EffiReasonTrans: RL-Optimized Reasoning for Code Translation

Yanlin Wang, Rongyi Ou, Yanli Wang, Mingwei Liu*, Jiachi Chen, Ensheng Shi, Xilin Liu, Yuchi Ma, and Zibin Zheng

Abstract—**Code translation is a crucial task in software development and maintenance. While recent advancements in Large Language Models (LLMs) have improved automated code translation accuracy, these gains often come at the cost of increased inference latency—hindering real-world development workflows that involve human-in-the-loop inspection. To address this trade-off, we propose EffiReasonTrans, a training framework designed to improve translation accuracy while balancing inference latency. We first construct a high-quality reasoning-augmented dataset by prompting a stronger language model DeepSeek-R1 to generate intermediate reasoning and target translations. Each (source code, reasoning, target code) triplet undergoes automated syntax and functionality checks to ensure reliability. Based on this dataset, we employ a two-stage training strategy: supervised fine-tuning on reasoning-augmented samples, followed by reinforcement learning to further enhance accuracy, which also helps balance inference latency. We evaluate EffiReasonTrans on six translation pairs. Experimental results show that EffiReasonTrans consistently improves translation accuracy (up to +49.2% CA and +27.8% CodeBLEU compared to the base model), while reducing the number of generated tokens (up to -19.3%) and lowering inference latency in most cases (up to -29.0%). Ablation studies further confirm the complementary benefits of the two-stage training framework. Additionally, EffiReasonTrans shows improvements of translation accuracy when integrated into agent-based frameworks. Our code and data are available at https://github.com/DeepSoftwareAnalytics/EffiReasonTrans.**

Index Terms—**Code Translation, Large Language Models, Reasoning, Efficiency**

I. Introduction

CODE translation is a crucial task in software development and maintenance, involving converting source code from one programming language into another to suit different application scenarios [1], [2], [3], [4], [5], [6], [7]. Recent studies reveal that automated code translation techniques based on Large Language Models (LLMs) are promising [8], [3], [9], [10], [7], [11], [12], [13], [14], [15], [16], [17]. For example, UniTrans[3] enhances code translation accuracy by generating test cases and iteratively repairing errors using LLMs, while hmCodeTrans[7] leverages human-machine collaboration to improve accuracy. While these approaches have improved translation accuracy, challenges remain in handling complex translation scenarios. To address such challenges, explicit reasoning methods like Chain-of-Thought (CoT) prompting have been explored, which has been shown to improve performance on a range of arithmetic, commonsense, and symbolic reasoning tasks [18], [19], [20], [21]. Recent studies have incorporated them into the code translation task to improve performance [9], [10]. More recently, models like DeepSeek-R1 have been designed to generate internal reasoning processes without prompting [22], reflecting continued interest in utilizing the powerful reasoning capabilities of LLMs.

However, the enhanced performance comes at the cost of longer inference chains, **resulting in substantial increases in inference latency**. Our preliminary experiments show that while the reasoning model (i.e., DeepSeek-R1) improves translation accuracy from 86% to 92% compared to the non-reasoning model (i.e., DeepSeek-V3) on UniTrans-Dataset [3], there is a 540% increase in inference latency. While higher accuracy may reduce manual review time, the total task duration is constrained by both subjective human review and objective machine latency. In interactive development scenarios, developers typically expect rapid and reliable feedback, making inference speed a critical factor alongside translation accuracy. Unlike the highly variable human review time, machine-side latency is an objective, controllable variable that directly dictates user experience. This is especially relevant in human-in-the-loop pipelines such as hmCodeTrans [7], where excessive machine-side delays can disrupt development continuity and increase user anxiety. Furthermore, as review tasks are increasingly integrated into automated agent-based workflows, the latency of machine generation accounts for a more significant portion of the total execution cycle. In large-scale industrial systems, this interaction is often indispensable, and high inference latency can significantly hinder the development workflow. Therefore, we aim to answer this question: **how to optimize the trade-off between accuracy and efficiency while harnessing LLMs' powerful reasoning capabilities?**

In this paper, we propose **EffiReasonTrans**, a training framework that integrates reasoning-augmented data synthesis with a two-stage training process to improve code translation accuracy while optimizing inference latency. Specifically, EffiReasonTrans consists of three key stages: data synthesis, supervised fine-tuning, and reinforcement learning. ① EffiReasonTrans first synthesizes high-quality (source code, reasoning, target code) triplets by leveraging a more capable language model (DeepSeek-R1 [22]), and filters out incorrect or incomplete samples through automated syntax validation and functional testing. This process produces **EffiReasonTrans-**

Yanlin Wang, Rongyi Ou, Yanli Wang, Mingwei Liu, Jiachi Chen, and Zibin Zheng are with the School of Software Engineering, Sun Yat-sen University, Zhuhai 519082, China (e-mail: wangylin36, liumw26, chenjch86, zhzibin@mail.sysu.edu.cn; oury, wangyli58@mail2.sysu.edu.cn).

Ensheng Shi, Xilin Liu, and Yuchi Ma are with Huawei Cloud Computing Technologies Co., Ltd., Beijing 100085, China (e-mail: shiensheng, liuxilin3, mayuchi1@huawei.com).

Corresponding author: Mingwei Liu (e-mail: liumw26@mail.sysu.edu.cn).

Data, a reliable reasoning-augmented code translation corpus that captures the semantic and logical migration from source to target language. ② Next, EffiReasonTrans employs supervised fine-tuning followed by reinforcement learning (using the GRPO algorithm [23]), guided by a custom reward strategy with dual objectives: (1) execution correctness (test case pass rate) and (2) output conciseness (length tolerance), jointly reducing latency without sacrificing accuracy.

To evaluate the effectiveness of EffiReasonTrans, we conduct extensive experiments on 6 translation pairs among Python, Java, and C++ and obtain the following findings. ① Experiments demonstrate that EffiReasonTrans achieves superior accuracy while optimizing latency. For instance, on Java $\rightarrow$ Python, it improves CA by 27.4%, APR by 23.1%, and CodeBLEU by 10.1%, while reducing latency by 29.0%. ② Ablation studies reveal that both supervised fine-tuning and reinforcement learning contribute to these gains, with RL further boosting CA by up to 34.0% and reducing latency by 25.4%. ③ Notably, EffiReasonTrans maintains strong performance under limited model capacity and benefits from multilingual training data, showcasing its generalizability. ④ Finally, when integrated into agent-based frameworks, EffiReasonTrans preserves its accuracy improvements in end-to-end pipelines.

Our main contributions in this work include:

- We propose EffiReasonTrans, a two-stage training framework that integrates reasoning-augmented data synthesis with supervised fine-tuning and reinforcement learning, aiming to improve the accuracy-efficiency trade-off in code translation.
- We design a task-driven data synthesis method that leverages a stronger language model to generate reasoning-augmented code translation triplets, coupled with automated syntax and functional tests to ensure data quality. The resulting high-quality dataset is EffiReasonTrans-Data.
- We design a dual-objective reward strategy for code translation that considers both execution correctness and output conciseness.
- We conduct extensive experiments to show that EffiReasonTrans effectively improves translation accuracy while reducing latency.

II. Related Work

A. Automated Code Translation

Automated code translation refers to the task of converting source code from one programming language to another, aiming to support software reuse, cross-platform compatibility, and long-term maintainability [1], [2], [3], [4], [5], [6], [7]. Early approaches were dominated by rule-based methods relying on handcrafted grammars and language-specific transformation rules [24], [25]. However, these methods often suffered from low readability and correctness [4], [6]. Therefore, a series of learning-based code transpilers have emerged to address these limitations [5], [26], [27], [28]. For example, some studies adopt unsupervised or weakly supervised strategies, leveraging large-scale monolingual corpora to train models. However, a critical limitation is that their translation performance remains insufficient for real-world deployment. Recently, with the advent of LLMs, automated code translation has gained renewed momentum. Recent studies reveal that LLM-based translation methods are particularly promising [8], [3], [9], [10], [7], [11], [29], [30], [31], [32], [33], [34], [12], [35], [36]. For instance, UniTrans [3] improves translation accuracy by generating test cases and iteratively repairing incorrect outputs, while hmCodeTrans [7] demonstrates the effectiveness of human-in-the-loop collaboration. These methods have shown notable improvements over earlier paradigms, particularly in terms of execution correctness and end-to-end automation.

B. Balancing Accuracy and Latency in Code Translation

While most prior work on automated code translation has improved translation accuracy, LLM-based systems still struggle with complex semantic transformations. Recent approaches have integrated Chain-of-Thought (CoT) prompting strategies into the code translation task [9], [10], encouraging LLMs to reason through multiple intermediate steps before producing the final output. More advanced models, such as DeepSeek-R1 [22], are designed to generate reasoning processes without explicit prompting. Compared to DeepSeek-V3, which produces direct outputs without intermediate reasoning, DeepSeek-R1 achieves higher accuracy (from 86% to 92%). However, this improvement comes at the cost of a substantial 540% increase in inference latency.

These observations reveal a growing tension between reasoning-enhanced translation quality and inference efficiency. However, to the best of our knowledge, no existing framework has been specifically designed to balance the trade-off between accuracy gains and inference latency in the domain of automated code translation.

C. Internalization of CoT

Since the introduction of Chain-of-Thought (CoT) prompting [18], numerous studies have explored ways to reduce the length of reasoning traces or internalize the reasoning process altogether [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51]. These efforts aim to retain the benefits of intermediate reasoning while mitigating the associated inference cost.

For example, Kang et al. [37] proposed C3oT, a CoT compression framework that generates shorter yet informative reasoning traces. By using a conditioned training strategy, the model learns to map long CoTs to their compressed counterparts. In a more radical shift, Hao et al. [40] introduced Coconut, a latent reasoning framework that replaces natural language CoTs with hidden states. Instead, the model performs reasoning directly in latent space, enabling breadth-first exploration and reduced token-level computation. These approaches demonstrate two promising directions: (1) compressing CoT sequences in the language space, and (2) fully internalizing reasoning into latent representations.

Building on these insights, our work explores an alternative internalization strategy tailored for code translation, which balances reasoning quality and inference efficiency through stepwise training.

III. Approach

A. Overview

In this section, we introduce EffiReasonTrans, a training framework for code translation that aims to enhance translation accuracy while balancing inference latency. The overview of EffiReasonTrans is shown in Figure 1, comprising three components: a *data synthesis stage*, followed by a two-stage model training process combining *supervised fine-tuning* and *reinforcement learning*. In the first stage, we construct a high-quality dataset by prompting a stronger LLM DeepSeek-R1 [22] to generate reasoning-augmented translation samples. Each sample includes a triplet of source code, explicit reasoning, and target code, and is filtered through automated syntax and functionality checks to ensure reliability. This execution-based filtering logic aligns with prior work [52], [53], [54], which leverage the correctness of final outputs to validate the quality of the underlying reasoning paths. Based on the synthesized dataset, called EffiReasonTrans-Data, we then perform a two-stage training process: first, the target model is fine-tuned to learn reasoning-aware code translation patterns; second, reinforcement learning is applied to further improve accuracy using reward signals derived from the model generations and the code execution results. This design aims to leverage explicit reasoning to enhance translation accuracy while mitigating the latency overhead typically introduced by longer inference chains.

B. Data Synthesis

The first component of EffiReasonTrans is a data synthesis stage, where we construct a high-quality dataset to support reasoning-aware code translation. This stage consists of two steps: collecting clean source programs with reliable test cases, and generating reasoning-augmented translation data using a reasoning-capable LLM.

1) Collecting Source Programs: We start from a publicly available dataset hosted on Hugging Face[1]. The dataset contains parallel functions implemented in Python, Java, and C++, along with associated test cases. We perform a series of steps to filter out low-quality samples: those that fail to compile or run, or those whose tests do not pass are removed. Additionally, to prevent data leakage, we exclude samples that overlap with the test dataset used in our evaluation. After these filtering steps, we retain a curated set of 180 parallel functions across the three languages. Each function is accompanied by at least ten test cases, with average code coverage exceeding 95% (in many cases reaching 100%). This ensures that the test cases provide sufficient functional validation, which is crucial for later evaluation. Notably, these test cases serve only as ground-truth signals during training; once trained, EffiReasonTrans performs inference independently without requiring any external test cases or execution environments [55], [56].

2) Generating Reasoning Augmented Translations: To generate reasoning-augmented training samples, we use the latest open-source version of DeepSeek-R1 (DeepSeek-R1-0120) [2]—a reasoning-oriented large language model that extends DeepSeek-R1-Zero [22]. DeepSeek-R1 introduces multi-stage training and cold-start data strategies prior to reinforcement learning, which address issues like language mixing and readability. These improvements enable the model to produce more reliable and explicit reasoning chains.

[1] https://huggingface.co/datasets/ziwenyd/transcoder-geeksforgeeks

[2] https://github.com/deepseek-ai/DeepSeek-Coder

TABLE I: Overview of EffiReasonTrans-Data.

Translation Pair	# Samples	# Avg. Tokens
Java → Python	1,258	1311.42
C++ → Java	1,074	1217.08
Python → C++	691	1269.12
Overall	**3,023**	**1,268.24**

For input prompts, we adopt the template as shown in Figure 2, which provides a structured translation instruction along with source code context. When queried with this prompt, DeepSeek-R1 naturally outputs two components: an explicit reasoning sequence describing the translation process, and the final translated target code.

3) Constructing Reasoning Augmented Triplets: Out of approximately 3,400 raw triplets initially generated by DeepSeek-R1, 89% survived our automated execution-based filtering pipeline (comprising syntax and functional checks) in about one hour. After validation, we construct a dataset EffiReasonTrans $\mathcal{D} = \{(C_s, R, C_t)\}$, including 3023 training samples, where each element is a triplet composed of:

- C_s: The source code snippet to be translated, covering diverse data structures and algorithmic patterns. These are selected from the filtered source program set.
- R: The explicit reasoning steps produced by DeepSeek-R1 during translation generation.
- C_t: The final translated code in the target language, functionally equivalent to C_s, and extracted from the answer section of the model's output.

EffiReasonTrans-Data offers supervision for training models to not only generate accurate translations but also to understand the rationale behind the translation process, thus laying a solid foundation for the subsequent training stages. As shown in Table I, it comprises 1,258 Java → Python samples, 1,074 C++ → Java samples, and 691 Python → C++ samples, covering a diverse range of translation scenarios.

C. Supervised Fine-Tuning

In this stage, we fine-tune a reasoning-capable language model to learn how to translate source code into the target language through step-by-step reasoning. The training is conducted on the synthesized dataset EffiReasonTrans-Data, where each triplet includes the source code C_s, the explicit reasoning process R, and the translated code C_t.

To construct each training sample, we design a structured prompt that guides the model to follow a reasoning path before producing the final output. As illustrated in Figure 3, the prompt consists of three parts: (1) task prompt: a task-specific instruction that indicates the translation direction (e.g.,

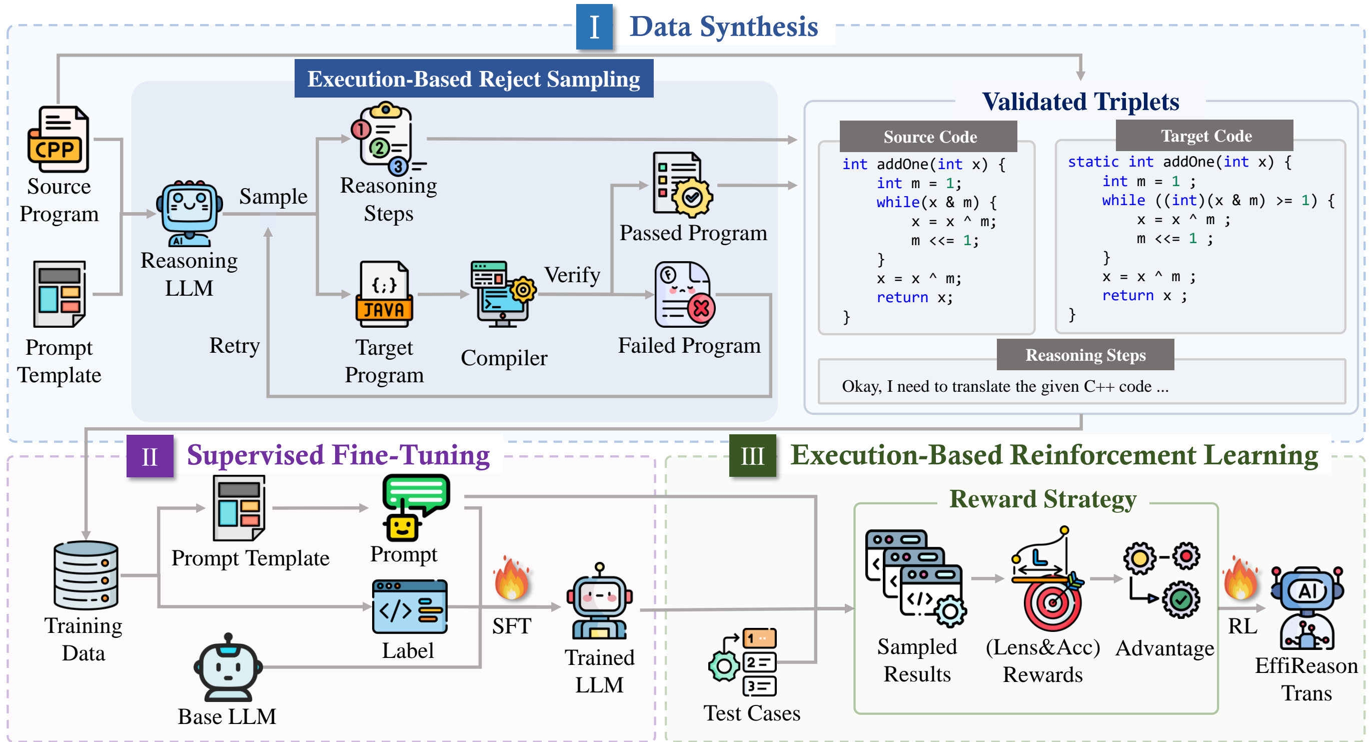


Fig. 1: Overview of EffiReasonTrans.

Prompt Template for Data Synthesis

```
## Role Description
You are a professional developer proficient in Java, Python, and C++.

## Task Description
Given a {Source Language} program:
```{Source Language}
{Program Code}
```
Please translate above {Source Language} program to {Target Language}.
```

Fig. 2: The prompt template for data synthesis.

Prompt Template for Supervised Fine-Tuning

```
## Role Description
You are a professional developer proficient in Java, Python, and C++.

## Task Description
Given a {Source Language} program:
```{Source Language}
{Program Code}
```
Please translate above {Source Language} program to {Target Language}.

Problem Analysis (Internal Process)
{Reasoning Steps}

Final Answer
```

Fig. 3: The prompt template for supervised fine-tuning.

"Translate the above C++ code into Java code") and provides the input code; (2) a problem analysis section that encourages the model to analyze the semantics of the input code and plan its translation; and (3) an answer cue (e.g., "Final Answer:") that prompts the model to begin generating the output. During supervised fine-tuning, although the model's output naturally includes intermediate reasoning steps R followed by the translated code C_t, we apply a masking mechanism to focus the optimization. Specifically, the training labels and the associated cross-entropy loss are computed solely on the tokens of the translated code C_t. The tokens within the reasoning section R are masked out during the loss computation, meaning they do not directly contribute to the gradient updates.

Despite this, we observe that the model continues to exhibit consistent reasoning behavior, which contributes to improved translation quality.

We use DeepSeek-R1-Distill-Qwen-1.5B [22] as the backbone model for fine-tuning. This model is distilled from DeepSeek-R1 [22], inheriting its reasoning ability. We select the 1.5B parameter scale primarily because it significantly reduces computational and memory costs. As shown in later experiments of RQ3, this lightweight model still achieves competitive translation performance compared to models with several times more parameters.

The fine-tuning is implemented using the Hugging Face Transformers library [57] with the `Trainer` API. We optimize the model using the standard cross-entropy loss [58]. To ensure proper supervision, input prompt tokens are masked during loss computation. This stage enables the model to learn reasoning-aware translation patterns in a fully supervised manner and serves as the foundation for subsequent reinforcement learning.

D. Execution-Based Reinforcement Learning

Algorithm 1 Execution-based Reward Strategy

Require: Completions C, Target Language L, Test Cases T
Ensure: Rewards R

```
1:  R ← ∅
2:  for all c in C do
3:      code ← ExtractCode(c, L)
4:      script ← PrepareTestScript(code, T, L)
5:      try
6:          result ← RunTestScript(script, L)
7:          reward ← CountPassedTests(result) / CountTotalTests(result)
8:      catch
9:          reward ← 0
10:     R ← R ∪ {reward}
11: end for
12: return R
```

Algorithm 2 Length-Based Reward Strategy

Require: Completions C, Ground Truth G, Max Length M, Tolerance τ
Ensure: Rewards R

```
1:  R ← ∅
2:  for all (c, g) in (C, G) do
3:      l_c ← Len(c), l_g ← Len(g)
4:      if l_c > M then
5:          reward ← 0
6:      else if |l_c − l_g| / l_g ≤ τ then
7:          reward ← 1 − |l_c − l_g| / (τ · l_g)
8:      else
9:          reward ← 0.1
10:     end if
11:     R ← R ∪ {reward}
12: end for
13: return R
```

In the second stage of training, we further optimize the model using reinforcement learning to enhance translation accuracy and reduce inference latency. The training data remains the same as in the previous stage, consisting of reasoning-augmented triplets $\{(C_s, R, C_t)\}$ that capture the translation process from source code to target code via intermediate reasoning. Following the insights from the DeepSeek-R1 [22], we initialize reinforcement learning from the supervised fine-tuned model, which ensures stable optimization and preserves the reasoning capability acquired during SFT. Then, we adopt the GRPO algorithm [23] for policy optimization, using the `GRPOTrainer` implementation provided by the TRL library [59]. The input prompt format follows the same template used in the SFT stage, which is shown in Figure 3, encouraging the model to generate outputs consisting of reasoning steps followed by final translated code.

1) Execution-based reward: To guide the policy updates, we design an execution-based reward function, shown in Algorithm 1. Specifically, after the model generates a response, we extract the translated code segment. This code is then validated against a set of test cases associated with the input function. For each sample, we calculate the reward as the fraction of test cases passed. For example, if the translated code passes 6 out of 10 test cases, the resulting reward is 0.6. This reward is used to update the model's generation policy via the GRPO algorithm.

2) Length reward: To constrain the verbosity of the model outputs and encourage concise generation, we introduce a length-based auxiliary reward, as shown in Algorithm 2. Here, the "ground truth length" l_g is defined as the total number of tokens in the reference output synthesized in Stage I, encompassing both the explicit reasoning trace and the translated code. By considering the combined length, we assign higher rewards to generations whose total lengths fall within a relative tolerance window (e.g., $\pm 20\%$ of l_g). This design choice is motivated by our goal to achieve holistic efficiency: the model is encouraged not only to internalize and shorten its reasoning path but also to avoid generating redundant code structures. Additional ablation results in the appendix further illustrate the impact of the tolerance hyperparameter τ. Specifically, outputs whose lengths fall outside the tolerance window are assigned a small reward, while outputs exceeding the predefined maximum length receive zero reward, to ensure that the essential logic for functional correctness is preserved.

Overall, by incorporating reward signals derived from actual execution outcomes, the reinforcement learning stage explicitly aligns the model's generation behavior with the ultimate objective of producing semantically and functionally correct translations. In addition to the execution-based reward, we also introduce a length-based auxiliary reward to encourage concise and efficient outputs. This reward penalizes responses that are either excessively long or significantly shorter than the reference solution, while assigning higher rewards to outputs whose lengths fall within an acceptable tolerance window.

IV. EXPERIMENTS

In this section, we evaluate EffiReasonTrans across six translation pairs using the UniTrans-Dataset. We provide the implementation details, dataset specifications, and evaluation metrics used to assess performance in terms of both effectiveness and efficiency.

A. Experimental Setup

1) Implementation Details: We implement EffiReasonTrans with a 2.0×10^{-6} learning rate and a cosine scheduler for 3 epochs (batch size 8, gradient accumulation 4). For GRPO, reward weights are set to 2.0 for execution and 0.5 for length. Decoding uses $T = 0.6$ and $top_p = 0.95$. Training is conducted on an NVIDIA A100-80GB GPU, while evaluation uses an NVIDIA RTX 3090 GPU (both with CUDA 12.1).

2) Dataset: To assess the effectiveness of EffiReasonTrans, we use UniTrans-Dataset[3]. This dataset contains 568 parallel functions implemented in Python, Java, and C++, each paired with corresponding unit tests to support execution-based evaluation. In total, the dataset includes 464 unit tests for Python, 482 for Java, and 467 for C++, depending on the availability of code samples in the source data. All samples

originate from GeeksforGeeks [60], a popular online platform that provides coding problems and solutions across multiple programming languages. We evaluate models across six translation pairs: C++ → Python, C++ → Java, Java → C++, Java → Python, Python → C++, and Python → Java. This diverse set of translation pairs allows us to thoroughly examine the generalizability and robustness of our method across different source-target language combinations.

3) Base Model: To evaluate the generalizability of our method while considering limited experimental resources, we adopt two representative reasoning-capable models as the base models: DeepSeek-R1-Distill-Qwen-1.5B [22] and DeepScaleR-1.5B-Preview [61]. The former is a distilled version of DeepSeek-R1 that retains its explicit reasoning capability while significantly reducing model size and computational cost. The latter is a high-performance reasoning model trained via large-scale reinforcement learning to further optimize problem-solving consistency and accuracy. In our experiments, we compare the vanilla models with versions fine-tuned using our proposed framework, analyzing performance improvements across all translation pairs.

4) Metrics: To comprehensively assess the performance of EffiReasonTrans, we design dual-dimensional evaluation metrics covering *effectiveness* and *efficiency*.

Effectiveness focuses on how well the translated code preserves functional correctness. We assess the effectiveness using the following three metrics.

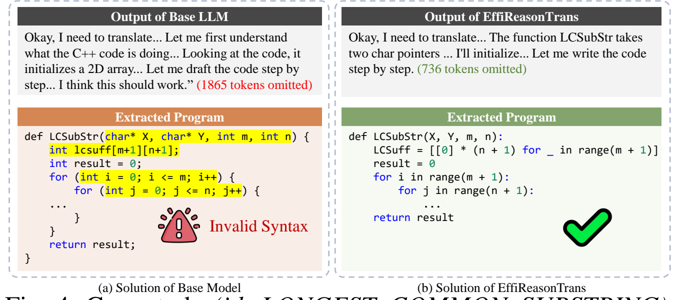
- **Computational Accuracy (CA)**: the proportion of functions whose translated code passes all associated unit tests. This and the following metric are originally proposed by Yang et al. [3].
- **Average Pass Rate (APR)**: the average proportion of passed test cases per function, which reflects the functional correctness of translated code by evaluating how many unit tests are passed on average.
- **CodeBLEU** [62]: an enhanced version of BLEU [63] tailored for code. It incorporates syntax (via abstract syntax trees) and semantics (via data-flow analysis), and has been widely adopted in multiple studies as a standard evaluation metric for code-related tasks [64], [65].

Efficiency is evaluated from two perspectives: the number of generated tokens and the inference latency:

- **Average Number of Generated Tokens (#Tokens)**: We report the average number of generated tokens per sample, denoted as:

$$\#\text{Tokens} = \frac{1}{N}\sum_{i=1}^{N} T_i$$

where T_i is the number of tokens generated for the i-th input and N is the total number of samples. A lower number of generated tokens typically indicates more concise output, which helps reduce memory consumption and decoding time. #Tokens provides a reproducible, hardware-agnostic proxy for generation conciseness and inference cost [66], [67].



(a) Solution of Base Model (b) Solution of EffiReasonTrans

Fig. 4: Case study *(id: LONGEST_COMMON_SUBSTRING)*.

- **Average Inference Latency (Latency)**: We measure the average inference latency (in seconds) across the dataset, calculated as:

$$\text{Latency} = \frac{1}{N}\sum_{i=1}^{N} t_i$$

where t_i is the time taken to generate a complete response for the i-th input. This metric directly reflects the responsiveness of the model, which is crucial for real-time applications and human-in-the-loop development scenarios. Unlike #Tokens, Latency measures end-to-end execution time. It accounts for non-linear factors such as prefilling and framework overhead, which prevent latency from being strictly proportional to token counts [68], [69].

To account for inference non-determinism and system fluctuations, we repeat each evaluation three times and report the average to ensure statistical reliability. When reporting the averaged results, we treat #Tokens and Latency as complementary efficiency metrics: #Tokens reflects algorithmic conciseness and approximate generation cost, while Latency captures the practical runtime behavior in the evaluated implementation. Together, these metrics provide a comprehensive view of translation quality and practical deployability.

B. Evaluation Results

In this section, we conduct experiments to investigate and answer the following research questions (RQs):

- **RQ1**: How effective is EffiReasonTrans in code translation?
- **RQ2**: How does each component impact the performance of EffiReasonTrans?
- **RQ3**: Can a small model trained with EffiReasonTrans rival the performance of a larger model?
- **RQ4**: How does multilingual training data impact model performance?
- **RQ5**: How does EffiReasonTrans generalize to agent-based frameworks?

*1) **Overall Effectiveness (RQ1)**:* To investigate whether EffiReasonTrans can effectively improve code translation accuracy while reducing inference latency, we compare its performance with the base models (e.g., DeepSeek-R1-Distill-Qwen-1.5B) on six translation pairs using UniTrans-Dataset. We evaluate the models on both effectiveness metrics—Computational

TABLE II: Ablation study results across different translation pairs based on the DeepSeek-R1-Distill-Qwen-1.5B model. Superscripts indicate relative improvements of EffiReasonTrans over the Base method.

Translation Pair	Method	CA (%)	APR (%)	CodeBLEU (%)	# Tokens	Latency (s)
Java → Python	Base	56.68	62.84	35.96	1389.48	73.59
	EffiReasonTrans	**72.20**$^{\uparrow 27.4\%}$	**77.31**$^{\uparrow 23.0\%}$	**39.55**$^{\uparrow 10.0\%}$	**1344.03**$^{\downarrow 3.3\%}$	**52.17**$^{\downarrow 29.1\%}$
C++ → Java	Base	38.17	40.74	37.93	1177.90	38.45
	EffiReasonTrans	**47.30**$^{\uparrow 23.9\%}$	**49.39**$^{\uparrow 21.2\%}$	**41.00**$^{\uparrow 8.1\%}$	**1077.24**$^{\downarrow 8.5\%}$	**33.69**$^{\downarrow 12.4\%}$
Python → C++	Base	26.55	28.35	28.98	1494.97	56.54
	EffiReasonTrans	**39.61**$^{\uparrow 49.2\%}$	**42.33**$^{\uparrow 49.2\%}$	**37.05**$^{\uparrow 27.8\%}$	**1205.30**$^{\downarrow 19.3\%}$	**44.06**$^{\downarrow 22.1\%}$
C++ → Python	Base	54.31	59.53	35.28	1483.46	56.15
	EffiReasonTrans	**64.22**$^{\uparrow 18.2\%}$	**70.58**$^{\uparrow 18.6\%}$	**38.59**$^{\uparrow 9.4\%}$	**1231.33**$^{\downarrow 17.0\%}$	**42.17**$^{\downarrow 24.9\%}$
Python → Java	Base	33.20	37.27	36.57	1347.48	44.61
	EffiReasonTrans	**45.85**$^{\uparrow 38.1\%}$	**49.08**$^{\uparrow 31.7\%}$	**40.53**$^{\uparrow 11.0\%}$	**1199.48**$^{\downarrow 11.1\%}$	**34.93**$^{\downarrow 21.6\%}$
Java → C++	Base	41.97	44.56	38.41	1188.10	46.29
	EffiReasonTrans	**50.54**$^{\uparrow 20.4\%}$	**53.04**$^{\uparrow 19.0\%}$	**43.34**$^{\uparrow 12.8\%}$	**967.68**$^{\downarrow 18.5\%}$	**37.31**$^{\downarrow 19.2\%}$

Note: All values are means of 3 independent runs. SDs are within: CA (±.22), APR (±.32), CodeBLEU (±.19), #Tokens (±7.15), Latency (±1.21). All improvements of EffiReasonTrans over Base are statistically significant at $p < 0.001$ (Welch's t-test).

TABLE III: Ablation study results across different translation pairs based on the DeepScaleR-1.5B-Preview model. Superscripts indicate relative improvements of EffiReasonTrans over the Base method.

Translation Pair	Method	CA (%)	APR (%)	CodeBLEU (%)	# Tokens	Latency (s)
Java → Python	Base	47.09	52.11	33.89	1615.90	43.49
	EffiReasonTrans	**70.11**$^{\uparrow 48.9\%}$	**76.10**$^{\uparrow 46.0\%}$	**39.13**$^{\uparrow 15.5\%}$	**1253.78**$^{\downarrow 22.4\%}$	**35.46**$^{\downarrow 18.5\%}$
C++ → Java	Base	41.42	44.04	37.94	1458.46	39.11
	EffiReasonTrans	**63.35**$^{\uparrow 52.9\%}$	**65.74**$^{\uparrow 49.3\%}$	**44.97**$^{\uparrow 18.5\%}$	**948.36**$^{\downarrow 35.0\%}$	**25.50**$^{\downarrow 34.8\%}$
Python → C++	Base	21.91	23.32	27.44	1716.06	46.01
	EffiReasonTrans	**24.69**$^{\uparrow 12.7\%}$	**27.39**$^{\uparrow 17.5\%}$	**33.04**$^{\uparrow 20.4\%}$	**1373.50**$^{\downarrow 20.0\%}$	**31.00**$^{\downarrow 32.6\%}$
C++ → Python	Base	50.07	54.79	33.97	1617.79	42.83
	EffiReasonTrans	**69.26**$^{\uparrow 38.3\%}$	**75.33**$^{\uparrow 37.5\%}$	**38.73**$^{\uparrow 14.0\%}$	**1331.92**$^{\downarrow 17.7\%}$	**38.05**$^{\downarrow 11.2\%}$
Python → Java	Base	34.92	37.03	34.92	1637.36	43.79
	EffiReasonTrans	**54.91**$^{\uparrow 57.2\%}$	**58.35**$^{\uparrow 57.6\%}$	**39.90**$^{\uparrow 14.3\%}$	**1108.83**$^{\downarrow 32.3\%}$	**30.08**$^{\downarrow 31.3\%}$
Java → C++	Base	32.62	35.06	32.23	1497.31	39.89
	EffiReasonTrans	**49.75**$^{\uparrow 52.5\%}$	**51.30**$^{\uparrow 46.3\%}$	**47.25**$^{\uparrow 46.6\%}$	**1161.78**$^{\downarrow 22.4\%}$	**30.70**$^{\downarrow 23.0\%}$

Note: All values are means of 3 independent runs. SDs are within: CA (±.24), APR (±.29), CodeBLEU (±.15), #Tokens (±7.97), Latency (±.92). All improvements of EffiReasonTrans over Base are statistically significant at $p < 0.01$ (Welch's t-test).

Accuracy (CA), Average Pass Rate (APR), CodeBLEU and efficiency metrics—#Tokens and Latency.

Table II and Table III summarize the experimental results. Across all six translation pairs and different base models, EffiReasonTrans consistently outperforms the base model in terms of translation accuracy. Specifically, CA improvements range from 12.7% to 57.2%, with similar trends observed in APR (17.5%–57.6%) and CodeBLEU (8.1%–46.6%), indicating better syntactic and semantic quality of generated code across various architectures.

Importantly, EffiReasonTrans achieves these accuracy improvements while substantially reducing inference latency. The average number of generated tokens decreases ranging from 3.3% to 35.0%. Correspondingly, average inference latency is consistently reduced ranging from 11.2% to 34.8% for all translation pairs under different base model settings.

For example, Figure 4 illustrates a representative case from the C++ → Python translation task. The left side shows the output of the base model. The model directly copies C++ constructs such as "int lcsuff[m+1][n+1];" and "for (int i=0; ...)", leading to syntax errors in Python due to mismatched language paradigms. In contrast, the right side demonstrates the output from EffiReasonTrans, which exhibits better understanding of Python's syntax. It allocates arrays using Pythonic list comprehensions and applies correct "for i in range(...)" loops. Moreover, after applying EffiReasonTrans, the reasoning part in this case becomes more concise (from 1865 tokens to 736 tokens). At the same time, the inference latency is reduced from 70.01 to 33.21, indicating that the simplification of reasoning contributes to lower latency.

Overall, these results demonstrate that EffiReasonTrans successfully balances accuracy and efficiency. By combining RL with SFT on reasoning-augmented data, EffiReasonTrans generates more accurate translations with fewer tokens, thereby reducing inference latency. This trade-off is critical for real-world deployment, where both high accuracy and low latency

are essential.

RQ1 Summary: EffiReasonTrans significantly improves translation accuracy (up to 57.2% CA ↑) while producing more concise outputs (up to 32.3% #Tokens ↓) across all six translation pairs and multiple base models, effectively balancing accuracy and efficiency.

*2) **Component Contribution (RQ2):*** To investigate how the main components (supervised fine-tuning and reinforcement learning) of EffiReasonTrans contribute to its performance, we conduct ablation studies by comparing the following five training settings:

- **Base**: The pretrained base model (DeepSeek-R1-Distill-1.5B) without fine-tuning.
- **SFT-Only**: Base model with only supervised fine-tuning on reasoning-augmented data.
- **RL-Only**: Base model with only reinforcement learning (no SFT).
- **EffiReasonTrans** $^{\text{w/o } R_L}$: The model trained with reinforcement learning but without the length-based auxiliary reward.
- **EffiReasonTrans**: Our full two-stage approach (SFT followed by RL).

All the five training strategies are implemented on the synthesized reasoning-augmented dataset EffiReasonTrans-Data. Based on the experimental results, we have the following observations:

(1) Supervised fine-tuning lays a strong foundation . Across all translation pairs, supervised fine-tuning leads to noticeable improvement over the base setting and RL-Only. For example, in the Java → Python task, SFT-Only improves CA from 56.68% (Base) and 50.72% (RL-Only) to 71.34%, achieving relative gains of 14.7% and 20.6% respectively. Similar trends are observed in other pairs. Compared to the base setting, SFT-Only also helps to decrease the number of generated tokens and inference latency in most cases, such as Python → C++ (#Tokens↓ 6.5%, Latency↓ 19.5%).

(2) Reinforcement learning directly shows limited improvement in accuracy. From the experimental results, we observe that directly applying reinforcement shows limited improvement in accuracy. For some translation pairs, such as Java → Python, RL-Only even decreases from 56.68% to 50.72% in CA. We attribute this to the lack of proper SFT initialization, causing RL to start from a weak policy with a large exploration space and unstable gradients. This often leads to suboptimal policies. Thus, prior SFT provides a stronger initialization, improving both accuracy and convergence speed.

(3) EffiReasonTrans offers improved effectiveness-efficiency trade-offs. The full two-stage approach generally achieves better performance by improving accuracy significantly while generally reducing the number of generated tokens and inference latency. For example, in the Python → C++ task, EffiReasonTrans achieves significant improvements from 26.55% to 39.61% (+49.2%) in CA while decreasing Latency from 56.54s to 44.06s (-22.1%). Similar trends are observed across other translation pairs.

(4) Necessity of Length-based Reward. To isolate the impact of the length reward (R_L), we compare EffiReasonTrans with its variant EffiReasonTrans $^{\text{w/o } R_L}$. As shown in Table IV, while SFT provides a concise logical baseline by imitating the Teacher model, the model tends to generate verbose reasoning traces if RL is guided solely by execution results. For instance, in the C++ → Java task, removing R_L causes the average #Tokens to surge from 1077.24 to 1258.45. This confirms that R_L is indispensable for pruning reasoning redundancy and ensuring optimal inference efficiency during reinforcement learning.

RQ2 Summary: Directly applying reinforcement learning to the base model yields limited improvements. In contrast, supervised fine-tuning plays a crucial role in the two-stage training process. Furthermore, the length-based reward in the reinforcement learning stage is essential for pruning reasoning redundancy and achieving a better accuracy-efficiency trade-off.

*3) **Performance Comparability Between Small-Scale And Larger-Scale Models (RQ3):*** To investigate whether a small-scale model equipped with EffiReasonTrans can achieve performance comparable to a larger-scale model, we compare two models of different sizes: the 8B-parameter DeepSeek-R1-Distill-Llama-8B and the 1.5B-parameter DeepSeek-R1-Distill-Qwen-1.5B [22]. According to the results in DeepSeek-R1 report [22], DeepSeek-R1-Distill-Llama-8B consistently outperforms DeepSeek-R1-Distill-Qwen-1.5B across several benchmarks, including AIME 2024 [70], MATH-500 [22], GPQA Diamond[71], LiveCode [72], CodeForces [73], LiveCodeBench [72].

As shown in Figure 5, our EffiReasonTrans-enhanced 1.5B model (denoted as “1.5B-EffiReasonTrans”) consistently narrows the performance gap with the 8B model across all translation pairs and metrics. Specifically, in the Java → Python task, the 1.5B-EffiReasonTrans model achieves a CA of 72.20%, surpassing the 8B model’s 68.53%, along with higher APR and CodeBLEU scores. A similar trend is observed in the C++ → Java task and the Python → Java task, where the smaller model also outperforms the 8B counterpart. Moreover, the 1.5B-EffiReasonTrans model leads to notable performance gains over the 1.5B-Base model in the remaining three translation tasks, significantly reducing the gap with the 8B model. These improvements are often accompanied by reductions in either the average number of output tokens or inference latency, demonstrating enhanced efficiency in addition to improved translation quality.

RQ3 Summary: EffiReasonTrans enables small models (1.5B) to achieve comparable or superior performance to larger models (8B), with $1.5\text{B}_{\text{EffiReasonTrans}}$ outperforming the 8B model in Java → Python and C++ → Java tasks while reducing latency by 3.2×. This demonstrates that reasoning optimization can effectively compensate for model scale reduction, making high-accuracy translation feasible on resource-constrained devices.

TABLE IV: Ablation study results across different translation pairs. Superscripts indicate relative improvements over Base. $w/o\ R_L$ denotes training without the length-based reward.

Translation Pair	Method	CA (%)	APR (%)	CodeBLEU (%)	#Tokens	Latency (s)
Java → Python	Base	56.68	62.84	35.96	1389.48	73.59
	RL-Only	$50.72^{\downarrow 10.5\%}$	$56.75^{\downarrow 9.7\%}$	$35.48^{\downarrow 1.3\%}$	$1235.15^{\downarrow 11.1\%}$	$72.85^{\downarrow 0.7\%}$
	SFT-Only	$71.34^{\uparrow 25.9\%}$	$76.06^{\uparrow 21.1\%}$	$39.13^{\uparrow 8.8\%}$	$1350.10^{\downarrow 2.8\%}$	$68.59^{\downarrow 6.6\%}$
	EffiReasonTrans $w/o\ R_L$	$69.54^{\uparrow 22.7\%}$	$73.85^{\uparrow 17.6\%}$	$39.22^{\uparrow 9.1\%}$	$1435.12^{\uparrow 3.3\%}$	$54.85^{\downarrow 25.3\%}$
	EffiReasonTrans	$\mathbf{72.20}^{\uparrow 27.4\%}$	$\mathbf{77.31}^{\uparrow 23.0\%}$	$\mathbf{39.55}^{\uparrow 10.0\%}$	$\mathbf{1344.03}^{\downarrow 3.3\%}$	$\mathbf{52.17}^{\downarrow 29.1\%}$
C++ → Java	Base	38.17	40.74	37.93	1177.29	38.45
	RL-Only	$39.49^{\uparrow 3.5\%}$	$41.72^{\uparrow 2.4\%}$	$38.01^{\uparrow 0.3\%}$	$1198.25^{\uparrow 1.8\%}$	$71.25^{\uparrow 85.3\%}$
	SFT-Only	$45.44^{\uparrow 19.0\%}$	$46.70^{\uparrow 14.6\%}$	$41.59^{\uparrow 9.7\%}$	$1028.18^{\downarrow 12.7\%}$	$45.74^{\uparrow 19.2\%}$
	EffiReasonTrans $w/o\ R_L$	$49.93^{\uparrow 30.8\%}$	$51.35^{\uparrow 26.0\%}$	$42.55^{\uparrow 12.3\%}$	$1258.45^{\uparrow 6.9\%}$	$46.25^{\uparrow 20.3\%}$
	EffiReasonTrans	$\mathbf{47.30}^{\uparrow 23.9\%}$	$\mathbf{49.39}^{\uparrow 21.2\%}$	$\mathbf{41.00}^{\uparrow 8.1\%}$	$\mathbf{1077.24}^{\downarrow 8.5\%}$	$\mathbf{33.69}^{\downarrow 12.4\%}$
Python → C++	Base	26.55	28.35	28.98	1494.97	56.54
	RL-Only	$25.34^{\downarrow 4.6\%}$	$27.05^{\downarrow 4.7\%}$	$30.75^{\uparrow 6.1\%}$	$1512.25^{\uparrow 1.2\%}$	$83.05^{\uparrow 46.9\%}$
	SFT-Only	$29.55^{\uparrow 11.3\%}$	$31.50^{\uparrow 11.0\%}$	$29.78^{\uparrow 2.8\%}$	$1397.53^{\downarrow 6.5\%}$	$45.34^{\downarrow 19.5\%}$
	EffiReasonTrans $w/o\ R_L$	$31.91^{\uparrow 20.2\%}$	$38.35^{\uparrow 35.3\%}$	$34.95^{\uparrow 20.6\%}$	$1588.45^{\uparrow 6.3\%}$	$50.15^{\downarrow 11.0\%}$
	EffiReasonTrans	$\mathbf{39.61}^{\uparrow 49.2\%}$	$\mathbf{42.33}^{\uparrow 49.2\%}$	$\mathbf{37.05}^{\uparrow 27.8\%}$	$\mathbf{1205.33}^{\downarrow 19.3\%}$	$\mathbf{44.06}^{\downarrow 22.1\%}$
C++ → Python	Base	54.31	59.53	35.28	1483.46	56.15
	RL-Only	$49.28^{\downarrow 9.3\%}$	$55.72^{\downarrow 6.4\%}$	$34.75^{\downarrow 1.5\%}$	$1505.15^{\uparrow 1.5\%}$	$82.55^{\uparrow 47.0\%}$
	SFT-Only	$63.15^{\uparrow 16.3\%}$	$68.23^{\uparrow 14.6\%}$	$37.91^{\uparrow 7.5\%}$	$1265.65^{\downarrow 14.7\%}$	$44.40^{\downarrow 20.9\%}$
	EffiReasonTrans $w/o\ R_L$	$67.31^{\uparrow 23.9\%}$	$73.65^{\uparrow 23.7\%}$	$38.85^{\uparrow 10.1\%}$	$1442.25^{\downarrow 2.8\%}$	$44.55^{\downarrow 20.6\%}$
	EffiReasonTrans	$\mathbf{64.22}^{\uparrow 18.2\%}$	$\mathbf{70.58}^{\uparrow 18.6\%}$	$\mathbf{38.59}^{\uparrow 9.4\%}$	$\mathbf{1231.33}^{\downarrow 17.0\%}$	$\mathbf{42.17}^{\downarrow 24.9\%}$
Python → Java	Base	33.20	37.27	36.57	1347.48	44.61
	RL-Only	$36.03^{\uparrow 8.5\%}$	$40.35^{\uparrow 8.3\%}$	$36.85^{\uparrow 0.8\%}$	$1368.15^{\uparrow 1.5\%}$	$77.55^{\uparrow 73.8\%}$
	SFT-Only	$42.74^{\uparrow 28.7\%}$	$46.41^{\uparrow 24.5\%}$	$40.08^{\uparrow 9.6\%}$	$1083.93^{\downarrow 19.6\%}$	$38.21^{\downarrow 14.3\%}$
	EffiReasonTrans $w/o\ R_L$	$39.28^{\uparrow 18.3\%}$	$42.52^{\uparrow 14.1\%}$	$38.05^{\uparrow 4.0\%}$	$1402.25^{\uparrow 4.1\%}$	$40.85^{\downarrow 8.4\%}$
	EffiReasonTrans	$\mathbf{45.85}^{\uparrow 38.1\%}$	$\mathbf{49.08}^{\uparrow 31.7\%}$	$\mathbf{40.53}^{\uparrow 10.8\%}$	$\mathbf{1199.48}^{\downarrow 11.0\%}$	$\mathbf{34.93}^{\downarrow 21.7\%}$
Java → C++	Base	41.97	44.56	38.41	1188.10	46.29
	RL-Only	$42.11^{\uparrow 0.3\%}$	$45.32^{\uparrow 1.7\%}$	$37.45^{\downarrow 2.5\%}$	$1235.15^{\uparrow 4.0\%}$	$72.95^{\uparrow 57.6\%}$
	SFT-Only	$44.97^{\uparrow 7.1\%}$	$48.20^{\uparrow 8.2\%}$	$42.16^{\uparrow 9.8\%}$	$985.86^{\downarrow 17.0\%}$	$49.95^{\uparrow 7.9\%}$
	EffiReasonTrans $w/o\ R_L$	$47.25^{\uparrow 12.6\%}$	$47.65^{\uparrow 6.9\%}$	$46.45^{\uparrow 20.9\%}$	$1178.15^{\downarrow 0.8\%}$	$41.25^{\downarrow 10.9\%}$
	EffiReasonTrans	$\mathbf{50.54}^{\uparrow 20.4\%}$	$\mathbf{53.04}^{\uparrow 19.0\%}$	$\mathbf{43.34}^{\uparrow 12.8\%}$	$\mathbf{967.68}^{\downarrow 18.5\%}$	$\mathbf{37.31}^{\downarrow 19.2\%}$

Note: All values are means of 3 independent runs. SDs are within: CA (±.25), APR (±.32), CodeBLEU (±.19), #Tokens (±9.15), Latency (±1.35). The performance differences across the evaluated settings are statistically significant at $p < 0.001$ (Welch's t-test).

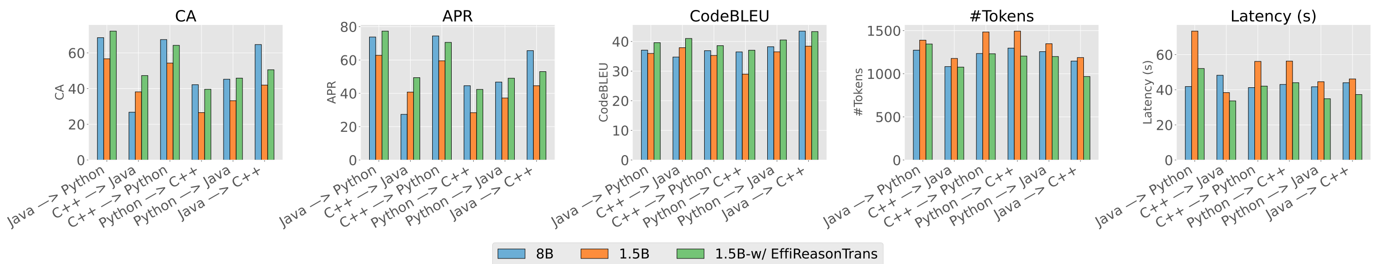


Fig. 5: Performance of EffiReasonTrans-enhanced 1.5B model vs. 8B model across six translation pairs.

*4) **Multilingual Training Impact (RQ4)**:* To investigate the impact of multilingual training data, we compare three variants of our method:

- **Base**: the original base model (DeepSeek-R1-Distill-1.5B) without any fine-tuning.
- **EffiReasonTrans-single**: the model trained by EffiReasonTrans on single-translation-pair data. Specifically, we collect 1,809 samples for the Java-to-Python translation pair based on the data synthesis procedure described earlier.
- **EffiReasonTrans-multi**: the model trained by EffiReasonTrans on multilingual data. We collect 1,809 samples in total, including 603 for Java-to-Python, 603 for C++-to-Java, and 603 for Python-to-C++, all constructed using the same data synthesis approach.

The experimental results are presented in Figure 6. We analyze the outcomes from two perspectives, as detailed below.

In terms of effectiveness, EffiReasonTrans-single exhibits large performance variance across tasks. For instance, EffiReasonTrans-single performs well in Java → Python but

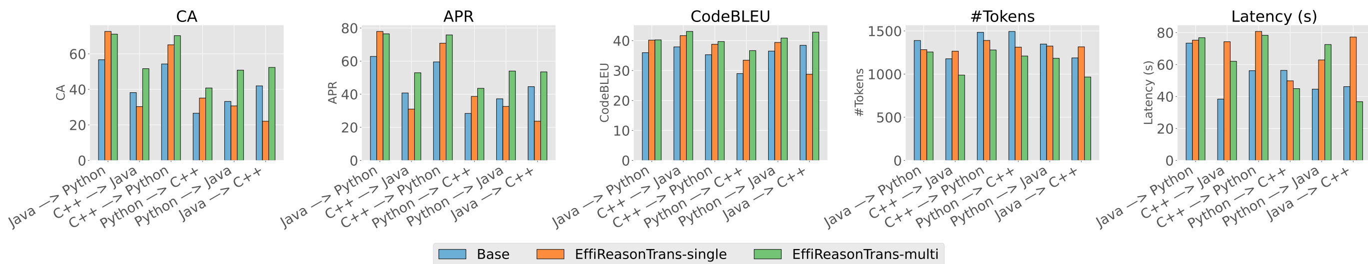


Fig. 6: Performance comparison of different compositions of training data.

poorly in Python → Java (CA drops to 30.71%, even lower than the base model's 33.20%). In contrast, EffiReasonTrans-multi maintains strong performance across all tasks. For instance, in the Python → Java task, EffiReasonTrans-multi achieves 50.83% CA, outperforming both the base model (33.20% CA) and EffiReasonTrans-single (30.71% CA). This suggests that training on a single translation pair lacks generalization ability, while EffiReasonTrans-multi benefits from cross-lingual knowledge and shows robust gains across diverse translation pairs.

In terms of efficiency, EffiReasonTrans-multi reduces the number of generated tokens and inference latency compared to the base model and EffiReasonTrans-single in some cases. For example, in the Java → C++ task, EffiReasonTrans-multi reduces the average number of tokens from 1187.64 (base) to 966.20, and latency from 46.16s to 36.74s. Similarly, in Python → C++, both token length and latency are significantly reduced (1494.25 → 1209.63 tokens; 56.33s → 44.92s).

> **RQ4 Summary:** In summary, training on multilingual data improves both the effectiveness and generalization of the model across diverse translation tasks, while also offering potential gains in inference efficiency.

*5) **Generalization to Agent-based Framework (RQ5)**:* Recent advances in code translation have introduced agent-based frameworks that leverage LLMs to iteratively refine translations based on execution feedback [3], [11]. In this subsection, we adopt the UniTrans framework [3] to assess the effectiveness of EffiReasonTrans in the agent-based setting. Specifically, UniTrans utilizes LLM-generated test cases to guide the translation and subsequent error-fixing process. Following its original configuration, we generate three test cases per example and conduct experiments over two interaction rounds across six translation pairs.

Table V shows the final results of UniTrans, comparing the base model (DeepSeek-R1-Distill-Qwen-1.5B) and our EffiReasonTrans-enhanced model. Based on the experimental results, we summarize the following observations:

(1) Substantial improvements in execution correctness. EffiReasonTrans consistently improves execution-based metrics (CA and APR) across nearly all translation pairs. For instance, in the Java → Python task, CA increases from 55.6% to 73.49% (+32.2%), and APR improves from 63.34% to 79.98% (+26.3%). These results demonstrate that EffiReasonTrans enables the model to generate more functionally correct programs, which is critical in agent-based frameworks where the overall performance largely depends on the deployed LLM.

(2) Mixed impact on efficiency-related metrics. While EffiReasonTrans brings substantial improvements in execution correctness, its performance on efficiency metrics such as #Tokens and Latency is less consistent. Specifically, in the Java → C++ task, EffiReasonTrans produces fewer generated tokens than the base model (1074.42 to 1069.00 #Tokens) and gains improvements of CA (36.53% to 40.04%), indicating that the model produces more concise and effective translations when guided by the agent framework. However, despite the reduced token length, the Latency is still notably higher than the base, possibly due to more computationally intensive reasoning steps induced by EffiReasonTrans. What's more, both #Tokens and Latency show a noticeable increase compared to the base model across most translation pairs. This degradation suggests that the additional round of interaction amplifies the computational burden introduced by EffiReasonTrans, possibly because the model generates more elaborate fixes in response to failed test cases or overfits to verbose reasoning patterns.

> **RQ5 Summary:** EffiReasonTrans improves execution correctness in agent-based frameworks, consistently enhancing Computational Accuracy and Average Pass Rate across translation pairs and rounds. However, it also increases inference latency and output length, particularly in later rounds, revealing a trade-off between accuracy and efficiency in multi-round workflows.

V. Threats to Validity

Despite our efforts to conduct a comprehensive evaluation, several factors may pose threats to the validity of our findings.

One potential threat lies in the reliance on reasoning-augmented data automatically generated during the data synthesis stage. To ensure reliability, we use a stronger language model to generate reasoning outputs and verify them through execution, filtering invalid results. However, the generated reasoning process may still contain hallucinated steps, and guaranteeing the correctness of all reasoning steps remains challenging due to the large amount of textual information. Future work could incorporate stronger verification methods to further improve the reliability of detailed reasoning.

Another potential threat to validity involves the diversity of base models used in our experiments. We evaluate EffiReasonTrans using DeepSeek-R1-Distill-Qwen-1.5B and DeepScaleR-1.5B-Preview, as these models offer strong reasoning capabilities and high computational efficiency, enabling comprehensive evaluations across multiple translation pairs

TABLE V: Performance of base and EffiReasonTrans across different translation pairs in the agent-based setting. Relative changes of EffiReasonTrans are shown as superscript arrows to the right of values.

Translation pair	Method	CA (%)	APR (%)	CodeBLEU (%)	#Tokens	Latency (s)
Java → Python	Base	55.60	63.34	34.58	**973.72**	**36.96**
	EffiReasonTrans	**73.49**$^{\uparrow 32.2\%}$	**79.98**$^{\uparrow 26.3\%}$	**39.82**$^{\uparrow 15.2\%}$	1235.12$^{\uparrow 26.8\%}$	69.98$^{\uparrow 89.3\%}$
C++ → Java	Base	42.36	44.73	36.52	**1030.29**	**46.31**
	EffiReasonTrans	**45.44**$^{\uparrow 6.8\%}$	**48.05**$^{\uparrow 6.9\%}$	**40.77**$^{\uparrow 10.4\%}$	1464.05$^{\uparrow 42.1\%}$	72.00$^{\uparrow 55.4\%}$
Python → C++	Base	21.20	23.40	28.19	**1050.44**	51.69
	EffiReasonTrans	**27.84**$^{\uparrow 31.3\%}$	**30.02**$^{\uparrow 28.3\%}$	**32.22**$^{\uparrow 14.3\%}$	1360.57$^{\uparrow 29.5\%}$	**33.22**$^{\downarrow 35.7\%}$
C++ → Python	Base	54.09	61.12	34.00	**1089.06**	**50.59**
	EffiReasonTrans	**72.63**$^{\uparrow 34.3\%}$	**79.07**$^{\uparrow 29.4\%}$	**39.27**$^{\uparrow 15.5\%}$	1112.02$^{\uparrow 2.1\%}$	68.81$^{\uparrow 36.0\%}$
Python → Java	Base	36.50	39.00	35.70	**1300.00**	**48.00**
	EffiReasonTrans	**45.23**$^{\uparrow 23.9\%}$	**47.82**$^{\uparrow 22.6\%}$	**39.82**$^{\uparrow 11.5\%}$	1458.43$^{\uparrow 12.2\%}$	61.67$^{\uparrow 28.5\%}$
Java → C++	Base	36.53	38.71	34.53	1074.42	**40.80**
	EffiReasonTrans	**40.04**$^{\uparrow 9.6\%}$	**41.88**$^{\uparrow 8.2\%}$	**36.67**$^{\uparrow 6.2\%}$	**1069.00**$^{\downarrow 0.5\%}$	62.52$^{\uparrow 53.2\%}$

within our available resources. However, testing on a broader range of architectures remains necessary to fully establish generalizability. Future work will explore larger-scale models to further assess the robustness of EffiReasonTrans.

Finally, a potential threat is the choice of programming languages used for evaluation. In this work, we focus on three widely-used languages: Python, Java, and C++. These languages are selected due to their high popularity in both academic and industrial settings, which also facilitates comparison with prior work. However, this choice may limit the generalizability of our conclusions to other languages, particularly low-resource programming languages (e.g., ArkTS). Extending the evaluation to a broader set of programming languages is a promising direction for future work and would further verify the robustness of our method in more diverse translation scenarios.

VI. CONCLUSION

In this paper, we propose EffiReasonTrans, a reasoning-enhanced training framework that balances accuracy and efficiency in code translation. It comprises three stages: reasoning-augmented data synthesis, supervised fine-tuning, and reinforcement learning. High-quality (source code, reasoning, target code) triplets are generated by a powerful LLM and filtered via syntax and functional tests. Moreover, a dual-objective reward strategy is introduced to optimize both execution correctness and output conciseness. Experiments on six translation pairs show that EffiReasonTrans consistently improves accuracy while generally reducing inference latency. Ablation and extension studies further highlight the contributions of each component and demonstrate effectiveness in multilingual and agent-based settings, suggesting EffiReasonTrans 's practical value for real-world development workflows.

ACKNOWLEDGEMENTS

This work is supported by the National Natural Science Foundation of China (Grant No. 92582202, No. 62302534, No. 62402113, No. 92582117).

REFERENCES

[1] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *ICSE*, 2010.

[2] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv:2006.03511*, 2020.

[3] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li, "Exploring and unleashing the power of large language models in automated code translation," *PACMSE*, 2024.

[4] F. Liu, J. Li, and L. Zhang, "Syntax and domain aware model for unsupervised program translation," in *ICSE*, 2023.

[5] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *NeurIPS*, 2018.

[6] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code (t)," in *ASE*, 2015.

[7] J. Liu, F. Zhang, X. Zhang, Z. Yu, L. Wang, Y. Zhang, and B. Guo, "hmcodetrans: Human-machine interactive code translation," *TSE*, 2024.

[8] P. Jana, P. Jha, H. Ju, G. Kishore, A. Mahajan, and V. Ganesh, "Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution," in *ECAI 2024*, 2024.

[9] Q. Tao, T. Yu, X. Gu, and B. Shen, "Unraveling the potential of large language models in code translation: How far are we?" *arXiv:2410.09812*, 2024.

[10] Z. Yuan, W. Chen, H. Wang, K. Yu, X. Peng, and Y. Lou, "Transagent: An llm-based multi-agent system for code translation," *arXiv:2409.19894*, 2024.

[11] A. R. Ibrahimzada, K. Ke, M. Pawagi, M. S. Abid, R. Pan, S. Sinha, and R. Jabbarvand, "Alphatrans: A neuro-symbolic compositional approach for repository-level code translation and validation," *arXiv:2410.24117*, 2024.

[12] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, and D. Kroening, "Towards translating real-world code with llms: A study of translating to rust," *arXiv:2405.11514*, 2024.

[13] H. Zhang, C. David, M. Wang, B. Paulsen, and D. Kroening, "Scalable, validated code translation of entire projects using large language models," *PACMPL*, 2025.

[14] X. Yin, C. Ni, T. N. Nguyen, S. Wang, and X. Yang, "Rectifier: Code translation with corrector via llms," *arXiv:2407.07472*, 2024.

[15] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *ICSE*, 2024.

[16] M. Macedo, Y. Tian, P. Nie, F. R. Cogo, and B. Adams, "Intertrans: Leveraging transitive intermediate translations to enhance llm-based code translation," *arXiv:2411.01063*, 2024.

[17] M. Bhattarai, J. E. Santos, S. Jones, A. Biswas, B. Alexandrov, and D. O'Malley, "Enhancing code translation in language models with few-shot learning via retrieval-augmented generation," in *HPEC*, 2024.

[18] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *NeurIPS*, 2022.

[19] N. Ho, L. Schmid, and S.-Y. Yun, "Large language models are reasoning teachers," *arXiv:2212.10071*, 2022.

[20] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, Y. Li, X. Wang, M. Dehghani, S. Brahma *et al.*, "Scaling instruction-finetuned language models," *JMLR*, 2024.

[21] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le *et al.*, "Least-to-most prompting enables complex reasoning in large language models," *arXiv:2205.10625*, 2022.

[22] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv:2501.12948*, 2025.

[23] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu *et al.*, "Deepseekmath: Pushing the limits of mathematical reasoning in open language models," *arXiv:2402.03300*, 2024.

[24] immunant, "c2rust: Migrate c code to rust," https://github.com/immunant/c2rust, n.d., accessed on 2025-07-10.

[25] gotranspile, "cxgo: Tool for transpiling c to go," https://github.com/gotranspile/cxgo, n.d., accessed on 2025-07-10.

[26] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Onward!*, 2014.

[27] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Mapping api elements for code migration with vector representations," in *ICSE*, 2016.

[28] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *ASE*, 2015.

[29] B. Wang, R. Li, M. Li, and P. Saxena, "Transmap: Pinpointing mistakes in neural code translation," in *FSE*, 2023.

[30] J. Pan, A. Sadé, J. Kim, E. Soriano, G. Sole, and S. Flamant, "Stelocoder: a decoder-only llm for multi-language to python code translation," *arXiv:2310.15539*, 2023.

[31] Y. Luo, R. Yu, F. Zhang, L. Liang, and Y. Xiong, "Bridging gaps in llm code translation: Reducing errors with call graphs and bridged debuggers," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024.

[32] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Understanding the effectiveness of large language models in code translation," *CoRR*, 2023.

[33] W. Luo, J. W. Keung, B. Yang, J. Klein, T. F. Bissyande, H. Tian, and B. Le, "Unlocking llm repair capabilities in low-resource programming languages through cross-language translation and multi-agent refinement," *arXiv:2503.22512*, 2025.

[34] M. T. Dearing, Y. Tao, X. Wu, Z. Lan, and V. Taylor, "Lassi: An llm-based automated self-correcting pipeline for translating parallel scientific codes," in *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, 2024.

[35] M. Macedo, Y. Tian, F. Cogo, and B. Adams, "Exploring the impact of the output format on the evaluation of large language models for code translation," in *FORGE*, 2024.

[36] V. Nitin, R. Krishna, and B. Ray, "Spectra: Enhancing the code translation ability of language models by generating multi-modal specifications," *arXiv:2405.18574*, 2024.

[37] Y. Kang, X. Sun, L. Chen, and W. Zou, "C3ot: Generating shorter chain-of-thought without compromising effectiveness," in *AAAI*, 2025.

[38] Y. Deng, Y. Choi, and S. Shieber, "From explicit cot to implicit cot: Learning to internalize cot step by step," *arXiv:2405.14838*, 2024.

[39] Y. Deng, K. Prasad, R. Fernandez, P. Smolensky, V. Chaudhary, and S. Shieber, "Implicit chain of thought reasoning via knowledge distillation," *arXiv:2311.01460*, 2023.

[40] S. Hao, S. Sukhbaatar, D. Su, X. Li, Z. Hu, J. Weston, and Y. Tian, "Training large language models to reason in a continuous latent space," *arXiv:2412.06769*, 2024.

[41] Q. Chen, L. Qin, J. Liu, D. Peng, J. Guan, P. Wang, M. Hu, Y. Zhou, T. Gao, and W. Che, "Towards reasoning era: A survey of long chain-of-thought for reasoning large language models," *arXiv:2503.09567*, 2025.

[42] Y. Sui, Y.-N. Chuang, G. Wang, J. Zhang, T. Zhang, J. Yuan, H. Liu, A. Wen, S. Zhong, H. Chen *et al.*, "Stop overthinking: A survey on efficient reasoning for large language models," *arXiv:2503.16419*, 2025.

[43] T. Han, Z. Wang, C. Fang, S. Zhao, S. Ma, and Z. Chen, "Token-budget-aware llm reasoning," *arXiv:2412.18547*, 2024.

[44] H. Xia, C. T. Leong, W. Wang, Y. Li, and W. Li, "Tokenskip: Controllable chain-of-thought compression in llms," *arXiv:2502.12067*, 2025.

[45] X. Ma, G. Wan, R. Yu, G. Fang, and X. Wang, "Cot-valve: Length-compressible chain-of-thought tuning," *arXiv:2502.09601*, 2025.

[46] S. Xu, W. Xie, L. Zhao, and P. He, "Chain of draft: Thinking faster by writing less," *arXiv:2502.18600*, 2025.

[47] S. A. Aytes, J. Baek, and S. J. Hwang, "Sketch-of-thought: Efficient llm reasoning with adaptive cognitive-inspired sketching," *arXiv:2503.05179*, 2025.

[48] S. Feng, G. Fang, X. Ma, and X. Wang, "Efficient reasoning models: A survey," *arXiv:2504.10903*, 2025.

[49] S. Nayab, G. Rossolini, M. Simoni, A. Saracino, G. Buttazzo, N. Manes, and F. Giacomelli, "Concise thoughts: Impact of output length on llm reasoning and cost," *arXiv:2407.19825*, 2024.

[50] X. Shen, Y. Wang, X. Shi, Y. Wang, P. Zhao, and J. Gu, "Efficient reasoning with hidden thinking," *arXiv:2501.19201*, 2025.

[51] Z. Shen, H. Yan, L. Zhang, Z. Hu, Y. Du, and Y. He, "Codi: Compressing chain-of-thought into continuous space via self-distillation," *arXiv:2502.21074*, 2025.

[52] E. Zelikman, Y. Wu, J. Mu, and N. Goodman, "Star: Bootstrapping reasoning with reasoning," in *NeurIPS*, 2022.

[53] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," 2023. [Online]. Available: https://arxiv.org/abs/2203.11171

[54] H. Lin, Q. Pei, X. Gao, Z. Pan, Y. Li, J. Li, C. He, and L. Wu, "Scaling code-assisted chain-of-thoughts and instructions for model reasoning," 2025. [Online]. Available: https://arxiv.org/abs/2510.04081

[55] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," 2022. [Online]. Available: https://arxiv.org/abs/2207.01780

[56] P. Shojaee, A. Jain, S. Tipirneni, and C. K. Reddy, "Execution-based code generation using deep reinforcement learning," 2023. [Online]. Available: https://arxiv.org/abs/2301.13816

[57] Hugging Face, "Transformers documentation," https://huggingface.co/docs/transformers, 2023.

[58] PyTorch Documentation Team, "torch.nn.CrossEntropyLoss — PyTorch Documentation," 2024. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

[59] Hugging Face, "Grpo trainer — trl," https://huggingface.co/docs/trl/main/grpo_trainer, 2024, accessed: 2025-07-15.

[60] GeeksforGeeks, "GeeksforGeeks," n.d. [Online]. Available: https://www.geeksforgeeks.org/

[61] M. Luo, S. Tan, J. Wong, X. Shi, W. Y. Tang, M. Roongta, C. Cai, J. Luo, L. E. Li, R. A. Popa, and I. Stoica, "Deepscaler: Surpassing o1-preview with a 1.5b model by scaling rl," 2025.

[62] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv:2009.10297*, 2020.

[63] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *ACL*, 2002.

[64] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv:2102.04664*, 2021.

[65] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv:2109.00859*, 2021.

[66] Z. Du, H. Kang, S. Han, T. Krishna, and L. Zhu, "Ockbench: Measuring the efficiency of llm reasoning," 2026. [Online]. Available: https://arxiv.org/abs/2511.05722

[67] Y. Li, Z. Lan, and J. Zhou, "Text or pixels? it takes half: On the token efficiency of visual text inputs in multimodal llms," 2025. [Online]. Available: https://arxiv.org/abs/2510.18279

[68] N. J. Eliopoulos, P. Jajal, J. C. Davis, G. Liu, G. K. Thiravathukal, and Y.-H. Lu, "Pruning one more token is enough: Leveraging latency-workload non-linearities for vision transformers on the edge," 2024. [Online]. Available: https://arxiv.org/abs/2407.05941

[69] NVIDIA, "GPU Performance Background User's Guide," https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html, accessed: 2026-05-06.

[70] Mathematical Association of America, "American Invitational Mathematics Examination (AIME) 2024," https://maa.org/math-competitions/american-invitational-mathematics-examination-aime, 2024.

[71] D. Rein, M. Tworkowski, X. Zhang, T. Khot, O. Tafjord, A. Bosselut, and H. Hajishirzi, "Gpqa: A graduate-level google-proof q&a benchmark," *arXiv:2311.12022*, 2023. [Online]. Available: https://arxiv.org/abs/2311.12022

[72] V. Jain, C. Choquette-Choo, C. White, X. Ma, A. Abid, A. Salehi, Y. Zhu, C. Raffel, and S. Hooker, "Livecodebench: Holistic and contamination-free evaluation of large language models for code," *arXiv:2403.07974*, 2024.

[73] Codeforces Community, "Codeforces Programming Contest Platform," https://codeforces.com/, accessed: July 2025.