

A Direction-Aware Study of LLM-Based Code Translation Across Eight Programming Languages

Quanhe Wang¹ 

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

Xiangyun Zhan¹ 

Guangzhou Institute of Technology, Xidian University, Guangzhou, China

Cheng Wen² 

Shenzhen Institute of Technology, Xidian University, Shenzhen, China

Bin Yu 

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

Ping Chen² 

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

Xingjian Han 

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

Abstract

Large language models (LLMs) are increasingly used for code translation, yet existing evaluations often collapse results across language pairs, thereby obscuring the direction of transfer. We present an emerging empirical study that treats multilingual code translation as a directed transfer problem. Our study uses 100 programming problems, each with reference implementations in eight languages, yielding a complete directed graph of 56 ordered source-target translation directions. Five contemporary LLMs generate candidate translations, which are evaluated through execution in the target language. The results reveal substantial directional asymmetry. In model-averaged pass@5, reversing a language pair changes the translation success rate by as much as 27.4 percentage points. We also observe systematic differences between source and target roles: Python3 and Java achieve higher average performance when used as source languages, whereas JavaScript, Rust, and Golang achieve higher average performance when used as target languages. These findings show that aggregate scores and unordered language-pair averages can conceal practically important transfer behavior. Rather than establishing a definitive model ranking, this study provides preliminary evidence that code-translation evaluations should preserve ordered source-target results, distinguish source and target language roles, and explicitly report directional gaps. This direction-aware perspective provides a more informative basis for subsequent failure analysis and the evaluation of realistic cross-language migration scenarios.

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

Keywords and phrases Large language models, code translation, multilingual programming, empirical software engineering, execution-based evaluation

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.63

¹ These authors contributed equally to this work and should be considered co-first authors.

² Corresponding author: Cheng Wen and Ping Chen.



© Quanhe Wang, Xiangyun Zhang, Cheng Wen, Bin Yu, Ping Chen, Xianjian Han, and Shengchao Qin; 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. 63; pp. 63:1–63:14



Leibniz International Proceedings in Informatics
LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Category Emerging Results, Vision & Reflection Track Paper

Supplementary Material *Software (Source Code)*: <https://doi.org/10.5281/zenodo.20456638>

Funding This work was supported in part by the Basic Research Foundation of Shenzhen City (No. JCYJ20250604184202003), the National Natural Science Foundation of China (Nos. 62302375, 62472339), and the China Postdoctoral Science Foundation funded project (No. 2023M723736), and the Open Project of China Electronic Product Reliability and Environmental Testing Research Institute (The Fifth Electronic Research Institute of MIIT).

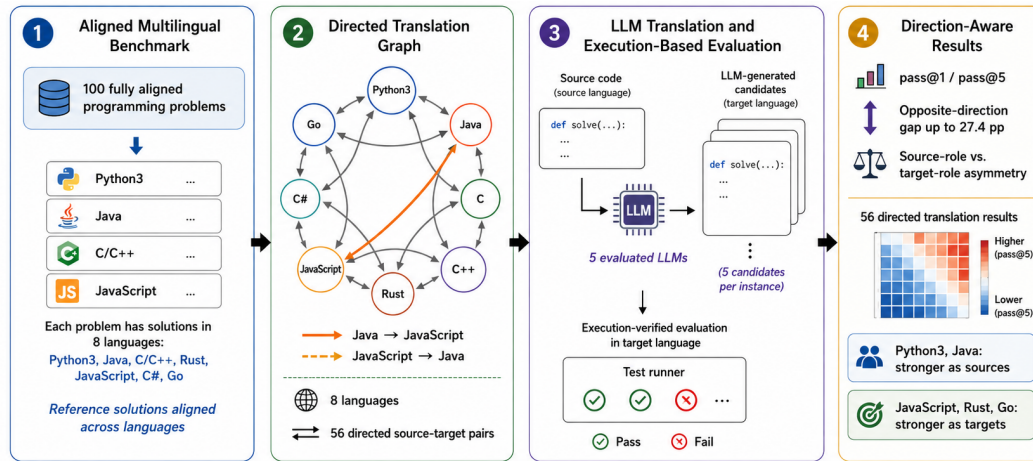
1 Introduction

Large language models (LLMs) have made code generation a mainstream research topic [1, 3, 5]. However, many practical software engineering scenarios require translating existing programs rather than producing new implementations from natural-language requirements [13, 6]. Developers translate code when migrating legacy systems, porting algorithmic implementations, reusing software across language ecosystems, teaching programming concepts in multiple languages, or moving prototypes toward production-oriented platforms [4]. Recent repository-level benchmarks further demonstrate that code migration is a practical software engineering problem rather than merely a programming-contest task [14, 15]. In these settings, an LLM must understand an existing source program, preserve its intended behavior, and realize that behavior as an executable implementation in a different programming language.

This paper argues that LLM-based code translation should be studied as a *directed* transfer problem. Translating from Java to JavaScript is not necessarily equivalent to translating from JavaScript to Java. The former starts from explicit type information and class structure and produces an implementation in a more flexible language, whereas the latter may require the model to recover types, interfaces, and data-structure choices that are implicit in the source code. Similar directional differences may arise between Python3 and Rust, C and Golang, or C++ and C#. Consequently, treating a language pair as unordered can hide important empirical behavior: a single pair-level average may conflate two translation tasks that impose different requirements on the model.

We conduct an emerging empirical study of direction-aware code translation across eight programming languages: Python3, Java, C, C++, Rust, JavaScript, C#, and Golang. Our evaluation set contains 100 programming problems, each with reference implementations in all eight languages. This design induces a complete directed translation graph in which programming languages are nodes and the 56 ordered source-target pairs are directed edges. For each evaluated model, problem, and translation direction, the model generates candidate programs in the specified target language. The resulting programs are submitted to the corresponding target-language execution environment, so correctness is determined through execution rather than textual similarity, following the broader movement toward execution-based code evaluation [3, 8, 7]. Because the same 100 problems are used across all translation directions, differences between source-target pairs can be examined under matched problem semantics.

Figure 1 summarizes the study setting. The central empirical object is the directed translation graph rather than a single aggregate model score. Each edge represents a concrete translation direction, such as Java-to-JavaScript or JavaScript-to-Java. Preserving this ordered relation throughout the evaluation allows us to examine whether opposite directions exhibit similar performance, which directions are comparatively difficult, and whether individual languages behave differently when serving as sources and targets.



■ **Figure 1** Overview of our direction-aware code translation study. Using 100 aligned multilingual problems across 8 languages, we form 56 directed source-target pairs, evaluate LLM outputs by execution, and analyze the resulting matrix for opposite-direction gaps and source/target role asymmetries.

As an emerging-results study, we center our investigation on one overarching question: *what does the 56-edge translation graph reveal about the behavior of LLM-based code translation?* Our results show that the graph is uneven, asymmetric, and role-dependent. Although the evaluated models differ in overall translation performance, aggregate model scores do not reveal where translation succeeds or fails. In model-averaged pass@5 results, several pairs exhibit opposite-direction differences exceeding 20 percentage points, with the largest observed gap reaching 27.4 percentage points. We also find that some languages achieve higher average performance when used as source languages, whereas others achieve higher average performance when used as target languages. These observations indicate that translation difficulty cannot be adequately characterized by an unordered language pair or by a single language-level score.

This paper makes two main contributions:

- We formulate multilingual code translation as a direction-aware evaluation problem, representing programming languages as nodes and ordered source-target translation tasks as edges in a directed graph.
- We provide execution-verified empirical evidence from 100 programming problems, eight languages, 56 directed translation pairs, and five contemporary LLMs, showing that aggregate translation scores can conceal substantial opposite-direction gaps and distinct source/target language roles.

Based on these findings, we derive reporting implications for empirical evaluations of LLM-based code translation. In particular, translation studies should preserve ordered source-target results, report source and target language roles separately, and distinguish first-attempt performance from multi-candidate translation success.

2 Code Translation as a Directed Transfer Problem

2.1 Direction-Aware Translation Perspective

Code generation and code translation are related but distinct capabilities. Code generation maps a natural-language problem description to an implementation in a target language and has been extensively studied through benchmarks such as HumanEval and its multilingual

extensions [3, 2, 20]. Broader code-intelligence datasets further support multilingual and multi-task evaluation [21, 7]. Code translation, by contrast, starts from an existing implementation and asks a model to preserve its behavior while realizing an executable program in another language. This task has motivated dedicated translation methods, datasets, and evaluation studies [12, 6, 18, 14]. A successful translation therefore requires the model to understand the source program, preserve its intended semantics, and satisfy the syntactic, type, library, and execution constraints of the target language. The broader challenge of preserving intent across representations also arises in LLM-assisted formal specification generation, where informal requirements are transformed into machine-checkable specifications [11, 10, 9]. Although code translation and specification generation produce different artifacts, both require semantic constraints expressed in one representation to be faithfully reconstructed in another.

These coupled requirements make the direction of transfer important. Translating from language A to language B is not necessarily equivalent to translating from B to A , because the two directions may expose different information and impose different reconstruction requirements. For example, a dynamically typed source may leave type information implicit, making translation into a statically typed target more demanding. A low-level source may expose memory management, indexing, or pointer-related details that must be re-expressed using higher-level abstractions. Conversely, translating from a high-level language into a systems language may require explicit choices concerning data structures, ownership, interfaces, and resource management. An aggregate score over an unordered language pair can therefore conflate two empirically and practically different translation tasks.

We model multilingual code translation as a weighted directed graph. Each programming language is represented as a node, and each ordered source-target pair is represented as a directed edge. For eight languages, the resulting graph contains $8 \times 7 = 56$ directed edges. This graph supports three complementary levels of analysis. At the *edge level*, we examine the translation success of an individual source-target direction. At the *pair level*, we compare two opposite edges, such as Java-to-JavaScript and JavaScript-to-Java, to quantify directional asymmetry. At the *role level*, we aggregate outgoing and incoming edges to study how a language behaves when used as a source or as a target.

This perspective complements conventional aggregate reporting. A model-level score over all translation instances remains useful as a coarse indicator of overall capability, but it does not answer the practical question faced by developers: *can this model translate from the language I have to the language I need?* Direction-aware evaluation addresses this question by preserving the ordered source-target relation throughout the analysis. It also complements execution-based multilingual code evaluation [7] by shifting attention from model-level or target-only averages to the edge-, pair-, and role-level structure of multilingual transfer.

2.2 Formal Problem Definition

Let $\mathcal{L} = \{\ell_1, \dots, \ell_m\}$ denote a set of programming languages and $\mathcal{P} = \{p_1, \dots, p_n\}$ denote a set of programming problems. Each problem $p \in \mathcal{P}$ has a reference implementation $s_{p,\ell}$ in every language $\ell \in \mathcal{L}$. A directed code-translation instance is represented as a triple (p, ℓ_s, ℓ_t) , where $\ell_s, \ell_t \in \mathcal{L}$ and $\ell_s \neq \ell_t$. Its input consists of the source program s_{p,ℓ_s} and the specified target language ℓ_t . The expected output is a target-language program $\hat{s}_{p,\ell_t}$ that preserves the behavior of s_{p,ℓ_s} and is executable in the target-language evaluation environment.

We view an LLM M as a stochastic translator:

$$M(s_{p,\ell_s}, \ell_t) \rightarrow \{\hat{s}_{p,\ell_t}^{(1)}, \dots, \hat{s}_{p,\ell_t}^{(k)}\},$$

where k denotes the number of generated candidates. Correctness is determined by an execution oracle E :

$$E(p, \ell_t, \hat{s}) = \begin{cases} 1, & \text{if } \hat{s} \text{ passes the target-language tests for } p, \\ 0, & \text{otherwise.} \end{cases}$$

Execution provides a uniform and practical correctness criterion, but its judgment is necessarily limited to the available test cases. Recent work on LLM-assisted formal specification generation suggests a complementary direction in which machine-checkable behavioral properties could support stronger validation of semantic preservation beyond test execution [16, 17, 19].

For a directed instance (p, ℓ_s, ℓ_t) , empirical pass@ k indicates whether at least one of the first k candidates is accepted:

$$\text{pass@}k(p, \ell_s, \ell_t, M) = \mathbb{I} \left[\max_{1 \leq i \leq k} E(p, \ell_t, \hat{s}_{p, \ell_t}^{(i)}) = 1 \right].$$

This measure represents empirical best-of- k success over the generated candidates. It differs from the unbiased pass@ k estimator commonly used when a larger candidate pool is sampled.

The edge weight for an ordered source-target pair (ℓ_s, ℓ_t) is defined as the average empirical pass@ k over the problem set:

$$w_k(\ell_s, \ell_t, M) = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \text{pass@}k(p, \ell_s, \ell_t, M).$$

This definition induces a model-specific weighted directed graph $G_k^M = (\mathcal{L}, \mathcal{E}, w_k)$, where

$$\mathcal{E} = \{(\ell_s, \ell_t) : \ell_s, \ell_t \in \mathcal{L}, \ell_s \neq \ell_t\}.$$

When results are aggregated over a set of evaluated models $\mathcal{M} = \{M_1, \dots, M_r\}$, we define the model-averaged edge weight as

$$\bar{w}_k(\ell_s, \ell_t) = \frac{1}{|\mathcal{M}|} \sum_{M \in \mathcal{M}} w_k(\ell_s, \ell_t, M).$$

To quantify asymmetry between two languages, we compare the corresponding opposite edges:

$$\Delta_k(\ell_a, \ell_b) = |\bar{w}_k(\ell_a, \ell_b) - \bar{w}_k(\ell_b, \ell_a)|.$$

A small Δ_k indicates that the two directions exhibit similar translation success under the evaluation setting, whereas a large value indicates that reversing the source and target languages substantially changes the observed performance.

We further characterize the source and target roles of a language ℓ by averaging its outgoing and incoming edges:

$$\begin{aligned} \text{Src}_k(\ell) &= \frac{1}{|\mathcal{L}| - 1} \sum_{\ell_t \in \mathcal{L} \setminus \{\ell\}} \bar{w}_k(\ell, \ell_t), \\ \text{Tgt}_k(\ell) &= \frac{1}{|\mathcal{L}| - 1} \sum_{\ell_s \in \mathcal{L} \setminus \{\ell\}} \bar{w}_k(\ell_s, \ell). \end{aligned}$$

Here, $\text{Src}_k(\ell)$ measures the average translation success when ℓ provides the source program, whereas $\text{Tgt}_k(\ell)$ measures the average success when ℓ is the requested target language. Their difference, $\text{Tgt}_k(\ell) - \text{Src}_k(\ell)$, indicates whether a language obtains higher performance in the target or source role.

In the remainder of the paper, we apply these definitions to the complete 56-edge translation graph. Our analysis proceeds from overall model performance to directed edge behavior, opposite-direction gaps, and source/target language roles.

■ **Table 1** Direction-aware code translation setting.

Item	Setting
Problems	100 fully aligned programming problems
Languages	Python3, Java, C, C++, Rust, JavaScript, C#, Golang
Translation graph	8 language nodes and 56 directed source-target edges
Candidates	5 candidates per problem-source-target setting
Candidates per model	28,000 translated programs
Evaluation protocol	Target-language execution-based checking
Main metrics	Accepted rate, pass@1, pass@5, opposite-direction gap

3 Study Design

3.1 Research Questions

Guided by the direction-aware formulation in Section 2.2, we organize the study around three research questions.

RQ1: How effectively do LLMs perform execution-verified code translation across eight programming languages? This question examines the overall translation capability of the evaluated models before considering individual source-target directions.

RQ2: To what extent does code-translation performance depend on the source-target direction? This question analyzes the 56 directed source-target edges and compares each edge with its reverse direction to quantify directional asymmetry.

RQ3: How do programming languages differ in their source and target roles? This question aggregates outgoing and incoming translation edges separately to examine whether a language behaves differently when providing the source program and when serving as the target language.

3.2 Evaluation Set and Translation Graph

The study evaluates 100 online-judge programming problems across eight programming languages: Python3, Java, C, C++, Rust, JavaScript, C#, and Golang. Each problem has a reference implementation in every language, allowing each language to serve as the source and every other language to serve as the target. The same 100 problem identities are used for all translation directions, ensuring that source-target pairs and their reverse directions are evaluated under matched problem semantics. This controlled design reduces the influence of problem-set composition when comparing translation directions.

Under the formulation in Section 2.2, $|\mathcal{L}| = 8$ and $|\mathcal{P}| = 100$. The eight languages form a complete directed graph with $8 \times 7 = 56$ ordered source-target edges. Each edge represents one translation direction, such as Java-to-JavaScript or JavaScript-to-Java, and each evaluation instance is a problem-source-target triple (p, ℓ_s, ℓ_t) . The resulting evaluation contains $100 \times 56 = 5,600$ directed translation instances per model before candidate sampling. Table 1 summarizes the study setting.

3.3 Models and Prompting

We evaluate five contemporary code-capable LLMs: qwen3-coder-480b-a35b-instruct, gpt-4o, glm-4, deepseek-v3.2, and claude-3-5-haiku-20241022. For each problem and ordered source-target pair, the prompt provides the reference implementation s_{p, ℓ_s} in the source language and instructs the model to produce an equivalent implementation in the target language

ℓ_t . The instruction template remains fixed across all translation directions; only the source program and target-language name vary. This common prompting protocol ensures that the 56 directed edges are evaluated under the same task formulation.

Each model generates five candidates for every problem-source-target instance. Consequently, one model produces $100 \times 56 \times 5 = 28,000$ translated programs, and the five evaluated models collectively produce 140,000 candidates. The complete prompt templates, model invocation configurations, and decoding parameters are provided in the accompanying artifact.

3.4 Execution-Verified Evaluation

Model outputs are converted into executable submissions using the corresponding target-language templates. When an output contains Markdown fences, post-processing selects the first complete code block matching the target language. Otherwise, it uses the first complete code block or the cleaned raw output. The extracted program is then inserted into the target-language submission template. Post-processing does not repair imports, rename identifiers, edit types, modify input-output logic, or manually correct algorithms.

A candidate is accepted only if the execution oracle returns $E(p, \ell_t, \hat{s}) = 1$. We report three complementary correctness measures: candidate-level accepted rate, empirical pass@1, and empirical pass@5. The acceptance rate is computed over all generated candidates. Pass@1 indicates whether the first candidate for a problem-source-target instance is accepted, whereas pass@5 indicates whether at least one of its five candidates is accepted. Thus, accepted rate is a candidate-level measure, while pass@1 and pass@5 are instance-level best-of- k measures.

For each model, the overall pass@5 result is computed over the 5,600 problem-source-target instances. The corresponding 95% confidence interval is estimated from these instance-level binary outcomes. Reporting both pass@1 and pass@5 allows us to distinguish first-attempt reliability from the benefit of generating multiple translation candidates.

For the direction-level analyses in RQ2 and RQ3, we use the model-averaged edge weight $\bar{w}_5(\ell_s, \ell_t)$ defined in Section 2.2. Opposite-direction asymmetry is measured using $\Delta_5(\ell_a, \ell_b)$, while source and target language roles are characterized using $\text{Src}_5(\ell)$ and $\text{Tgt}_5(\ell)$. Unless otherwise stated, these analyses average pass@5 over the five evaluated models. The resulting edge-, pair-, and role-level values are interpreted as empirical observations about the evaluated 56-edge translation graph rather than as intrinsic rankings of programming languages.

4 Preliminary Results

4.1 RQ1: Overall Translation Performance

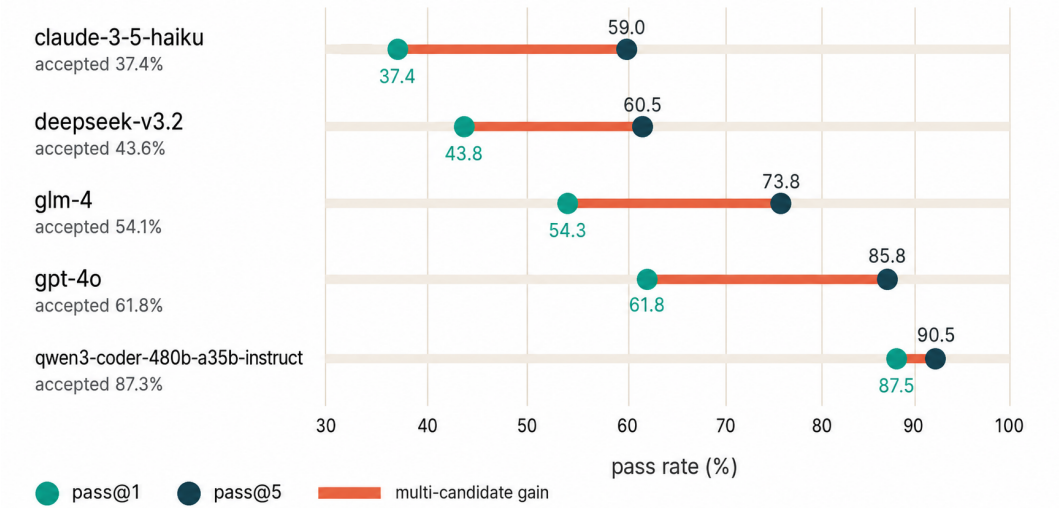
We first examine overall execution-verified translation performance before analyzing the structure of individual source-target directions. Table 2 reports the results of each model aggregated over the 5,600 problem-source-target instances defined by the 56 directed translation edges. Among the evaluated models, qwen3-coder-480b-a35b-instruct achieves the highest pass@5 at 90.5%, followed by gpt-4o at 85.8%. The other models also demonstrate nontrivial translation capability: glm-4 reaches 73.8%, while deepseek-v3.2 and claude-3.5-haiku-20241022 achieve 60.5% and 59.0%, respectively.

Figure 2 compares first-attempt translation success with best-of-five success. Although accepted rate and pass@1 are numerically similar for several models in Table 2, they are computed over different units. Accepted rate is the proportion of successful candidates

■ **Table 2** Overall results for directed code translation.

Model	Candidates	Accepted	Accepted Rate	Pass@1	Pass@5	95% CI
qwen3-coder-480b-a35b	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	28,000	10,477	37.4%	37.4%	59.0%	[57.7, 60.3]

Note: qwen3-coder-480b-a35b abbreviates qwen3-coder-480b-a35b-instruct.



■ **Figure 2** Overall pass@1 and pass@5 of the five evaluated LLMs across the 56 directed translation edges.

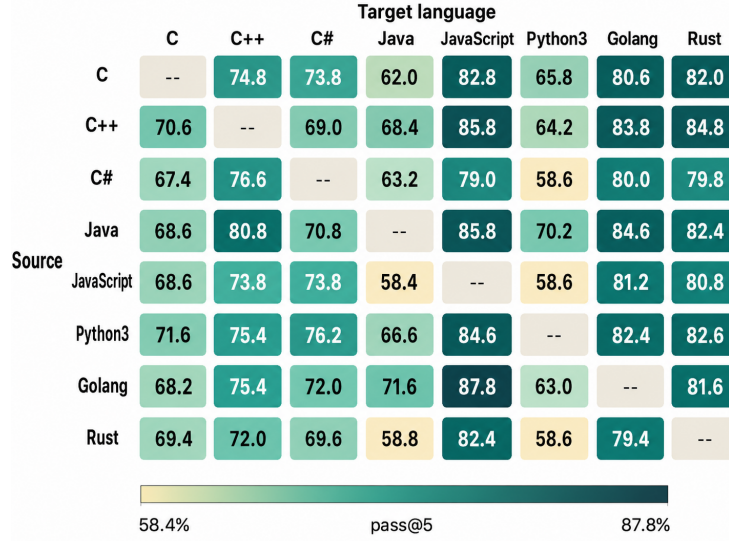
among all 28,000 generated programs per model, whereas pass@1 is the proportion of the 5,600 problem-source-target instances for which the first candidate is accepted. The more informative sampling comparison is therefore between pass@1 and pass@5.

For qwen3-coder-480b-a35b-instruct, pass@5 exceeds pass@1 by only 3.0 percentage points, indicating that most of its successful translation instances are already solved by the first candidate. In contrast, gpt-4o improves from 61.8% pass@1 to 85.8% pass@5, a gain of 24.0 percentage points. glm-4 and claude-3-5-haiku-20241022 gain 19.5 and 21.7 percentage points, respectively. These differences show that first-attempt reliability and multi-candidate success capture distinct aspects of translation capability. A model with a relatively modest pass@1 may still achieve substantially higher coverage when several candidates are available.

Finding 1. The evaluated LLMs achieve substantially different levels of execution-verified translation success across eight languages. Moreover, the gains from pass@1 to pass@5 vary considerably across models, showing that first-attempt reliability and best-of- k success should be reported separately.

4.2 RQ2: Directed Source-Target Effects

The aggregate results in RQ1 summarize overall model capability but collapse the 56 source-target directions into a small number of model-level scores. To expose the directional structure behind these aggregates, Figure 3 reports the model-averaged edge weight $\bar{w}_5(\ell_s, \ell_t)$ for every



■ **Figure 3** Model-averaged pass@5, $\bar{w}_5(\ell_s, \ell_t)$, for each directed source-target edge. Diagonal cells are excluded.

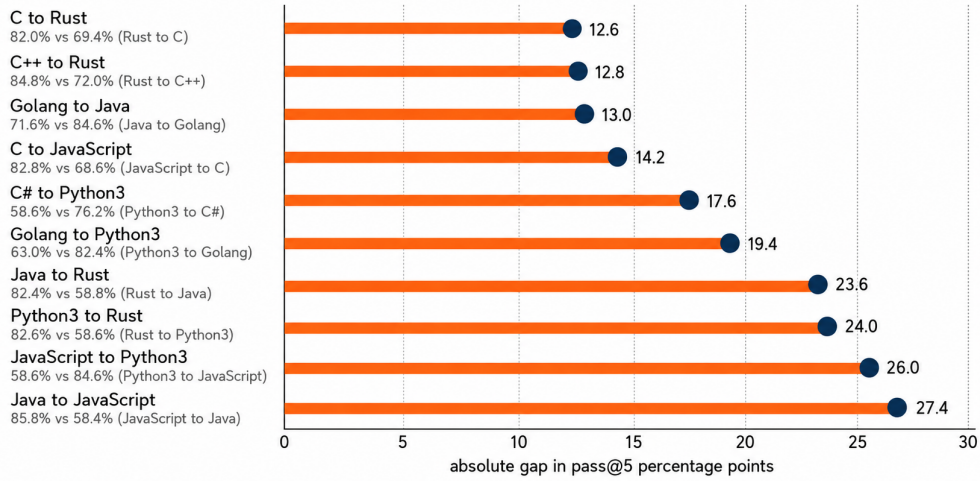
■ **Table 3** Largest observed opposite-direction gaps in code translation, with pass@5 averaged over the five evaluated models.

Direction Pair	Forward Pass@5	Reverse Pass@5	Gap
Java to JavaScript vs. JavaScript to Java	85.8%	58.4%	27.4 pp
Python3 to JavaScript vs. JavaScript to Python3	84.6%	58.6%	26.0 pp
Python3 to Rust vs. Rust to Python3	82.6%	58.6%	24.0 pp
Java to Rust vs. Rust to Java	82.4%	58.8%	23.6 pp
Python3 to Golang vs. Golang to Python3	82.4%	63.0%	19.4 pp

ordered source-target pair. The resulting matrix shows substantial variation across edges. It is also visibly asymmetric: reversing the source and target languages can markedly change the observed translation success rate.

Table 3 and Figure 4 further report the largest opposite-direction differences, measured by $\Delta_5(\ell_a, \ell_b)$. For example, Java-to-JavaScript reaches 85.8% model-averaged pass@5, whereas JavaScript-to-Java reaches 58.4%, producing a gap of 27.4 percentage points. Python3-to-JavaScript and JavaScript-to-Python3 differ by 26.0 percentage points. Similarly, Python3-to-Rust reaches 82.6%, while Rust-to-Python3 reaches 58.6%, yielding a gap of 24.0 percentage points. An unordered average for any of these language pairs would merge two empirically different translation settings and conceal which direction presents the greater difficulty.

The observed gaps should be interpreted as diagnostic evidence rather than as causal explanations. They identify language pairs for which source comprehension and target realization appear to impose different empirical burdens. For example, the JavaScript-to-Java direction may involve reconstructing explicit types, method signatures, and target-library choices that are less explicit in the source implementation. Likewise, translation between Python3 and Rust may involve substantial differences in typing, ownership, data structures, and implementation conventions. However, the edge-level pass@5 results alone cannot determine which language features or model behaviors cause a particular gap. Establishing such mechanisms would require direction-specific failure analysis and controlled experiments.



■ **Figure 4** The ten largest opposite-direction gaps in model-averaged pass@5, ranked by $\Delta_5(\ell_a, \ell_b)$.

■ **Table 4** Average pass@5 when each language is used as source or target in directed translation, averaged over the five evaluated models.

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

Finding 2. The evaluated translation graph exhibits substantial source-target asymmetry. Several opposite directions differ by more than 20 percentage points in model-averaged pass@5, demonstrating that translation results should preserve ordered source-target pairs rather than collapse them into unordered pair averages.

4.3 RQ3: Source and Target Language Roles

RQ2 examines asymmetry between opposite edges of individual language pairs. We next aggregate outgoing and incoming edges to determine whether each language behaves differently in its source and target roles. Following the definitions in Section 2.2, Table 4 reports $\text{Src}_5(\ell)$, $\text{Tgt}_5(\ell)$, and their difference for each language.

Python3 and Java obtain substantially higher averages when used as source languages than when used as target languages. Python3 achieves 77.1% in the source role and 62.7% in the target role, while Java achieves 77.6% and 64.1%, respectively. C also obtains a moderately higher source-role average.

The opposite pattern appears for JavaScript, Rust, and Golang. JavaScript achieves 70.7% when used as a source but 84.0% when used as a target. Rust increases from 70.0% in the source role to 82.0% in the target role, while Golang increases from 74.2% to 81.7%. C++ and C# are comparatively balanced, with source-target differences close to zero.

This role-level analysis complements the opposite-edge comparisons in RQ2. Directionality is not confined to a small number of exceptional language pairs; it is also reflected in systematic differences between the outgoing and incoming edges associated with individual languages. The results therefore caution against characterizing a programming language as simply “easy” or “hard” for code translation. A language may yield relatively high success when its implementations provide the source representation but substantially different results when the same language is requested as the translation target.

These role differences should nevertheless be interpreted as properties of the evaluated models, problems, prompts, and execution environments rather than as intrinsic properties of the programming languages themselves. Accordingly, direction-aware evaluations should report source-role and target-role results separately.

Finding 3. Programming languages exhibit distinct source and target roles in the evaluated translation graph. Python3 and Java achieve higher source-role averages, whereas JavaScript, Rust, and Golang achieve higher target-role averages. C++ and C# remain comparatively balanced across the two roles.

5 Discussion and Threats to Validity

5.1 Implications for Direction-Aware Evaluation

Our results show that LLM-based code translation should not be summarized solely by an aggregate pass@ k score. Although such a score provides a coarse measure of overall capability, it conceals variation across ordered source-target directions. This distinction matters in practice because translating an existing JavaScript implementation into Java is a different task from translating Java into JavaScript.

We derive three reporting implications. First, translation evaluation should be *direction-aware*: source-target settings should be reported as ordered pairs rather than collapsed into unordered language-pair averages. Second, it should be *role-aware*: a language’s performance as a source should be distinguished from its performance as a target. Third, it should be *sampling-aware*: pass@1 and pass@ k should be reported separately because first-attempt reliability and multi-candidate success can produce different assessments of model capability.

These analyses complement rather than replace aggregate model-level results. Model-level scores support broad comparison, while edge-, pair-, and role-level results reveal where translation performance varies. The directed graph therefore serves not only as a visualization, but also as an organizational structure for source-target-specific evaluation and validation.

5.2 Interpreting the Translation Graph

An evaluation over eight languages induces $8 \times 7 = 56$ distinct translation directions rather than merely eight target-language settings. Each edge may involve a different combination of source comprehension and target realization requirements, including typing, memory management, ownership, library conventions, class structure, and submission interfaces.

The role-level results should be interpreted as evaluation indicators rather than language rankings. The Target–Source column in Table 4 measures the difference between a language’s target-role and source-role averages. A negative value indicates higher observed performance in the source role, whereas a positive value indicates higher performance in the target role. For example, Python3 and Java obtain higher source-role averages, while JavaScript, Rust,

and Golang obtain higher target-role averages. These observations do not imply that a language is intrinsically easy or difficult to translate. Instead, they indicate that source understanding and target implementation should be treated as distinct empirical roles.

Likewise, large opposite-direction gaps should be viewed as diagnostic signals rather than causal explanations. A gap such as JavaScript-to-Java versus Java-to-JavaScript may be associated with type recovery, interface reconstruction, or library mapping, but the aggregate results do not identify the underlying mechanism. Establishing such explanations requires direction-specific failure analysis and controlled experiments.

5.3 Relation to Practical Migration

This study uses online-judge programming problems because they provide multilingual reference implementations and executable correctness oracles, a design also adopted in multilingual code-evaluation benchmarks [7]. Using the same problems across all ordered source-target pairs enables controlled comparison of translation directions under matched problem semantics.

However, online-judge translation differs substantially from repository-level software migration. Real migration tasks involve multi-file structure, dependencies, build systems, framework APIs, test suites, performance constraints, maintainability, and human review [14]. Our findings should therefore be interpreted as evidence of direction-dependent behavior in controlled executable tasks rather than as a complete assessment of industrial migration quality.

Nevertheless, the direction-aware perspective extends naturally to repository-level settings. Future evaluations can treat each migration direction as a distinct configuration and report build success, test passing, API compatibility, performance, and maintainability separately for each source-target direction.

5.4 Threats to Validity

Study scope. The evaluation uses 100 online-judge programming problems with reference implementations in all eight languages. Although this design enables matched comparison across 56 translation directions, the problems may not represent the full range of difficulty levels, algorithmic topics, or language-specific idioms. The results should therefore be interpreted as preliminary evidence from the evaluated setting rather than as definitive benchmark rankings.

Evaluation setting. Online-judge tasks provide executable oracles and controlled multilingual comparison, but they do not capture the structural and organizational complexity of repository-level migration, including dependencies, build configurations, framework APIs, and maintainability requirements.

Implementation choices. Prompt wording, decoding parameters, post-processing rules, target-language templates, and source-implementation style may affect the measured results. We apply a common prompting and post-processing protocol across all directions, but the absolute pass rates may remain sensitive to these design choices.

Training exposure and correctness oracle. The public programming problems and reference implementations may have appeared in model training data, potentially affecting absolute performance and individual languages unevenly. Moreover, execution-based checking is

limited by the available tests and does not establish full semantic equivalence. These limitations caution against over-interpreting absolute scores, while leaving the observed need for direction-aware reporting intact.

6 Conclusion

This paper studied LLM-based code translation as a directed multilingual transfer problem. Using 100 fully aligned problems across eight languages, we evaluated 56 ordered source-target directions through execution-based checking. The results reveal substantial opposite-direction asymmetries and distinct source/target language roles, showing that aggregate scores can conceal important translation behavior. These findings motivate direction-aware reporting and further analysis of the causes of translation asymmetry.

References

- 1 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.
- 2 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.
- 3 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.
- 4 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.
- 5 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.
- 6 Mingsheng Jiao, Tingrui Yu, Xuan Li, Guan jie 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.
- 7 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.
- 8 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.
- 9 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.
- 10 Zhi Ma, Cheng Wen, Zhexin 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.

- 11 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.
- 12 Baptiste Roziere, Marie-Anne Lachaux, Lowik Chansussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, volume 33, pages 20601–20611, 2020.
- 13 Qingxiao Tao, Tingrui Yu, Xiaodong Gu, and Beijun Shen. Unraveling the potential of large language models in code translation: How far are we? In *2024 31st Asia-Pacific Software Engineering Conference (APSEC)*, pages 353–362. IEEE, 2024. doi:10.1109/APSEC65559.2024.00046.
- 14 Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, et al. Repotransbench: A real-world multilingual benchmark for repository-level code translation. *IEEE Transactions on Software Engineering*, 2025.
- 15 Yanlin Wang, Rongyi Ou, Yanli Wang, Mingwei Liu, Jiachi Chen, Ensheng Shi, Xilin Liu, Yuchi Ma, and Zibin Zheng. Effireasontrans: RL-optimized reasoning for code translation. *IEEE Transactions on Software Engineering*, 2026.
- 16 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.
- 17 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.
- 18 Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. Codetransocean: A comprehensive multilingual benchmark for code translation. *arXiv preprint*, 2023. doi:10.48550/arXiv.2310.04951.
- 19 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.
- 20 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.
- 21 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.