

An Empirical Study of Problem-Aligned Multilingual Code Generation and Directed Code Translation by Large Language Models

Quanhe Wang 

Shenzhen Institute of Technology, Xidian University, Shenzhen, China
School of Computer Science and Technology, Xidian University, Xi'an, China

Cheng Wen¹ 

Shenzhen Institute of Technology, Xidian University, Shenzhen, China

Dugang Liu 

College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China

Xingjian Han¹ 

China Electronic Product Reliability and Environmental Testing Research Institute (The fifth electronic research institute of MIIT), Guangzhou, China

Bin Yu 

School of Computer Science and Technology, Xidian University, Xi'an, China

Ping Chen 

China Electronic Product Reliability and Environmental Testing Research Institute (The fifth electronic research institute of MIIT), Guangzhou, China

Shengchao Qin 

School of Computer Science and Technology, Xidian University, Xi'an, China

Cong Tian 

School of Computer Science and Technology, Xidian University, Xi'an, China

Abstract

Background. Large language models (LLMs) can generate executable code, but their reliability across programming languages remains difficult to characterize.

Aims. We investigate multilingual code capability under problem alignment, focusing on functional reliability, cross-language consistency, translation behavior, problem and sampling effects, and execution quality.

Method. We present MCIBENCH, covering 1,489 programming problems and eight languages, of which 1,466 have execution-validated coverage in all eight languages. We evaluate representative LLMs through multilingual generation and analyze source-code-conditioned translation on 100 randomly selected all-language-covered problems.

Results. Performance varies substantially across models and target languages, while source-code-conditioned translation also exhibits marked variation across source-target settings. Problem difficulty and candidate budget materially affect correctness, strong aggregate performance does not necessarily imply cross-language consistency, and accepted programs differ in runtime and memory efficiency.

Conclusions. Aggregate pass@*k* alone is insufficient for characterizing multilingual code capability, motivating evaluation that considers language consistency, problem sensitivity, sampling behavior, translation settings, and post-correctness execution quality.

2012 ACM Subject Classification Software and its engineering → General programming languages; Computing methodologies → Artificial intelligence

¹ **Corresponding author:** Cheng Wen and Xingjian Han.



© Quanhe Wang, Cheng Wen, Dugang Liu, Xingjian Han, Bin Yu, Ping Chen, Shengchao Qin, and Cong Tian;

licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 25; pp. 25:1–25:21



Leibniz International Proceedings in Informatics

LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases Large language models, code generation, code translation, multilingual programming, empirical software engineering

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.25

Category Technical Track Paper

Acknowledgements This work was supported in part by the National Natural Science Foundation of China (Nos. 62472339, 62302375), the Basic Research Foundation of Shenzhen City (No. JCYJ20250604184202003), and the Open Project of China Electronic Product Reliability and Environmental Testing Research Institute (The Fifth Electronic Research Institute of MIIT).

1 Introduction

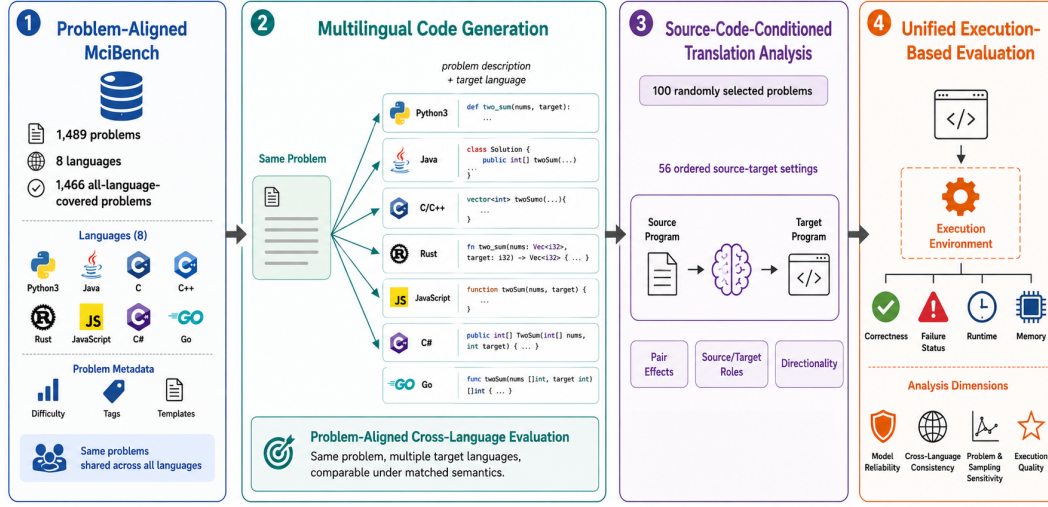
Large language models (LLMs) have shown strong capabilities in generating executable programs from natural-language descriptions [8, 2]. However, their reliability across programming languages remains difficult to characterize [6]. Many existing evaluations either focus on a single language or aggregate results over heterogeneous problem sets [4, 1, 5, 12]. Such results provide useful overall indicators, but they make it difficult to determine whether a model solves the same problems consistently across languages, how its performance varies with problem characteristics and candidate budget, and whether functionally correct programs exhibit comparable execution quality.

A key challenge is *problem comparability*. When different languages are evaluated on different problem sets, observed performance gaps may reflect task composition rather than target-language effects. We therefore adopt a problem-aligned design in which the same underlying programming problems are evaluated across target languages under a unified execution-based protocol. This design enables multilingual behavior to be compared under matched problem semantics and supports analyses beyond aggregate pass@ k , including language consistency, failure behavior, sampling effects, and post-correctness execution quality.

To support this study, we construct MCIBENCH, a problem-aligned evaluation infrastructure containing 1,489 algorithmic programming problems across eight languages: Python3, Java, C, C++, Rust, JavaScript, C#, and Golang. Among them, 1,466 problems have execution-validated program coverage in all eight languages. MCIBENCH integrates problem descriptions, difficulty labels, algorithmic tags, language-specific templates, generated outputs, executable submissions, and execution records, allowing model behavior to be analyzed consistently across problems and languages.

Figure 1 summarizes the evaluation setting. Our primary study focuses on multilingual code generation, where the same problem description is used to generate solutions in each target language. We additionally randomly select 100 all-language-covered problems for a complementary analysis of source-code-conditioned translation across 56 ordered source-target settings. This analysis examines how translation performance varies across source-target combinations, language roles, and reversed translation directions. All generated and translated programs are evaluated through execution, with correctness, failure status, runtime, and memory recorded when available.

We evaluate representative advanced LLMs together with a Qwen3 scaling series. Our analyses examine overall reliability and failure modes, consistency across target languages, sensitivity to problem difficulty and algorithmic characteristics, the benefit of additional candidates, source-code-conditioned translation behavior, and runtime-memory efficiency of accepted programs. The results show substantial variation across models and languages, and



■ **Figure 1** Problem-aligned multilingual evaluation for code generation and code translation.

demonstrate that strong aggregate pass@ k does not necessarily imply consistent multilingual behavior. Problem difficulty and candidate budget also materially affect correctness, while accepted programs can differ in execution efficiency even when they are functionally correct.

This paper makes three contributions:

- We construct MciBENCH, a problem-aligned multilingual code evaluation infrastructure containing 1,489 programming problems across eight languages, including 1,466 problems with execution-validated coverage in all eight languages.
- We provide a unified execution-based pipeline that integrates multilingual code generation, source-code-conditioned translation, failure categorization, candidate sampling, and runtime/memory collection.
- We conduct a large-scale empirical study of multilingual code capability across models, target languages, translation settings, problem characteristics, candidate budgets, and post-correctness execution quality.

2 Related Work

2.1 General Code Generation Evaluation

LLMs have demonstrated strong capabilities in generating executable code from natural-language requirements [21, 19]. Benchmarks such as HumanEval [4] and MBPP [1] evaluate functional correctness through execution on controlled programming tasks. While these benchmarks have established widely used evaluation protocols, they primarily focus on individual programming languages and provide limited support for studying whether the same model behaves consistently on the same problems across multiple languages.

2.2 Multilingual and Cross-Language Code Evaluation

Recent benchmarks extend code evaluation to multilingual settings. HumanEval-based resources, including HumanEval-X [29], HumanEvalPack [16], MultiPL-E [3], HumanEval-XL [17], and mHumanEval [18], provide code-generation tasks in multiple programming languages. Other benchmarks, such as XLCost [30], XCODEEVAL [10], and PseudoEval [26],

■ **Table 1** Comparison with representative code generation and code translation benchmarks.

Benchmark	Year	Scale	#Lang.	Aligned Coverage	Code Gen.	Code Trans.
MBPP [1]	2021	974 tasks	1	–	✓	–
HumanEval [4]	2021	164 problems	1	–	✓	–
XLCoST [30]	2022	901 problems	7	Partial	✓	✓
HumanEval-X [29]	2023	164 problems	5	164 aligned	✓	–
HumanEvalPack [16]	2023	164 problems	6	164 aligned	✓	–
MultiPL-E [3]	2023	1,138 problems	18	Partial	✓	–
G-TransEval [9]	2023	400 pairs	5	–	–	✓
HumanEval-XL [17]	2024	80 problems	12	80 aligned	✓	–
XCODEEVAL [10]	2024	7.5K problems	11	Partial	✓	✓
mHumanEval [18]	2025	164 problems	25	164 aligned	✓	–
PseudoEval [26]	2025	357 problems	3	333 aligned	✓	–
McIBench	2026	1,489 problems	8	1,466 all-language-covered	✓	✓

*) “Aligned Coverage” indicates whether the same programming problem has corresponding multilingual solutions or execution-validated program coverage across languages. “Partial” means that multilingual solutions are available only for some problems, languages, or language pairs.

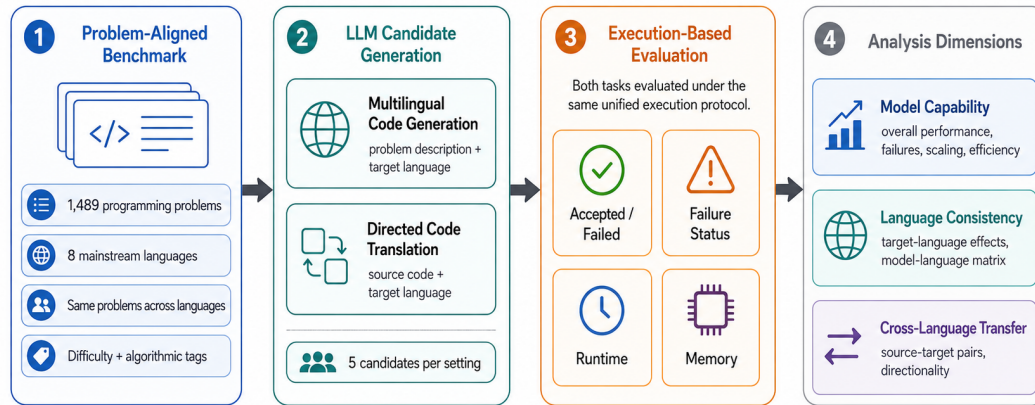
further support multilingual generation, cross-language tasks, or aligned evaluation settings. As summarized in Table 1, these resources vary considerably in scale, language coverage, problem alignment, and supported evaluation tasks.

Problem alignment is particularly important for cross-language comparison because performance differences measured on different problem sets may reflect task composition rather than programming-language effects. MCIBENCH therefore evaluates the same underlying problems across target languages, enabling direct comparison under matched problem semantics. It additionally supports source-code-conditioned translation on problems with multilingual program coverage, allowing generation and translation to be studied under a common execution-based framework. Prior translation benchmarks such as G-TransEval [9] provide complementary support for evaluating cross-language program transformation.

2.3 Execution-Based and Quality-Aware Evaluation

Execution-based evaluation is central to code generation and translation because textual similarity is a weak proxy for functional correctness [4, 1, 11]. Metrics such as $\text{pass}@k$ measure whether generated programs satisfy executable tests rather than whether they resemble reference implementations [4]. In multilingual settings, execution also exposes language-specific failures related to compilation, type systems, libraries, runtime behavior, and submission interfaces. LLMs have also been explored for software-quality analysis beyond code generation, including large-scale inspection of static bug warnings [22, 20]. Beyond execution-based testing, recent LLM-assisted formal-methods research has explored translating natural-language requirements into LTL specifications and synthesizing program specifications through static analysis and neuro-symbolic reasoning [13, 14, 23, 24].

Functional correctness, however, does not fully characterize generated code. Accepted programs may still differ in runtime and memory consumption, motivating post-correctness evaluation of execution quality. Compared with existing benchmarks, MCIBENCH combines problem-aligned multilingual generation with execution records that support failure analysis, candidate-sampling analysis, source-code-conditioned translation analysis, and runtime-memory evaluation of accepted programs.



■ **Figure 2** Overview of the MCIBENCH evaluation workflow and analysis dimensions.

3 Study Design

3.1 Overview

Figure 2 summarizes the MCIBENCH evaluation workflow. Starting from a problem-aligned benchmark, we evaluate LLMs on two programming tasks: multilingual code generation and source-code-conditioned translation. For both tasks, each model generates multiple candidates, which are converted into executable submissions and evaluated under the same execution-based protocol. The resulting records capture functional correctness, failure status, runtime, and memory, supporting analyses from model-, language-, problem-, and execution-quality perspectives.

Our primary evaluation focuses on multilingual code generation, where the same programming problems are evaluated across eight target languages under matched problem semantics. We further conduct a complementary source-code-conditioned translation analysis on a randomly selected set of all-language-covered problems. Using the same 100 problems across all 56 ordered source-target settings enables controlled analysis of source-target variation, language roles, and translation directionality.

3.2 Benchmark Construction

MCIBENCH organizes executable programming problems, multilingual program coverage, language-specific templates, model outputs, and execution records under a unified evaluation framework. For each problem, we store its platform identifier, description, difficulty label, algorithmic tags, and language-specific submission templates.

Starting from 2,029 collected problem records, we retain 1,489 algorithmic programming problems covering Python3, Java, C, C++, Rust, JavaScript, C#, and Golang. The retained set contains 481 Easy, 723 Medium, and 285 Hard problems. Problems are retained if they support executable submission in the selected languages and can be evaluated through a stable submission-and-check protocol.

For multilingual code generation, all 1,489 problems are evaluated across the eight target languages using the same problem identities and execution protocol. Because generated candidates are checked directly through execution, generation does not require an existing reference implementation in every target language.

To support code translation, MCIBENCH further maintains multilingual program implementations across languages. As shown in Table 2, the initially collected reference solutions are highly uneven across languages, and only 331 problems have reference implementations in

■ **Table 2** Coverage of initially collected reference solutions in MCIBENCH before execution-validated solution enhancement.

Per-language solution coverage			Per-problem solution alignment	
Language	#Problems	Coverage	#Collected Lang.	#Problems
C++	1,448	97.2%	1	10
Python3	1,353	90.9%	2	168
Java	1,338	89.9%	3	98
JavaScript	1,163	78.1%	4	81
C	1,120	75.2%	5	212
C#	1,014	68.1%	6	420
Golang	568	38.1%	7	169
Rust	371	24.9%	8	331

■ **Table 3** Overview of MCIBENCH.

Item	Description
Problems	1,489 algorithmic programming problems
Difficulty	481 Easy, 723 Medium, 285 Hard
Languages	Python3, Java, C, C++, Rust, JavaScript, C#, Golang
Code generation	1,489 problems $\times$ 8 target languages
Initial all-reference coverage	331 problems with solutions in all eight languages
All-language coverage	1,466 execution-validated problems
Translation analysis	100 randomly selected problems $\times$ 56 ordered source-target settings
Candidates	5 candidates per setting
Evaluation protocol	Execution-based submission and checking
Evaluation outputs	Correctness, failure status, runtime, memory
Metadata	Difficulty levels and algorithmic tags

all eight languages. We therefore expand multilingual coverage through execution-validated solution enhancement, retaining additional problem-language implementations only after they pass the execution-based checker. This process results in 1,466 problems with validated program coverage in all eight languages. For the translation analysis in this study, we randomly select 100 problems from this all-language-covered set.

With eight languages, each selected problem supports 56 ordered source-target translation settings. For the translation analyses reported in this paper, the same 100 problems are evaluated across all 56 ordered source-target settings, enabling controlled comparison of translation behavior across language combinations. Table 3 summarizes the benchmark scale and evaluation settings.

3.3 Evaluation Tasks

Task 1: Multilingual Code Generation. Given a problem description and a target language, the model generates candidate solutions in the requested language. The same problem identities are evaluated across all eight target languages, enabling direct comparison of multilingual behavior under matched problem semantics.

Task 2: Source-Code-Conditioned Translation. Given an execution-validated source-language implementation and a target language, the model generates a semantically equivalent program in the target language. The eight languages define 56 ordered source-target settings. In this study, we analyze these translation results across models, source-target combinations, source/target language roles, and reversed translation directions.

■ **Table 4** Evaluated models.

Model	Role	Generation	Translation
qwen3-coder-480b-a35b-instruct	Cross-model comparison	✓	✓
gpt-4o	Cross-model comparison	✓	✓
glm-4	Cross-model comparison	✓	✓
deepseek-v3.2	Cross-model comparison	✓	✓
claude-3-5-haiku-20241022	Cross-model comparison	✓	✓
qwen3-32b	Scaling analysis	✓	–
qwen3-14b	Scaling analysis	✓	–
qwen3-8b	Scaling analysis	✓	–
qwen3-4b	Scaling analysis	✓	–
qwen3-1.7b	Scaling analysis	✓	–

*) For qwen3-coder-480b-a35b-instruct, the model has 480B total parameters, with 35B parameters activated during inference.

3.4 Research Questions

We organize the empirical study around five research questions.

RQ1: How reliable are LLMs in problem-aligned multilingual code generation?

We examine overall functional correctness, model scaling, and failure profiles in the full generation setting.

RQ2: How consistently do LLMs solve the same problems across programming languages? We analyze target-language performance and model-language variation under aligned problem semantics.

RQ3: How does source-code-conditioned translation vary across models and language settings? We analyze overall translation performance, source-target pair effects, source/target language roles, and differences between reversed translation directions on the 100-problem translation set.

RQ4: How sensitive is multilingual code-generation reliability to problem characteristics and candidate budget? We study performance across difficulty levels and algorithmic topics, together with the benefit of generating multiple candidates.

RQ5: Does functional correctness imply comparable execution quality? We compare runtime and memory characteristics among accepted generated programs to examine post-correctness execution quality.

4 Experimental Setup

4.1 Evaluated Models

Table 4 summarizes the evaluated models. Five representative advanced LLMs are evaluated on both multilingual generation and source-code-conditioned translation. We additionally evaluate a Qwen3 scaling series on multilingual generation to examine how model scale affects cross-language coding performance.

4.2 Candidate Generation

Each model generates five candidates per evaluation setting, enabling empirical pass@ k evaluation and analysis of candidate-budget effects. For multilingual generation, the prompt contains the problem description and requested target language. For translation, the prompt contains an execution-validated source implementation and the requested target language. The complete prompt templates, decoding parameters, and model invocation configurations are included in the accompanying artifact.

Model outputs are post-processed into executable submissions using the corresponding target-language templates. If fenced code blocks are present, we select the first block matching the target language; otherwise, we use the first complete code block or cleaned raw output. Post-processing only extracts code and fills submission templates; it does not repair program logic or manually correct generated outputs. Outputs that cannot be converted into executable submissions are treated as failed candidates.

Following prior execution-based code evaluation [4], empirical $\text{pass}@k$ is the proportion of evaluation instances for which at least one of the first k generated candidates passes the execution-based tests. Unless otherwise stated, we report $\text{pass}@1$ and $\text{pass}@5$.

4.3 Execution and Result Collection

Each generated or translated program is submitted through the target-language execution interface under a common submission-and-check protocol. A candidate is accepted only if it passes all available target-language tests. For failed candidates, we record the returned status, including wrong answer, compile error, runtime error, time limit exceeded, memory limit exceeded, and output limit exceeded. For accepted candidates, runtime and memory are recorded when available.

The same execution protocol is applied across models, tasks, and target languages. The resulting records include raw model outputs, extracted submissions, execution status, failure type, runtime, and memory measurements.

4.4 Statistical Analysis

The evaluation unit depends on the task. For multilingual generation, an instance is a problem-target-language pair. For translation, an instance is a problem-source-target triple. We report aggregate pass rates over the corresponding instances and use 95% Wilson confidence intervals for the main $\text{pass}@5$ results [25].

For comparisons across models and target languages, we use the same underlying problem instances whenever possible to reduce confounding from problem composition. For the translation analyses in RQ3, the same 100 problems are evaluated across all 56 ordered source-target settings. Pair-level, role-level, and direction-level results are therefore computed over matched problem identities, reducing variation caused by different problem compositions.

Runtime and memory analyses are conducted only on accepted generated programs. Because raw measurements are not directly comparable across problems or target languages, we normalize them within each problem-language group, comparing accepted programs only with other accepted programs solving the same problem in the same target language. Confidence intervals for the normalized efficiency results are estimated by bootstrap resampling. These measurements are interpreted as relative, platform-reported post-correctness indicators rather than machine-independent performance measures.

5 RQ1: Reliability of Multilingual Code Generation

RQ1 examines the functional reliability of LLMs in problem-aligned multilingual code generation. We first compare overall correctness across models and then examine how unsuccessful candidates fail. Target-language effects, candidate-budget effects, problem characteristics, and execution efficiency are analyzed in subsequent RQs.

■ **Table 5** Overall results for multilingual code generation. Each model generates 59,560 candidates.

Model	Accepted	Cand. Acc.	Pass@1	Pass@5	Pass@5 95% CI
qwen3-coder-480b-a35b-instruct	53,022	89.0%	89.1%	92.1%	[91.6, 92.6]
gpt-4o	46,050	77.3%	77.2%	88.6%	[88.0, 89.2]
qwen3-32b	44,608	74.9%	74.7%	87.6%	[87.0, 88.2]
glm-4	40,073	67.3%	67.4%	82.6%	[81.9, 83.3]
deepseek-v3.2	27,855	46.8%	46.4%	77.8%	[77.0, 78.5]
claude-3-5-haiku-20241022	38,855	65.2%	65.1%	77.2%	[76.4, 77.9]
qwen3-14b	31,790	53.4%	53.1%	64.6%	[63.7, 65.5]
qwen3-8b	29,942	50.3%	50.2%	52.1%	[51.2, 53.0]
qwen3-4b	24,645	41.4%	41.4%	43.2%	[42.3, 44.1]
qwen3-1.7b	13,686	23.0%	22.9%	24.1%	[23.3, 24.9]

5.1 Overall Model Performance

Each model is evaluated on 1,489 problems and eight target languages, with five candidates generated for each problem-language pair. This yields 11,912 evaluation instances and 59,560 generated candidates per model. We report candidate-level acceptance rate together with empirical pass@1 and pass@5. The 95% confidence intervals in Table 5 are Wilson intervals computed over the instance-level pass@5 outcomes.

Table 5 shows substantial differences in multilingual generation reliability. qwen3-coder-480b-a35b-instruct achieves the highest pass@5 at 92.1%, followed by gpt-4o at 88.6% and qwen3-32b at 87.6%. At the other end of the spectrum, qwen3-1.7b reaches only 24.1%. Within the Qwen3 scaling series, pass@5 increases consistently with model size, from 24.1% for qwen3-1.7b to 87.6% for qwen3-32b, indicating a strong association between model scale and multilingual code-generation reliability.

Candidate-level acceptance and pass@5 capture different aspects of performance. For example, deepseek-v3.2 accepts 46.8% of individual candidates, while its instance-level pass@5 reaches 77.8%. This difference illustrates that candidate-level success rates and best-of- k reliability should not be interpreted interchangeably. We analyze the effect of candidate budget in detail in RQ4.

5.2 Failure Mode Analysis

Pass@ k measures whether a model eventually produces an accepted program, but does not explain why unsuccessful candidates fail. We therefore categorize rejected candidates according to the execution status returned by the evaluation platform: wrong answer, compile error, runtime error, time limit exceeded, memory limit exceeded, and output limit exceeded.

Figure 3 reveals heterogeneous failure profiles across models. Wrong answers account for a substantial fraction of failures, particularly for several smaller Qwen3 models, and may arise from incorrect algorithms, missed corner cases, or misunderstood problem constraints. Compile and runtime errors instead reflect failures in realizing an executable target-language implementation, such as syntax, typing, library, or interface mismatches. Resource-limit failures indicate another class of problems related to computational or implementation efficiency. Thus, similar aggregate pass rates can arise from qualitatively different failure patterns.

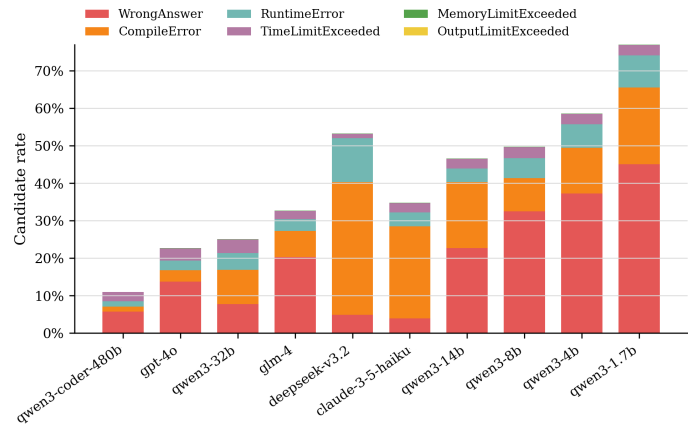


Figure 3 Failure type distribution in multilingual code generation.

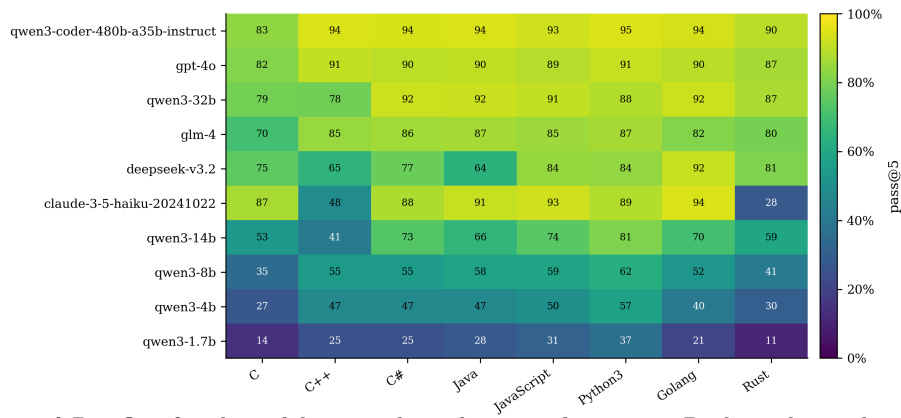


Figure 4 Pass@5 of each model across the eight target languages. Darker colors indicate higher pass@5.

Finding 1 (RQ1). LLMs exhibit substantial differences in problem-aligned multilingual code-generation reliability. Within the Qwen3 scaling series, larger models consistently achieve higher pass@5. Failure analysis further shows that unsuccessful generations arise from distinct semantic, implementation, and resource-related errors, demonstrating that aggregate correctness alone does not fully characterize model reliability.

6 RQ2: Multilingual Consistency across Programming Languages

RQ2 examines whether LLMs behave consistently across programming languages when solving the same underlying problems. Because all 1,489 problems are evaluated in the same eight target languages, observed differences can be analyzed under aligned problem semantics rather than heterogeneous task composition. We first examine target-language performance and then compare the cross-language consistency of individual models.

6.1 Target-Language Performance

Figure 4 presents the model-language pass@5 matrix. Performance varies substantially across target languages, and the magnitude of these differences is strongly model-dependent. Some models remain relatively stable across languages, whereas others exhibit pronounced weaknesses on particular targets.

■ **Table 6** Average generation performance by target language.

Language	Tasks	Pass@1	Pass@5	95% CI	Gain
Python3	14,890	66.3%	77.0%	[76.3, 77.7]	10.7%
JavaScript	14,890	66.4%	74.9%	[74.2, 75.6]	8.5%
C#	14,890	62.6%	72.8%	[72.1, 73.5]	10.1%
Golang	14,890	63.9%	72.6%	[71.9, 73.3]	8.7%
Java	14,890	62.4%	71.7%	[71.0, 72.4]	9.3%
C++	14,890	50.1%	63.0%	[62.2, 63.8]	12.9%
C	14,890	50.4%	60.7%	[59.9, 61.5]	10.3%
Rust	14,890	48.0%	59.2%	[58.4, 60.0]	11.2%

■ **Table 7** Cross-language consistency in multilingual code generation.

Model	Mean Pass@5	Weakest Lang.	Strongest Lang.	Range	Std.
qwen3-coder-480b-a35b-instruct	92.1%	C (83.4%)	Python3 (94.6%)	11.2%	3.6%
gpt-4o	88.6%	C (81.6%)	Python3 (91.3%)	9.7%	3.0%
qwen3-32b	87.6%	C++ (78.4%)	C# (92.2%)	13.8%	5.3%
glm-4	82.6%	C (70.2%)	Python3 (86.7%)	16.5%	5.2%
deepseek-v3.2	77.8%	Java (64.2%)	Golang (91.6%)	27.4%	8.8%
claude-3-5-haiku-20241022	77.2%	Rust (27.8%)	Golang (94.3%)	66.5%	23.3%
qwen3-14b	64.6%	C++ (40.7%)	Python3 (80.6%)	39.9%	12.2%
qwen3-8b	52.1%	C (35.1%)	Python3 (62.4%)	27.3%	8.7%
qwen3-4b	43.2%	C (27.3%)	Python3 (57.3%)	30.0%	9.5%
qwen3-1.7b	24.1%	Rust (10.6%)	Python3 (36.7%)	26.1%	8.0%

Table 6 summarizes generation performance averaged over the evaluated models. Python3 achieves the highest average pass@5 at 77.0%, followed by JavaScript, C#, Golang, and Java, while C++, C, and Rust obtain lower scores. Rust has the lowest average pass@5 at 59.2%, producing a 17.8-point gap from Python3. Because the same problem set is evaluated in every language, this difference cannot be explained simply by variation in problem composition.

The observed target-language differences likely reflect several interacting factors, including language syntax and type constraints, library and interface conventions, evaluation templates, and the models’ familiarity with different programming ecosystems. For example, lower performance on C, C++, and Rust may be associated with stricter compilation requirements or more explicit implementation constraints. However, these results should not be interpreted as intrinsic rankings of programming-language difficulty.

The benefit of additional candidates also differs across target languages. For example, C++ exhibits a larger pass@1-to-pass@5 gain than JavaScript or Golang. We defer a systematic analysis of candidate-budget effects to RQ4.

6.2 Model-Level Cross-Language Consistency

Language-level averages do not reveal whether an individual model performs reliably across all target languages. A model may achieve a high overall pass@5 while still exhibiting a substantial weakness in one or more languages. We therefore characterize each model using its mean pass@5, strongest and weakest target languages, range, and standard deviation across the eight languages.

Table 7 reveals that high average performance does not necessarily imply strong cross-language consistency. GPT-4o has a relatively narrow 9.7-point range across languages, while qwen3-coder-480b-a35b-instruct spans 11.2 points. By contrast, claude-3-5-haiku-20241022 exhibits a 66.5-point range, largely due to its substantially lower Rust performance relative to Golang and JavaScript. Deepseek-v3.2 also shows marked cross-language variation, performing strongly on Golang but considerably worse on Java.

■ **Table 8** Overall results for directed code translation.

Model	Candidates	Accepted	Accepted Rate	Pass@1	Pass@5	95% CI
qwen3-coder-480b-a35b-instruct	28,000	24,433	87.3%	87.5%	90.5%	[89.7, 91.2]
gpt-4o	28,000	17,295	61.8%	61.8%	85.8%	[84.9, 86.7]
glm-4	28,000	15,141	54.1%	54.3%	73.8%	[72.6, 74.9]
deepseek-v3.2	28,000	12,215	43.6%	43.8%	60.5%	[59.2, 61.8]
claude-3-5-haiku-20241022	28,000	10,477	37.4%	37.4%	59.0%	[57.7, 60.3]

These differences demonstrate that model ranking and multilingual consistency capture distinct properties. Two models with similar average pass@5 can differ substantially in their weakest target language and cross-language variance. For practical model selection, the intended programming language may therefore matter as much as aggregate multilingual performance.

Finding 2 (RQ2). Problem-aligned evaluation reveals substantial and model-dependent variation across target languages. Python3 achieves the highest average pass@5 and Rust the lowest, with a 17.8-point difference under the same problem set. Moreover, strong average performance does not necessarily imply cross-language consistency: some models remain stable across languages, whereas others exhibit pronounced language-specific weaknesses.

7 RQ3: Directed Cross-Language Transfer

RQ3 examines source-code-conditioned translation as a complementary multilingual programming setting. Unlike direct generation, translation must preserve the behavior of an existing implementation while producing executable code in another language. We analyze the 56 ordered source-target settings from overall, pair-level, role-level, and direction-level perspectives. Unless otherwise stated, the pair-, role-, and direction-level translation analyses report pass@5 averaged over the five models evaluated on the translation task.

7.1 Translation Subset

For translation, we randomly select 100 problems from the all-language-covered set described in Section 3. Each problem has an execution-validated implementation in all eight languages, allowing every language to serve as a source and every other language as a target. The same 100 problems are used across all 56 source-target settings, and correctness is determined through target-language execution.

7.2 Overall Translation Performance

Each model is evaluated on 100 problems and 56 ordered source-target settings, with five candidates per instance, yielding 5,600 translation instances and 28,000 candidates per model. As in generation, we report candidate-level acceptance together with instance-level pass@1 and pass@5.

Table 8 shows substantial differences across models. qwen3-coder-480b-a35b-instruct achieves the highest pass@5 at 90.5%, followed by gpt-4o at 85.8%, while glm-4, deepseek-v3.2, and claude-3-5-haiku-20241022 reach 73.8%, 60.5%, and 59.0%, respectively. The broad ordering resembles the generation results in RQ1, suggesting that general code-generation capability remains relevant when source code is provided. However, aggregate scores do not reveal how translation performance varies across language settings.

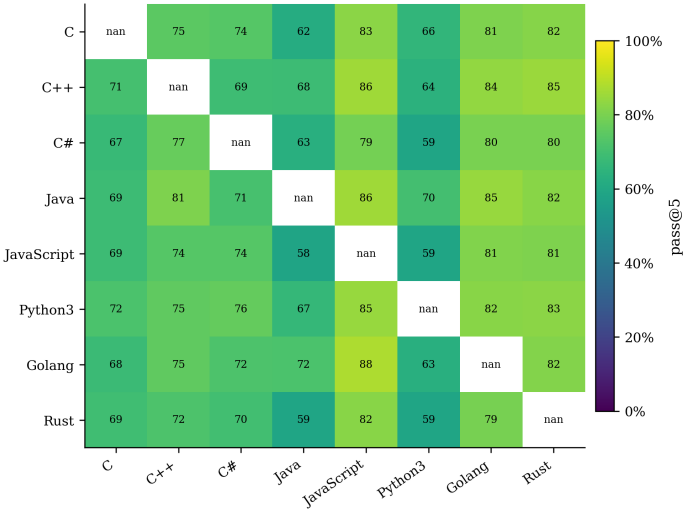


Figure 5 Directed source-target pass@5 matrix for code translation.

Table 9 Average pass@5 when each language is used as source or target in directed translation.

Language	As Source	As Target	Target-Source (pp)
Python3	77.1%	62.7%	-14.4
Java	77.6%	64.1%	-13.5
C	74.5%	69.2%	-5.3
C++	75.2%	75.5%	0.3
C#	72.1%	72.2%	0.1
Golang	74.2%	81.7%	7.5
Rust	70.0%	82.0%	12.0
JavaScript	70.7%	84.0%	13.3

7.3 Source-Target Pair Effects

We next aggregate pass@5 by ordered source-target pair to examine variation hidden by model-level averages.

Figure 5 shows substantial variation across source-target combinations. The same target language can obtain markedly different pass@5 depending on the source language, indicating that translation performance depends on their combination rather than on the target alone. Such variation may reflect differences in source comprehension, target realization, type adaptation, library mapping, and interface reconstruction.

7.4 Source and Target Language Roles

We further summarize each language by its average pass@5 when used as a source and when used as a target.

Table 9 reveals clear role differences. Python3 and Java achieve higher averages as source languages than as targets, whereas JavaScript, Rust, and Golang perform better in the target role. C++ and C# are comparatively balanced. These results suggest that a single language-level translation score can obscure differences between source interpretation and target-language realization.

■ **Table 10** Largest directional gaps in code translation.

Direction	Pass@5	Reverse Pass@5	Gap (pp)
Java→JavaScript	85.8%	58.4%	27.4
JavaScript→Python3	58.6%	84.6%	26.0
Python3→Rust	82.6%	58.6%	24.0
Java→Rust	82.4%	58.8%	23.6
Golang→Python3	63.0%	82.4%	19.4

■ **Table 11** Generation performance by problem difficulty.

Difficulty	Tasks	Pass@1	Pass@5	Pass@5 95% CI	Gain
Easy	38,480	73.4%	81.8%	[81.4, 82.2]	8.4%
Medium	57,840	57.7%	68.2%	[67.8, 68.6]	10.5%
Hard	22,800	36.8%	49.3%	[48.7, 49.9]	12.5%

7.5 Directionality

Finally, we compare each source-target setting with its reverse direction. If translation behavior were symmetric, the two directions of a language pair would have similar pass@5; the results show otherwise.

Table 10 reports the largest differences between reversed language pairs. For example, Java→JavaScript reaches 85.8% pass@5, compared with 58.4% for JavaScript→Java, while Python3→Rust reaches 82.6% compared with 58.6% in the reverse direction. Several reversed pairs differ by more than 20 percentage points, showing that pair-level averages can conceal substantial variation.

Together, the pair-, role-, and direction-level results show that source-code-conditioned translation is highly heterogeneous across language settings. These aggregate observations do not establish the causes of individual gaps, but they identify settings where source interpretation and target realization warrant more detailed analysis.

Finding 3 (RQ3). Source-code-conditioned translation varies substantially across models and language settings. Translation success depends on the source-target combination, languages exhibit different source and target roles, and several reversed pairs differ by more than 20 pass@5 points. Thus, aggregate translation scores alone provide an incomplete view of cross-language transfer.

8 RQ4: Effects of Problem Attributes and Candidate Sampling

RQ4 examines how sensitive multilingual code-generation reliability is to problem characteristics and candidate budget. We analyze performance across difficulty levels and algorithmic topics, and then quantify how additional candidates change measured success.

8.1 Impact of Problem Difficulty

We first group generation results by the platform-provided difficulty labels: Easy, Medium, and Hard. Table 11 reports pass@1, pass@5, 95% confidence intervals, and the gain from five candidates.

Difficulty is strongly associated with generation success. Pass@5 decreases from 81.8% on Easy problems to 68.2% on Medium and 49.3% on Hard problems, while pass@1 drops from 73.4% to 36.8%. The gain from pass@1 to pass@5 also increases with difficulty, from

■ **Table 12** High-performing algorithmic tags in multilingual code generation.

Tag	Tasks	Pass@1	Pass@5	Gain
Counting Sort	640	73.6%	82.5%	8.9%
Simulation	7,280	64.9%	75.1%	10.2%
Randomized	720	65.7%	72.8%	7.1%
Counting	8,400	62.8%	72.4%	9.6%
Reservoir Sampling	160	63.7%	71.9%	8.1%
Two Pointers	10,080	62.1%	71.6%	9.6%
String	38,080	59.1%	69.3%	10.3%
Sorting	19,600	58.7%	69.2%	10.5%
Math	24,320	59.4%	69.1%	9.8%
Array	72,080	58.7%	69.0%	10.3%

8.4 percentage points on Easy problems to 12.5 points on Hard problems. Thus, additional candidates provide greater benefit on harder problems, but do not close the substantial Easy–Hard performance gap.

8.2 Impact of Algorithmic Tags

Difficulty labels provide only a coarse characterization of problem complexity. We therefore further group generation results by algorithmic tags. Because a problem may have multiple tags, the same problem-language instance can contribute to multiple groups. Table 12 reports the highest-performing tags by pass@5.

Higher-performing categories include Counting Sort, Simulation, Counting, Two Pointers, String, Sorting, Math, and Array. Many of these tags correspond to frequently occurring algorithmic patterns, which is consistent with models performing better on familiar implementation structures. However, tags overlap and are correlated with factors such as difficulty, problem length, and data-structure requirements. The observed differences should therefore be interpreted as descriptive topic-level variation rather than independent causal effects.

8.3 Benefit of Multiple Candidates

We next examine how the candidate budget affects measured reliability. Pass@1 reflects first-attempt success, whereas pass@5 captures whether an accepted solution appears within five attempts.

Table 13 and Figure 6 show that the benefit of additional candidates varies substantially across models. qwen3-coder-480b-a35b-instruct begins with high pass@1 and gains only 3.0 percentage points by pass@5. In contrast, deepseek-v3.2 improves by 31.3 points, indicating that later attempts frequently recover instances missed by the first candidate. Several smaller Qwen3 models gain little from additional sampling, showing that a larger candidate budget does not benefit all models equally.

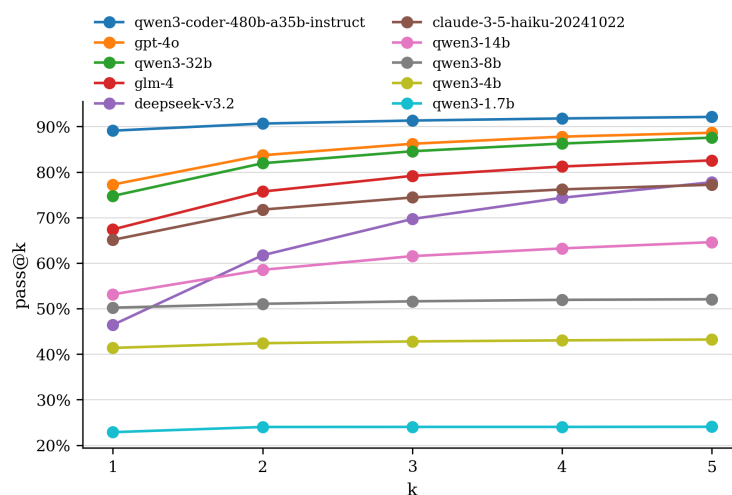
The translation results exhibit the same general phenomenon. For example, gpt-4o improves from 61.8% pass@1 to 85.8% pass@5, while claude-3-5-haiku-20241022 and glm-4 gain 21.7 and 19.5 percentage points, respectively. These results reinforce that single-candidate evaluation can substantially underestimate models whose successful programs appear in later attempts.

8.4 Relative Magnitude of Empirical Effects

Finally, we place the observed sources of variation in context using descriptive range-based gaps. These quantities summarize different analyses and should not be interpreted as causal effect sizes or directly comparable estimates of importance.

■ **Table 13** Pass@5 gains over pass@1 from generating five candidates.

Task	Model	Pass@1	Pass@5	Gain
Generation	qwen3-coder-480b-a35b-instruct	89.1%	92.1%	3.0%
Generation	gpt-4o	77.2%	88.6%	11.4%
Generation	qwen3-32b	74.7%	87.6%	12.8%
Generation	glm-4	67.4%	82.6%	15.2%
Generation	deepseek-v3.2	46.4%	77.8%	31.3%
Generation	claude-3-5-haiku-20241022	65.1%	77.2%	12.1%
Generation	qwen3-14b	53.1%	64.6%	11.5%
Generation	qwen3-8b	50.2%	52.1%	1.8%
Generation	qwen3-4b	41.4%	43.2%	1.8%
Generation	qwen3-1.7b	22.9%	24.1%	1.2%
Translation	qwen3-coder-480b-a35b-instruct	87.5%	90.5%	3.0%
Translation	gpt-4o	61.8%	85.8%	24.0%
Translation	glm-4	54.3%	73.8%	19.5%
Translation	deepseek-v3.2	43.8%	60.5%	16.7%
Translation	claude-3-5-haiku-20241022	37.4%	59.0%	21.7%



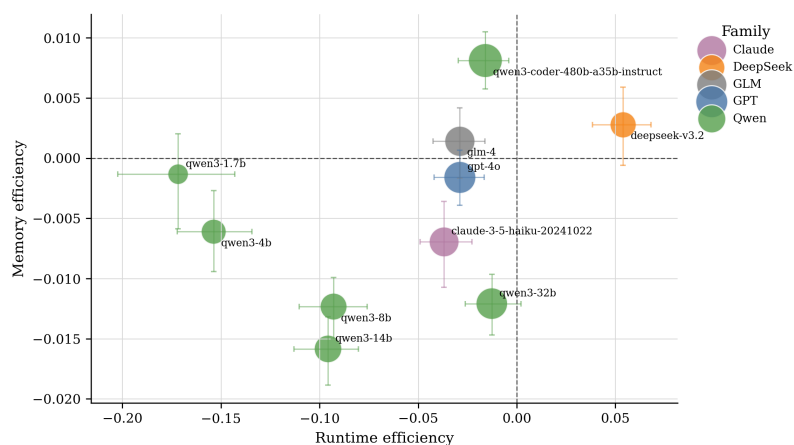
■ **Figure 6** Generation pass@k curves by model.

Table 14 shows that model capability produces the largest observed generation gap, with 68.0 pass@5 points between the strongest and weakest models. Problem difficulty produces a 32.5-point Easy–Hard gap, while the spread in generation sampling gains reaches 30.1 points. For context, the largest translation directional gap observed in RQ3 is 27.4 points, and the generation target-language range is 17.8 points. These results illustrate that measured performance depends not only on model choice, but also on the problem set, target language, and candidate budget.

Finding 4 (RQ4). Multilingual code-generation reliability is highly sensitive to both problem characteristics and candidate budget. Harder problems substantially reduce pass@1 and pass@5, algorithmic tags reveal topic-level variation, and the benefit of additional candidates differs markedly across models and difficulty levels. Therefore, aggregate pass@k should be interpreted together with problem- and sampling-aware analyses.

■ **Table 14** Magnitude of major empirical effects observed in MCIBENCH.

Factor	Comparison	Observed Gap (pp)
Model capability in generation	best vs. weakest model pass@5	68.0
Problem difficulty in generation	Easy vs. Hard pass@5	32.5
Candidate sampling in generation	largest vs. smallest pass@5 gain	30.1
Model capability in translation	best vs. weakest model pass@5	31.5
Translation directionality	largest A→B vs. B→A gap	27.4
Target language in generation	easiest vs. hardest language pass@5	17.8



■ **Figure 7** Runtime-memory efficiency of accepted generated programs.

9 RQ5: Runtime-Memory Efficiency of Accepted Code

RQ5 examines whether functionally correct programs generated by different LLMs exhibit comparable execution quality. Because pass@ k treats all accepted programs as equally successful, we conduct an accepted-only analysis of runtime and memory efficiency.

9.1 Efficiency Measurement

We analyze accepted generated programs only. For each problem-language pair and model, if multiple candidates are accepted, we retain the first accepted candidate, corresponding to a practical first-success strategy. Because raw runtime and memory measurements are not directly comparable across problems or target languages, we normalize them within each problem-language group. Specifically, for a measurement x (runtime or memory), we compute its relative efficiency as $e(x) = \frac{\bar{x} - x}{\bar{x}}$, where $\bar{x}$ is the mean measurement of accepted programs within the same problem-language group. Each accepted program is therefore compared only with other accepted programs solving the same problem in the same language. Higher runtime-efficiency and memory-efficiency scores indicate faster execution and lower memory consumption, respectively. For each model, we aggregate these normalized scores by averaging over its accepted problem-language instances.

9.2 Runtime-Memory Efficiency across Models

Figure 7 summarizes post-correctness execution quality across models. Each point represents a model; the x- and y-axes denote normalized runtime and memory efficiency, respectively, point size represents candidate acceptance rate, and error bars show bootstrap confidence intervals. Larger values indicate better relative efficiency.

Correctness and execution efficiency do not necessarily align. qwen3-coder-480b-a35b-instruct achieves the highest candidate acceptance rate and competitive memory efficiency, but does not dominate runtime efficiency. Conversely, deepseek-v3.2 has substantially lower correctness than the strongest models, yet its accepted programs achieve the highest relative runtime efficiency in this analysis. Differences in memory efficiency are smaller, but accepted programs are nevertheless not resource-equivalent.

These results show that pass@5 captures whether a model can produce a correct program, but not the execution quality of that program. A model with high correctness may generate relatively expensive accepted solutions, while a lower-performing model may produce efficient programs on the subset of problems it solves. Since the analysis is conditioned on successful outputs and relies on platform-reported measurements, these results should be interpreted as relative post-correctness efficiency rather than unconditional model efficiency.

Finding 5 (RQ5). Functional correctness does not imply comparable execution quality. Accepted programs exhibit meaningful runtime-memory differences across models, and models with higher pass@ k do not necessarily dominate post-correctness efficiency. Runtime and memory therefore provide complementary information beyond correctness, while remaining accepted-only and platform-dependent measures.

10 Discussion

Our results show that multilingual LLM code capability cannot be adequately characterized by a single aggregate pass@ k score. Even under problem-level alignment, performance varies across target languages, translation settings, problem characteristics, candidate budgets, and post-correctness execution quality. These observations lead to several implications for multilingual code evaluation. First, evaluation should be *language-aware*, because models with similar average performance may exhibit substantially different strengths and weaknesses across target languages. Second, task formulation should be considered explicitly: direct generation and source-code-conditioned translation impose different requirements, and translation performance can further vary across source-target settings. Third, evaluation should be *problem-aware*, since difficulty levels and algorithmic topics reveal non-uniform behavior across problems. Fourth, it should be *sampling-aware*, because pass@1 and pass@5 capture different aspects of reliability, and the benefit of additional candidates varies considerably across models. Finally, it should be *quality-aware*, as functionally correct programs can still differ in runtime and memory efficiency. Model selection for multilingual programming should therefore consider the intended language, translation setting, problem characteristics, candidate budget, and execution-quality requirements in addition to aggregate correctness.

Threats to Validity. Several threats affect the interpretation of our results. First, MCIBENCH focuses on online algorithmic programming problems in eight mainstream languages, so the findings may not generalize to repository-level development, domain-specific languages, or production maintenance tasks. Second, the evaluated problems are publicly available and may have appeared in model training data; our results should therefore be interpreted as benchmark performance under a problem-aligned execution protocol rather than evidence of generalization to unseen tasks. Third, prompt design, decoding settings, post-processing rules, language templates, and submission interfaces may affect measured performance, although we apply a common protocol across models and languages and release the corresponding artifacts for inspection. Fourth, the translation analysis uses 100 randomly selected problems

from the all-language-covered set, and the execution-validated source implementations may differ in provenance, style, idioms, and complexity; these factors can influence translation performance and limit the generality of source-target comparisons. Finally, execution-based correctness depends on the available tests and does not establish full semantic equivalence. Recent work combining LLMs with formal verification and proof synthesis suggests complementary mechanisms for stronger semantic validation [15, 28, 7, 27]. Runtime and memory are analyzed only for accepted programs and are platform-dependent. Accordingly, the efficiency results should be interpreted as relative post-correctness indicators rather than unconditional or machine-independent measures of model quality.

11 Conclusion

This paper presented a large-scale empirical study of problem-aligned multilingual code generation and source-code-conditioned translation by LLMs. Across 1,489 programming problems and eight languages, our generation results reveal substantial variation across models, target languages, problem characteristics, candidate budgets, and execution quality; a complementary analysis on 100 problems further shows substantial variation across source-target translation settings. These findings demonstrate that aggregate pass@ k alone provides an incomplete view of multilingual code capability. Reliable evaluation should therefore consider cross-language consistency, translation settings, problem and sampling effects, and post-correctness execution quality.

Data Availability Statement

The benchmark data, prompts, model outputs, executable submissions, execution records, and analysis scripts are publicly available at <https://github.com/Xidian-ICTT-GZ/MCIBench>.

References

- 1 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint*, 2021. [arXiv:2108.07732](https://arxiv.org/abs/2108.07732).
- 2 Stefano Bistarelli, Marco Fiore, Ivan Mercanti, and Marina Mongiello. Usage of large language model for code generation tasks: A review. *SN Computer Science*, 6(6):673, 2025. doi:10.1007/S42979-025-04241-5.
- 3 Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multiple: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023. doi:10.1109/TSE.2023.3267446.
- 4 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. Evaluating large language models trained on code. *arXiv preprint*, 2021. [arXiv:2107.03374](https://arxiv.org/abs/2107.03374).
- 5 Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.
- 6 Chao Jiang, Dugang Liu, Cheng Wen, Zhiwu Xu, Hua Zheng, Muhammad Sadiq, Jawwad Ahmed Shamsi, Shengchao Qin, and Zhong Ming. Large language models for multilingual code intelligence: A survey. In *International Conference on Knowledge Science, Engineering and Management*. Springer, 2026.

- 7 Chao Jiang, Ding Wang, Dugang Liu, Zhiwu Xu, Cheng Wen, Zhong Ming, Yuwei Liu, Xinyi Wan, and Lin Huang. Starverus: Llm-powered multi-agent collaboration for industrial rust code verification automation. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 7444–7455, 2026.
- 8 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, 2026. doi:10.1145/3747588.
- 9 Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. On the evaluation of neural code translation: Taxonomy and benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1529–1541. IEEE, 2023. doi:10.1109/ASE56229.2023.00114.
- 10 Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805, 2024. doi:10.18653/V1/2024.ACL-LONG.367.
- 11 Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, 2023.
- 12 Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint*, 2021. arXiv:2102.04664.
- 13 Zhi Ma, Xiao Liang, Cheng Wen, Rui Chen, Bin Gu, Shengchao Qin, Cong Tian, and Mengfei Yang. Automated ltl specification generation from industrial aerospace requirements. In *International Symposium on Formal Methods*, pages 670–690. Springer, 2026. doi:10.1007/978-3-032-26220-2_33.
- 14 Zhi Ma, Cheng Wen, Zhixin Su, Xiao Liang, Cong Tian, Shengchao Qin, and Mengfei Yang. Bridging natural language and formal specification - automated translation of software requirements to ltl via hierarchical semantics decomposition using llms. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, 2025.
- 15 Zhi Ma, Cheng Wen, Bin Yu, and Jie Su. Integrating ensemble learning and large language models for efficient formal verification of ip-based aerospace systems. *Information Fusion*, 125:103466, 2026. doi:10.1016/J.INFFUS.2025.103466.
- 16 Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. In *International Conference on Learning Representations*, volume 2024, pages 7604–7623, 2024.
- 17 Qiwei Peng, Yekun Chai, and Xuhong Li. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. *arXiv preprint*, 2024. doi:10.48550/arXiv.2402.16694.
- 18 Md Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. mhumaneval-a multilingual benchmark to evaluate large language models for code generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 11432–11461, 2025. doi:10.18653/V1/2025.NAACL-LONG.570.
- 19 Jonathan Ullrich, Matthias Koch, and Andreas Vogelsang. From requirements to code: Understanding developer practices in llm-assisted software engineering. In *2025 IEEE 33rd International Requirements Engineering Conference (RE)*, pages 257–266. IEEE, 2025. doi:10.1109/RE63999.2025.00032.
- 20 Ying Wang, Jie Su, Cheng Wen, Rong Wang, Cong Tian, Zhenhua Dun, and Shengchao Qin. Synergizing llm-driven semantic reasoning with assertion-guided analysis for enhanced vulnerability detection. In *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–6. IEEE, 2026. doi:10.1109/SANER67736.2026.00075.

- 21 Bingyang Wei. Requirements are all you need: From requirements to code with llms. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, pages 416–422. IEEE, 2024. doi:10.1109/RE59067.2024.00049.
- 22 Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Cong Tian. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *ACM Transactions on Knowledge Discovery from Data*, 18(7):1–34, 2024.
- 23 Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*. Springer, 2024.
- 24 Cheng Wen, Junjie Hu, YiKun Hu, Jie Su, Bin Yu, Dugang Liu, Zhiwu Xu, Weidi Sun, Shengchao Qin, and Cong Tian. Formally specifying the intended behavior of the program: Llm-driven neuro-symbolic program specification synthesis. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 672–682, 2026. doi:10.18653/V1/2026.ACL-DEMO.66.
- 25 Edwin B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.
- 26 Jiarong Wu, Songqiang Chen, Jialun Cao, Hau Ching Lo, and Shing-Chi Cheung. Isolating language-coding from problem-solving: Benchmarking llms with pseudoeval. *arXiv preprint*, 2025. doi:10.48550/arXiv.2502.19149.
- 27 Shaolin Zhang, Hongjin Liu, Zhi Ma, Xiao Liang, and Cheng Wen. Formal verification of aerospace software ip components: a multi-tool case study. In *Proceedings of the 2026 6th International Conference on Computer Network Security and Software Engineering*, pages 219–228, 2026.
- 28 Yuchen Zhang, Cheng Wen, Zhiwu Xu, Dugang Liu, et al. Enhancing llm-based proof synthesis for rust programs via semantic chunking and hierarchical context expansion. In *International Symposium on Theoretical Aspects of Software Engineering*, pages 81–100, 2026.
- 29 Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 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*, pages 5673–5684, 2023.
- 30 Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint*, 2022. doi:10.48550/arXiv.2206.08474.