prompts as a general setup for all LLMs.

Pass@{1, 5, 10}. Increasing attempts also bring more chances of generating correct codes in the setup of pseudocode-based code generation, in particular for smaller LLMs like Llama-3.2-3B. For larger LLMs, 5 attempts may be appropriate considering cost-effectiveness.

***v.s.* Generating from Problem.** (*abbr.* direct) It is clear that Pass@1 rates of all LLMs when generating from pseudocode have already surpassed the effort of 10 attempts when generating from problem descriptions. The finding again echos our conclusion that *problem-solving is the key bottleneck of current LLMs in code generation*. Besides, with pseudocode to hint the solution logic, *one* attempt enables all LLMs except Phi-3.5 and Llama3B to outperform the Pass@10 rates achieved by the commercial GPT-4o-mini generating from problems.

	Source Code			Pseudocode	
	Lang	LoC	Tokens	LoC	Tokens
Manual	C++	21.64	222.16	12.58 *(-42%)*	151.45 *(-32%)*
DeepSeek-V3	C++	21.64	222.16	18.84 *(-13%)*	172.91 *(-23%)*
DeepSeek-R1	C++	21.64	222.16	13.20 *(-39%)*	122.31 *(-45%)*
DeepSeek-R1	Rust	18.45	219.33	12.71 *(-31%)*	124.51 *(-43%)*
DeepSeek-R1	Python	15.47	156.89	11.93 *(-23%)*	111.29 *(-29%)*

Table 3: Loc and tokens of a subset (55) of LCB tasks

4.4 RQ4: Automatically-generated v.s. Manually-written pseudocode

We compare the pseudocode annotated by humans and DeepSeek-R1 on 55 sampled programming tasks. The manual annotation involved six developers with over five years of C++ coding experience; one pseudocode is annotated by an annotator and validated with another two. We mainly compare the simplicity and effectiveness of the two sets of pseudocode, which are essential qualities clear to measure based on lengths and pass rates.

Simplicity. Table 3 lists the lengths of pseudocode annotated manually and automatically in the number of BPE tokens (Meta, 2023) and lines of codes ignoring blank lines and comments. It is found that pseudocode, particularly the ones annotated by humans and DeepSeek-R1 (*abbr.* R1), are much shorter than the source codes. This suggests that *pseudocode is a simplified format to express the solution of programs*. Besides, interestingly, although the manually annotated pseudocode include fewer lines than R1-generated ones, the manual ones are found to include more tokens than R1's. The reason is R1 tends to describe the logic more concisely.

Effectiveness. We compare how effectively the automatically and manually annotated pseudocode guide LLMs in generating correct codes. As presented in Table 4, the manual pseudocode help LLMs generate slightly more compilable codes on the validation tasks. Meanwhile, surprisingly, *the pseudocode generated by R1 contribute to higher Pass@k rates than the human-written ones*. The cause may be the gap between the expression style

	Manual Pseudocode				DeepSeek-R1 Pseudocode			
	C@1	P@1	P@5	P@10	C@1	P@1	P@5	P@10
GPT-4o-mini	0.97	0.58	0.70	0.73	0.96	0.78	0.83	0.83
Qwen32B	1.00	0.65	0.71	0.71	0.99	0.81	0.86	0.86
Qwen32Bq4	0.99	0.63	0.71	0.75	0.99	0.79	0.87	0.88
Qwen14B	0.97	0.65	0.72	0.74	0.97	0.77	0.86	0.88
Qwen7B	0.94	0.55	0.66	0.69	0.92	0.64	0.75	0.78
Gemma9B	0.90	0.46	0.50	0.52	0.89	0.61	0.63	0.64
Llama8B	0.89	0.45	0.60	0.64	0.87	0.47	0.58	0.63
Llama3B	0.87	0.26	0.43	0.49	0.83	0.30	0.44	0.47
Phi4	0.95	0.60	0.70	0.73	0.94	0.63	0.75	0.80
Phi3.5	0.85	0.30	0.41	0.45	0.82	0.28	0.41	0.46
Average	0.93	0.51	0.61	0.65	0.92	0.61	0.70	0.72

Table 4: Compilation (C) and Pass (P) Rates of C++ code generation with pseudocode on 55 LCB tasks

and knowledge preferences of humans and LLMs as reported by existing studies (Gao et al., 2024).

Based on these detailed clues, we consider current SOTA reasoning LLMs like DeepSeek-R1 an effective helper to abstract reference codes into concise pseudocode with high accuracy. They may offer feasible automation to abstract existing valuable code resources in GitHub or code generation benchmarks and facilitate studies on pseudocode.

5 Related Work

Benchmarking End-to-End Code Generation. Various benchmarks have been developed to assess LLMs in end-to-end code generation – some benchmarks *broader the programming languages* to evaluate. Classical benchmarks focus on Python programming (Chen et al., 2021; Cassano et al., 2022); later, benchmarks considering other programming languages, e.g., Java (Cao et al., 2024) and even multilingual (Zheng et al., 2023), emerge. Some studies evaluate LLM programming *across different contexts*, such as class-level (Du et al., 2023), project-level (Li et al., 2024), and repository-level (Zhang et al., 2023a; Liao et al., 2024), pushing the boundaries of LLM capabilities in real-world scenarios. The performance of generating *code in different domains* also attracts studies (Zhu et al., 2024). Several recent studies explored LLMs' code-generation capabilities *incorporating external techniques*, for example, using RAG to retrieve codes (Liu et al., 2024; Chen et al., 2024) and documents (Jain et al., 2024b; Zhao et al., 2024), allowing LLMs to code with external resources.

Though these studies assess LLM's performance in various scenarios, they reveal relatively limited information about LLM's ability at steps within the end-to-end pipeline, e.g., coding a solution logic.

Benchmarking Code Generation Using Pseudocode. Only a few works have studied translating pseudocode into code. Dirgahayu et al. (2017) propose a conceptual framework that breaks down pseudocode into XML elements. Kulal et al. (2019b) explore potential mappings of pseudocode and C++ code using test cases. The SPoC dataset with 18K line-to-line mappings is built in the work. However, the fairly trivial line-by-line pseudocode may not accurately reflect the human-written pseudocode typically appearing in real-world software development. SPoC was later utilized by Acharjee et al. (2022) to train two basic deep-learning models for pseudocode-to-code translation. These studies worked on relatively small and trivial pseudocode snippets. They also barely compared the performance of code generators (in particular the advanced LLMs) or discussed the detailed abilities.

6 Insights from Study Results

❶ Code generation bottleneck differs across programming languages (PLs). One can improve end-to-end LLM programming performance for popular PLs like Python by boosting problem-solving abilities, whereas for less-trained languages like Rust, enhancing language-coding skills is crucial.

❷ Problem-solving ability may transfer across PLs, which may allow LLMs' coding performance to be improved in a unified manner across PLs.

❸ Reasoning models can effectively handle the code-to-pseudocode transformation. This enables easy creation of up-to-date benchmarks focusing on problem-solving capability, which may help relieve the current bottleneck and support cross-PL tasks.

These insights may shed light on enhancing LLMs in code generation and other cross-PL tasks, as well as guide human-LLM collaboration in the era of AI-driven low/zero-code development.

7 Conclusion

To understand the bottlenecks in end-to-end code generation for LLMs, we introduce PSEUDOEVAL, a multilingual code generation benchmark incorporating pseudocode as input, isolating the evaluation of language-coding from problem-solving capabilities. Empirical study results with PSEUDOEVAL reveal key insights about the bottlenecks identified for different programming languages, broad applicability of pseudocode across programming languages, and exceptional quality of automatically derived pseudocode.

8 Limitations

Pseudocode Samples. Due to the limited access to DeepSeek-R1, the latency of response of reasoning models, and the costs of the subsequence inference, this study only sample one pseudocode for each problem. As revealed in Section 4.4, a small portion of the generated pseudocode could be not semantic preserving and is filtered out from the final benchmark. The thorough study on whether sampling multiple pseudocode or using a majority vote mechanism can further improve the pseudocode quality is left as future work.

Problem Domain. The current PSEUDOEVAL selects subjects from LiveCodeBench and their solutions on LeetCode, which are mainly algorithmic code for programming puzzles. Although this meets the purpose of using pseudocode to present algorithms in practice, the daily software development scenarios such as implementing business logic are not covered. It is unclear whether the performance gap between problem-to-code generation and pseudocode-to-code generation is also significant in such scenarios. The future work to understanding this problem can be extending the workflow of PSEUDOEVAL to code generation benchmarks in different scenarios.

Involved Programming Languages. The programming languages studied in this paper are Python, C++, and Rust. They represent three popular imperative programming languages, with a major difference in the type system. Python is dynamic, C++ is static but weakly typed, and Rust is known for having a rigorous type checking mechanism. The results in RQ2 may shed light on similar languages such as Java, but may not apply to functional languages such as Haskell or low-resourced languages such as domain-specific languages.

References

Uzzal Kumar Acharjee, Minhazul Arefin, Kazi Mojammel Hossen, Mohammed Nasir Uddin, Md. Ashraf Uddin, and Linta Islam. 2022. Sequence-to-sequence learning-based conversion of pseudo-code to source code using neural translation approach. *IEEE Access*, 10:26730–26742.

Jialun Cao, Yuk-Kit Chan, Zixuan Ling, Wenxuan Wang, Shuqing Li, Mingwei Liu, Ruixi Qiao, Yuting Han, Chaozheng Wang, Boxi Yu, et al. 2025. How should i build a benchmark? revisiting code-related benchmarks for llms. *arXiv e-prints*, pages arXiv–2501.

Jialun Cao, Zhiyong Chen, Jiarong Wu, Shing-Chi Cheung, and Chang Xu. 2024. Javabench: A benchmark of object-oriented code generation for evaluating large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, pages 870–882. ACM.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. A scalable and extensible approach to benchmarking nl2code for 18 programming languages. *CoRR*, abs/2208.08227.

Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. 2024. Code search is all you need? improving code suggestions with code search. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 73:1–73:13. ACM.

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

Google DeepMind. 2024. Gemma 2 is now available to researchers and developers. Blog. Online; accessed 27-Jun-2024.

Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. 2023. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495*.

Teduh Dirgahayu, Sheila Nurul Huda, Zainudin Zukhri, and Chanifah Indah Ratnasari. 2017. Automatic translation from pseudocode to source code: A conceptual-metamodel approach. In *2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*, pages 122–128. IEEE.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *CoRR*, abs/2308.01861.

Xinyu Gao, Yun Xiong, Deze Wang, Zhenhan Guan, Zejian Shi, Haofen Wang, and Shanshan Li. 2024. Preference-guided refactored tuning for retrieval augmented code generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, pages 65–77. ACM.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024a. Livecodebench: Holistic and contamination free evaluation of large language models for code. *CoRR*, abs/2403.07974.

Nihal Jain, Robert Kwiatkowski, Baishakhi Ray, Murali Krishna Ramanathan, and Varun Kumar. 2024b. On mitigating code LLM hallucinations with API documentation. *CoRR*, abs/2407.09726.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019a. Spoc: Search-based pseudocode to code. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 11883–11894.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019b. Spoc: Search-based pseudocode to code. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 11883–11894.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. 2024. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 3603–3614. Association for Computational Linguistics.

Junyi Li, Xiaoxue Cheng, Wayne Xin Zhao, Jian-Yun Nie, and Ji-Rong Wen. 2023. Halueval: A large-scale hallucination evaluation benchmark for large language models. *arXiv preprint arXiv:2305.11747*.

Dianshu Liao, Shidong Pan, Xiaoyu Sun, Xiaoxue Ren, Qing Huang, Zhenchang Xing, Huan Jin, and Qinying Li. 2024. $\mathbf{A^3}$a3-codgen: A repository-level code generation framework for code reuse with local-aware, global-aware, and third-party-library-aware. *IEEE Transactions on Software Engineering*, 50(12):3369–3384.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. Graphcoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, pages 570–581. ACM.

AI Meta. 2023. Tiktoken: a fast bpe tokeniser for use with openai's models. URL.

AI Meta. 2024. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models. Blog. Online; accessed 25-Sept-2024.

Microsoft. 2024. Introducing phi-4: Microsoft's newest small language model specializing in complex reasoning. Blog. Online; accessed 13-Dec-2024.

OpenAI. 2024. Gpt-4o mini: advancing cost-efficient intelligence. Blog. Online; accessed 18-July-2024.

Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2024. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 2471–2484. Association for Computational Linguistics.

Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023b. Siren's song in the ai ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219*.

Ziyao Zhang, Yanli Wang, Chong Wang, Jiachi Chen, and Zibin Zheng. 2024. LLM hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *CoRR*, abs/2409.20550.

Shengming Zhao, Yuheng Huang, Jiayang Song, Zhijie Wang, Chengcheng Wan, and Lei Ma. 2024. Towards understanding retrieval accuracy and prompt quality in RAG systems. *CoRR*, abs/2411.19463.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, pages 5673–5684. ACM.

Qiming Zhu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Le Sun, and Shing-Chi Cheung. 2024. DOMAINEVAL: an auto-constructed benchmark for multi-domain code generation. *CoRR*, abs/2408.13204.

A Dataset

A.1 Legal Compliance and License

The problems we use are from the LiveCodeBench, and the solutions we use to generate pseudocode are from LeetCode, which are the publicly visible portions. We did not include the user-submitted solutions in our final benchmark but their extracted pseudocode. Following Hendrycks et al. (2021) and LiveCodeBench (Jain et al., 2024a), we abide by Fair Use 107: "the fair use of a copyrighted work, including such use by … scholarship, or research, is not an infringement of copyright", where fair use is determined by the purpose and character of the use, including whether such use is of a commercial nature or is for nonprofit educational purposes", "the amount and substantiality of the portion used in relation to the copyrighted work as a whole", and "the effect of the use upon the potential market for or value of the copyrighted work." The collected data in PSEUDOEVAL is used only for academic purposes. Moreover, PSEUDOEVAL is used for benchmarking, and we do not use it for training models.

A.2 Basic Stats

Table 5 shows the number of files (pseudocode) extracted from different source languages. The statistics is consistent with the sampled subset in Section 4.4. Each pseudocode corresponds to a problem in LiveCodeBench and can use its testcases to test the correctness of the code generated from the pseudocode.

Source	LoC	Tokens	#Files
C++	13.59	129.15	355
Rust	14.09	135.12	347
Python	11.64	111.93	357

Table 5: LoC and Tokens of pseudocode in PSEUDOEVAL from different sources

B Human Annotations

Six programmers with more than five years of C++ experience participate in the annotation of pseudocode on 55 sampled C++ solution code. Each annotated piece of pseudocode is validated by two other participants from the same group.

The approval from the ethics review board is exempted because the annotation procedure is not physically or mentally harmful and does not impose an intense workload in a short time. The participants have been compensated according to the local legislation. The consent to use the annotated data has been obtained from the participants.

C Case Study

C.1 Motivating Example

Listing 1 lists the full problem and Listing 2 lists the user-submitted C++ solution where the pseudocode is converted from. Note that the pseudocode simplifies the solution by replacing the map structure with a set structure.

C.2 Simplifying Common Procedures

Listing 3 shows a user-submitted C++ solution with detailed steps, and Listing 4 shows a concise but semantic-preserving pseudocode converted from the long solution. Powerful LLMs such as GPT-4o-mini and Qwen32B can implement code correctly in all three languages, while smaller LLMs such as Phi-3.5 have lower success rates and even drop to zero when writing Rust codes.

C.3 Underflow in Rust

Listing 5 shows an example of a user-submitted Rust solution with the subtraction underflow problem. Specifically, the variable `pos` is from `.len()` (line 5) and should be a `usize` (unsigned) variable. The user uses a nonstandard way to control the loop termination: when `pos` is 0 and subtracts 1 from it in the release mode, it becomes the biggest unsigned integer, so the loop terminates because `pos > arr.len()`. However, such a coding style is not encouraged in Rust. In the debug mode, the Rust program will panic (i.e., running into an invalid state because `pos` is unsigned and should not underflow) and terminate the execution.

The pseudocode generated by DeepSeeek-R1 (Listing 6) focuses on the solution logic, which does not contain such detailed type information and uses a more standard coding style (loop until `pos` is negative). Based on the pseudocode, only Qwen32B notices the possible sign problem and can generate code that correctly converts the type as `isize` (line 6, Listing 7), while all other less powerful models failed to do so.

C.4 Worsening Pseudocode

Listing 8, 9, and 10 show a case where the pseudocode generated from a Python solution misleads LLMs and causes a lower pass@1 compared with generating Python code from the problem.

C.5 Failure of Pseudocode Generation

Listing 11 and 12 show a case of a Python solution and its generated pseudocode that is not semantic preserving. The problem is at the last line, where the Python code will return `max_sum` if it is negative but not `-inf`, while the pseudocode incorrectly assumes `max_sum` to be non-negative, possibly due to the hallucination problem in LLMs.

D Prompts

Generating Pseudocode. Listing 13 is the prompt (a single user query as suggested by the DeepSeek team (Guo et al., 2025)) we use to query DeepSeek-R1 to generate pseudocode from Python code. The prompts to generate pseudocode from C++ and Rust are similar with minor difference in the example code snippets.

Generating Code from Pseudocode. Listing 14 is the zero-shot prompt, and Listing 15 is the one-shot prompt for generating Python code. The prompts to generate C++ and Rust code are similar with language difference in the one-shot example.

E Additional Experiment Results

Figure 4, 5, and 6 show the pass@k of code generation from pseudocode from C++, Python, and Rust, respectively, compared with the Pass@k of code generation from the problem.

```
You are given an integer array nums and an integer k.
An integer h is called valid if all values in the array that are strictly greater than h are identical.
For example, if nums = [10, 8, 10, 8], a valid integer is h = 9 because all nums[i] > 9 are equal to 10, but 5 is not a valid integer.
You are allowed to perform the following operation on nums:

Select an integer h that is valid for the current values in nums.
For each index i where nums[i] > h, set nums[i] to h.

Return the minimum number of operations required to make every element in nums equal to k. If it is impossible to make all elements equal to k, return -1.

Example 1:

Input: nums = [5,2,5,4,5], k = 2
Output: 2
Explanation:
The operations can be performed in order using valid integers 4 and then 2.

Example 2:

Input: nums = [2,1,2], k = 2
Output: -1
Explanation:
It is impossible to make all the values equal to 2.

Example 3:

Input: nums = [9,7,5,3], k = 1
Output: 4
Explanation:
The operations can be performed using valid integers in the order 7, 5, 3, and 1.


Constraints:

1 <= nums.length <= 100
1 <= nums[i] <= 100
1 <= k <= 100
```

Listing 1: Full problem in the motivating example

```cpp
class Solution {
public:
    int minOperations(vector<int>& nums, int k) {
        int mn = *min_element(nums.begin(), nums.end());
        if (mn < k) {
            return -1;
        }
        unordered_map<int,int> mp;
        for (auto &it: nums) {
            mp[it] = 1;
        }
        int ans = mp.size();
        if (mp[k]) {
            ans--;
        }
        return ans;
    }
};
```

Listing 2: User-submitted C++ solution to Listing 1

```cpp
class Solution {
public:
    int nonSpecialCount(int l, int r) {
        // Calculate the limit up to which we need to find prime numbers
        int lim = (int)(sqrt(r));

        // Create a boolean array to mark primes up to lim using Sieve of Eratosthenes
        vector<bool> v(lim + 1, true);
        v[0] = v[1] = false; // 0 and 1 are not prime numbers

        // Sieve of Eratosthenes to mark non-prime numbers
        for (int i = 2; i * i <= lim; i++) {
            if (v[i]) {
                for (int j = i * i; j <= lim; j += i) {
                    v[j] = false;
                }
            }
        }

        // Count special numbers in the range [l, r]
        int cnt = 0;
        for (int i = 2; i <= lim; i++) {
            if (v[i]) {
                int square = i * i;
                if (square >= l && square <= r) {
                    cnt++;
                }
            }
        }

        // Total numbers in the range [l, r]
        int totalCount = r - l + 1;

        // Calculate non-special numbers
        return totalCount - cnt;
    }
};
```

Listing 3: A C++ solution that can be simplified

```
function nonSpecialCount(l, r):
    lim = floor of sqrt(r)
    generate sieve for primes up to lim using Sieve of Eratosthenes
    cnt = count of primes i in 2..lim where i^2 is in [l, r]
    return (r - l + 1) - cnt
```

Listing 4: Pseudocode from Listing 3

```
 1 impl Solution {
 2   pub fn maximum_points(mut arr: Vec<i32>, mut curr: i32) -> i64 {
 3     arr.sort_unstable();
 4     let mut res = 0;
 5     let mut pos = arr.len() - 1;
 6 
 7     while pos <= arr.len() && arr[0] <= curr {
 8       res += (curr / arr[0]) as i64;
 9       curr %= arr[0];
10       curr += arr[pos];
11       pos -= 1;
12     }
13 
14     return res;
15   }
16 }
```

Listing 5: Example user-submitted Rust code with subtraction underflow problem (line 11)

```
function maximum_points(arr, curr)
    sort arr in ascending order
    res = 0
    pos = arr.len() - 1

    while pos >= 0 and arr[0] <= curr:
        res += curr // arr[0]
        curr = curr % arr[0]
        curr += arr[pos]
        pos -= 1

    return res
```

Listing 6: Pseudocode converted from Listing 5

```
impl Solution {
    pub fn maximum_points(enemy_energies: Vec<i32>, current_energy: i32) -> i64 {
        let mut arr = enemy_energies.clone();
        arr.sort();
        let mut res: i64 = 0;
        let mut pos = arr.len() as isize - 1;
        let mut curr = current_energy;

        while pos >= 0 && arr[0] <= curr {
            res += (curr / arr[0]) as i64;
            curr = curr % arr[0];
            if pos >= 0 {
                curr += arr[pos as usize];
            }
            pos -= 1;
        }

        res
    }
}
```

Listing 7: Correct Rust code implemented from Listing 6

```
class Solution:
    def maximumSubarraySum(self, nums: List[int], k: int) -> int:
        max_sum, prefix_sum, val_to_min_prefix_sum = -inf, 0, defaultdict(lambda: inf)
        for i, num in enumerate(nums):
            if val_to_min_prefix_sum[num] > prefix_sum:
                val_to_min_prefix_sum[num] = prefix_sum
            prefix_sum += num
            max_sum = max(max_sum, prefix_sum - val_to_min_prefix_sum[num + k], prefix_sum -
                ↪ val_to_min_prefix_sum[num - k])
        return max_sum if max_sum > -inf else 0
```

Listing 8: Python code whose converted pseudocode has negative impact

```
function circularGameLosers(n, k):
    s = {1, 2, ..., n}
    for each mul in cumulative sums of integers from 0 to n-1:
        position = (k * mul) mod n + 1
        if position not in s:
            break
        remove position from s
    return s as a list
```

Listing 9: Pseudocode converted from Listing 8

```
class Solution:
    def circularGameLosers(self, n: int, k: int) -> List[int]:
        players = set(range(1, n + 1))
        mul = 0
        while True:
            position = (k * mul) % n + 1
            if position not in players:
                break
            players.remove(position)
            mul += 1
        return sorted(players)
```

Listing 10: Wrong Python code generated from Listing 9

```
class Solution:
    def maximumSubarraySum(self, nums: List[int], k: int) -> int:
        max_sum, prefix_sum, val_to_min_prefix_sum = -inf, 0, defaultdict(lambda: inf)
        for i, num in enumerate(nums):
            if val_to_min_prefix_sum[num] > prefix_sum:
                val_to_min_prefix_sum[num] = prefix_sum
            prefix_sum += num
            max_sum = max(max_sum, prefix_sum - val_to_min_prefix_sum[num + k], prefix_sum -
                ↪ val_to_min_prefix_sum[num - k])
        return max_sum if max_sum > -inf else 0
```

Listing 11: Python code being wrongly converted into pseudocode

```
 1  function maximumSubarraySum(nums, k):
 2      max_sum = -infinity
 3      prefix_sum = 0
 4      val_to_min_prefix = empty map // default value is infinity
 5      for each num in nums:
 6          if val_to_min_prefix[num] > prefix_sum:
 7              val_to_min_prefix[num] = prefix_sum
 8          prefix_sum += num
 9          candidate1 = prefix_sum - val_to_min_prefix[num + k]
10          candidate2 = prefix_sum - val_to_min_prefix[num - k]
11          max_sum = max(max_sum, candidate1, candidate2)
12      return max(max_sum, 0)
```

Listing 12: Pseudocode converted from Listing 11 with errors (line 12)

~~~
I am a Python programmer.
Please help me convert Python code into a semantic-preserving and concise pseudocode.
Instead of translating line by line, you should simplify the pseudocode as much as possible
    ↪ and also readable.
Below are specific rules:

1. Use indents to represent control structures.
```
if a == b:
    c += 1
```

2. The pseudocode should not be tied to a specific programming language and should not
    ↪ contain any language-specific stuffs such as `yield` in Python.

3. The pseudocode does not need to preserve concrete type info: (a) The concrete names such
    ↪ as `vector` and `i64` should not appear. Usually, general names such as array/list
    ↪ and int are enough for describing algorithms. (b) Do not involve type casting.

4. You should omit the implementation of common algorithms/data structures/operations.

For example, the customized binary search subroutine
```
def search_square_geq(nums, val):
    left = 0
    right = len(nums) - 1
    while left < right:
        mid = left + (right - left) // 2
        if nums[mid]**2 < val:
            left = mid + 1
        else:
            right = mid
    return left

target = search_square_geq(xs, 9)
```
can be simplified as
```
target = binary search for the index i such that xs[i] * xs[i] >= 9
```

5. You can use natural language to simplify code, in particular loops. For example,
```
for x in xs:
    if x == 233:
        flag = true
```
can be simplified as `flag = whether 233 exists in xs`

6. Do not use natural language if that is verbose. For example, `let n be the size of list_a`
    ↪  is less compact and readable than `n = list_a.size()`

7. A function definition should be formatted like `function max(a, b)`. Functions can be
    ↪ nested and can use variables in the outer scope.

Finally, recall that the principles are **semantic-preserving** and **concise and readable**.
Do not change the name of the given function.
You can iterate the writing of pseudocode to ensure it follows the above rules.
Wrap only the final version with code blocks (```) in the response.

Below is the Python code to convert into pseudocode.
{code}
~~~

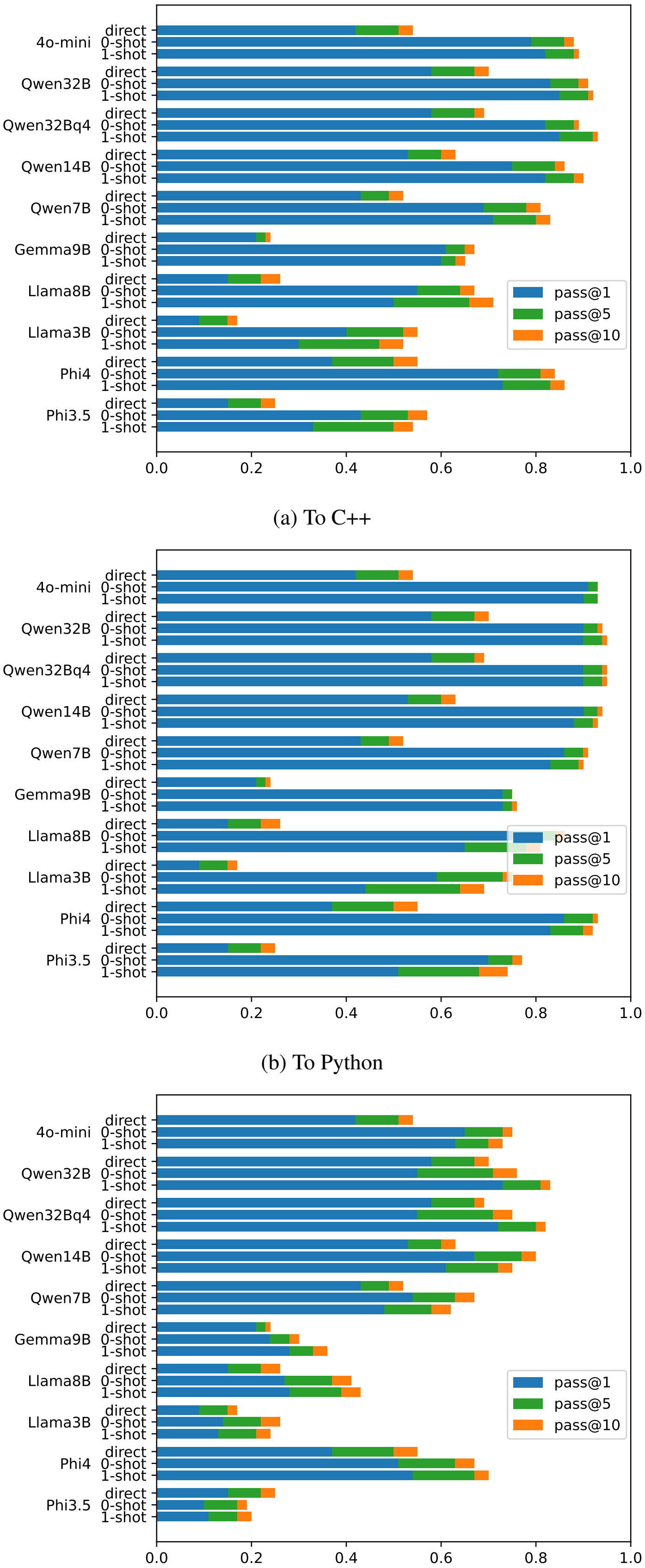
Listing 13: Prompt (a single user query) to generate pseudocode from DeepSeek-R1

```
===System===
You are a proficient {lang} programmer and familiar with various algorithms.
Your task is to implement a {lang} code given a pseudocode illustrating an algorithm and a {
    ↪ lang} signature.
If some subroutines are provided in the standard library, you can use them.
For subroutines that are not provided in the standard library, you need to implement them.

===User===
```
{pseudocode}
```

Please implement the above pseudocode with the following template.
```
{signature}
```
```

Listing 14: Zero-shot prompt to generate code from pseudocode

~~~
===System===
You are a proficient Python programmer and familiar with various algorithms.
Your task is to implement a Python code given a pseudocode illustrating an algorithm and a
    ↪ Python signature.
If some subroutines are provided in the standard library, you can use them.
For subroutines that are not provided in the standard library, you need to implement them.
Below is a simple example.

===User===
```pseudocode
Input:
    nums is a list of positive integers;
    k is a positive integer
Output:
    return x such that x is the smallest in nums and x * x > k
function findTarget(nums, k):
    sorted = sort nums in ascending order
    res = binary search on sorted for the leftmost value res such that res * res > k
    return res
```

Please implement the function with the following template.
```python
class Solution:
    def findTarget(self, nums: List[int], k: int) -> int:

```

===Assistant===
The pseudocode involves two subroutines, sort and binary search.
Sort can be implemented using the standard libray, and binary search needs to be implemented
    ↪ by ourselves.
Finally, the Python implementation is as follows.

```python
class Solution:
    def findTarget(self, nums: List[int], k: int) -> int:
        nums = sorted(nums)
        left = 0
        right = len(nums) - 1
        while left < right:
            mid = left + (right - left) // 2
            if nums[mid] * nums[mid] <= k:
                left = mid + 1
            else:
                right = mid
        return nums[left]
```

===User===
```
{pseudocode}
```

Please implement the function with the following template.
```
{signature}
```
~~~

Listing 15: One-shot prompt to generate code from pseudocode

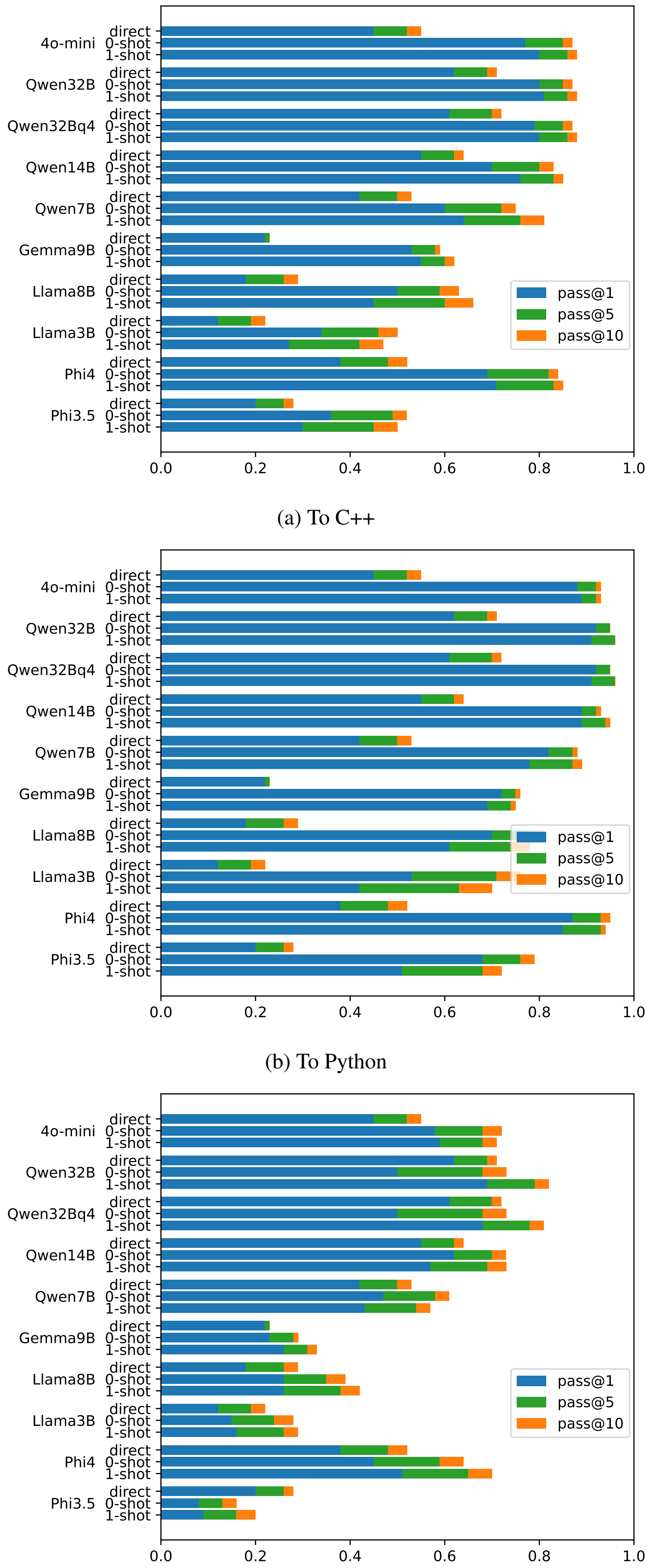


(a) To C++

(b) To Python

(c) To Rust

Figure 4: Pass@k of code generation from pseudocode from C++ to all languages, compared with direct generation from problems

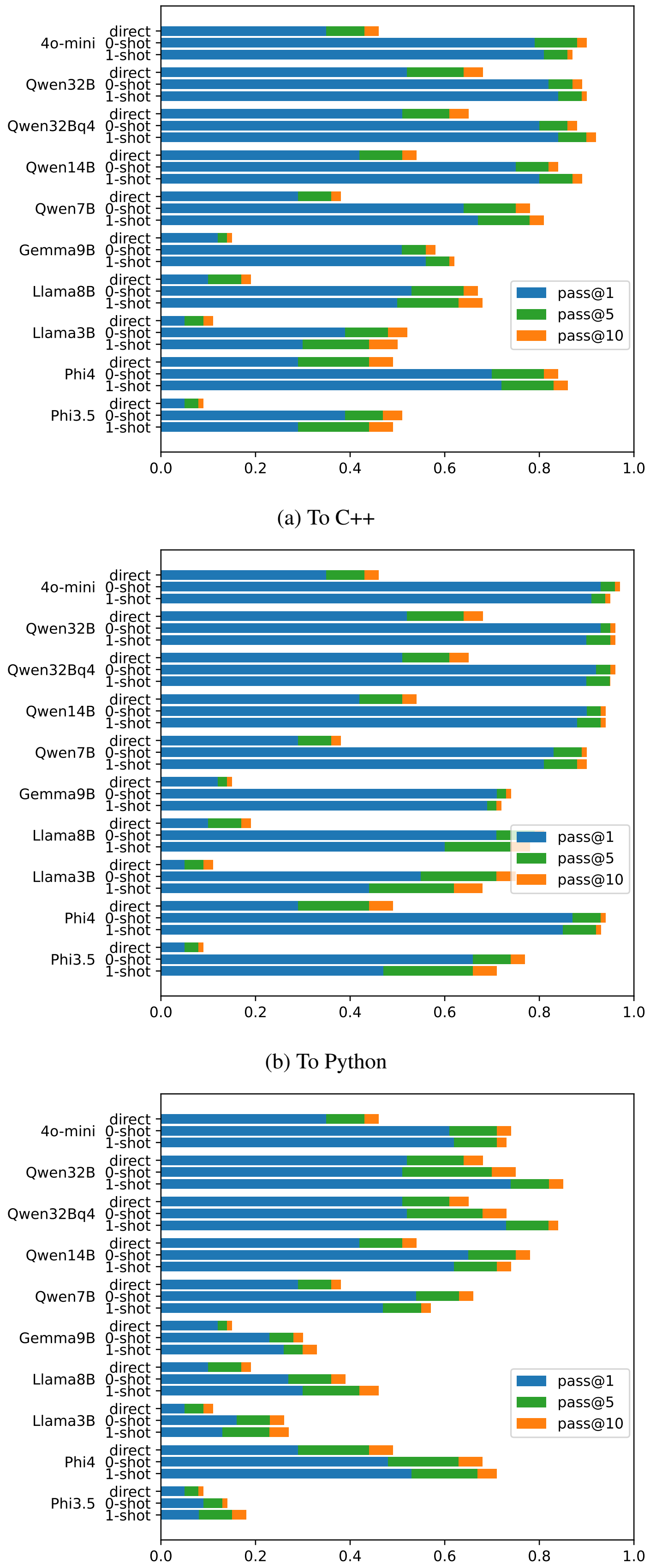


(a) To C++

(b) To Python

(c) To Rust

Figure 5: Pass@k of code generation from pseudocode from Python to all languages, compared with direct generation from problems



(a) To C++

(b) To Python

(c) To Rust

Figure 6: Pass@k of code generation from pseudocode from Rust to all languages, compared with direct generation from problems