

CODEPIVOT: Bootstrapping Multilingual Transpilation in LLMs via Reinforcement Learning with Reduced Reliance on Parallel Corpora

Shangyu Li¹ Juyong Jiang² Meibo Ren¹ Sizhe Zhong¹ Huiji Tan¹ Yunhao Gou¹
Xu Han² Chun Yong Chong³ Yun Peng⁴ Jiasi Shen¹

¹The Hong Kong University of Science and Technology ³Monash University Malaysia

²The Hong Kong University of Science and Technology (Guangzhou)

⁴The Chinese University of Hong Kong

{sliaw, mrenad, sizhe.zhong, htanaj, ygou}@connect.ust.hk Chong.chunyong@monash.edu
{jjiang472, xhan696}@connect.hkust-gz.edu.cn ypeng@cse.cuhk.edu.hk sjs@cse.ust.hk

Abstract

Transpilation, or code translation, aims to convert source code from one programming language (PL) to another. Recent large language model (LLM)-based approaches have demonstrated immense potential for code transpilation. Training-based methods are important because LLMs currently do not effectively adapt to domain-specific settings that suffer from a lack of knowledge without targeted training. However, existing training-based approaches rely on a pairwise transpilation paradigm, making it impractical to support a diverse range of PLs. Furthermore, these methods shape the reinforcement learning (RL) reward linearly in the test-case pass rate, which underweights the earliest partial progress. To address these limitations, we propose CODEPIVOT, a training framework that leverages Python as an intermediate representation (IR), augmented by a novel RL reward mechanism, *Aggressive-Partial-Functional* reward, to bootstrap the model’s multilingual transpilation ability with reduced reliance on parallel corpora. Experiments involving 10 PLs show that the resulting 7B model, trained on Python-to-Others tasks, consistently improves performance across both general and low-resource PL-related transpilation tasks. It outperforms substantially larger mainstream models with hundreds of billions more parameters, such as DeepSeek-R1 and Qwen3-235B-A22B-Instruct-2507, on Python-to-Others tasks and Others-to-All tasks, respectively, and surpasses its counterpart trained directly on Any-to-Any tasks on general transpilation tasks. Code and data: <https://github.com/lishangyu-hkust/CodePivot>.

1 Introduction

Background. Transpilation, or code translation, is the conversion of source code from one programming language (PL) to another, with each ordered source-target PL pair constituting a distinct *transpilation task*. It supports key practical needs such as migrating legacy systems to modern platforms (Sneed, 2010; Ziftci et al., 2025), reducing long-term maintenance costs (Sneed, 2010), strengthening security by moving from unsafe PLs to safer ones (Emre et al., 2021; Ling et al., 2022; Hans et al., 2025; Sneed, 2010; Hong and Ryu, 2025, 2024; Nitin et al., 2025), and creating parallel corpora that improve code-synthesis training and mitigate data scarcity for low-resource PLs (Pan et al., 2024).

Why training-based methods. Despite decades of research, high-quality transpilation remains challenging: classical rule-based approaches (Wang et al., 2023) depend on heavy static analysis and extensive engineering, and program synthesis-based methods (Mariano et al., 2022; Terekhov and Verhoef, 2000) scale poorly due to search complexity and the cost of validation (§ 2). Recent large language models (LLMs) offer real-time transpilation (Macedo et al., 2025; Yang et al., 2024b) and may alleviate the scalability limitations of synthesis (Zhang et al., 2025; Wang et al., 2025b; Yuan et al., 2025). Among these approaches, training-based methods are particularly important because, without targeted training, LLMs often do not effectively adapt to domain-specific settings that suffer from a lack of knowledge (Cassano et al., 2024; Gururangan et al., 2020). This limitation is especially evident for low-resource PLs (Wang et al., 2025a) and in tasks such as migrating COBOL

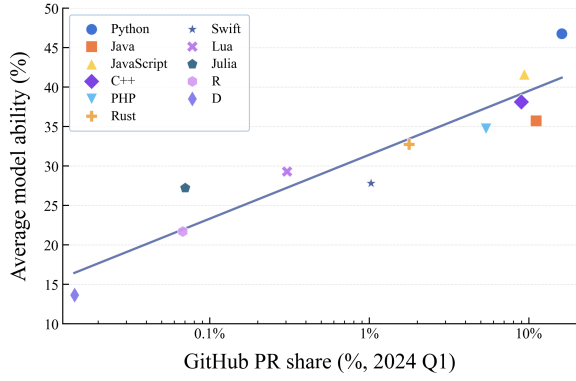


Figure 1: Correlation between programming language resource abundance and average model coding proficiency. The y-axis denotes the coding capability scores averaged across models evaluated by the Big Code Models Leaderboard (BigCode, 2023; Li et al., 2023), while the x-axis (logarithmic scale) represents the resource availability of each language, quantified by its respective percentage of GitHub pull requests in 2024 Q1 (Beuke, 2026). A logarithmic trendline is fitted to highlight the positive correlation.

to Java (Hans et al., 2025; Sneed, 2010), where COBOL remains critical to financial systems yet is difficult to maintain because of its decades-old design and limited developer expertise. Furthermore, with tools such as GitHub Copilot (Microsoft, 2026) and Codex (Chen et al., 2021), developer productivity has substantially improved, yet limited support for low-resource languages places their programmers at a disadvantage (Zheng et al., 2023; Zhang et al., 2024b).

As illustrated in Figure 1, a model’s overall coding proficiency exhibits a strong positive correlation with the resource availability of the target programming language. This dependency underscores a critical limitation of current models, which consistently suffer from degraded performance when applied to low-resource PLs.

Limitations of existing training-based methods. Existing training-based approaches rely on a pairwise transpilation paradigm, making it impractical to support a wide variety of PLs. Because these methods learn each task from data for that task (Wang et al., 2026b; Jana et al., 2024), supporting N languages requires coverage of $N(N - 1)$ tasks, whose number grows as $O(N^2)$, making it difficult in practice to construct sufficient data to improve the models’ transpilation capabilities. This challenge is further exacerbated by the scarcity of high-quality parallel data for low-resource PLs. Furthermore, although prior work (Wang et al.,

2026b; Jana et al., 2024) has demonstrated the potential of reinforcement learning (RL) for transpilation, existing approaches remain constrained by suboptimal reward formulations. These formulations either depend on heavyweight symbolic compiler feedback (Jana et al., 2024) or use an execution reward that is linear in the test-case pass rate (Wang et al., 2026b; Jana et al., 2024). The latter assigns credit uniformly across the pass-rate range and thus provides limited additional incentive during the earliest stages of partial correctness, when a transpilation attempt begins to pass only a small fraction of test cases.

Python as an IR. To close these gaps, we propose CODEPIVOT, a training framework that employs Python as an intermediate representation (IR) along with a novel reward mechanism, *Aggressive-Partial-Functional* reward, to bootstrap the model’s multilingual transpilation abilities with reduced reliance on parallel corpora and to harness the potential of RL. To solve a transpilation task from a source PL l_{src} to a target PL l_{tar} , we train a model using transpilation data consisting of Python-to- l_{src} and Python-to- l_{tar} tasks. In other words, we introduce Python as an IR or shared representation (Johnson et al., 2017) during training to facilitate transfer to the original transpilation task from l_{src} to l_{tar} . Through this shared Python representation, CODEPIVOT implicitly aligns the underlying semantics of diverse PLs within the model’s latent space, thereby enabling zero-shot transfer to unseen transpilation tasks.

Differences from prior IR-based methods. Some prior IR-based methods learn source-to-IR mappings from large parallel corpora, but either degrade on unseen transpilation tasks (Szafraniec et al., 2023) or primarily target code completion and understanding (Paul et al., 2024). Others explicitly translate through a pivot language at inference time (Macedo et al., 2025). In contrast, CODEPIVOT adopts a one-to-many training scheme in which a high-resource PL serves only as a shared semantic representation during training, while the model directly generates the target program at inference time. Training on Python-to-Others tasks improves not only the trained tasks but also unseen Others-to-Python and Others-to-Others tasks, avoiding both an inference-time pivot and the pairwise data construction required by Any-to-Any training.

Why an IR, and why Python. The motivation for using an IR stems from prior work (Szafraniec

et al., 2023), which demonstrated that a low-level compiler IR, such as LLVM IR (Lattner and Adve, 2004), can facilitate transpilation. However, this approach is fundamentally constrained to PLs that can be compiled to LLVM IR, such as C, C++, Java, and Rust, lacking extensibility to a broader spectrum of PLs. It also requires a massive number of program-to-IR pairs simply for the model to acquire a foundational understanding of the low-level IR (Paul et al., 2024; Szafraniec et al., 2023), rendering it vulnerable to data scarcity when applied to low-resource PLs.

These limitations motivate the idea that, rather than training models to understand low-level compiler IRs that current LLMs often do not master, it is more effective to adopt a high-resource PL as the IR. We select Python because it is widely recognized as a high-resource PL (Yang et al., 2025; Cassano et al., 2023) underpinned by abundant high-quality training corpora (Lozhkov et al., 2024; Ben Allal et al., 2024), it features concise and natural-language-inspired syntax (Yun et al., 2026), and code LLMs generally exhibit strong performance on Python compared with many other PLs (Cassano et al., 2023; Wang et al., 2026a). Other candidates such as JavaScript are also plausible; a systematic comparison of alternative IRs is orthogonal to our contribution and left to future work. Because Python is used only to abstract core functional behavior, our approach also avoids manually modeling language-specific features such as Rust’s ownership rules or pointer-level memory semantics. Under this design, we apply a Python-IR-pivoted supervised fine-tuning (SFT) strategy to bootstrap multilingual transpilation across 10 PLs, including low-resource ones, using only Python-to-Others pairs.

Aggressive-Partial-Functional reward. To further enhance performance during the RL phase, our newly proposed *Aggressive-Partial-Functional* reward emphasizes rollouts with programs that pass at least one but not all test cases (partially functionally correct), encouraging the model to turn more rollouts with programs that pass no test cases (functionally incorrect) into rollouts with partially functionally correct ones. This mechanism does not rely on heavyweight compiler symbolic feedback (Jana et al., 2024). Unlike prior execution-based rewards that scale linearly with the test-case pass rate (Wang et al., 2026b; Jana et al., 2024), our concave gating function assigns larger rewards to improvements at low pass rates. Consequently,

early transitions from incorrectness to partial correctness receive stronger learning signals, thereby accelerating training progress in the initial stages (Appendix H).

Dataset and experiments. We build a file-level transpilation dataset $\mathcal{D}_{\text{PY-TRANS}}$ based on PyEduR (Li et al., 2025a; Ben Allal et al., 2024; Lozhkov et al., 2024), containing roughly 1M Python-to-Others task instances. Target PLs span 9 languages with diverse paradigms, compilation models, type systems, and resource levels: C++, C#, Java, JavaScript, Golang, Perl, Ruby, Rust, and Haskell. Each instance contains a Python source program, language-agnostic JSON-formatted test cases, and metadata.

From $\mathcal{D}_{\text{PY-TRANS}}$ we derive two benchmarks: $\mathcal{D}_{\text{PY2OTHERSBENCH}}$, 900 Python-to-Others task instances evenly distributed across the 9 target PLs; and $\mathcal{D}_{\text{OTHERS2ALLBENCH}}$, an 8.1K-instance Others-to-All suite obtained by pairing 100 oracle programs per non-Python PL across the 9 source PLs with 10 target PLs (including Python). For training, we sample fewer than 30K instances without replacement, excluding all benchmark instances, and instantiate CODEPIVOT on a 7B-parameter base model Qwen2.5-Coder-7B-Instruct, trained exclusively on Python-to-Others transpilation.

Experiments show consistent improvements across general transpilation tasks and tasks involving low-resource PLs. The resulting model outperforms substantially larger mainstream models, including Qwen3-235B-A22B-Instruct-2507 and DeepSeek-R1, gaining 10.67 *Pass@1* percentage points (hereafter “points”) on tasks involving low-resource PLs and 15.19 points on Others-to-All tasks relative to Qwen3-235B-A22B-Instruct-2507, and outperforming DeepSeek-R1 by 5.34 points on Python-to-Others. In addition, the model trained on Python-to-Others transpilation is more effective than its counterpart trained directly on Any-to-Any pairs.

Contributions. Our main contributions are:

- To the best of our knowledge, we are the first to propose a training framework using Python as an intermediate representation to bootstrap the model’s multilingual transpilation capabilities without direct parallel corpora for every transpilation task. Our approach achieves multilingual transpilation training of as many as 10 PLs. Experiments show that it works well even with low-resource PLs whose training

data is scarce.

- We propose a novel RL reward, *Aggressive-Partial-Functional* reward, to enable our framework to improve models’ multilingual transpilation capabilities. Experiments show that by incorporating the reward for training Qwen2.5-Coder-7B-Instruct, our framework yields a 7B-parameter model that surpasses DeepSeek-R1 (671B) in our experiments.
- We conduct comprehensive experiments showing the effectiveness of CODEPIVOT in the Others-to-All and Python-to-Others settings, in which it surpasses models with hundreds of billions more parameters, and the efficiency of Python as an IR, which enables it to surpass its counterpart trained on Any-to-Any pairs by 5.83 *Pass@1* points.

2 Related work

Code translation. Transpilation, or code translation, has been studied for decades (Terekhov and Verhoef, 2000; Sneed, 2010) and has motivated extensive research. Early efforts primarily relied on rule-based methods (Wang et al., 2023; Sharpen, 2026; C2Rust, 2026; Cxgo, 2026; J2cstranslator, 2026) that require hand-crafted rules for each translation setting, which limit scalability and make it difficult to generalize across languages and complex programming patterns. More recently, learning-based and LLM-based approaches have attracted increasing research attention (Xue et al., 2025; Pan et al., 2024; Bhattarai et al., 2024; Ou et al., 2024; Ramos et al., 2024; Cassano et al., 2023; Feischl and Kern, 2025). Prior work strengthens the core code translation capability of LLMs by leveraging large-scale data and improved training algorithms (Puri et al., 2021; Zheng et al., 2023; Roziere et al., 2020; Paul et al., 2024; Szafraniec et al., 2023; Cassano et al., 2024), while other studies rely on prompting and pipeline orchestration to handle challenging cases (Macedo et al., 2025; Yang et al., 2024b). Although these methods often generalize well, they typically provide weaker guarantees of correctness. Some approaches investigate the synergy between rule-based methods and LLMs to balance correctness with generalization (Ibrahimzada et al., 2025; Bhatia et al., 2024; Yang et al., 2024a).

Reinforcement learning for code. Reinforcement learning (RL) has become an effective ap-

proach for improving the reasoning capabilities of LLMs (Guo et al., 2025; Shao et al., 2024; Li et al., 2022; Wang et al., 2024), with particularly notable gains in code- and mathematics-oriented domains. Recent studies have applied RL to enhance code generation, including using execution feedback for code synthesis (Gehring et al., 2025; Shojaee et al., 2023), training for Verilog program generation (Zhu et al., 2025), and leveraging synthesized test cases (Zeng et al., 2025). Beyond code generation, RL has also been explored for broader programming tasks, including issue resolution (Wei et al., 2025), code translation (Jana et al., 2024; Wang et al., 2026b), reasoning and planning (Alon and David, 2025), and source code summarization (Wang et al., 2020). Among these, the two RL-based transpilation approaches are the closest to our work. CoTran (Jana et al., 2024) rewards the policy using compiler feedback together with a symbolic-execution check that runs tests on a round-trip reconstruction of the source program; this requires a compiler and a test-generation tool for each language pair, and it has been instantiated only for Java and Python. EffiReasonTrans (Wang et al., 2026b) adopts a lighter execution-based reward equal to the fraction of test cases passed, and covers six transpilation tasks among Python, Java, and C++. Both learn each task from data for that task, whereas CODEPIVOT trains only on Python-to-Others tasks and shapes the reward concavely to emphasize the earliest partial progress.

3 Methodology

This section establishes the foundational concepts and outlines the design of CODEPIVOT.

3.1 Problem Formulation

CODEPIVOT focuses on source-to-source transpilation: translating a program from a source PL to a target PL.

Definition 1 (Test Cases). A *test case* (s_i, s_o) is a pair of input and output strings.

To simplify the execution interface, we format the test case inputs and outputs as JSON strings (Appendix D), which currently support numbers, booleans, strings, arrays, and objects. Extending to other data types or datasets requires only a task-specific serializer and execution harness.

Definition 2 (Programs). A *program* p is modeled as a function $p(s_i) = s_o$ that accepts an input string s_i and returns an output string s_o .

For simplicity, we focus exclusively on deterministic programs, since nondeterministic programs may produce different outputs for the same input and lead to unpredictable evaluation behavior. Extending CODEPIVOT to non-deterministic programs involves replacing the current exact-match execution harness with a distributional verifier.

Definition 3 (Functional Correctness). Given a set of test cases τ and a program p , p is *functionally correct* iff p passes all test cases in τ ; *partially functionally correct* iff p passes at least one but not all; and *functionally incorrect* iff p passes none.

Definition 4 (Behavioral Equivalence). Given τ and two programs p_1 and p_2 , p_1 is *behaviorally equivalent* to p_2 modulo τ iff they pass the same subset of test cases in τ .

Definition 5 (Transpilation Task Instance). A *transpilation task instance* is a tuple $(p_{\text{src}}, \tau, l_{\text{src}}, l_{\text{tar}})$, where p_{src} is the source program in a source PL l_{src} and τ is a finite set of test cases specifying its intended I/O behavior. A target program p_{tar} is a *valid* result iff it is in l_{tar} and is behaviorally equivalent to p_{src} modulo τ .

3.2 IR-pivoted Supervised Fine-Tuning

We use a hypothetico-deductive approach to formulate an IR-to-Others training paradigm. Let L denote the set of considered PLs and $r \in L$ the IR. Treating r as a shared semantic representation (Johnson et al., 2017), we posit the following hypotheses.

Hypothesis H1: Transfer to the reverse task. When training improves the l_s -to- l_t task, the learned relationship between the two PLs also improves the reverse l_t -to- l_s task, even though the latter is absent from the training data.

Hypothesis H2: IR-mediated transfer. For distinct non-IR languages $l_s, l_t \in L \setminus \{r\}$, improvements in both l_s -to- r and r -to- l_t jointly improve l_s -to- l_t transpilation without training on the l_s -to- l_t task.

Under these hypotheses, training only on the r -to- l task for every $l \in L \setminus \{r\}$ directly targets the IR-to-Others tasks. H1 predicts transfer to every untrained l -to- r task. H2 then combines the gains on these reverse tasks with those on the trained r -to- l tasks and predicts transfer to every l_s -to- l_t task among the non-IR PLs. Thus, training on only $|L| - 1$ tasks yields the testable prediction of improvements across all $|L|(|L| - 1)$ tasks, without

constructing task-specific training data for each one. We test H1 and H2 empirically in § 4.

Since our primary focus is on functional correctness, we use the IR only to abstract core functional behavior rather than to model language-specific features such as Rust’s ownership rules or pointer-level memory semantics. As discussed in § 1, we select Python as the IR because it is a widely used, high-resource PL (Yang et al., 2025; Wang et al., 2026a), has concise and natural-language-inspired syntax (Yun et al., 2026), and is a language on which code LLMs often exhibit strong performance (Cassano et al., 2023; Wang et al., 2026a). Other high-resource PLs could, in principle, serve the same role, and we leave a systematic comparison of alternative IRs to future work.

Leveraging Python, we apply a Python-IR-pivoted supervised fine-tuning (SFT) approach to bootstrap multilingual transpilation. We sample Python-to-Others transpilation tasks without replacement from $\mathcal{D}_{\text{PY-TRANS}}$ and use this subset to distill Claude-Sonnet-4. We employ execution-based rejection sampling, retaining only responses yielding a functionally correct p_{tar} . Each validated response paired with its prompt forms a distillation pair, populating the SFT dataset $\mathcal{D}_{\text{PY2OTHERS-SFT}}$ with approximately 20K pairs distributed across the 9 target PLs. CODEPIVOT then performs SFT strictly on these Python-to-Others pairs as illustrated in Figure 2, bootstrapping the model’s general transpilation capability without requiring extensive parallel corpora for every language combination.

Consequently, this methodology bootstraps the model’s general transpilation capability while mitigating the need to train on extensive parallel corpora for every language combination.

3.3 Reinforcement Learning Augmented by Aggressive-Partial-Functional Reward

During the *Aggressive-Partial-Functional* reward-augmented RL stage (Figure 2), for each target PL we sample without replacement 800 Python-to-Others task instances distinct from those in $\mathcal{D}_{\text{PY2OTHERS-SFT}}$ to construct the RL training set $\mathcal{D}_{\text{PY2OTHERS-RL}}$.

Given a transpilation task instance $(p_{\text{src}}, \tau, l_{\text{src}}, l_{\text{tar}})$, let q denote its natural-language prompt and $\pi_\theta(\cdot | q)$ a conditional text-generation model. The instance is successfully solved by π_θ iff there exists $o \sim \pi_\theta(\cdot | q)$ that contains a program p_{tar} written in l_{tar} and behaviorally

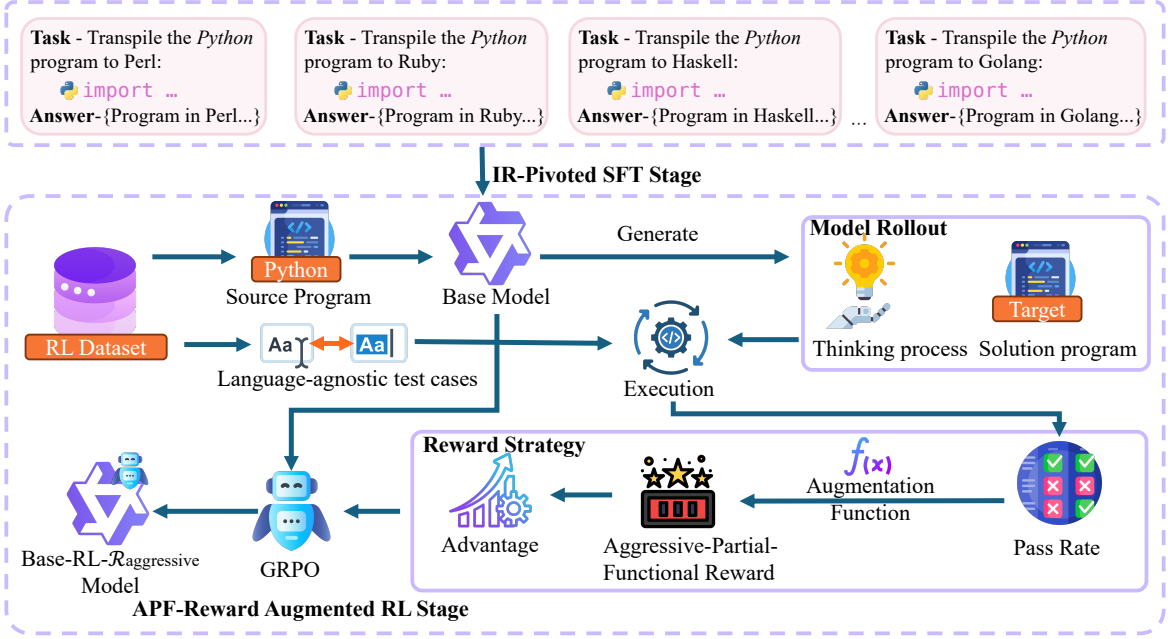


Figure 2: The workflow of the training process of CODEPIVOT. The core components of the workflow consist of the IR-pivoted SFT stage (Section 3.2) and the *Aggressive-Partial-Functional* reward (APF) augmented RL stage (Section 3.3).

equivalent to p_{src} .

During RL, after π_θ generates a rollout o , we compute its reward. Conventional linear execution-based schemes assign rewards purely based on the number of passed test cases and ignore the underlying program state. We classify the extracted p_{tar} into three states: functionally incorrect (lacks essential main logic), partially functionally correct (largely implements the main logic but fails some corner cases), and functionally correct. We hypothesize that a functionally incorrect program cannot easily become partially functionally correct, while a partially functionally correct program can more readily become functionally correct because it already has nearly correct main logic.

Based on this intuition, we propose the *Aggressive-Partial-Functional* reward that assigns a higher reward weight to rollouts whose programs are partially functionally correct.

Definition 6 (*Aggressive-Partial-Functional Reward*).

$$\mathcal{R}(o) = \sigma\left(\frac{1}{|\tau|} \left| \{(s_i, s_o) \in \tau \mid p_o(s_i) = s_o\} \right| \right) + \mathcal{R}_0, \quad (1)$$

where p_o is extracted from o and $\mathcal{R}_0$ is the format-compliance reward. We check for the required `<think>` and `<answer>` tags and a fenced code

block. $\mathcal{R}_0 = 0.2$ when both checks pass, 0.1 when one passes, and 0 otherwise. The concave gate function σ is defined as follows:

$$\sigma(x) = \frac{1 - \lambda^{-x}}{1 - \lambda^{-1}}, \quad \lambda > 1, \quad (2)$$

with λ controlling the extent to which partial functional correctness is rewarded.

This reward encourages the policy to convert rollouts with functionally incorrect programs into rollouts with partially functionally correct ones. We apply the standard GRPO algorithm (Shao et al., 2024) for RL training; details in Appendix H.

4 Evaluation

This section evaluates CODEPIVOT across multiple dimensions: its effectiveness across multiple PLs, the effectiveness of its training pipeline with *Aggressive-Partial-Functional* reward, the efficiency of the IR-based design, and the impact of different RL reward designs via ablation.

4.1 Experimental Setup

Implementation. We use LLaMAFactory (Zheng et al., 2024) for SFT and verl (Sheng et al., 2025) for RL on 8 H800-80G GPUs. Full implementation details, including hyperparameters and the program-execution environment, are in Appendix C.

Evaluation metrics. We adopt $Pass@k$: given a transpilation task instance $(p_{src}, \tau, l_{src}, l_{tar})$ and k generated programs, $Pass@k$ indicates that at least one generated program is a valid result of the instance. Since each p_{src} in our benchmarks is functionally correct modulo τ , $Pass@k$ also indicates one of the k programs is functionally correct. Throughout, differences between two $Pass@1$ values are reported in percentage points, and % is reserved for $Pass@1$ levels, proportions, and relative changes.

LLM settings. Beyond our trained model, we evaluate existing mainstream LLMs that vary in institution, parameter count, open-source availability, data cutoff, and pretraining objective. We set temperature to 0 (greedy decoding) for consistency. Untrained baselines are accessed through official APIs; our trained models are served via VLLM (Kwon et al., 2023). We report the average $Pass@1$ over three runs.

Dataset and benchmarks. $\mathcal{D}_{PY-TRANS}$ is a file-level Python-to-Others transpilation dataset of roughly 1M task instances built on PyEdu-R (Li et al., 2025a; Ben Allal et al., 2024; Lozhkov et al., 2024). Each instance contains a Python source program p_{src} , language-agnostic JSON-formatted test cases, and metadata. Source programs span algorithms, logic puzzles, math, scientific computation, and system modeling, and they invoke external APIs (e.g., container management and file I/O) to reflect real-world migration scenarios. Construction details and sampling strategy are in Appendix D.

For training, we sample fewer than 30K instances from $\mathcal{D}_{PY-TRANS}$ (excluding benchmark instances) to form $\mathcal{D}_{PY2OTHERS-SFT}$ and $\mathcal{D}_{PY2OTHERS-RL}$ due to computational overhead. To systematically evaluate CODEPIVOT, we construct two benchmarks. $\mathcal{D}_{PY2OTHERSBENCH}$ contains 900 Python-to-Others task instances evenly distributed across 9 tasks, sampled from $\mathcal{D}_{PY-TRANS}$ with structurally complex Python sources. $\mathcal{D}_{OTHERS2ALLBENCH}$ contains 8.1K Others-to-All task instances spanning 81 tasks (100 instances per task): we distill Claude-Sonnet-4 on Python-to-Others tasks and execution-filter the results to obtain 100 oracle programs per non-Python PL, then pair them with each distinct target PL among the 10 PLs considered.

In contrast to existing benchmarks that are limited to function- or class-level transpilation (Cas-

sano et al., 2023; Xue et al., 2025), cover narrow PL sets (Ahmad et al., 2023; Roziere et al., 2020; Zheng et al., 2023), or exhibit limited problem diversity (Ahmad et al., 2023; Roziere et al., 2020), our benchmarks offer file-level scope, multilingual coverage, and richer diversity. We use the mainstream LLMs listed in Table 6 (Appendix F) as baselines. No directly comparable IR-based approach exists for our setting: a full-coverage LLVM comparison is infeasible because JavaScript, Python, Perl, and Ruby lack a uniform off-the-shelf LLVM pipeline, and restricting coverage would exclude the dynamically typed and low-resource PLs we target; IRCoder (Paul et al., 2024) performs continued pretraining on source code and LLVM IR for code completion and understanding rather than file-level transpilation.

4.2 Main Results

Training Qwen2.5-Coder-7B-Instruct exclusively on Python-to-Others tasks yields an SFT variant Base-7B-SFT on $\mathcal{D}_{PY2OTHERS-SFT}$ and an RL variant Base-7B-RL- $\mathcal{R}_{aggressive}$ via *Aggressive-Partial-Functional* reward-augmented RL on $\mathcal{D}_{PY2OTHERS-RL}$. We evaluate zero-shot generalization on $\mathcal{D}_{OTHERS2ALLBENCH}$.

Table 1 presents representative results for Base-7B, Base-7B-SFT, Base-7B-RL- $\mathcal{R}_{aggressive}$, and three baselines (Qwen3-235B-A22B-Instruct-2507, DeepSeek-V3.1, and Qwen3-Next-80B-A3B-Instruct). Base-7B-SFT outperforms Base-7B on all 81 Others-to-All tasks, by 44.49 points on average. Base-7B-RL- $\mathcal{R}_{aggressive}$ further improves upon Base-7B-SFT on 98.77% of the tasks, adding 8.55 points on average. Despite having substantially fewer parameters, Base-7B-RL- $\mathcal{R}_{aggressive}$ outperforms the three baselines on 91.36%, 90.12%, and 93.83% of the tasks, by 15.19, 20.83, and 21.22 points on average. The baselines remain stronger on dynamically typed targets (Python, Perl, JavaScript) and a subset of Rust tasks, which may be partly attributed to Rust’s paradigm-blending design (Yun et al., 2026; Matsakis and Klock, 2014).

Hypothesis assessment. These results support both hypotheses: Base-7B-SFT improves over Base-7B on all nine untrained Others-to-Python tasks, supporting H1, and on all 72 untrained Others-to-Others tasks, supporting H2. Together, they support our prediction that training on the nine Python-to-Others tasks transfers to all 81 untrained

Table 1: Performance comparison in *Pass@1* (%) of models on **representative** tasks of $\mathcal{D}_{\text{OTHERS2ALLBENCH}}$. Columns specify source PLs from **C++** to **Haskell**, and row blocks specify target PLs. Base-7B denotes the base model, and Base-7B-SFT and Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ denote the resulting models further trained. Full results in Table 5 in Appendix E.

Languages	Model	From C++	From Java	From C#	From JS	From Golang	From Rust	From Perl	From Ruby	From Haskell
To Perl	Qwen3-235B-A22B-Instruct-2507	66.00	65.67	64.67	64.00	60.00	80.33	N/A	64.67	68.33
	DeepSeek-V3.1	82.00	69.33	81.67	80.67	75.67	78.67	N/A	71.33	70.67
	Qwen3-Next-80B-A3B-Instruct	72.33	63.67	73.67	67.00	69.33	68.00	N/A	58.67	69.33
	Base-7B (Qwen2.5-Coder-7B-Instruct)	29.33	27.67	40.00	32.00	34.00	39.00	N/A	19.67	11.67
	Base-7B-SFT	73.67	64.33	72.00	75.33	65.33	64.00	N/A	66.00	54.00
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$ (Ours)	74.00	70.33	83.33	81.00	76.33	69.00	N/A	72.00	71.00
To Ruby	Qwen3-235B-A22B-Instruct-2507	66.67	65.67	69.33	58.00	55.33	66.67	59.00	N/A	58.67
	DeepSeek-V3.1	44.00	57.33	62.33	25.67	46.00	45.00	43.33	N/A	62.00
	Qwen3-Next-80B-A3B-Instruct	57.00	55.00	58.67	49.00	53.00	51.67	43.00	N/A	58.00
	Base-7B (Qwen2.5-Coder-7B-Instruct)	51.00	47.33	40.67	43.67	42.00	43.00	41.67	N/A	25.67
	Base-7B-SFT	78.00	72.67	60.00	74.67	68.00	63.00	71.00	N/A	54.67
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$ (Ours)	84.33	83.33	78.00	86.00	78.00	78.00	80.00	N/A	67.67
To Haskell	Qwen3-235B-A22B-Instruct-2507	10.33	15.67	17.00	9.67	12.67	20.33	11.00	11.33	N/A
	DeepSeek-V3.1	15.67	12.00	12.33	10.00	8.67	17.33	14.33	10.33	N/A
	Qwen3-Next-80B-A3B-Instruct	0.67	1.00	1.00	0.00	1.00	2.67	1.00	0.00	N/A
	Base-7B (Qwen2.5-Coder-7B-Instruct)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	N/A
	Base-7B-SFT	30.33	25.00	21.00	29.00	22.33	36.33	21.00	24.67	N/A
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$ (Ours)	31.33	28.67	36.00	41.00	37.67	43.33	31.33	34.33	N/A

tasks.

Performance on tasks involving low-resource PLs. Figure 3 compares High-to-Low (H-L), Low-to-High (L-H), and Low-to-Low (L-L) tasks that pair high-resource PLs (Python, C++, Java, JavaScript) with low-resource PLs (Rust, Perl, Haskell) (Cassano et al., 2023; TIOBE, 2026; Cassano et al., 2024). Averaging the three per-task means, Base-7B-SFT outperforms Base-7B by 42.00 points and Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ adds a further 7.99 points; Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ surpasses the three baselines by 10.67, 14.09, and 18.16 points, indicating that CODEPIVOT advances low-resource transpilation while mitigating parallel-corpora scarcity. Per-task breakdowns are in Appendix E.

4.3 Effectiveness of the Training Pipeline

We directly evaluate CODEPIVOT’s training pipeline on $\mathcal{D}_{\text{PY2OTHERSBENCH}}$. Table 2 reports Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ alongside mainstream LLMs. Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ achieves the highest overall *Pass@1*, surpassing DeepSeek-R1 (671B) and Claude-Sonnet-4 by 5.34 and 14.08 points. It is best on 6 of 9 Python-to-Others tasks, with an average margin of 6.78 points over the second-best model per language. The short-falls on Python-to-JavaScript, Python-to-Golang, and Python-to-Rust are attributable to JavaScript’s high-resource prevalence and to Golang/Rust’s

paradigm-blending designs (Yun et al., 2026; Matsakis and Klock, 2014); even so, relative to Base-7B, Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ improves *Pass@1* by 1671.77% on Python-to-Golang and 1555.67% on Python-to-Rust. Full results are in Appendix F.

4.3.1 Ablation of Different RL Rewards

We compare *Aggressive-Partial-Functional* reward against three alternative rewards trained from Base-7B-SFT on $\mathcal{D}_{\text{PY2OTHERS-RL}}$: (1) a *linear execution* reward $\sigma(x) = x$; (2) a *Conservative-Partial-Functional* reward with a convex gate $\sigma(x) = \frac{\lambda^x - 1}{\lambda - 1}$ ($\lambda > 1$) that emphasizes fully correct roll-outs; and (3) a *discrete* reward $\sigma(x) = [x = 1]$ that rewards only fully correct programs. We fix $\lambda = 20$ throughout. At 25/50/75/100% pass rates, the functional reward is about 0.555/0.817/0.941/1, emphasizing early progress while still distinguishing partially correct programs.

Table 3 shows that all four RL variants outperform Base-7B-SFT on every Python-to-Others task, confirming the value of RL. Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ achieves the best overall *Pass@1* (64.67%), leads on 7 of 9 PLs, and yields a 12.71-point overall RL gain. This corresponds to absolute gains of 3.41, 3.78, and 4.11 points over the linear, conservative, and discrete rewards, respectively, or 36.67%, 42.33%, and 47.79% larger RL-stage improvements. There are two exceptions. Base-7B-RL- $\mathcal{R}_{\text{linear}}$, the second-best variant overall, leads on Python-to-Rust; we attribute this to

Table 2: Performance comparison in terms of $Pass@1$ (%) of representative models on $\mathcal{D}_{PY2OTHERSBENCH}$. Columns correspond to the nine Python-to-Others tasks, from **Python-to-C++** through **Python-to-Haskell**. The **Total** column presents the overall $Pass@1$ score across $\mathcal{D}_{PY2OTHERSBENCH}$. Full results in Table 6 in Appendix F.

Model	C++	Java	C#	JavaScript	Golang	Rust	Perl	Ruby	Haskell	Total
Claude-Sonnet-4	20.33	64.67	60.67	68.00	49.67	65.33	67.33	24.33	35.00	50.59
DeepSeek-R1	35.67	70.00	63.33	77.00	67.67	63.00	67.33	67.67	22.33	59.33
Qwen3-235B-A22B-Instruct-2507	32.33	40.00	21.67	30.33	25.33	28.33	26.67	20.00	6.67	25.70
Base-7B (Qwen2.5-Coder-7B-Instruct)	1.33	3.67	0.00	5.67	3.33	3.00	4.33	10.67	0.00	3.56
Base-7B-RL- $\mathcal{R}_{aggressive}$ (Ours)	68.67	70.00	69.00	74.33	59.00	49.67	72.00	84.00	35.33	64.67

Table 3: Performance comparison in $Pass@1$ (%) of models trained with different rewards on $\mathcal{D}_{PY2OTHERSBENCH}$. The **Total** column reports the overall $Pass@1$ across $\mathcal{D}_{PY2OTHERSBENCH}$; the value in parentheses is the gain in $Pass@1$ percentage points over Base-7B-SFT. Further trained on Base-7B-SFT using $\mathcal{D}_{PY2OTHERS-RL}$, Base-7B-RL- $\mathcal{R}_{linear}$, Base-7B-RL- $\mathcal{R}_{conservative}$, Base-7B-RL- $\mathcal{R}_{discrete}$, and Base-7B-RL- $\mathcal{R}_{aggressive}$ denote the resulting models using the linear execution reward, the Conservative-Partial-Functional reward, the discrete reward, and the *Aggressive-Partial-Functional* reward in RL, respectively.

Model	C++	Java	C#	JavaScript	Golang	Rust	Perl	Ruby	Haskell	Total
Base-7B-SFT	56.67	54.67	54.67	65.33	43.00	40.33	64.00	65.67	23.33	51.96
Base-7B-RL- $\mathcal{R}_{linear}$	62.67	67.00	66.33	66.00	58.33	53.67	69.33	80.67	27.33	61.26 (+9.30)
Base-7B-RL- $\mathcal{R}_{conservative}$	61.33	65.67	67.67	69.33	56.00	47.67	74.00	78.33	28.00	60.89 (+8.93)
Base-7B-RL- $\mathcal{R}_{discrete}$	65.00	66.33	61.67	69.00	57.00	51.00	70.33	76.67	28.00	60.56 (+8.60)
Base-7B-RL- $\mathcal{R}_{aggressive}$ (Ours)	68.67	70.00	69.00	74.33	59.00	49.67	72.00	84.00	35.33	64.67 (+12.71)

Table 4: Performance comparison in $Pass@1$ (%) on $\mathcal{D}_{OTHERS2ALLBENCH}$ between two models fine-tuned on Python-to-Others pairs and Any-to-Any pairs, respectively.

SFT Data Type	Model Performance (%)
Python-to-Others pairs	53.85
Any-to-Any pairs	48.02

Rust’s syntactic complexity (Yun et al., 2026; Matsakis and Klock, 2014), which makes it difficult to convert rollouts with partially functionally correct programs into those with functionally correct ones. Base-7B-RL- $\mathcal{R}_{conservative}$ leads only on Python-to-Perl (74.00%), which we attribute to Perl’s simpler syntax and high initial performance (64.00%), so that partially correct rollouts are already plentiful and emphasizing them adds little. Conversely, on Python-to-Haskell, the weakest task, *Aggressive-Partial-Functional* reward improves $Pass@1$ from 23.33% to 35.33%, a 12-point gain that is three times the 4-point gain of the linear reward. Emphasizing partial test-case success is particularly beneficial where baseline performance is low.

4.4 Efficiency of IR-pivoted Training

We evaluate the efficiency of the IR-based design by comparing SFT on Python-to-Others pairs against SFT on Any-to-Any pairs. From

$\mathcal{D}_{PY2OTHERS-SFT}$ (~20K Python-to-Others pairs), we build an Any-to-Any SFT dataset by permuting functionally correct target programs across languages within each source-program group, since all programs in a group solve the same problem. This yields ~160K Any-to-Any pairs, not balanced across tasks because distillation does not always yield a correct program for every target.

We fine-tune Qwen2.5-Coder-7B-Instruct on each dataset for three epochs. Table 4 presents $\mathcal{D}_{OTHERS2ALLBENCH}$ performance: the model trained on 20K Python-to-Others pairs outperforms the one trained on 160K Any-to-Any pairs by 5.83 points, indicating that IR-pivoted training is more data-efficient than training directly on parallel pairs across languages.

5 Conclusion

We present CODEPIVOT, combining Python-IR-pivoted SFT, *Aggressive-Partial-Functional* reward-augmented RL, and language-agnostic test cases to bootstrap multilingual transpilation without parallel corpora for every task. Our results show that (1) Python as the IR transfers Python-to-Others gains to untrained tasks; (2) concave reward shaping strengthens RL by emphasizing early partial progress; and (3) together, they enable a 7B model to outperform substantially larger models on our benchmarks.

Limitations

In our evaluation and training, we rely on currently mainstream large language models, which may limit the validity of our conclusions for future model generations. To mitigate this risk, we evaluate multiple models from different institutions with varying parameter scales and open-source availability, ensuring that our findings reflect a broad set of models. We train the model on only 10 PLs that are mainstream at present, and it remains unclear whether the approach can scale to a substantially larger set of languages. To mitigate this limitation, we select languages to cover diverse type systems, resource availability, and programming paradigms, and we focus on widely used and well-known languages that are broadly representative. Due to the computational overhead, we sample only around 10 test cases for each program to keep reinforcement learning computationally feasible. The limited number of test cases may not fully characterize the function of the program, but our approach is to approximate the behavior represented by the given set of test cases, which the experimental results demonstrate to be effective, and the approach naturally incorporates more test cases when more computational resources are available.

Acknowledgments

This work was supported in part by the Hong Kong Research Grants Council (Project No. 26216025) and the Guangdong Science and Technology Department (Internal Project No. G_2025_082). The views expressed in this work are those of the authors and do not necessarily reflect the views of the funding agencies.

References

- Wasi Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2023. Avatar: A parallel corpus for java-python program translation. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 2268–2281.
- Yoav Alon and Cristina David. 2025. [Integrating large language models and reinforcement learning for non-linear reasoning](#). *Proc. ACM Softw. Eng.*, 2(FSE).
- Loubna Ben Allal, Anton Lozhkov, Guilherme Penedo, Thomas Wolf, and Leandro von Werra. 2024. SmolLM-Corpus. <https://huggingface.co/datasets/HuggingFaceTB/SmolLM-corpus>. Dataset.
- Fabian Beuke. 2026. Github 2.0: Github language statistics. <https://madnight.github.io/github/#/>. GitHub repository.
- Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit A Seshia, and Alvin Cheung. 2024. Verified code transpilation with llms. *Advances in Neural Information Processing Systems*, 37:41394–41424.
- Manish Bhattarai, Javier E Santos, Shawn Jones, Ayan Biswas, Boian Alexandrov, and Daniel O’Malley. 2024. Enhancing code translation in language models with few-shot learning via retrieval-augmented generation. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE.
- BigCode. 2023. Bigcode models leaderboard. <https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard>. Accessed: 2024-05-24.
- C2Rust. 2026. [C2rust](#).
- Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge transfer from high-resource to low-resource programming languages for code llms. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):677–708.
- Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. [Multipl-e: A scalable and polyglot approach to benchmarking neural code generation](#). *IEEE Trans. Softw. Eng.*, 49(7):3675–3691.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Wei Chen, Bowen Zhang, Chengpeng Wang, Wensheng Tang, and Charles Zhang. 2025. Seal: Towards diverse specification inference for linux interfaces from security patches. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1246–1262.
- Cxgo. 2026. [Cxgo](#).
- Jack W Davidson and Christopher W Fraser. 1980. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(2):191–202.
- Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating c to safer rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29.

- Christof Feischl and Roman Kern. 2025. Large language models for code translation: An in-depth analysis of code smells and functional correctness. *ACM Transactions on Software Engineering and Methodology*.
- Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2025. [RLEF: Grounding code LLMs in execution feedback with reinforcement learning](#). In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 19034–19055. PMLR.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. [Deepseek-r1 incentivizes reasoning in llms through reinforcement learning](#). *Nature*, 645(8081):633–638.
- Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. 2020. [Don’t stop pretraining: Adapt language models to domains and tasks](#). In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8342–8360, Online. Association for Computational Linguistics.
- Sandeep Hans, Atul Kumar, Toshikai Yasue, Kouichi Ono, Saravanan Krishnan, Devika Sondhi, Fumiko Satoh, Gerald Mitchell, Sachin Kumar, and Diptikalyan Saha. 2025. Automated testing of cobol to java transformation. *arXiv preprint arXiv:2504.10548*.
- Jaemin Hong and Sukyoung Ryu. 2024. To tag, or not to tag: Translating c’s unions to rust’s tagged unions. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 40–52.
- Jaemin Hong and Sukyoung Ryu. 2025. Type-migrating c-to-rust translation using a large language model. *Empirical Software Engineering*, 30(1):3.
- Harry D Huskey, Maurice H Halstead, and R McArthur. 1960. Neliac—dialect of algol. *Communications of the ACM*, 3(8):463–468.
- Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2025. Alpha-trans: A neuro-symbolic compositional approach for repository-level code translation and validation. *Proceedings of the ACM on Software Engineering*, 2(FSE):2454–2476.
- J2cstranslator. 2026. [J2cstranslator](#).
- Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2024. Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution. In *ECAI 2024*, pages 4011–4018. IOS Press.
- Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2017. [Google’s multilingual neural machine translation system: Enabling zero-shot translation](#). *Transactions of the Association for Computational Linguistics*, 5:339–351.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.
- Chris Lattner and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE.
- Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. 2025a. [CodeIO: Condensing reasoning patterns via code input-output prediction](#). In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 34471–34489. PMLR.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.
- Shangyu Li, Juyong Jiang, Tiancheng Zhao, and Jiasi Shen. 2026. [OSVBench: Benchmarking LLMs on specification generation tasks for operating system verification](#). *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(37):31698–31707.
- Shangyu Li, Zhaoyang Zhang, Sizhe Zhong, Diyu Zhou, and Jiasi Shen. 2025b. A sound static analysis approach to i/o api migration. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):584–614.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.
- Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R Cordy, and Ahmed E Hassan. 2022. In rust we trust: a transpiler from unsafe c to safer rust. In *Proceedings of the ACM/IEEE 44th international conference on software engineering: companion proceedings*, pages 354–355.
- Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79.

- Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–32.
- Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.
- Marcos Macedo, Yuan Tian, Pengyu Nie, Filipe R. Cogo, and Bram Adams. 2025. *InterTrans: Leveraging Transitive Intermediate Translations to Enhance LLM-Based Code Translation*, page 1153–1164. IEEE Press.
- Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işıl Dillig. 2022. Automated transpilation of imperative to functional code using neural-guided program synthesis. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27.
- Nicholas D Matsakis and Felix S Klock. 2014. The rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*, pages 103–104.
- Jason Merrill. 2003. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Summit*, pages 171–180.
- Microsoft. 2026. *GitHub Copilot – Your AI pair programmer*. GitHub repository.
- Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 252–269.
- Vikram Nitin, Rahul Krishna, Luiz Lemos do Valle, and Baishakhi Ray. 2025. C2saferrust: Transforming c projects into safer rust with neurosymbolic techniques. *arXiv preprint arXiv:2501.14257*.
- Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Xin Peng, and Zibin Zheng. 2024. Repository-level code translation benchmark targeting rust. *arXiv preprint arXiv:2411.13990*.
- Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.
- Maulishree Pandey, Steve Oney, and Andrew Begel. 2024. Towards inclusive source code readability based on the preferences of programmers with visual impairments. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pages 1–18.
- Indraneil Paul, Goran Glavaš, and Iryna Gurevych. 2024. *IRCoder: Intermediate representations make language models robust multilingual code generators*. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15023–15041, Bangkok, Thailand. Association for Computational Linguistics.
- Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, and 1 others. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.
- Daniel Ramos, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2024. Batfix: Repairing language model-based transpilation. *ACM Transactions on Software Engineering and Methodology*, 33(6):1–29.
- Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanasot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS '20*, Red Hook, NY, USA. Curran Associates Inc.
- John Schulman. 2020. *Approximating KL divergence*. Accessed: 2025-02-22.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.
- Sharpen. 2026. *Sharpen*.
- Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297.
- Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 693–706.
- Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. *Execution-based code generation using deep reinforcement learning*. *Transactions on Machine Learning Research*.
- Harry M Sneed. 2010. Migrating from cobol to java. In *2010 IEEE International Conference on Software Maintenance*, pages 1–7. IEEE.

- J Strong, J Wegstein, A Tritter, J Olsztyn, O Mock, and T Steel. 1958. The problem of programming communication with changing machines: a proposed solution. *Communications of the ACM*, 1(8):12–18.
- Yulei Sui and Jingling Xue. 2016a. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 460–473.
- Yulei Sui and Jingling Xue. 2016b. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266.
- Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 254–264.
- Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122.
- Marc Szafraniec, Baptiste Roziere, Hugh James Leather, Patrick Labatut, Francois Charton, and Gabriel Synnaeve. 2023. [Code translation with compiler representations](#). In *The Eleventh International Conference on Learning Representations*.
- Huiri Tan, Juyong Jiang, and Jiasi Shen. 2025. [Prefix: Improving profile-guided optimization in compilers with graph neural networks](#). In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pages 118278–118303. Curran Associates, Inc.
- Wensheng Tang, Dejun Dong, Shijie Li, Chengpeng Wang, Peisen Yao, Jinguo Zhou, and Charles Zhang. 2024. Octopus: Scaling value-flow analysis via parallel collection of realizable path conditions. *ACM Transactions on Software Engineering and Methodology*, 33(3):1–33.
- Andrey A. Terekhov and Chris Verhoef. 2000. [The realities of language conversions](#). *IEEE Softw.*, 17(6):111–124.
- TIOBE. 2026. [Tiobe](#).
- Bo Wang, Aashish Kolluri, Ivica Nikolić, Teodora Bala, and Prateek Saxena. 2023. User-customizable transpilation of scripting languages. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):201–229.
- Chaozheng Wang, Zongjie Li, Cuiyun Gao, Wenxuan Wang, Ting Peng, Hailiang Huang, Yuetang Deng, Shuai Wang, and Michael Lyu. 2026a. [Exploring multi-lingual bias of large code models in code generation](#). *ACM Trans. Softw. Eng. Methodol.*, 35(8).
- Jun Wang, Chenghao Su, Yijie Ou, Yanhui Li, Jialiang Tan, Lin Chen, and Yuming Zhou. 2025a. Translating to a low-resource language with compiler feedback: A case study on cangjie. *IEEE Transactions on Software Engineering*.
- Junqiao Wang, Zeng Zhang, Yangfan He, Zihao Zhang, Xinyuan Song, Yuyang Song, Tianyu Shi, Yuchen Li, Hengyuan Xu, Kunyu Wu, and 1 others. 2024. Enhancing code llms with reinforcement learning in code generation: A survey. *arXiv preprint arXiv:2412.20367*.
- Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip S Yu, and Guandong Xu. 2020. Reinforcement-learning-guided source code summarization using hierarchical attention. *IEEE Transactions on software Engineering*, 48(1):102–119.
- Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and 1 others. 2025b. Repotransbench: A real-world multilingual benchmark for repository-level code translation. *IEEE Transactions on Software Engineering*.
- Yanlin Wang, Rongyi Ou, Yanli Wang, Mingwei Liu, Jiachi Chen, Ensheng Shi, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2026b. [Effreasontrans: RL-optimized reasoning for code translation](#). *IEEE Transactions on Software Engineering*, 52(8):2354–2366.
- Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, LINGMING ZHANG, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. 2025. [Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution](#). In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pages 78500–78525. Curran Associates, Inc.
- Pengyu Xue, Linhao Wu, Zhen Yang, Chengyi Wang, Xiang Li, Yuxiang Zhang, Jia Li, Ruikai Jin, Yifei Pei, Zhaoyan Shen, and 1 others. 2025. Classeval-t: Evaluating large language models in class-level code translation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1421–1444.
- Aidan ZH Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. 2024a. Vert: Verified equivalent rust transpilation with large language models as few-shot learners. *arXiv preprint arXiv:2404.18852*.
- Jian Yang, Shawn Guo, Lin Jing, Wei Zhang, Aishan Liu, Chuan Hao, Zhoujun Li, Wayne Xin Zhao, Xianglong Liu, Weifeng Lv, and 1 others. 2025. Scaling laws for code: Every programming language matters. *arXiv preprint arXiv:2512.13472*.
- Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024b. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1585–1608.

- Zhiqiang Yuan, Wenjun Mao, Zhuo Chen, Xiyue Shang, Chong Wang, Yiling Lou, and Xin Peng. 2025. Project-level c-to-rust translation via synergistic integration of knowledge graphs and large language models. *arXiv preprint arXiv:2510.10956*.
- Shangbo Yun, Xiaodong Gu, Jianghong Huang, and Beijun Shen. 2026. [Beyond language boundaries: Uncovering programming language families for code language models](#). *Proc. ACM Softw. Eng.*, 3(FSE).
- Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhui Chen. 2025. [ACECODER: Acing coder RL via automated test-case synthesis](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12023–12040, Vienna, Austria. Association for Computational Linguistics.
- Bowen Zhang, Wei Chen, Peisen Yao, Chengpeng Wang, Wensheng Tang, and Charles Zhang. 2024a. Siro: empowering version compatibility in intermediate representations via program synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 882–899.
- Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. 2025. [Scalable, validated code translation of entire projects using large language models](#). *Proc. ACM Program. Lang.*, 9(PLDI).
- Jipeng Zhang, Jianshu Zhang, Yuanzhe Li, Renjie Pi, Rui Pan, Runtao Liu, Ziqiang Zheng, and Tong Zhang. 2024b. Bridge-coder: Unlocking llms’ potential to overcome language gaps in low-resource code. *arXiv preprint arXiv:2410.18957*.
- Yifan Zhang and Kevin Leach. 2025. [Training large language models to comprehend llvm ir via feedback-driven optimization](#). In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion ’25*, page 1477–1478, New York, NY, USA. Association for Computing Machinery.
- Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, and 1 others. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.
- Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyang Luo. 2024. [LlamaFactory: Unified efficient fine-tuning of 100+ language models](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410, Bangkok, Thailand. Association for Computational Linguistics.
- Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Pengwei Jin, and 1 others. 2025. Qimeng-codev-r1: Reasoning-enhanced verilog generation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, and Max Kim. 2025. Migrating code at scale with llms at google. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 162–173.

A Discussion

Future work. Potential future work includes systematically investigating the application of *Aggressive-Partial-Functional* reward in RL training for coding-related tasks such as code generation. In addition, the use of the high-resource PL Python as an intermediate representation to address low-resource PL transpilation warrants further exploration, particularly by leveraging such IRs to enhance the code generation capabilities of current LLMs. Furthermore, while investigating alternative PLs as potential IRs remains a compelling direction, the dearth of comparable high-quality corpora for other PLs constrains current efforts, and we therefore leave this exploration to future work. Additionally, while $D_{PY-TRANS}$ provides high-quality, complex file-level transpilation tasks, it intentionally excludes repository-level complexities, such as multi-file structures and build systems, which are beyond the current scope and are left for future work.

B Additional Related Work

Intermediate representation. An intermediate representation is an abstract form of a program and has been a foundational concept since the early development of compiler design (Strong et al., 1958; Huskey et al., 1960; Merrill, 2003; Davidson and Fraser, 1980; Lattner and Adve, 2004). LLVM IR stands out for its language agnosticism, well-defined semantics, and rich toolchain support, which have made it a widely adopted foundation for modern research in static analysis for bug detection (Sui and Xue, 2016a; Shi et al., 2018; Sui and Xue, 2016b; Sui et al., 2012, 2014; Tang et al., 2024; Chen et al., 2025), program synthesis and migration (Zhang et al., 2024a; Li et al., 2025b), verification (Nelson et al., 2017; Lopes et al., 2021; Li et al., 2026), and compiler optimization (Lopes et al., 2015; Tan et al., 2025). More recently, LLVM IR has also been used to study and enhance the multilingual capabilities of code models. Prior

work (Zhang and Leach, 2025) trains models to learn compiler IR, while other studies (Szafraniec et al., 2023) leverage LLVM IR to improve transpilation performance. In addition, some studies (Paul et al., 2024) investigate the use of compiler intermediate representations to improve the multilingual capabilities of Code LMs and facilitate cross-lingual transfer.

C Implementation Details

During the IR-pivoted supervised fine-tuning (SFT) stage, we use LLaMAFactory (Zheng et al., 2024) to perform SFT of the open-source base model Qwen2.5-Coder-7B-Instruct on $\mathcal{D}_{\text{PY2OTHERS-SFT}}$. We train the model for 3 epochs with a learning rate of 1×10^{-5} and a batch size of 32. The distilled model is used for later RL.

During the RL stage, we implement CODEPIVOT with a non-linear *Aggressive-Partial-Functional* reward in the verl (Sheng et al., 2025) framework and further train the distilled model on $\mathcal{D}_{\text{PY2OTHERS-RL}}$ using GRPO (Shao et al., 2024). We set the total context length to 22,528, train with a batch size of 32, a clipping range ϵ of 0.2, and a learning rate of 1×10^{-6} , and run for about 350 steps until the reward curves converge. The rollout temperature is fixed at 1.0, and the maximum lengths are set to 5,120 tokens for the instruction and 16,384 tokens for the response.

All experiments are conducted on 8 H800-80G GPUs. The supervised fine-tuning (SFT) stage requires approximately 4 hours, while the RL stage requires about 50 hours. All generated programs are executed on an Ubuntu 20.04 LTS server equipped with two 16-core Intel(R) Xeon(R) Gold 6444Y processors (45M cache, 3.60 GHz base clock) and 256 GiB of DDR5-4800 ECC RDIMM memory.

D Dataset Construction and Sampling Strategy

Programming language resource-level definition. We adopted a definition based on GitHub 2.0 (2024 Q1) (Beuke, 2026), computed across all 69 PLs with complete pull-request, issue, push, and star records. For each indicator, a PL received a high-resource vote at or above the 95th percentile and a low-resource vote below the 85th percentile; intermediate values received neither. For each indicator, the top 5% account for about 50% of activity, the next 10% for about 35%, and the remaining 85%

for only about 15%. We aggregated the four votes using a three-out-of-four rule: a PL was classified as high-resource if it received at least three high-resource votes, low-resource if it received at least three low-resource votes, and medium-resource otherwise. For our experiments, we selected Python, C++, Java, and JS from the high-resource PLs; C#, Go, and Ruby from the medium-resource PLs; and Rust, Perl, and Haskell from the low-resource PLs.

Construction of $\mathcal{D}_{\text{PY-TRANS}}$. We build $\mathcal{D}_{\text{PY-TRANS}}$ on top of PyEdu-R(easoning) (Li et al., 2025a), a reasoning-focused subset of Python-Edu containing approximately 369K Python code files sourced from The Stack v2 (Lozhkov et al., 2024). We start by extracting high-quality Python code snippets and adding a main function that formats inputs and outputs as JSON. We then use LLMs to automatically synthesize test inputs for each program with prompts that emphasize edge cases, execute each program twice, and discard programs whose outputs disagree across runs, which is an effective heuristic for filtering non-deterministic programs. Each retained deterministic program p_{src} is paired with its test cases $(s_i, s_o) \in \tau$ and an automatically assigned problem category, and target PLs are designated from C++, C#, Java, JavaScript, Golang, Perl, Ruby, Rust, and Haskell. Iterating this pipeline yields $\mathcal{D}_{\text{PY-TRANS}}$, comprising roughly 1M Python-to-Others transpilation task instances with problem-diverse, high-quality source programs that incorporate real-world constructs such as external API invocation and exception handling.

Language-agnostic test cases. Inconsistencies in I/O formats across PLs have long hindered scaling of transpilation datasets (e.g., decimal handling, integer widths, string/encoding boundaries, container iteration order). We adopt language-agnostic JSON-formatted I/O test cases so that the same p_{src} and τ can validate transpiled programs in any target PL, enabling $\mathcal{D}_{\text{PY-TRANS}}$ to be fully exploited during RL training across all target PLs.

Sampling weight function. Given a program p , let $P(p)$ denote the specific problem solved by p , and let $C(\cdot)$ be a function that maps a problem to its corresponding problem class. We define the

sampling weight function as follows:

$$w(p) = \left| \left\{ (p_{\text{src}}, \tau, l_{\text{src}}, l_{\text{tar}}) \in \mathcal{D}_{\text{PY-TRANS}} \mid C(P(p_{\text{src}})) \neq C(P(p)) \right\} \right|.$$

Sampling strategy. To construct the task subset, we sample $\mathcal{D}_{\text{PY-TRANS}}$ without replacement based on the defined weight function. We restrict our selection to transpilation tasks where p_{src} has over 50 lines of code, distributed uniformly across the target PLs.

E Complete Main Results

In this section, we conduct an experiment studying the effectiveness of CODEPIVOT in improving transpilation capabilities across multiple PLs and on tasks involving low-resource PLs.

By exclusively training the Base-7B model, Qwen2.5-Coder-7B-Instruct, on Python-to-Others transpilation tasks, we obtain an SFT variant trained on $\mathcal{D}_{\text{PY2OTHERS-SFT}}$, Base-7B-SFT, and an RL variant, Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$, which is further optimized via *Aggressive-Partial-Functional* reward-augmented RL on $\mathcal{D}_{\text{PY2OTHERS-RL}}$. We subsequently evaluate the zero-shot generalization capabilities of these models on Others-to-All transpilation tasks, without any direct training, using the benchmark $\mathcal{D}_{\text{OTHERS2ALLBENCH}}$.

Table 5 presents the performance of six models: Base-7B, Base-7B-SFT, Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$, and three baseline models, namely Qwen3-235B-A22B-Instruct-2507, DeepSeek-V3.1, and Qwen3-Next-80B-A3B-Instruct. Compared with Base-7B, Base-7B-SFT consistently improves performance across all Others-to-All tasks and yields an average gain of 44.49 points. Relative to Base-7B-SFT, Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ improves performance on 98.77% of Others-to-All tasks, with only one task remaining unchanged. Overall, the RL stage yields an average improvement of 8.55 points. These results suggest that Python serves as an effective IR for enhancing code understanding across multiple PLs.

Despite possessing hundreds of billions fewer parameters, Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ outperforms the baseline models Qwen3-235B-A22B-Instruct-2507, DeepSeek-V3.1, and Qwen3-Next-80B-A3B-Instruct on 91.36%, 90.12%, and 93.83% of the evaluated tasks, respectively. Furthermore, it achieves average performance improvements of

15.19, 20.83, and 21.22 points relative to each respective baseline.

This substantial gain underscores the effectiveness of CODEPIVOT in systematically enhancing a model’s general transpilation capabilities. However, the baselines retain an advantage in specific tasks involving translation into dynamically typed languages, such as Python, Perl, and JavaScript. We hypothesize that this may be partly attributed to their structural flexibility and natural-language-inspired syntax (Pandey et al., 2024; Yun et al., 2026), as code LLMs often exhibit strong performance on PLs with such characteristics (Cassano et al., 2023; Wang et al., 2026a). Furthermore, Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ ’s underperformance in certain Rust transpilation tasks can be ascribed to the PL’s paradigm-blending design (Yun et al., 2026) and complex feature set (Matsakis and Klock, 2014), which introduce intricate semantic challenges that hinder the model’s learning process.

Performance on tasks involving low-resource PLs. Figure 3 presents a performance comparison across tasks involving low-resource PLs. These tasks are categorized into High-to-Low (H-L), Low-to-High (L-H), and Low-to-Low (L-L), pairing high-resource PLs, such as Python, C++, Java, and JavaScript, with low-resource PLs, such as Rust, Perl, and Haskell (Cassano et al., 2023; TIOBE, 2026; Cassano et al., 2024). The evaluation results indicate that both Base-7B-SFT and Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ consistently enhance performance on all tasks involving low-resource PLs. In particular, Base-7B-SFT outperforms Base-7B by an average margin of 42.00 points, while Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ yields a further average increase of 7.99 points relative to Base-7B-SFT. Notably, Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ outperforms the baseline models, Qwen3-235B-A22B-Instruct-2507, DeepSeek-V3.1, and Qwen3-Next-80B-A3B-Instruct, by considerable margins of 10.67, 14.09, and 18.16 points, respectively. Collectively, these findings indicate the effectiveness of CODEPIVOT in advancing models’ low-resource transpilation capabilities while mitigating the challenge of data scarcity for low-resource PLs.

F Effectiveness of CODEPIVOT’s training pipeline with *Aggressive-Partial-Functional* reward

In this section, we directly evaluate the effectiveness of the training pipeline of CODEPIVOT,

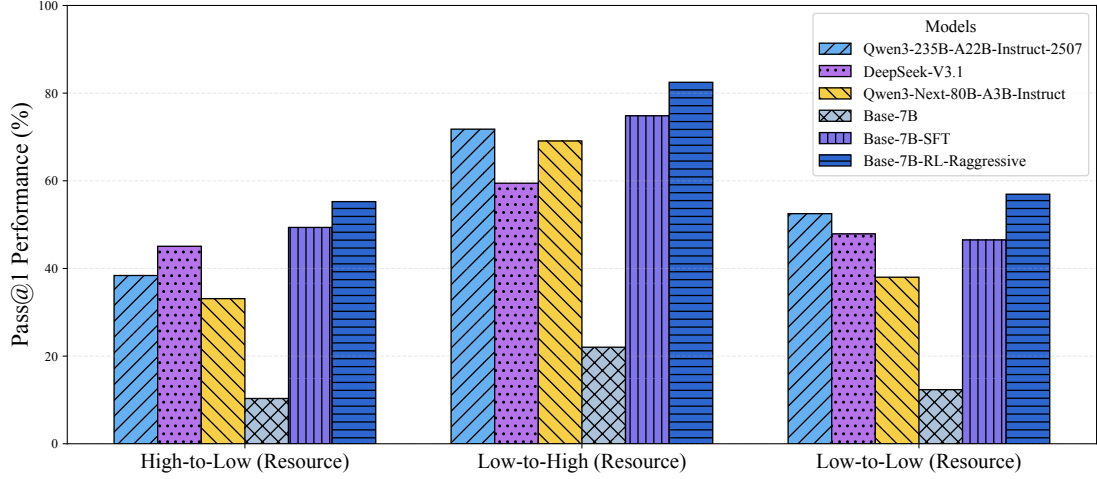


Figure 3: Performance of models on transpilation tasks involving low-resource PLs.

Table 6: Performance comparison in terms of $Pass@1$ (%) of different models on $\mathcal{D}_{PY2OTHERSBENCH}$. Columns correspond to the nine Python-to-Others tasks, from **Python-to-C++** through **Python-to-Haskell**. The **Total** column presents the overall $Pass@1$ score across $\mathcal{D}_{PY2OTHERSBENCH}$. Base-7B-RL- $\mathcal{R}_{aggressive}$ is the model trained using our training pipeline with the *Aggressive-Partial-Functional* reward from the base model, Base-7B. Bold values indicate the highest $Pass@1$ rate in each column. * denotes closed-source models.

Model	C++	Java	C#	JavaScript	Golang	Rust	Perl	Ruby	Haskell	Total
GPT-4o*	17.33	37.00	27.33	41.33	33.00	29.33	45.00	38.00	0.33	29.85
GPT-4.1*	24.33	65.00	60.00	73.33	51.67	58.67	68.33	39.00	24.67	51.67
Gemini-2.5-Flash*	54.00	50.33	53.67	75.00	44.33	49.67	56.00	54.67	17.00	50.52
Claude-Sonnet-4*	20.33	64.67	60.67	68.00	49.67	65.33	67.33	24.33	35.00	50.59
GPT-4.1-mini*	15.33	50.33	44.67	65.67	46.67	59.33	66.33	22.67	30.33	44.59
DeepSeek-R1	35.67	70.00	63.33	77.00	67.67	63.00	67.33	67.67	22.33	59.33
Qwen3-235B-A22B-Instruct-2507	32.33	40.00	21.67	30.33	25.33	28.33	26.67	20.00	6.67	25.70
Qwen3-Coder-480B-A35B-Instruct	22.33	41.33	32.67	43.00	28.67	33.67	41.33	19.00	11.33	30.37
DeepSeek-V3.1	13.67	43.33	42.67	46.00	34.33	46.00	48.33	47.67	12.33	37.15
DeepSeek-V3.2	17.33	47.00	43.67	49.67	34.33	45.00	55.00	43.33	8.00	38.15
Qwen3-Next-80B-A3B-Instruct	25.33	28.00	23.00	47.67	13.00	28.67	33.00	39.33	0.33	26.48
Qwen2.5-72B-Instruct	6.67	7.33	14.67	12.00	3.33	9.67	18.33	26.67	0.00	10.96
Qwen3-30B-A3B-Instruct-2507	10.00	24.33	13.33	25.00	7.00	20.33	23.33	29.00	0.00	16.93
Qwen3-14B	2.33	5.67	11.67	10.67	3.67	4.67	16.00	28.00	0.33	9.22
Qwen3-8B	2.00	6.67	7.33	8.33	3.00	5.00	14.00	8.00	0.00	6.04
Base-7B (Qwen2.5-Coder-7B-Instruct)	1.33	3.67	0.00	5.67	3.33	3.00	4.33	10.67	0.00	3.56
Base-7B-RL- $\mathcal{R}_{aggressive}$ (Ours)	68.67	70.00	69.00	74.33	59.00	49.67	72.00	84.00	35.33	64.67

including the Python-IR-pivoted SFT stage and the *Aggressive-Partial-Functional* reward augmented RL stage, in enhancing the models' performance on Python-to-Others transpilation tasks in $\mathcal{D}_{PY2OTHERSBENCH}$. We do not evaluate Python as an IR.

Table 6 presents the performance of Base-7B-RL- $\mathcal{R}_{aggressive}$ and mainstream LLMs on Python-to-Others transpilation tasks. Across all evaluated models, Base-7B-RL- $\mathcal{R}_{aggressive}$ achieves the highest overall $Pass@1$ on $\mathcal{D}_{PY2OTHERSBENCH}$. It outperforms other mainstream LLMs, such as DeepSeek-R1 (671B) and Claude-Sonnet-4, by 5.34 and 14.08 points

in overall $Pass@1$, respectively. This indicates that Base-7B-RL- $\mathcal{R}_{aggressive}$ has the best multi-lingual transpilation capability. In particular, Base-7B-RL- $\mathcal{R}_{aggressive}$ delivers the best results in 6 out of 9 Python-to-Others tasks and yields an average margin of 6.78 points over the second-best model for each PL. We attribute these gains to the effectiveness of the training pipeline of CODEPIVOT, which enables a smaller model to better understand programs and to develop stronger reasoning-driven transpilation capabilities across multiple PLs.

For the Python-to-JavaScript task, Base-7B-RL- $\mathcal{R}_{aggressive}$ exhibits a 2.67-point

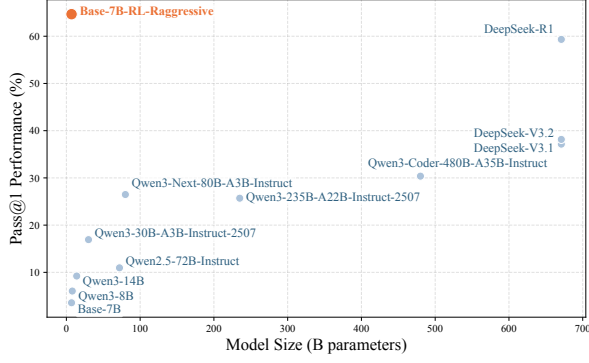


Figure 4: Scatter plot illustrating the relationship between model size (B) and $Pass@1$ performance (%) on $\mathcal{D}_{PY2OTHERSBENCH}$.

decrease in $Pass@1$ relative to the best-performing model. We attribute this to the high-resource nature of JavaScript, which constitutes a large share of the training corpora of current LLMs. Base-7B-RL- $\mathcal{R}_{aggressive}$ does not achieve the best results for the Python-to-Golang and Python-to-Rust tasks. This is attributed to the fact that Golang and Rust are modern multi-paradigm languages (Yun et al., 2026) with paradigm-blending designs that combine functional, procedural, and object-oriented approaches. Such paradigm blending increases syntactic and grammatical complexity, particularly in Rust (Matsakis and Klock, 2014), often requiring more elaborate constructs to express equivalent semantics and introducing substantial divergence from other languages. This hinders the easy conversion of partially correct programs into functionally correct ones. Despite this, compared to the initial $Pass@1$ performance of Base-7B, the performance of Base-7B-RL- $\mathcal{R}_{aggressive}$ improves by 1671.77% on the Python-to-Golang task and by 1555.67% on the Python-to-Rust task, further indicating the effectiveness of the CODEPIVOT training pipeline with *Aggressive-Partial-Functional* reward.

Figure 4 presents the relationship between model size and $Pass@1$ across open-source models. It indicates that the model trained with the CODEPIVOT pipeline attains the highest $Pass@1$ with the fewest parameters, underscoring the pipeline’s effectiveness.

G Evaluation Beyond the Distillation Teacher

The original $\mathcal{D}_{OTHERS2ALLBENCH}$ benchmark was distilled from Claude-Sonnet-4. Since our model is also trained on data distilled from Claude-Sonnet-4,

we constructed a new version of $\mathcal{D}_{OTHERS2ALLBENCH}$ using Gemini 3.5 Flash and the same execution filtering described in § 3.2 to evaluate our trained model beyond its distillation teacher.

Table 7: Performance comparison in $Pass@1$ (%) on $\mathcal{D}_{OTHERS2ALLBENCH}$ constructed using Gemini 3.5 Flash. Base-7B denotes the base model, while Base-7B-SFT and Base-7B-RL- $\mathcal{R}_{aggressive}$ denote the models obtained through further training.

Model	$Pass@1$ (%)
Qwen3-235B-A22B-Instruct-2507	52.75
Qwen3-Next-80B-A3B-Instruct	41.09
Base	18.91
Base-SFT	50.86
Base-RL (Ours)	60.36

Table 7 presents the results, indicating that CODEPIVOT’s gains persist with a different benchmark teacher.

H Training Dynamics

When applying the standard GRPO algorithm (Shao et al., 2024) for RL training, the policy LLM $\pi_\theta(\cdot | q)$ produces multiple rollouts o for each input prompt q . Given the group size G , the rollouts are denoted by $\{o_i\}_{i=1}^G \sim \pi_{\theta_{old}}(\cdot | q)$. Therefore, the policy LLM aims to optimize the following GRPO objective:

$$\mathcal{J}(\theta) = \mathbb{E} \left[\frac{1}{G} \sum_{i=1}^G \frac{1}{|o_i|} \sum_{t=1}^{|o_i|} \ell_{i,t} \right],$$

$$\ell_{i,t} = \min \left[\begin{aligned} &\rho_{i,t} \hat{A}_{i,t}, \\ &\text{clip}(\rho_{i,t}, 1 - \epsilon, 1 + \epsilon) \hat{A}_{i,t} \end{aligned} \right] \quad (3)$$

$$- \beta D_{KL}(\pi_\theta(\cdot | q) \parallel \pi_{\text{ref}}(\cdot | q)),$$

$$\rho_{i,t} = \frac{\pi_\theta(o_{i,t} | q, o_{i,<t})}{\pi_{\theta_{old}}(o_{i,t} | q, o_{i,<t})},$$

where ϵ and β are hyperparameters, and $\pi_\theta(\cdot | q)$ and $\pi_{\text{ref}}(\cdot | q)$ are the probability distributions of the policy and reference models, respectively. The reward $r_i = \mathcal{R}(o_i)$ and the advantages $\hat{A}_{i,t} = \frac{r_i - \text{mean}(r_1, \dots, r_G)}{\text{std}(r_1, \dots, r_G)}$ are calculated using the normalized rewards within each group following GRPO. D_{KL} denotes the estimated KL-divergence (Schulman, 2020).

Figure 5 presents the trajectories of the mean reward during reinforcement learning (RL) training under four distinct reward formulations. Notably, all four curves show upward trends, indicating that the policy progressively improves code quality and test-case pass rates across all formulations.

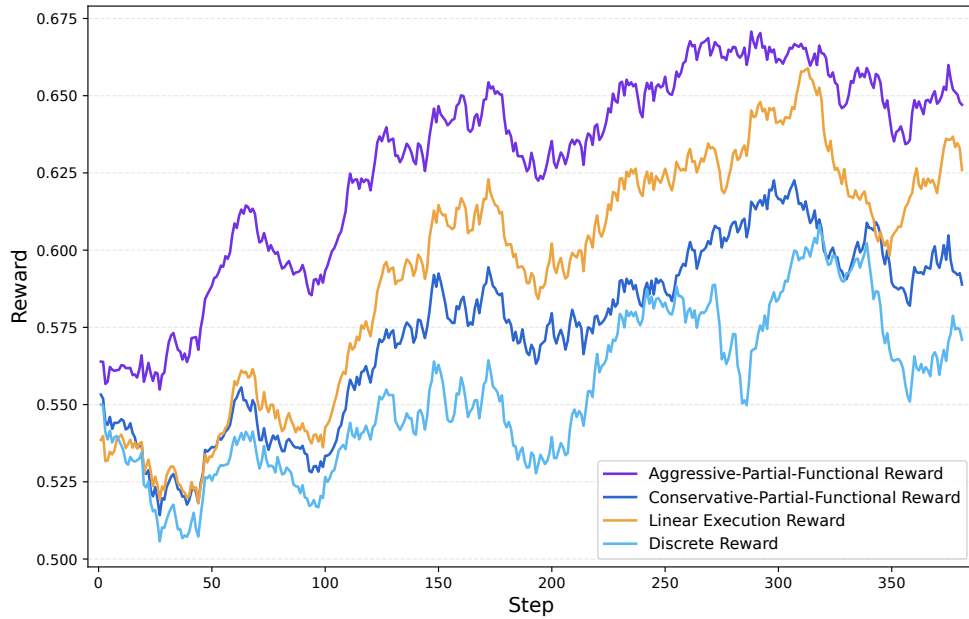


Figure 5: Comparison of mean reward trajectories during RL training under four distinct reward formulations. All curves are smoothed using an exponential moving average for clarity.

The discrete reward exhibits highly unstable behavior and is prone to plateaus. Among the four formulations, APF achieves the highest mean reward throughout most of the training phase, followed by Linear Execution (LE), Conservative-Partial Functional (CPF), and Discrete rewards. This suggests that reward formulations providing denser and more continuous signals support smoother optimization and more stable training progress in program translation. By contrast, the linear execution reward exhibits a highly unstable trajectory with a pronounced drop during the early stages, resulting in lower overall reward growth. A similar pattern is observed for CPF, whose trajectory consistently remains below that of APF throughout the training process.

Both reward designs yield substantially lower value trajectories and weaker late-stage improvement.

Furthermore, the Discrete reward consistently exhibits the lowest values with a relatively flat learning curve. This stark contrast highlights the inherent challenges of sparse reward optimization. Because the Discrete strategy assigns rewards only to functionally correct outputs ($x = 1$), the resulting reward scores lack meaningful gradation during the early exploration stage. Consequently, the model finds it much more difficult to escape initially suboptimal behaviors than under formulations with richer and more informative reward signals.

Table 5: Performance comparison in *Pass@1* (%) of models across the tasks in $\mathcal{D}_{\text{OTHERS2ALLBENCH}}$. Columns specify source PLs from **C++** to **Haskell**, and row blocks specify target PLs. Base-7B denotes the base model, and Base-7B-SFT and Base-7B-RL- $\mathcal{R}_{\text{aggressive}}$ denote the resulting models further trained. Values highlighted in bold indicate the highest *Pass@1* rate for each task.

Languages	Model	From C++	From Java	From C#	From JS	From Golang	From Rust	From Perl	From Ruby	From Haskell
To Python	Qwen3-235B-A22B-Instruct-2507	90.33	92.67	96.00	94.67	91.00	92.33	95.67	90.00	92.67
	DeepSeek-V3.1	93.00	97.67	95.33	95.00	92.00	96.00	94.67	98.33	92.33
	Qwen3-Next-80B-A3B-Instruct	93.33	94.00	95.00	93.00	90.33	97.00	78.67	96.00	88.00
	Base-7B (Qwen2.5-Coder-7B-Instruct)	65.00	54.67	71.67	66.00	62.67	57.33	68.33	71.00	46.00
	Base-7B-SFT	89.33	86.67	84.00	82.33	83.00	72.33	88.33	88.00	77.00
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$	93.67	95.00	96.33	95.33	92.33	83.67	96.33	90.67	83.33
To C++	Qwen3-235B-A22B-Instruct-2507	N/A	57.00	63.33	51.67	55.67	52.33	45.00	62.67	56.33
	DeepSeek-V3.1	N/A	46.67	16.00	17.67	56.67	30.00	10.00	8.33	36.33
	Qwen3-Next-80B-A3B-Instruct	N/A	43.33	50.00	54.00	53.33	60.00	49.33	43.67	52.33
	Base-7B (Qwen2.5-Coder-7B-Instruct)	N/A	0.00	0.00	0.00	0.00	0.00	0.67	0.00	0.00
	Base-7B-SFT	N/A	62.00	68.67	76.33	78.33	73.00	74.00	67.33	64.00
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$	N/A	76.67	87.67	80.33	82.33	87.00	76.67	69.00	75.00
To Java	Qwen3-235B-A22B-Instruct-2507	60.67	N/A	45.33	40.67	54.67	50.00	58.33	41.00	56.33
	DeepSeek-V3.1	46.00	N/A	45.67	37.00	31.67	38.67	39.67	42.00	43.00
	Qwen3-Next-80B-A3B-Instruct	51.67	N/A	42.00	41.67	36.00	59.00	45.00	42.00	42.00
	Base-7B	30.33	N/A	15.00	9.67	24.00	20.00	10.00	8.00	19.00
	Base-7B-SFT	66.67	N/A	39.00	77.00	81.00	76.33	70.67	71.33	63.33
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$	82.33	N/A	66.67	77.00	87.00	84.33	75.67	74.67	73.33
To C#	Qwen3-235B-A22B-Instruct-2507	33.67	28.00	N/A	54.33	45.67	63.00	63.67	50.33	54.33
	DeepSeek-V3.1	49.67	57.67	N/A	54.33	35.33	52.00	54.33	56.33	56.33
	Qwen3-Next-80B-A3B-Instruct	49.00	22.00	N/A	46.33	60.67	54.33	45.33	27.00	50.33
	Base-7B (Qwen2.5-Coder-7B-Instruct)	1.33	0.67	N/A	1.67	0.00	0.33	1.67	1.33	0.33
	Base-7B-SFT	55.00	62.67	N/A	71.33	51.33	66.67	65.33	61.67	52.67
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$	66.00	72.00	N/A	78.33	75.67	69.67	72.33	66.33	67.33
To JS	Qwen3-235B-A22B-Instruct-2507	91.67	81.33	90.00	N/A	81.67	90.33	87.33	76.67	84.67
	DeepSeek-V3.1	87.33	73.00	71.00	N/A	84.00	78.67	70.67	54.00	83.33
	Qwen3-Next-80B-A3B-Instruct	89.00	82.33	91.00	N/A	81.00	88.67	85.67	79.67	83.33
	Base-7B (Qwen2.5-Coder-7B-Instruct)	29.67	15.67	18.00	N/A	31.33	21.33	10.33	7.33	11.33
	Base-7B-SFT	84.00	77.33	83.67	N/A	77.33	83.67	80.67	78.00	74.67
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$	93.00	82.33	91.33	N/A	85.00	92.00	83.67	81.33	78.67
To Golang	Qwen3-235B-A22B-Instruct-2507	37.67	38.67	30.67	27.33	N/A	40.67	29.33	25.00	37.33
	DeepSeek-V3.1	28.33	35.33	36.33	38.00	N/A	27.00	24.00	33.33	28.67
	Qwen3-Next-80B-A3B-Instruct	27.00	27.67	30.00	24.00	N/A	27.67	19.00	25.33	26.33
	Base-7B (Qwen2.5-Coder-7B-Instruct)	6.00	13.33	4.33	7.33	N/A	21.67	6.00	8.33	9.00
	Base-7B-SFT	58.00	56.67	65.00	67.33	N/A	62.00	61.67	62.33	59.00
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$	64.00	61.67	75.00	68.00	N/A	71.33	74.00	65.67	60.33
To Rust	Qwen3-235B-A22B-Instruct-2507	57.33	57.00	59.33	53.33	60.67	N/A	64.67	60.67	70.33
	DeepSeek-V3.1	66.67	60.00	43.33	37.67	61.00	N/A	64.33	57.00	42.00
	Qwen3-Next-80B-A3B-Instruct	54.67	38.33	46.33	37.67	49.00	N/A	39.00	44.67	48.00
	Base-7B (Qwen2.5-Coder-7B-Instruct)	9.00	9.33	15.33	9.33	14.00	N/A	8.67	12.00	14.67
	Base-7B-SFT	59.67	54.67	56.00	52.67	45.67	N/A	50.33	55.33	53.33
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$	61.00	60.33	65.67	58.33	61.33	N/A	65.00	57.33	62.00
To Perl	Qwen3-235B-A22B-Instruct-2507	66.00	65.67	64.67	64.00	60.00	80.33	N/A	64.67	68.33
	DeepSeek-V3.1	82.00	69.33	81.67	80.67	75.67	78.67	N/A	71.33	70.67
	Qwen3-Next-80B-A3B-Instruct	72.33	63.67	73.67	67.00	69.33	68.00	N/A	58.67	69.33
	Base-7B (Qwen2.5-Coder-7B-Instruct)	29.33	27.67	40.00	32.00	34.00	39.00	N/A	19.67	11.67
	Base-7B-SFT	73.67	64.33	72.00	75.33	65.33	64.00	N/A	66.00	54.00
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$	74.00	70.33	83.33	81.00	76.33	69.00	N/A	72.00	71.00
To Ruby	Qwen3-235B-A22B-Instruct-2507	66.67	65.67	69.33	58.00	55.33	66.67	59.00	N/A	58.67
	DeepSeek-V3.1	44.00	57.33	62.33	25.67	46.00	45.00	43.33	N/A	62.00
	Qwen3-Next-80B-A3B-Instruct	57.00	55.00	58.67	49.00	53.00	51.67	43.00	N/A	58.00
	Base-7B (Qwen2.5-Coder-7B-Instruct)	51.00	47.33	40.67	43.67	42.00	43.00	41.67	N/A	25.67
	Base-7B-SFT	78.00	72.67	60.00	74.67	68.00	63.00	71.00	N/A	54.67
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$	84.33	83.33	78.00	86.00	78.00	78.00	80.00	N/A	67.67
To Haskell	Qwen3-235B-A22B-Instruct-2507	10.33	15.67	17.00	9.67	12.67	20.33	11.00	11.33	N/A
	DeepSeek-V3.1	15.67	12.00	12.33	10.00	8.67	17.33	14.33	10.33	N/A
	Qwen3-Next-80B-A3B-Instruct	0.67	1.00	1.00	0.00	1.00	2.67	1.00	0.00	N/A
	Base-7B (Qwen2.5-Coder-7B-Instruct)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	N/A
	Base-7B-SFT	30.33	25.00	21.00	29.00	22.33	36.33	21.00	24.67	N/A
	Base-7B-RL-$\mathcal{R}_{\text{aggressive}}$	31.33	28.67	36.00	41.00	37.67	43.33	31.33	34.33	N/A

I Prompt Template

[System] You are a specialist in transpiling code between programming languages, tasked with translating the provided reference code from the source language into a functionally equivalent implementation in the target language. Ensure that variable names are preserved wherever the variables in both implementations share the same semantic meaning. If the transpiled program contains an entry class, it should be named `Main` whenever permissible. The specifications for the code execution runtime environments and versions are outlined below. Please ensure strict adherence to these requirements:

- The Python execution environment is Python 3.8 with the standard library and the following third-party packages installed: NumPy 1.24.4 and pandas 2.0.3. For JSON parsing, use the built-in `json` module.
- The C# execution environment is .NET 9 SDK (9.0.203). For JSON parsing, use the built-in standard library.
- The C++ environment is Ubuntu 20.04 with GCC 9.4.0 (g++), libstdc++ with C++17 support, and glibc 2.31 for runtime support. For JSON parsing, use the `JsonCpp` library and include the header `<json/json.h>`.
- The Java execution environment is OpenJDK 17.0.15. For JSON parsing, use the Jackson (FasterXML) library.
- The JavaScript execution environment is JavaScript Node.js v22.18.0. For JSON parsing, use the built-in JSON global object.
- The Golang execution environment is Go version go1.24.4 linux/amd64. For JSON parsing, use the standard library.
- The Ruby execution environment is Ruby 3.2.2 x86_64-linux. For JSON parsing, use the standard library `json` module.
- The Rust execution environment is based on Ubuntu 20.04, using `rustc` 1.75.0 for compiling Rust code and `cargo` 1.75.0 for package management and build automation. For JSON parsing, use the `serde_json` package.
- The Haskell execution environment is the Glorious Glasgow Haskell Compilation System (GHC) version 8.6.5. For JSON parsing, use the `aeson` package.
- The Perl execution environment is Perl version 5.30.0. For JSON parsing, use the `JSON` module from the `libjson-perl` package.

As a helpful AI Assistant, you provide well-reasoned and detailed responses by first thinking through the reasoning process as an internal monologue and then presenting the user with the answer. Provide your response in the following structured format: `<think> ... </think> <answer> ```{language} {code} ``` </answer>`. In the section enclosed by `<answer>` and `</answer>` tags, ensure that only the transpiled code is included in accordance with the given format, such as ````python code ````.

[User] Transpile the provided {source language} implementation into a functionally equivalent implementation in {target language}:

```
```{source language} {source program} ```
```

Figure 6: Prompt template used for the transpilation task in both training and evaluation. The system prompt defines the task and the LLM’s role, specifies the detailed target code-execution runtime environments, and prescribes the required output format. The user prompt provides explicit instructions to transpile a given source program.