Bootstrapping Code Translation with Weighted Multilanguage Exploration

Yuhan Wu[†] **Huan Zhang**[†] **Wei Cheng**[†] **Chen Shen**[†] **Jingyue Yang**[†] **Wei Hu**[†,‡,*]
[†] State Key Laboratory for Novel Software Technology, Nanjing University, China
[‡] National Institute of Healthcare Data Science, Nanjing University, China
{yhwu,zhanghuan,wchengcs,cshen,jyyang}.nju@gmail.com, whu@nju.edu.cn

Abstract

Code translation across multiple programming languages is essential yet challenging due to two vital obstacles: scarcity of parallel data paired with executable test oracles, and optimization imbalance when handling diverse language pairs. We propose BootTrans, a bootstrapping method that resolves both obstacles. Its key idea is to leverage the functional invariance and cross-lingual portability of test suites, adapting abundant pivot-language unit tests to serve as universal verification oracles for multilingual reinforcement learning (RL) training. Our method introduces a dual-pool architecture with seed and exploration pools to progressively expand training data via execution-guided experience collection. Furthermore, we design a language-aware weighting mechanism that dynamically prioritizes harder translation directions based on relative performance across sibling languages, mitigating optimization imbalance. Extensive experiments on the HumanEval-X and TransCoder-Test benchmarks demonstrate substantial improvements over baseline LLMs across all translation directions, with ablation studies validating the effectiveness of both bootstrapping and weighting components.

1 Introduction

Large Language Models (LLMs) have shown remarkable progress in coding tasks, revolutionizing contemporary software engineering workflows. Code translation, migrating code from a source programming language to a target while ensuring the syntax and semantics correctness, is pivotal for legacy system modernization and cross-platform interoperability (Nguyen et al., 2014; Roziere et al., 2020). Despite the advancements, code translation usually relies on abundant parallel code of high quality, which may not always be available. Even when available, they are rarely equipped with aligned, executable test cases (Roziere et al., 2022; Jiao et al., 2023; Zhu et al., 2024).

To resolve such reliance, existing works (Huang et al., 2023; Liu et al., 2023; Szafraniec et al., 2023) explore code structure information to learn representations for unsupervised translation. However, they typically demand enormous amounts of monolingual corpora to establish robust cross-lingual alignment. Moreover, these methods generally do not leverage executable test cases during training, and thus cannot directly optimize translation quality based on functional correctness. As a result, the learning objective is often restricted to syntactic conversion rather than functional equivalence.

Recently, Reinforcement Learning from Verifiable Rewards (RLVR) offers a promising paradigm shift by optimizing models based on execution feedback (Le et al., 2022; Shojaee et al., 2023; Jana et al., 2024). While high-quality parallel code is scarce, unit tests are inherently transferable (Roziere et al., 2022). As unit tests often follow template-based patterns, they allow for highly reliable translation through rule-based methods (Cassano et al., 2023), and thus provide a viable way to guarantee consistent functional verification across different programming languages. This observation suggests that by translating test oracles from a resource-rich language (e.g., Python) to target languages, we can construct a rigorous RL environment for multilingual translation without ground-truth references.

However, realizing this potential in multilingual code translation presents two fundamental challenges, as illustrated in Figure 1. First, there is a severe shortage of multilingual datasets that provide unit tests across diverse programming languages to serve as starting points for RLVR. While test oracles can be migrated from a resource-rich language, high-quality code paired with these oracles remains predominantly available in a single pivot language. It is rare to find verified, functionally

* Corresponding author

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 36247–36259

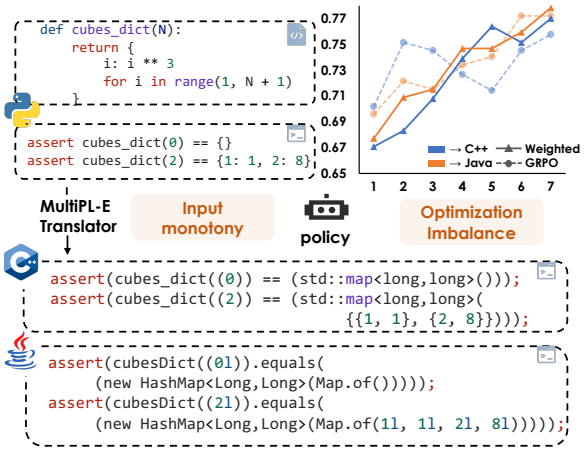


Figure 1: Illustration of challenges in scaling RLVR for multilingual code translation. (i) Input monotony: verifiable seeds are predominantly confined to a single pivot language. (ii) Optimization imbalance: varying task difficulties lead to biased learning signals.

equivalent source code in other target languages, so only unidirectional translation is enabled. Relying solely on a static source language dataset leaves a critical void in training data for reverse direction translations. While existing benchmarks like HumanEval-X (Zheng et al., 2023) provide multilingual source code, their scale remains limited for training robust models. Although synthetic data offer a viable workaround, they are intrinsically constrained by reduced diversity and potential bias. In addition, scaling up high-quality code across various programming languages remains computationally expensive and technically demanding. Bootstrapping verified training source code in all target languages is a key for adapting RL to code translation (Yan et al., 2023; Wang et al., 2024).

Second, optimizing for multiple programming languages simultaneously introduces optimization imbalance. Different translation directions (e.g., Python→Java vs. Python→C++) exhibit varying levels of difficulty due to syntactic and semantic discrepancies (Zhu et al., 2022b; Yan et al., 2023; Du et al., 2024). When optimizing uniformly across these tasks, the model tends to be dominated by easier translation directions where rewards are more readily accessible. Thus, the model rapidly improves on easier language pairs but often exhibits performance oscillation or stagnation on harder ones, causing suboptimal multilingual proficiency.

In this work, we address these challenges by proposing BootTrans. To overcome data sparsity, we leverage one language (e.g., Python) as a strategic pivot with abundant source code accompanied by unit tests and propagate test oracles to other languages. Then, we expand the RL curriculum through experience collection by utilizing verified rollouts from the policy model. To mitigate optimization imbalance, we introduce a language-aware weighting optimization mechanism that dynamically adjusts the learning focus based on the relative difficulty and performance of each target language. We conduct extensive experiments on pairwise code translation among C++, Java, and Python. Results demonstrate that BootTrans outperforms existing open-source LLMs.

Our main contributions are outlined as follows:

- We leverage a resource-rich pivot language to bootstrap a verifiable multilingual corpus, effectively overcoming the reliance on parallel code in code translation.
- We design a language-aware weighting optimization mechanism to mitigate optimization imbalance across translation directions by dynamically adjusting learning focus for different target languages.
- We conduct extensive experiments on pairwise code translation among C++, Java, and Python, showing that BootTrans consistently outperforms its corresponding base model by up to 26.82% on the HumanEval-X dataset and 7.46% on the TransCoder-Test dataset. Code is accessible at https://github.com/nju-websoft/BootTrans/.

2 Problem Formulation

We study the problem of multilingual code translation across a set of programming languages $\mathcal{L} = \{L_1, \ldots, L_M\}$. The goal is to learn a unified policy $\pi_\theta(y \mid x, L_{\text{src}}, L_{\text{tgt}})$ translating a source code x in language $L_{\text{src}} \in \mathcal{L}$ to a functionally equivalent target code y in language $L_{\text{tgt}} \in \mathcal{L}$ ($L_{\text{src}} \neq L_{\text{tgt}}$).

Unlike traditional supervised settings that rely on parallel corpora for language pairs, we operate under a *monolingual-pivot bootstrapping* scenario. We assume access to a seed dataset $\mathcal{D}_{\text{seed}}$ consisting of code-test pairs $\{(x, T)\}$ solely in a pivot language $L_{\text{pivot}} \in \mathcal{L}$ (e.g., Python). The set of unit tests T serves as a transferable verification oracle. Our objective is to leverage $\mathcal{D}_{\text{seed}}$ to progressively explore the multilingual space and optimize π_θ to maximize the expected success rate of translation across all directions in $\mathcal{L} \times \mathcal{L}$.

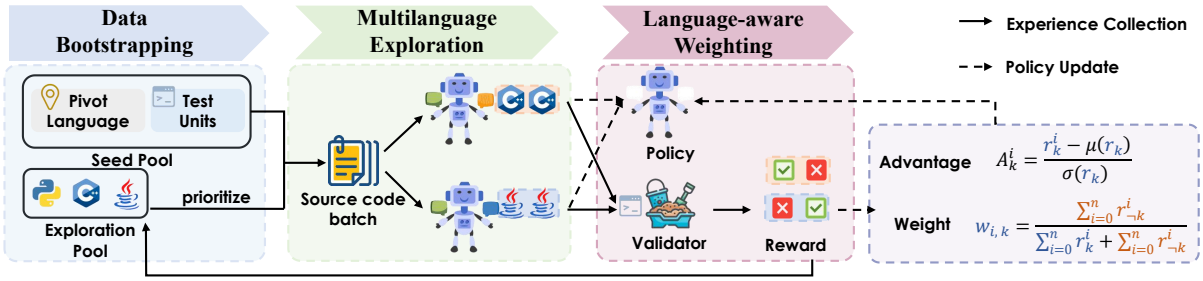


Figure 2: Overview of BootTrans. It comprises two key components: (i) **Bootstrapping Multilanguage Exploration** (left and center panels), which expands the training set via execution-verified translations; (ii) **Language-aware Weight Optimization** (right panels), which dynamically re-weights the loss via cross-lingual performance gaps.

3 Our Method

Figure 2 shows the proposed method consisting of bootstrapping multilanguage exploration and language-aware weight optimization.

3.1 Bootstrapping Multilanguage Exploration

To break the dependency on curated parallel corpora, we formulate the RL training for code translation as an evolving exploration process. Instead of confining the model to static pivot code-test pairs, the bootstrapping mechanism empowers the model to curate its own rollouts through execution-guided experience collection. These verified experiences serve as augmented source inputs, enabling the model to unlock training for translation directions originating from non-pivot languages.

3.1.1 Dual-Pool Architecture

We maintain two distinct data pools to manage the training curriculum:

Seed pool ($\mathcal{D}_{\text{seed}}$). This pool contains the original dataset $\mathcal{D}_{\text{seed}} = \{(x, T) \mid x \in L_{\text{pivot}}\}$, where x is the source code in the pivot language and T is the corresponding accessible test suite for all languages. In fact, T is readily obtainable as they only need to remain consistent with the execution behavior of x. Diverse and edge-case-rich inputs can be synthesized using automated techniques such as fuzzing or LLM-based synthesis. The corresponding canonical outputs are then obtained by executing x on these inputs, serving as the function output. Moreover, such a test suite is highly portable to other programming languages by rule-based conversion (Cassano et al., 2023), which naturally establishes a unified execution environment for RL. For example, we leverage Python's rich resource as a seed, providing a cold start for exploring the solution space of target languages.

Exploration pool ($\mathcal{D}_{\text{explore}}$). This pool dynamically stores the policy model's successful translations during rollout. Formally, we define $\mathcal{D}_{\text{explore}} = \{(y, T) \mid y \in \mathcal{L} \setminus \{L_{\text{pivot}}\}\}$, where y is a generated code that has passed all test cases in T. To ensure efficient exploration and prevent pool saturation, we manage $\mathcal{D}_{\text{explore}}$ as a First-In-First-Out (FIFO) queue with a capacity of $|\mathcal{L} \setminus \{L_{pivot}\}|$ times the rollout batch size. To prevent semantic drift, only verified rollouts stemming from D_{seed} are enqueued, ensuring execution consistency between the test suites and the evolving data pool.

In each training step, we prioritize drawing the full batch from $\mathcal{D}_{\text{explore}}$. If the remaining items are insufficient, the batch is supplemented with examples from the seed pool $\mathcal{D}_{\text{seed}}$. This strategy compels the policy model to exhaustively exploit the current exploration frontier for a given pivot language code before introducing new code-test pairs in $\mathcal{D}_{\text{seed}}$.

3.1.2 Verification Oracle and Reward

We optimize the policy model using execution feedback from unit tests. Given a generated candidate y and its associated test suite T in the target language, we define a binary verifiable reward:

$$R(y, T) = \mathbb{1}\Big[y \text{ compiles and passes all tests in } T\Big], \quad (1)$$

where $\mathbb{1}$ denotes the indicator function. In practice, compilation errors, runtime errors, and timeouts yield $R = 0$. This setting aligns the optimization objective with functional correctness rather than surface-form similarity.

3.1.3 Multilingual Expansion Training

Algorithm 1 shows the training process. We structure the exploration as a progressive expansion rooted in the pivot language, which follows a standard RL pipeline, consisting of rollout, reward com-

Algorithm 1: BootTrans

Input: Seed dataset $\mathcal{D}_{\text{seed}}$, test units T, initial policy $\pi_{\theta_{\text{init}}}$, hyperparameters N, B, G
Output: Optimized policy π_θ

$\pi_\theta \leftarrow \pi_{\theta_{\text{init}}}, \mathcal{D}_{\text{explore}} \leftarrow \emptyset$;
for *step* $\leftarrow 1$ **to** N **do**
  ▷ Sample a batch of source code
  $X \leftarrow \{x_1, \ldots, x_B\}$ from $\mathcal{D}_{\text{seed}} \cup \mathcal{D}_{\text{explore}}$;
  foreach *source code* $x_i \in X$ **do**
    Let L^i_{src} be the language of x_i, define target languages $\mathcal{L}^i_{\text{tgt}} = \mathcal{L} \setminus \{L^i_{\text{src}}\}$;
    foreach *target language* $L_k \in \mathcal{L}^i_{tgt}$ **do**
      ▷ Generate G candidates
      $\{y^1_{i,k}, \ldots, y^G_{i,k}\} \sim \pi_\theta(\cdot \mid x_i, L_k)$;
      ▷ Verify and reward computation
      Get $R(y^j_{i,k}, T)$ using Eq. (1);
      ▷ Expand exploration pool
      $\mathcal{D}_{\text{explore}} \leftarrow \cdot \cup \{y^j_{i,k} \mid R(y^j_{i,k}, T) = 1\}$;
    foreach $L_k \in \mathcal{L}^i_{tgt}$ **do**
      ▷ Compute cumulative reward
      $\mathcal{R}_{i,k} \leftarrow \sum_{j=1}^{G} R(y^j_{i,k}, T)$;
      ▷ Compute sibling reward
      $\mathcal{R}_{i,\neg k} \leftarrow \sum_{L_j \in \mathcal{L}^i_{\text{tgt}}, j \neq k} \mathcal{R}_{i,m}$;
      ▷ Compute language-aware weight
      $w_{i,k} \leftarrow \frac{\mathcal{R}_{i,\neg k}}{\mathcal{R}_{i,k} + \mathcal{R}_{i,\neg k}}$;
  Update π_θ by maximizing Eq. (3);

putation, and policy update. In each training iteration, we construct a batch by sampling source code x from both $\mathcal{D}_{\text{seed}}$ and $\mathcal{D}_{\text{explore}}$. Given x in language L_{src}, we obtain rollouts from the policy π_θ by generating G candidate translations $\mathcal{Y}_k = \{y^1_k, \ldots, y^G_k\}$ for each target language $L_k \in \mathcal{L} \setminus \{L_{\text{src}}\}$. We verify these candidates against the test oracle T and compute rewards for $\mathcal{Y}_k$. Subsequently, we perform experience collection to update the exploration pool $\mathcal{D}_{\text{explore}}$ immediately following the reward computation. If multiple candidates in $\mathcal{Y}_k$ pass T, we randomly retain one for $\mathcal{D}_{\text{explore}}$. Conversely, if no candidate passes T, no entry is added, pruning the exploration of tasks beyond the model's current reach. These verified translations are eligible to be sampled as *source* inputs in subsequent iterations. This enables the model to learn reverse translations (e.g., Java→Python) and cross-lingual translations (e.g., Java→C++) that are absent in the seed data. As a result, we progressively populate the $\mathcal{L} \times \mathcal{L}$ translation task matrix.

3.2 Language-aware Weight Optimization

To address the imbalance in optimization caused by varying translation capabilities across target languages, we introduce a language-aware weight optimization mechanism. Intuitively, the learning signal for a specific target language L_k should be amplified when the model underperforms on L_k despite demonstrating high proficiency in other "sibling" languages for the same source code.

For each specific target language $L_k \in \mathcal{L}^i_{\text{tgt}}$ for source code x_i, let $\mathcal{R}_{i,k} = \sum_{y \in \mathcal{Y}_{i,k}} R(y, T)$ denote the cumulative reward obtained by the candidate group $\mathcal{Y}_{i,k}$. We define the sibling reward $\mathcal{R}_{i,\neg k} = \sum_{L_j \in \mathcal{L}^i_{\text{tgt}}, j \neq k} \mathcal{R}_{i,j}$ as the aggregate performance on all other target languages. The optimization weight $w_{i,k}$ for the translation task $L_{\text{src}} \to L_k$ is computed as:

$$w_{i,k} = \frac{\mathcal{R}_{i,\neg k}}{\mathcal{R}_{i,k} + \mathcal{R}_{i,\neg k}}. \tag{2}$$

When $\mathcal{R}_{i,k} + \mathcal{R}_{i,\neg k} = 0$ (i.e., all candidates fail across all target languages for x_i), the weight is undefined, and in this case, we skip x_i in the policy update. This weighting scheme dynamically prioritizes lagging directions: if the model demonstrates semantic understanding via sibling languages (high $\mathcal{R}_{i,\neg k}$) but struggles with L_k (low $\mathcal{R}_{i,k}$), $w_{i,k}$ increases, forcing the model to focus on the syntactic or idiomatic hurdles of L_k. We employ the Group Relative Policy Optimization (GRPO) algorithm to optimize the policy with language-aware weighting. The policy is updated by maximizing:

$$\mathcal{J}(\theta) = \mathbb{E}\Big[\sum_{i,k} w_{i,k} \frac{1}{G} \sum_{j=1}^{G} \frac{1}{|o_j|} \sum_{t=1}^{|o_j|} \big(\min(r_{j,t}\hat{A}^j_{i,k,t}, \tilde{c}_{j,t}\hat{A}^j_{i,k,t}) - \beta \mathbb{D}_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}})\big)\Big], \tag{3}$$

where $r_{j,t} = \frac{\pi_\theta(y^j_{i,k,t} \mid x_i, y^j_{i,<t}, L_k)}{\pi_{\theta_{\text{old}}}(y^j_{i,k,t} \mid x_i, y^j_{i,<t}, L_k)}$ denotes the importance sampling ratio for the t-th token of the j-th candidate translation $y^j_{i,k} \in L_k$, and $\tilde{c}_{j,t} = \text{clip}\big(r_{j,t}, 1-\epsilon, 1+\epsilon\big)$ denotes the clipped probability ratio, where ϵ controls the clipping range. $w_{i,k}$ is the language-aware weight from Eq. (2), $|o_j|$ is the length of $y^j_{i,k}$, and β is the KL penalty coefficient. $\hat{A}^j_{i,k,t}$ is the advantage of t-th token in $y^j_{i,k}$ estimated by the same target language group:

$$\hat{A}^j_{i,k,t} = \frac{R\big(y^j_{i,k}, T\big) - \text{mean}\big(\{R(y,T)\}_{y \in \mathcal{Y}_{i,k}}\big)}{\text{std}\big(\{R(y,T)\}_{y \in \mathcal{Y}_{i,k}}\big)}. \tag{4}$$

4 Experiments and Results

4.1 Experiment Settings

Training data. The training dataset for BootTrans is constructed based on KodCode (Xu et al., 2025).

LLMs	HumanEval-X							TransCoder-Test						
	P→J	P→C	J→P	J→C	C→J	C→P	Avg	P→J	P→C	J→P	J→C	C→J	C→P	Avg
Qwen3-1.7B	54.27	43.29	82.32	58.54	75.00	72.56	64.33	75.52	76.66	80.17	81.37	83.61	80.17	79.58
Qwen3-32B	68.29	**64.63**	86.59	62.20	60.98	65.24	67.99	57.47	67.88	78.02	72.81	59.75	76.08	68.67
BootTrans Qwen3-1.7B	**73.78**	60.37	**87.20**	**70.73**	**77.44**	**78.66**	**74.70**	**79.88**	**85.87**	**83.62**	**91.86**	**84.85**	**82.11**	**84.70**
Llama-3.1-8B-Instruct	37.20	57.32	84.76	47.56	73.17	70.73	61.79	75.10	79.44	75.00	87.58	80.29	72.20	78.27
Llama-3.1-70B-Instruct	**87.80**	**78.66**	**86.59**	**83.54**	**87.20**	**84.76**	**84.76**	79.25	**90.15**	80.39	90.15	81.95	79.53	83.57
BootTrans Llama-3.1-8B-Instruct	73.78	66.46	85.98	76.83	84.76	82.32	78.36	**78.01**	86.51	**84.05**	**92.29**	**82.78**	**81.03**	**84.11**
Qwen2.5-7B-Instruct	51.22	69.51	86.59	59.15	64.02	80.49	68.50	86.51	88.87	87.72	92.29	89.00	84.91	88.22
Qwen2.5-32B-Instruct	62.80	74.39	**90.85**	55.49	65.24	82.32	71.85	86.72	**92.51**	**89.22**	**93.15**	89.00	85.99	**89.43**
BootTrans Qwen2.5-7B-Instruct	**81.71**	**76.83**	90.24	**82.32**	**89.02**	**82.93**	**83.84**	**86.72**	89.72	88.79	92.72	**90.87**	**86.64**	89.24

Table 1: CA@1 scores on HumanEval-X and TransCoder-Test benchmarks. "C", "J", and "P" denote C++, Java, and Python, respectively. For each model family, we show: (i) base model, (ii) large-scale reference (shaded), and (iii) our method BootTrans (**bold when highest**). "Avg" denotes the average score across six translation directions.

We pick the KodCode-RL-10K subset and extract Python solutions and test cases. We leverage the MultiPL-E (Cassano et al., 2023) translator to extend these test cases to Java and C++. Following its usage, we apply the translation templates to map Python unit-test scaffolds (e.g., entrypoint signatures and assertions) into target-language test harnesses, and discard those translated tests that fail to compile/execute or have ambiguous entrypoint signatures. To prevent data leakage, we further remove any training instances whose function names overlap with HumanEval-X or TransCoder-Test. Ultimately, we curate a dataset comprising 5,584 Python source code, each accompanied by test suites in Python (avg. 8.11 cases), Java (avg. 8.09), and C++ (avg. 8.09). See Appendix A for more details.

Implementation. The training process uses AdamW optimizer with a learning rate of 1×10^{-6}. For GRPO, the rollout macro batch size is set to 256 with $G = 8$, and the micro batch size for actor training is 8. The KL penalty coefficient $\beta = 0.01$ and the clipping range $\epsilon = 0.2$. During inference, we use greedy decoding for all evaluations to ensure deterministic and reproducible results.

Baselines. We compare our method against two categories of baselines. (i) *Competing open-source LLMs*. We select three widely adopted instruction-tuned checkpoints: Qwen3-32B, Qwen2.5-32B-Instruct, and Llama-3.1-70B-Instruct. They span from 32B to 70B parameters, providing competitive zero-shot code translation capabilities. (ii) *Representative finetuning methods*:

- **EffiReasonTrans** (Wang et al., 2025), a reasoning-enhanced code translation method that conducts RL to optimize CoT paths.
- **CoTran** (Jana et al., 2024), a collaborative RL approach that aligns source and target languages by maximizing execution-based rewards and compiler rewards.
- **MultiPL-T** (Cassano et al., 2024), a multilingual code data synthesis framework with a powerful teacher model to generate candidate programs and rejection sampling to curate a dataset for supervised finetuning. We synthesize Java and C++ implementations by ensembling Qwen3-32B and Llama-3.1-70B-Instruct, retaining only those passing all test cases. Together with the original Python solutions, this process yields 28,570 translation pairs in total for supervised finetuning.
- **PPOCoder** (Shojaee et al., 2023), an RL-based approach leveraging the PPO algorithm to optimize code translation performance.
- **OORL** (Wu et al., 2025), a method integrating online RL objectives with offline group DPO training objectives derived from intermediate representations.

These baselines allow us to assess whether BootTrans can achieve superior cross-lingual generalization compared to both massive-scale general models and specialized finetuning methods. More details are provided in Appendix B.

Evaluation benchmarks and metrics. We employ the HumanEval-X and TransCoder-Test benchmarks and choose Python, Java, and C++ programming languages. See Appendix C for more details. We use top-1 Computational Accuracy (CA@1) as the efficacy metric.

4.2 Main Results

To investigate whether our BootTrans can improve code translation accuracy, we compare its perfor-

Methods	HumanEval-X							TransCoder-Test						
	P→J	P→C	J→P	J→C	C→J	C→P	Avg	P→J	P→C	J→P	J→C	C→J	C→P	Avg
EffiReasonTrans	57.32	38.41	82.32	60.37	**77.44**	75.61	65.25	78.22	73.88	83.40	81.16	84.44	81.03	80.36
CoTran	54.88	43.29	82.32	56.71	74.39	72.56	64.03	75.10	76.87	79.96	79.66	83.40	79.96	79.16
MultiPL-T	64.02	45.73	73.78	62.20	75.00	67.68	64.74	74.48	**86.08**	74.48	79.66	80.91	73.49	78.18
PPOCoder	68.29	54.27	82.32	62.80	73.78	73.78	69.21	75.10	79.44	80.60	86.51	84.44	80.17	81.04
OORL	71.34	59.15	79.27	60.98	76.83	71.95	69.92	73.86	76.02	72.84	84.37	81.74	62.50	75.22
BootTrans (ours)	**73.78**	**60.37**	**87.20**	**70.73**	**77.44**	**78.66**	**74.70**	**79.88**	85.87	**83.62**	**91.86**	**84.85**	**82.11**	**84.70**

Table 2: CA@1 scores of different methods on HumanEval-X and TransCoder-Test, using Qwen3-1.7B.

Methods	HumanEval-X							TransCoder-Test						
	P→J	P→C	J→P	J→C	C→J	C→P	Avg	P→J	P→C	J→P	J→C	C→J	C→P	Avg
BootTrans	**73.78**	**60.37**	**87.20**	**70.73**	**77.44**	**78.66**	**74.70**	**79.88**	**85.87**	**83.62**	**91.86**	**84.85**	**82.11**	**84.70**
– Exploration	70.73	56.10	81.71	62.20	76.22	77.44	70.73	79.25	83.08	81.03	88.87	84.23	80.82	82.88
– Weighting	68.90	57.93	85.98	68.29	76.22	75.61	72.16	79.57	82.66	81.90	89.94	83.61	81.47	83.19

Table 3: CA@1 scores of ablation study with Qwen3-1.7B.

mance with its base model on six translation tasks.

Table 1 presents the experimental results. Across all benchmarks and translation tasks, BootTrans consistently outperforms its base model. These gains are not confined to translation tasks from Python, showing that BootTrans successfully propagates the learning signal beyond the pivot-language data. A closer look reveals that the weaker the base model, the greater the lift. This is precisely the expected pattern of BootTrans: multi-language exploration first mines high-confidence translations from the previous iteration and re-feeds them as synthetic source code, instantly expanding the search space; subsequent weight optimization then amplifies the learning signal for the weaker translation tasks.

The gains also generalize across different model families. As shown in Table 1, on HumanEval-X and TransCoder-Test, BootTrans yields average improvements of 10.37% and 5.12% for Qwen3-1.7B, 16.57% and 5.84% for Llama-3.1-8B-Instruct, 15.35% and 1.03% for Qwen2.5-7B-Instruct, respectively. Compared with the larger-scale sibling in the same model family, it achieves comparable overall performance and even excels in several specific directions despite much smaller parameters.

Moreover, we compare BootTrans with strong finetuning code-translation baselines in Table 2. To guarantee a strictly fair comparison, all methods are initialized from the same Qwen3-1.7B base model and trained on the same dataset with BootTrans, except for EffiReasonTrans, which uses its released dataset. BootTrans outperforms these methods on both benchmarks on average, with particularly large gains in J→C. Overall, these results demonstrate that BootTrans is an effective RLVR method for multilingual code translation, leveraging the scalability of unit tests as verifiable oracles.

4.3 Ablation Study

To measure the individual contributions of bootstrapping multilanguage exploration and language-aware weight optimization, we design two variants:

- **w/o Exploration** removes $\mathcal{D}_{\text{explore}}$, restricting the RL training on the initial pivot seed $\mathcal{D}_{\text{seed}}$, which only covers P→J/C training data.
- **w/o Weighting** removes the language-aware weighting by setting uniform weights (i.e., $w_{i,k} = 1$ for all tasks), and thus all samples contribute equally to the RL objective.

Table 3 reports CA@1 on HumanEval-X and TransCoder-Test with Qwen3-1.7B. Removing either component consistently degrades performance across all six directions. On HumanEval-X, disabling exploration leads to an average drop of 4%, showing that bootstrapped multilingual instances are crucial for improving overall performance. Without the exploration pool, the model can only learn from Python-to-X translations, missing critical reverse and cross-lingual patterns (e.g., J→P, J→C). This validates that multilingual bootstrapping is essential in evolving beyond pivot-to-X constraints toward broader multilingual translation tasks.

Removing language-aware weighting also decreases performance by 2.5% on average, indicating that reweighting is crucial for improving hard

directions. This effect is most evident on the challenging P→C tasks. Without adaptive weighting, the model tends to over-optimize on easier translation pairs while neglecting harder ones, leading to imbalanced multilingual proficiency. The weighting mechanism acts as a curriculum that prevents the policy from getting trapped in exploiting simpler patterns for higher rewards while neglecting the harder tasks.

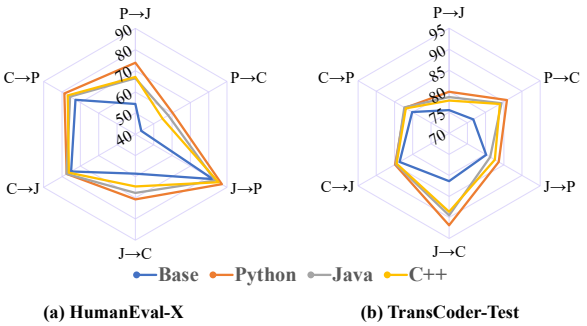
4.4 Different Pivot Languages

To see how the choice of pivot language influences performance, we repeat the full training pipeline of BootTrans with Qwen3-1.7B by substituting the default Python pivot with Java and C++. Since the original training split contains only Python code as the starting point, we adopt the synthesized Java and C++ code used in MultiPL-T. This includes 5,254 Java and 4,555 C++ source programs, which serve as cross-lingual counterparts to a subset of the original Python references for training. We categorize these as "silver-standard" references compared to the "gold-standard" Python seeds.

As depicted in Figure 3, our method achieves consistent performance gains compared with the base model regardless of the chosen pivot language, indicating that the core mechanism of BootTrans is effective. The radar chart visualizes the CA@1 scores across all six translation directions on both HumanEval-X and TransCoder-Test benchmarks, where each axis represents a specific translation task. While BootTrans is language-agnostic, using Python as the pivot yields the superior overall performance, with the largest coverage area in the radar chart. We attribute this advantage to two factors. The first advantage stems from the fact that our Python seed data is inherently more abundant and covers a significantly broader range of algorithmic logic and functional scenarios, providing a higher-quality initialization for exploration. Second, LLMs are typically optimized with a higher proportion of Python corpora during pre-training. Consequently, starting the exploration from Python leverages the model's strongest internal representations, providing more potential for cross-lingual knowledge transfer.

4.5 Compatibility to Existing Framework

Recent breakthroughs in code translation focus on inference-time strategies, which are orthogonal to the training-time optimizations of BootTrans. To further demonstrate the compatibility of BootTrans



(a) HumanEval-X (b) TransCoder-Test

Figure 3: Performance comparison of different pivot languages with Qwen3-1.7B.

to inference-time enhancement strategies, we evaluate it by integrating it with two distinct paradigms:

- Path-based scaling via **InterTrans** (Macedo et al., 2025), which explores multiple translation routes. This method capitalizes on the model's inherent multilingual translation proficiency by expanding the sampling budget. The performance of InterTrans serves as a direct indicator of the base model's multilingual code translation effectiveness.

- Iterative refinement via **UniTrans** (Yang et al., 2024), which utilizes execution feedback to progressively refine the output. It exploits the model's self-refine ability.

As presented in Table 4, BootTrans serves as a solid foundation for InterTrans and UniTrans. While BootTrans improves the success rate at first attempt, the integration with InterTrans and UniTrans further unlocks its potential. Specifically, when integrated with InterTrans, the CA@1 scores exhibit a substantial further improvement, providing more reliable candidates across diverse translation paths.

Furthermore, the integration with UniTrans reveals that despite undergoing a translation-specific RL process, the model preserves its intrinsic self-refine ability. This suggests that BootTrans is additive to existing test-time compute-scaling and feedback-driven methods.

4.6 Language Extension

To further validate the generalization of BootTrans, we extend our evaluation to a broader spectrum of programming languages on the HumanEval-X benchmark. Specifically, we investigate whether BootTrans can effectively generalize to unseen languages, which are not encountered during the train-

Methods	HumanEval-X							TransCoder-Test						
	P→J	P→C	J→P	J→C	C→J	C→P	Avg	P→J	P→C	J→P	J→C	C→J	C→P	Avg
BootTrans	73.78	60.37	87.20	70.73	77.44	78.66	74.70	79.88	85.87	83.62	91.86	84.85	82.11	84.70
+ InterTrans	84.15	76.22	90.85	76.22	84.15	84.76	82.73	86.31	88.65	84.91	93.15	88.80	83.19	87.50
+ UniTrans	78.05	60.98	87.80	71.34	80.49	81.71	76.73	89.83	88.87	87.72	93.36	91.70	86.85	89.72

Table 4: Compatibility with inference-time enhancement strategies.

Unseen	J→Go	C→Go	JS→Go	Go→J	Go→C	Go→JS
Qwen3-1.7B	45.73	54.88	53.05	70.12	52.44	92.07
+BootTrans	**60.37**	**57.93**	**66.46**	**75.0**	**57.93**	**96.34**
Low-res.	P→D	P→R	J→D	J→R	C→D	C→R
Qwen3-1.7B	23.08	12.18	21.79	13.46	25.00	16.03
+BootTrans	**41.67**	**28.85**	**33.33**	**23.08**	**41.67**	**25.64**

Table 5: CA@1 scores of extended language pairs. "JS", "D", and "R" denote JavaScript, Dlang, and Racket, respectively.

ing phase. We select Go as a representative. Furthermore, acknowledging that LLMs often falter when dealing with low-resource languages due to the scarcity of training corpora (Oida-Onesa and Ballera, 2024), we assess the performance of BootTrans on Dlang and Racket. For these low-resource scenarios, we leverage the MultiPL-E (Cassano et al., 2023) to extend the test cases.

As illustrated in Table 5, BootTrans consistently outperforms the base Qwen3-1.7B model across all tested pairs. Notably, in the low-resource scenarios such as P→D, BootTrans achieves a significant performance uplift, demonstrating its superior capability in bridging the gap for languages with limited data availability. This performance leap suggests that BootTrans effectively enhances cross-lingual alignment capabilities, even for languages that are not explicitly optimized during training or those with a minimal data footprint.

4.7 Class-level Code Translation

To further evaluate the robustness and generalizability of BootTrans in complex, real-world scenarios, we conduct an additional experiment on ClassEval-T (Xue et al., 2025), a significantly more challenging benchmark for class-level code translation. Unlike conventional method-level or program-level tasks, the benchmark involves longer contexts, intricate cross-method dependencies, and external library calls, requiring the model to maintain consistency across an entire class. Adopting the standard protocol for this benchmark, we employ class-level CA (CA_c) and method-level CA (CA_m) to evaluate performance.

As shown in Table 6, even without direct training on class-level data, BootTrans achieves improvements across most translation pairs. We observe no gains in the J→C task, which we attribute to the inherent capacity limits of the base model in handling C++'s complex memory management and syntax at scale.

4.8 Case Study

To investigate the qualitative impact of BootTrans beyond numerical metrics, we analyze specific translation instances to understand how BootTrans shifts the model's translation behavior.

As shown in Figure 4, in the top string encoding case, the C++ expression *w + 2* increments the ASCII code of w by 2, yielding another single character. However, Qwen3-1.7B translates it into *+ '2'*, which is a string concatenation operation. It alters both the length and the content of the output. In contrast, BootTrans faithfully reproduces the original source C++ code's behavior with *ord/chr + 2* . This demonstrates that BootTrans preserves the exact logic of the source code.

The bottom GCD case highlights the aggressive nature of our exploration strategy. Qwen3-1.7B remains a literal translation of % operator. Conversely, BootTrans uses the Pythonic *divmod* idiom. Despite misalignment in return types, BootTrans empowers the model to search for higher-level algorithmic equivalents. Collectively, both cases suggest that BootTrans generates more idiomatic and native-like translations. While exploration entails risks, as seen in the idiomatic misuse, it represents a vital opportunity for achieving more complex translations, such as built-in function mapping and API adaptation. See Appendix E for more details about the categorization of translation errors.

5 Related Work

Code translation research has evolved through multiple paradigms. Early unsupervised methods, e.g., TransCoder (Roziere et al., 2020), TransCoder-ST (Roziere et al., 2022), and structure-aware variants (Szafraniec et al., 2023; Huang et al., 2023; Liu

Models	C→P		J→P		P→J		C→J		P→C		J→C	
	CA_c	CA_m	CA_c	CA_m	CA_c	CA_m	CA_c	CA_m	CA_c	CA_m	CA_c	CA_m
Qwen3-1.7B	18.09	28.72	17.02	25.53	3.19	30.85	6.38	38.30	4.26	10.64	0.00	0.00
BootTrans Qwen3-1.7B	**23.40**	**35.11**	**19.15**	**32.98**	**8.51**	**46.81**	**9.57**	**56.38**	**4.26**	**10.64**	**0.00**	**1.06**
Qwen2.5-7B-Instruct	36.17	48.93	28.72	**38.30**	4.26	32.98	4.26	32.99	3.19	6.38	**1.06**	**1.06**
BootTrans Qwen2.5-7B-Instruct	**40.43**	**52.13**	**29.79**	**38.30**	**18.08**	**68.09**	**14.89**	**74.47**	**4.26**	**9.57**	**1.06**	**1.06**

Table 6: CA_c and CA_m scores on ClassEval-T benchmark.

Source code

```
string encode(string message) {
  string vowels = "aeiouAEIOU";
  string out = "";
  for (int i = 0; i < message.length(); i++) {
    char w = message[i];
    if (find(vowels.begin(), vowels.end(), w)
        != vowels.end()) w = w + 2;
  }
}
```

Qwen3-1.7B

```
def encode(message):
  vowels = "aeiouAEIOU"
  out = ""
  for char in message:
    if char in vowels:
      char = char.upper() + '2'
```

BootTrans Qwen3-1.7B

```
def encode(message):
    vowels = "aeiouAEIOU"
    out = ""
    for char in message:
      if char in vowels:
        char = chr(ord(char) + 2)
```

```
int greatest_common_divisor(int a, int b) {
    int m;
    while (true) {
      if (a < b) {
        m = a; a = b; b = m;
      }
      a = a % b;
      if (a == 0) return b;
    }
}
```

```
def greatest_common_divisor(a:
int, b: int) -> int:
    while True:
        if a < b:
            a, b = b, a
        a, b = a % b, b
        if a == 0:
            return b
```

```
def greatest_common_divisor(a:
int, b: int) -> int:
    while True:
        if a < b:
            a, b = b, a
        a, b = divmod(a, b)
        if b == 0:
            return a
```

Figure 4: Two examples of C++ translated into Python. BootTrans succeeds in the top case but fails in the bottom.

et al., 2023) avoid parallel data dependency but require massive corpora and heavy computation. With the availability of larger parallel corpora (Zhu et al., 2022a; Yan et al., 2023; Khan et al., 2024; Yan et al., 2024) and pre-trained models (Wang et al., 2021; Guo et al., 2021; Zheng et al., 2023; Feng et al., 2020), supervised finetuning has become the dominant paradigm. Recent work emphasizes executable, semantics-oriented evaluation: MultiPL-E (Cassano et al., 2023) enables scalable compile-and-run testing; reliability analyses and richer testing further study functional correctness (Pan et al., 2024; Enişer et al., 2024).

Beyond supervised finetuning, RLVR offers a promising direction. CodeRL (Le et al., 2022) pioneers execution-based signals for code generation, while CoTran (Jana et al., 2024), PPOCoder (Shojaee et al., 2023), and OORL (Wu et al., 2025) integrate compiler and symbolic-execution feedback. EffiReasonTrans (Wang et al., 2025) combines reasoning augmentation with RL to balance accuracy and latency. These methods apply PPO or GRPO (Schulman et al., 2017; Shao et al., 2024) for reward-based optimization. However, existing RLVR methods often rely on uniform optimization across language pairs and may be sensitive to test coverage. BootTrans addresses the challenges of data scarcity and multilingual optimization imbalance through bootstrapping multilanguage exploration and language-aware weighting. Test-time improvements such as explanations, iterative feedback or transitive intermediate translations can also boost quality (Tang et al., 2023; Macedo et al., 2025; Yang et al., 2024), motivated by multilingual software co-evolution (Zhang et al., 2023).

6 Conclusion

This paper proposes BootTrans, a novel method for multilingual code translation to address data scarcity and optimization imbalance challenges. By leveraging functional invariance and portable test oracles, we establish universal oracles for multilingual RL training. We integrate a dual-pool architecture and a language-aware weighting mechanism to bootstrap multilanguage exploration and achieve a balanced optimization. Our experiments demonstrate substantial improvements over strong baselines, with ablations confirming the effectiveness of both key components. Future work will extend BootTrans to functional languages and explore more sophisticated reward mechanisms.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62272219), the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118), and the Cooperation Fund of Huawei Cooperation Project (No. TC20230202021-2024-12).

Ethical Considerations

The datasets, benchmarks, and LLMs used in this paper are public with permissive licenses.

Limitations

While BootTrans demonstrates substantial improvements in multilingual code translation, several limitations warrant discussion. First, our evaluation mainly focuses on three imperative languages (Python, Java, C++) and may require additional adaptation for domain-specific languages with fundamentally different paradigms. Second, our language-aware weighting relies on execution-based binary rewards, which may not capture nuanced aspects of code quality such as readability, maintainability, or idiomatic style. Finally, the efficacy of BootTrans is partially influenced by the test suite size as shown in Appendix F. A scarcity of test cases in the initial pivot language may lead to a performance degradation.

References

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. In *OOPSLA2*, pages 677–708, New York, NY, USA. ACM.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A scalable and extensible approach to benchmarking neural code generation. *IEEE Trans. Softw. Eng.*, pages 3675–3691.

Yali Du, Hui Sun, and Ming Li. 2024. A joint learning model with variational interaction for multilingual program translation. In *ASE*, pages 1907–1918, Sacramento, CA, USA. ACM.

Hasan Ferit Enişer, Valentin Wüstholz, and Maria Christakis. 2024. Automatically testing functional properties of code translation models. In *AAAI*, volume 38, pages 21055–21062.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *EMNLP*, pages 1536–1547, Virtual. ACL.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *ICLR*, pages 1–18, Virtual. OpenReview.net.

Yufan Huang, Mengnan Qi, Yongqiang Yao, Maoquan Wang, Bin Gu, Colin Clement, and Neel Sundaresan. 2023. Program translation via code distillation. In *EMNLP*, pages 10903–10914, Singapore. ACL.

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*, pages 1–15, Santiago de Compostela, Spain. IOS Press.

Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the evaluation of neural code translation: Taxonomy and benchmark. In *ASE*, pages 1529–1541, Kirchberg, Luxembourg. IEEE.

Mohammad Abdullah Matin Khan, M. Saiful Bari, Xuan Do Long, Weishi Wang, Md. Rizwan Parvez, and Shafiq Joty. 2024. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *ACL*, pages 6766–6805, Bangkok, Thailand. ACL.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C.H. Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *NeurIPS*, pages 1–15, New Orleans, LA, USA. Curran Associates, Inc.

Fang Liu, Jia Li, and Li Zhang. 2023. Syntax and domain aware model for unsupervised program translation. In *ICSE*, pages 755–767, Melbourne, VIC, Australia. IEEE.

Marcos Macedo, Yuan Tian, Pengyu Nie, Filipe R. Cogo, and Bram Adams. 2025. InterTrans: Leveraging transitive intermediate translations to enhance LLM-based code translation. In *ICSE*, pages 1153–1164, Ottawa, ON, Canada. IEEE.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Migrating code with statistical machine translation. In *ICSE Companion*, pages 544–547, Hyderabad, India. ACM.

Rosel Oida-Onesa and Melvin A. Ballera. 2024. Fine tuning language models: A tale of two low-resource languages. *Data Intelligence*, 6(4):946–967.

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 *ICSE*, pages 1–13.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *NeurIPS*, pages 20601–20611, Vancouver, BC, Canada. Curran Associates Inc.

Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging automated unit tests for unsupervised code translation. In *ICLR*, pages 1–20, Virtual. OpenReview.net.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *CoRR*, 1707.06347:1–12.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y.K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, 2402.03300:1–30.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based code generation using deep reinforcement learning. *Trans. Mach. Learn. Res.*, 2023:1–26.

Marc Szafraniec, Baptiste Roziere, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2023. Code translation with compiler representations. In *ICLR*, pages 1–20, Kigali, Rwanda. OpenReview.net.

Zilu Tang, Mayank Agarwal, Alexander Shypula, Bailin Wang, Derry Wijaya, Jie Chen, and Yoon Kim. 2023. Explain-then-translate: an analysis on improving program translation with self-generated explanations. In *EMNLP Findings*, pages 1741–1788.

Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2024. RepoTransBench: A real-world benchmark for repository-level code translation. *CoRR*, 2412.17744:1–23.

Yanlin Wang, Rongyi Ou, Yanli Wang, Mingwei Liu, Jiachi Chen, Ensheng Shi, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. EffiReasonTrans: RL-optimized reasoning for code translation. *arXiv*, 2510.18863:1–30.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, pages 8696–8708, Punta Cana, Dominican Republic. ACL.

Haoyuan Wu, Rui Ming, Jilong Gao, Hangyu Zhao, Xueyi Chen, Yikai Yang, Haisheng Zheng, Zhuolun He, and Bei Yu. 2025. On-policy optimization with group equivalent preference for multi-programming language understanding. In *NeurIPS*, pages 1–38, Sydney, Australia. OpenReview.net.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. 2025. KodCode: A diverse, challenging, and verifiable synthetic dataset for coding. In *ACL Findings*, pages 6980–7008, Vienna, Austria. ACL.

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. *ISTTA*, 2:1421–1444.

Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Hari Sundaram, and Shuiguang Deng. 2024. CodeScope: An execution-based multilingual multitask multidimensional benchmark for evaluating LLMs on code understanding and generation. In *ACL*, pages 5511–5558, Bangkok, Thailand. ACL.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. CodeTransOcean: A comprehensive multilingual benchmark for code translation. In *EMNLP*, pages 5067–5089, Singapore. ACL.

Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proc. ACM Softw. Eng.*, 1:1585–1608.

Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual code co-evolution using large language models. In *FSE*, pages 695–707.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *KDD*, pages 5673–5684, Long Beach, CA, USA. ACM.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022a. XLCoST: A benchmark dataset for cross-lingual code intelligence. *CoRR*, 2206.08474:1–20.

Ming Zhu, Mohimenul Karim, Ismini Lourentzou, and Daphne Yao. 2024. Semi-supervised code translation overcoming the scarcity of parallel code data. In *ASE*, pages 1545–1556, Sacramento, CA, USA. IEEE/ACM.

Ming Zhu, Karthik Suresh, and Chandan K. Reddy. 2022b. Multilingual code snippets training for program translation. In *AAAI*, volume 36, pages 11783–11790, Virtual. AAAI.

A Details of Training Data Construction

In the original KodCode dataset, not all Python code snippets are accompanied by function signatures. We use Qwen3-32B to annotate the function signatures, verify the correctness of these annotated function signatures, and ultimately retain only the Python source code snippets with valid function signatures. The test cases provided with the Python code in the original KodCode dataset have all been verified to be correct. Therefore, we directly use MultiPL-E to translate these test cases, and filter out the test cases that failed to be translated (the proportion of such failed cases is very small, only 0.2%). To avoid data leakage, we filtered out all Python code snippets whose function entry points overlap with those in HumanEval-X and TransCoder-Test. Finally, we obtained a total of 5,584 valid data samples.

B Details of Baseline Implementation

Unlike prior studies that assume the availability of parallel training data, our setting operates under a monolingual-pivot regime. To ensure a fair comparison, we adapt these baselines to align with our non-parallel training constraints while preserving their original algorithmic essence.

For EffiReasonTrans, it relies on costly chain-of-thought distillation from DeepSeek-R1 through multi-step reasoning annotations. Due to the prohibitive expense of reproducing this pipeline, we instead utilize its publicly released reasoning-augmented dataset to ensure its competitive edge. We first perform supervised finetuning, followed by RL using GRPO, maintaining the same reward function and hyperparameter configurations as BootTrans for parity.

For MultiPL-T, which originally uses a single teacher model (e.g., StarCoder) for data synthesis, we enhance its data curation process by employing two powerful open-source models, Llama-3.1-70B and Qwen3-32B, to perform rejection sampling on BootTrans's seed dataset. Specifically, we generate candidate translations from both models for the to-Java and to-C++ tasks and retain only valid translations that pass all unit tests. We then pair these translations with Python in seed dataset and construct a clean parallel code translation dataset for superised finetuning. This modified pipeline better aligns with our non-parallel training constraint while preserving the core idea of MultiPL-T.

For CoTran, we follow the original implementation, including its forward–backward policy architecture and reward formulation. However, we observe a significant performance degradation compared to both our method and other adapted baselines. We attribute this to a training-reward mismatch: the backward policy is optimized using the forward policy's outputs without execution-based filtering. Consequently, the resulting reward signal becomes noisy and misleading, as the backward policy may reward translations that are reconstructible but functionally incorrect.

For PPOCoder, its original reward mechanism relies on reference translations, which are unavailable in our setting. To adapt it fairly, we replace its reference-based reward with our unit test pass rate (execution feedback). We then apply the standard PPO algorithm to optimize the base model, serving as a representative baseline for conventional RL-based code translation without parallel supervision.

For OORL, we use the same dataset as BootTrans for the online RL component. For the offline component, since the original data is not publicly available, we randomly select 9,600 curated C-to-IR (intermediate representation) groups from the SLTrans dataset.

C Details of Benchmarks

We select HumanEval-X and TransCoder-Test as the evaluation datasets, due to their widespread recognition and adoption within the research community. The HumanEval-X dataset contains 164 source code snippets for each programming language. Thus, across the six cross-translation tasks among C++, Python, and Java, it constitutes a total of 984 test samples, with an average of 6.9 test cases per sample. For the TransCoder-Test dataset, the number of samples for the tasks of translating to Java, Python, and C++ are 482, 464, and 467, respectively. It constitutes a total of 2,826 test samples, with an average of 10 test cases per sample.

D Details of Prompt

We follow the code translation setting of HumanEval-X, where we leverage declarations (shared across both source and target languages) and translate the solution from the source language to the target language. The prompt template adopted for model training and inference is below:

Prompt

```
Please translate source {{source_lang}}
code to target {{target_lang}} code:
```{{source_lang}}
{{source_code}}```
The translated {{target_lang}} code should
be:
```{{target_lang}}
{{target_signature}}```
```

E Classification of Translation Errors

To analyze how BootTrans alters the model's translation behavior, we categorize translation errors into four primary types:

- **Logical inconsistency:** The translated code exhibits different runtime behavior or logic compared to the source code;
- **Syntactic invalidity:** The generated code violates the grammatical rules of the target programming language and fails to compile or parse;
- **API misuse:** Inappropriate usage of APIs in the target language, such as using deprecated functions, wrong argument order, or non-existent library calls;
- **Type mismatch:** The translated code employs data types or structures that are incompatible with the target language's type system or the intended operations, such as improper casting or incorrect collection types;
- **Signature mismatch:** Discrepancies between the generated function signatures and the expected entry point in the unit tests, which prevent the code from being correctly invoked.

As shown in Figure 5, the error distribution analysis reveals that BootTrans significantly outperforms the base Qwen3-1.7B model across multiple failure modes, particularly reducing API misuse and logical inconsistency. The dramatic 60% reduction in API-related errors underscores the effectiveness of BootTrans.

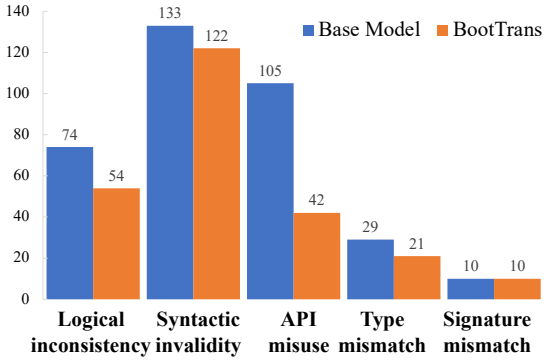


Figure 5: Error classification on HumanEval-X.

F Sensitivity to Unit Tests Scale

Since BootTrans relies on execution-based feedback as the primary signal for RL and data bootstrapping, the quality and quantity of test cases are critical. To investigate the BootTrans's sensitivity to the comprehensiveness of the verification oracle, we conduct experiments using different subsets of the original test suites. Specifically, we subsample the test cases for each problem to 50% and 25% of their original sizes randomly.

Based on the full set of test cases, BootTrans achieves the highest accuracy across all languages, as the dense test suite provides the most rigorous "grounding" for functional correctness. When the test cases are reduced to 50%, the average performance remains remarkably stable compared to the full-set setting; however, the impact on individual translation directions is mixed. The robustness at 50% test scale suggests that a representative subset is sufficient to capture the core functional logic. BootTrans effectively leverages this high-quality sparse feedback to maintain reliable weighting $w_{i,k}$. The 50% scale acts as a natural regularizer; it smooths the reward landscape and prevents extreme weight polarization. This leads to a more balanced performance. When the test cases are reduced to 25%, we observe a moderate performance degradation.

	P→J	P→C	J→P	J→C	C→J	C→P	Avg
Full	73.78	60.37	87.20	70.73	77.44	78.66	74.70
50%	74.39	65.85	84.15	72.56	72.56	78.66	74.70
25%	73.17	59.15	83.54	70.12	75.00	75.00	72.66

Table 7: Sensitivity analysis of BootTrans w.r.t. test suite size on HumanEval-X. We compare the translation performance on HumanEval-X using the full test suite against reduced versions containing 50% and 25% of the original test cases.