

EffiReasonTrans: RL-Optimized Reasoning for Code Translation

Yanlin Wang
wangylin36@mail.sysu.edu.cn
Sun Yat-sen University
Zhuhai, China

Mingwei Liu
liumw26@mail.sysu.edu.cn
Sun Yat-sen University
Zhuhai, China

Xilin Liu
liuxilin3@huawei.com
Huawei Cloud Computing
Technologies Co., Ltd.
China

Rongyi Ou
oury@mail2.sysu.edu.cn
Sun Yat-sen University
Zhuhai, China

Jiachi Chen
chenjch86@mail.sysu.edu.cn
Sun Yat-sen University
Zhuhai, China

Yuchi Ma
mayuchi1@huawei.com
Huawei Cloud Computing
Technologies Co., Ltd.
China

Yanli Wang
wangyli58@mail2.sysu.edu.cn
Sun Yat-sen University
Zhuhai, China

Ensheng Shi
shiensheng@huawei.com
Huawei Cloud Computing
Technologies Co., Ltd.
Beijing, China

Zibin Zheng
zhzibin@mail.sysu.edu.cn
Sun Yat-sen University
Guangzhou, China

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>.

1 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 [8, 40, 43, 45, 58, 95, 108]. Recent studies reveal that automated code translation techniques based on Large Language Models (LLMs) are promising [2, 16, 33, 36, 45, 53, 66, 80, 95–97, 101]. For

example, UniTrans[95] enhances code translation accuracy by generating test cases and iteratively repairing errors using LLMs, while hmCodeTrans[45] 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 [9, 29, 90, 109]. Recent studies have incorporated them into the code translation task to improve performance [80, 97]. More recently, models like DeepSeek-R1 have been designed to generate internal reasoning processes without prompting [23], 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 reasoning model (i.e., DeepSeek-R1) improves translation accuracy from 86% to 92% compared to non-reasoning model (i.e., DeepSeek-V3) on Unitrans-Dataset [95], there is a 540% increase in inference latency. In interactive development scenarios, developers typically expect rapid and reliable feedback, making inference speed a critical factor alongside translation accuracy. This is especially relevant in human-in-the-loop pipelines such as hmCodeTrans [45], where developers participate in reviewing and correcting model-generated translations. 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 balance 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 [23]), filtered via automated syntax validation and functional testing to filter out incorrect or incomplete samples. This process produces **EffiReasonTrans-Data**, a reliable reasoning-augmented code translation corpus that capture 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 [72]), 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.

2 Related Work

2.1 Automated Code Translation

In recent years, large language models (LLMs) have developed rapidly, with increasing application to tasks such as code translation [33, 45, 62, 63, 89, 95, 97, 107], code generation [6, 15, 22, 25, 28, 41, 42, 44, 46, 47, 85, 88, 98, 99, 103, 105, 106], code summarization [75, 76, 84, 86], code search [14, 19, 30, 77, 83, 104], issue resolution [24, 81, 93, 100], vulnerability detection and repair [3–5, 39, 55, 102], etc. Among these, 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 [8, 40, 43, 45, 58, 95, 108]. Early approaches are dominated by rule-based methods, which

relied on handcrafted grammars and language-specific transformation rules [21, 34]. These methods often suffered from low readability and correctness, especially in the face of modern programming paradigms and dynamic typing [43, 58]. Therefore, a series of learning-based code transpilers have emerged to address the limitations of traditional rule-based methods [8, 38, 59, 61]. For example, some studies adopt unsupervised or weakly supervised strategies, leveraging large-scale monolingual corpora to train models. While these approaches have achieved substantial improvements over earlier heuristic-based techniques, several challenges still remain. Specifically, one of the most critical limitation is that their translation performance still falls short of the requirements for real-world deployment. Recently, with the advent of LLMs, which demonstrate powerful generality across a wide range of code-related tasks, automated code translation has gained renewed momentum. Recent studies reveal that LLM-based translation methods are particularly promising [11, 16, 33, 36, 45, 49, 50, 52, 60, 64, 65, 80, 82, 95, 97]. For instance, UniTrans [95] improves translation accuracy by generating test cases and iteratively repairing incorrect outputs, while hmCodeTrans [45] demonstrates the effectiveness of human-in-the-loop collaboration in enhancing translation quality. These methods have shown notable improvements over earlier methods, particularly in terms of execution correctness and their ability to perform end-to-end translation automatically.

2.2 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 [80, 97], encouraging LLMs to reason through multiple intermediate steps before producing the final output. More advanced models, such as DeepSeek-R1 [23], 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.

2.3 Internalization of CoT

Since the introduction of Chain-of-Thought (CoT) prompting [90], numerous studies have explored ways to reduce the length of reasoning traces or internalize the reasoning process altogether [1, 7, 12, 13, 17, 26, 27, 37, 51, 56, 73, 74, 78, 92, 94]. 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, enabling over 50%

reduction in CoT length without compromising performance. In a more radical shift, Hao et al. [27] introduced Coconut, a latent reasoning framework that eliminates natural language CoTs altogether. Instead, the model performs reasoning directly in latent space using internal hidden states as “continuous thoughts,” which are recursively fed back into the model. This paradigm enables breadth-first exploration of reasoning paths and reduces token-level computation, showing superior performance on tasks requiring planning and backtracking. These approaches demonstrate two promising directions for reducing the cost of reasoning: (1) compressing CoT sequences in the language space, and (2) fully internalizing reasoning into latent representations. Both directions provide insights into building efficient, high-performing LLMs suitable for latency-sensitive applications.

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.

3 Approach

3.1 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 [23] 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. 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.

3.2 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.

3.2.1 Collecting Source Programs. We start from a publicly available dataset hosted on Hugging Face¹. 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

¹<https://huggingface.co/datasets/ziwenyid/transcoder-geeksforgEEKS>

Table 1: 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

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.

3.2.2 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 [23]. 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.

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.

3.2.3 Constructing Reasoning Augmented Triplets. After validation, we construct a dataset EffiReasonTrans $\mathcal{D} = \{(C_s, R, C_t)\}$, including 3032 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 1, it comprises 1,258 Java → Python samples, 1,074 C++ → Java samples, and 691 Python → C++ samples, covering a diverse range of translation scenarios.

3.3 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

²<https://github.com/deepseek-ai/DeepSeek-Coder>

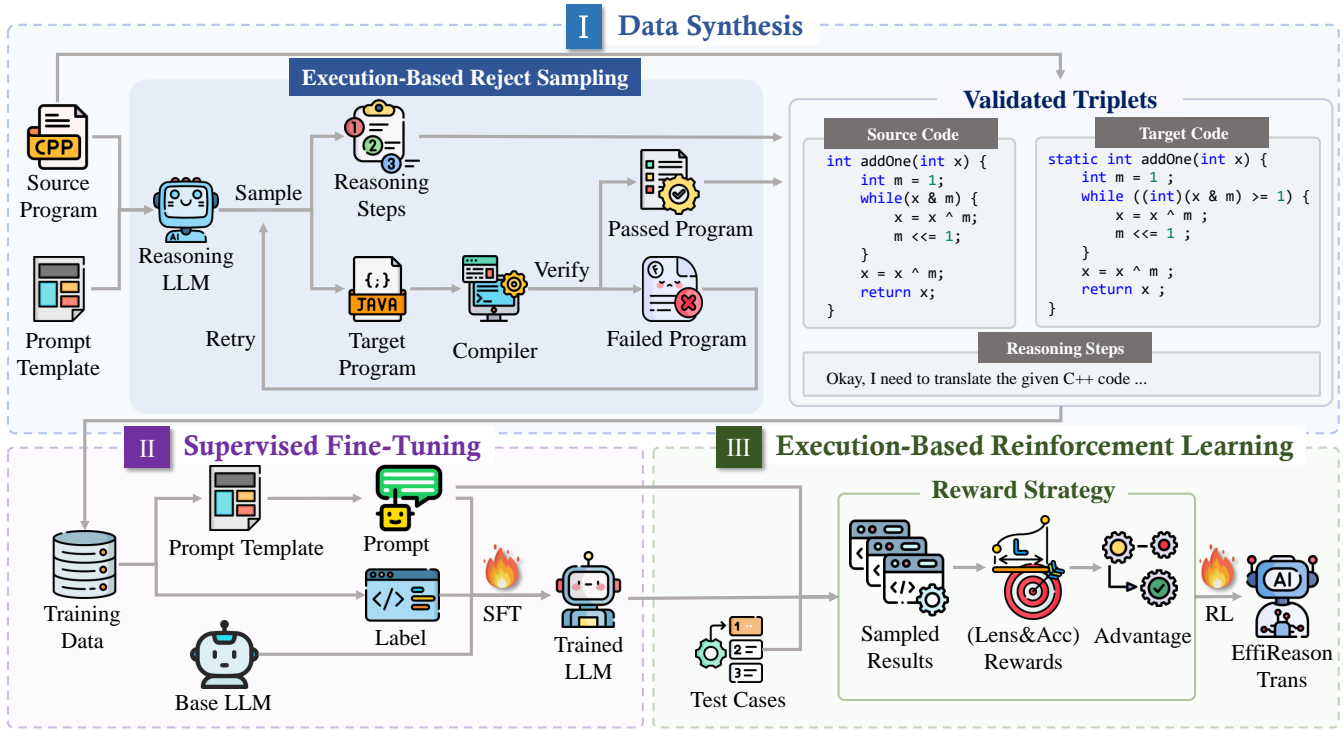


Figure 1: Overview of EffiReasonTrans.

```

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}.
    
```

Figure 2: The prompt template for data synthesis.

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., “Translate the above C++ code into Java code”) and provides the input code; (2) an 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, the training labels consist solely of the translated code C_t , while the chosen model, due to its inherent reasoning capability, naturally generates intermediate reasoning steps R as part of its output. Since these reasoning steps

```

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
    
```

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

are not present in the training labels, they are treated as unconstrained outputs and are included in the decoding process without direct supervision. 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 [23] as the backbone model for fine-tuning. This model is distilled from DeepSeek-R1 [23], 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 [31] with the Trainer API. We use the standard cross-entropy loss over the entire target sequence, which includes both the reasoning steps and the translated code. 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.

3.4 Execution-Based Reinforcement Learning

Algorithm 1 Execution-based Reward Strategy

Require: Completions C , Target Language L , Test Cases T

Ensure: Rewards R

```

1:  $R \leftarrow \emptyset$ 
2: for all  $c$  in  $C$  do
3:    $code \leftarrow \text{ExtractCode}(c, L)$ 
4:    $script \leftarrow \text{PrepareTestScript}(code, T, L)$ 
5:   try
6:      $result \leftarrow \text{RunTestScript}(script, L)$ 
7:      $reward \leftarrow \frac{\text{CountPassedTests}(result)}{\text{CountTotalTests}(result)}$ 
8:   catch
9:      $reward \leftarrow 0$ 
10:   $R \leftarrow R \cup \{reward\}$ 
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:  $R \leftarrow \emptyset$ 
2: for all  $(c, g)$  in  $(C, G)$  do
3:    $l_c \leftarrow \text{Len}(c), l_g \leftarrow \text{Len}(g)$ 
4:   if  $l_c < l_g$  or  $l_c > M$  then
5:      $reward \leftarrow 0$ 
6:   else if  $\frac{l_c - l_g}{l_g} \leq \tau$  then
7:      $reward \leftarrow 1 - \frac{l_c - l_g}{\tau \cdot l_g}$ 
8:   else
9:      $reward \leftarrow 0.1$ 
10:  end if
11:   $R \leftarrow R \cup \{reward\}$ 
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 [23], we initialize reinforcement learning from the supervised fine-tuned model, which ensuring stable optimization and preserves the reasoning capability acquired during SFT. Then, we adopt the GRPO algorithm [72] for policy optimization, using the GRPOTrainer implementation provided by the trl library [32]. The input prompt format follows the same template used in the SFT stage, which showed in Figure 3, encouraging the model to generate outputs consisting of reasoning steps followed by final translated code.

3.4.1 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.

3.4.2 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. Given the reference solution length, we assign higher rewards to generations whose lengths fall within a relative tolerance window (e.g., $\pm 20\%$ of the ground truth length). Specifically, outputs that are significantly shorter than the reference or exceed a predefined maximum length are assigned zero reward.

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.

4 EXPERIMENTS

In this section, we provide a detailed overview of the experimental setup. We begin by describing the computational environment, followed by an introduction to the evaluation dataset. Next, we present the baseline models used for comparison and outline the evaluation metrics employed to comprehensively assess model performance across various aspects of the code translation task.

4.1 Experimental Setup

4.1.1 Computational Environment. For training, we use a Linux server running Ubuntu 20.04, equipped with an NVIDIA A100-SXM4-80GB GPU and CUDA 12.1. For evaluation, a separate Ubuntu 20.04 server with an NVIDIA RTX 3090 GPU (24GB VRAM) and CUDA 12.1 is used.

4.1.2 Dataset. To assess the effectiveness of EffiReasonTrans, we use Unitrans-Dataset[95]. 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++. All samples originate from GeeksforGeeks [18], a popular online platform that provides coding problems and solutions across multiple programming languages. We evaluate models across six translation pairs: C++ \rightarrow Python, C++ \rightarrow Java, Java \rightarrow C++, Java \rightarrow Python, Python \rightarrow C++, and Python \rightarrow 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.

4.1.3 Base Model. As the base model, we adopt DeepSeek-R1-Distill-Qwen-1.5B [23], a distilled version of DeepSeek-R1 that retains its explicit reasoning capability while significantly reducing model size and computational cost. In our experiments, we compare the vanilla model with versions fine-tuned using our proposed framework, analyzing performance improvements across all translation pairs.

4.1.4 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. [95].
- **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** [69]: an enhanced version of BLEU [67] 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 [48, 87].

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:

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

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.

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

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

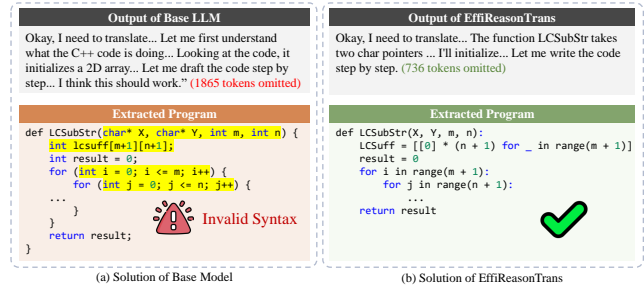


Figure 4: Case study (*id: LONGEST_COMMON_SUBSTRING*).

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.

Together, these metrics provide a comprehensive view of translation quality and practical deployability.

4.2 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?

4.2.1 Overall Effectiveness (RQ1). To investigate whether EffiReasonTrans can effectively improve code translation accuracy while reducing inference latency, we compare its performance with the base model (DeepSeek-R1-Distill-Qwen-1.5B) on six translation pairs using Unitrans-Dataset. We evaluate the models on both effectiveness metrics—Computational Accuracy (CA), Average Pass Rate (APR), CodeBLEU and efficiency metrics—#Tokens and Latency.

Table 2 summarizes the experimental results. Across all six translation pairs, EffiReasonTrans consistently outperforms the base model in terms of translation accuracy. Specifically, CA improvements range from 18.2% to 49.2%, with similar trends observed in APR (18.6%–49.2%) and CodeBLEU (8.2%–27.8%), indicating better syntactic and semantic quality of generated code.

Importantly, EffiReasonTrans achieves these accuracy improvements while substantially reducing inference latency. The average number of generated tokens decreases ranging from 3.2% to 19.3%, which directly contributes to lower latency. Correspondingly, average inference latency is consistently reduced (ranging from 12.3% to 29.0%) for all translation pairs.

For example, Figure 4 illustrates a representative case from the C++ \rightarrow Python translation task. The left side shows the output of

Table 2: Performance comparison across different translation pairs. Superscripts indicate relative improvements of EffiReasonTrans over the Base model.

| Translation Pair | Method | CA (%) | APR (%) | CodeBLEU (%) | # Tokens | Latency (s) |
|------------------|-----------------|--------------------------------|--------------------------------|--------------------------------|----------------------------------|--------------------------------|
| Java → Python | Base | 56.68 | 62.82 | 35.96 | 1389.08 | 73.40 |
| | EffiReasonTrans | 72.20 ^{↑27.4%} | 77.35 ^{↑23.1%} | 39.59 ^{↑10.1%} | 1344.82 ^{↓3.2%} | 52.10 ^{↓29.0%} |
| C++ → Java | Base | 38.17 | 40.75 | 37.90 | 1177.29 | 38.37 |
| | EffiReasonTrans | 47.30 ^{↑23.9%} | 49.42 ^{↑21.3%} | 41.01 ^{↑8.2%} | 1077.05 ^{↓8.5%} | 33.66 ^{↓12.3%} |
| Python → C++ | Base | 26.55 | 28.39 | 28.98 | 1494.25 | 56.33 |
| | EffiReasonTrans | 39.61 ^{↑49.2%} | 42.36 ^{↑49.2%} | 37.04 ^{↑27.8%} | 1205.24 ^{↓19.3%} | 44.02 ^{↓21.9%} |
| C++ → Python | Base | 54.31 | 59.53 | 35.28 | 1483.81 | 56.14 |
| | EffiReasonTrans | 64.22 ^{↑18.2%} | 70.58 ^{↑18.6%} | 38.59 ^{↑9.4%} | 1231.41 ^{↓17.0%} | 42.07 ^{↓25.1%} |
| Python → Java | Base | 33.20 | 37.22 | 36.50 | 1347.86 | 44.57 |
| | EffiReasonTrans | 45.85 ^{↑38.1%} | 49.07 ^{↑31.8%} | 40.51 ^{↑11.0%} | 1199.68 ^{↓11.0%} | 34.89 ^{↓21.7%} |
| Java → C++ | Base | 41.97 | 44.56 | 38.42 | 1187.64 | 46.16 |
| | EffiReasonTrans | 50.54 ^{↑20.4%} | 53.04 ^{↑19.0%} | 43.34 ^{↑12.8%} | 967.63 ^{↓18.5%} | 37.27 ^{↓19.3%} |

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 the trade-off between accuracy and efficiency. By integrating reinforcement learning with supervised fine-tuning on reasoning-augmented data, EffiReasonTrans generates more accurate translations while producing fewer tokens, thereby reducing inference latency. This balance is critical for deploying code translation models in real-world development environments, where high accuracy and low latency are both essential.

RQ1 Summary: EffiReasonTrans significantly improves translation accuracy (18.2% ~ 49.2% CA ↑) while reducing inference latency (12.3% ~ 29.0% time ↓) across all six translation pairs, effectively balancing accuracy and efficiency.

4.2.2 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 four 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:** Our full two-stage approach (SFT followed by RL).

All the three training strategies are implemented on the synthesized reasoning augmented dataset EffiReasonTrans-Data. Based on the experimental results shown in Table 3, 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 50.86% (Base) and 56.68% (RL-Only) to 71.34%, achieving relative gains of 40.0% and 25.9% respectively. Similar trends are observed in other pairs. Compared to the base setting, SFT-Only also help 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 decrease from 56.68% to 50.86% in CA. We speculate that this phenomenon can be attributed to the absence of a proper supervised fine-tuning (SFT) initialization, causing the reinforcement learning process to start from a relatively random or weak policy. This leads to a large exploration space and unstable gradient updates, which often result in the model converging to suboptimal policies or local minima. Therefore, incorporating an SFT stage prior to RL provides a stronger initialization, ultimately improving both the overall accuracy and the 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 decreases Latency from 56.33s to 44.02s (-21.9%). Similar trends are observed across other translation pairs.

Table 3: Ablation study results across different translation pairs. Superscripts indicate relative improvements over Base.

| Translation Pair | Method | CA (%) | APR (%) | CodeBLEU (%) | #Tokens | Latency (s) |
|------------------|------------------------|--------------------------------|--------------------------------|--------------------------------|----------------------------------|--------------------------------|
| Java → Python | Base | 56.68 | 62.82 | 35.96 | 1389.08 | 73.40 |
| | RL-Only | 50.86 ^{↓10.3%} | 56.90 ^{↓9.4%} | 35.55 ^{↓1.1%} | 1230.52 ^{↓11.4%} | 72.36 ^{↓1.4%} |
| | SFT-Only | 71.34 ^{↑25.9%} | 76.06 ^{↑21.1%} | 39.13 ^{↑8.8%} | 1350.10 ^{↓2.8%} | 68.59 ^{↓6.6%} |
| | EffiReasonTrans | 72.20 ^{↑27.4%} | 77.35 ^{↑23.1%} | 39.59 ^{↑10.1%} | 1344.82 ^{↓3.2%} | 52.10 ^{↓29.0%} |
| C++ → Java | Base | 38.17 | 40.75 | 37.90 | 1177.29 | 38.37 |
| | RL-Only | 39.42 ^{↑3.3%} | 41.64 ^{↑2.2%} | 37.96 ^{↑0.2%} | 1194.64 ^{↑1.5%} | 70.56 ^{↑83.9%} |
| | SFT-Only | 45.44 ^{↑19.0%} | 46.70 ^{↑14.6%} | 41.59 ^{↑9.7%} | 1028.18 ^{↓12.7%} | 45.74 ^{↑19.2%} |
| | EffiReasonTrans | 47.30 ^{↑23.9%} | 49.42 ^{↑21.3%} | 41.01 ^{↑8.2%} | 1077.05 ^{↓8.5%} | 33.66 ^{↓12.3%} |
| Python → C++ | Base | 26.55 | 28.39 | 28.98 | 1494.25 | 56.33 |
| | RL-Only | 25.48 ^{↓4.0%} | 27.17 ^{↓4.3%} | 30.89 ^{↓6.6%} | 1509.31 ^{↑1.0%} | 82.48 ^{↑46.4%} |
| | SFT-Only | 29.55 ^{↑11.3%} | 31.50 ^{↑11.0%} | 29.78 ^{↑2.8%} | 1397.53 ^{↓6.5%} | 45.34 ^{↑19.5%} |
| | EffiReasonTrans | 39.61 ^{↑49.2%} | 42.36 ^{↑49.2%} | 37.04 ^{↑27.8%} | 1205.24 ^{↓19.3%} | 44.02 ^{↓21.9%} |
| C++ → Python | Base | 54.31 | 59.53 | 35.28 | 1483.81 | 56.14 |
| | RL-Only | 49.14 ^{↓9.5%} | 55.56 ^{↓6.7%} | 34.61 ^{↓1.9%} | 1499.25 ^{↑1.0%} | 81.95 ^{↑46.0%} |
| | SFT-Only | 63.15 ^{↑16.3%} | 68.23 ^{↑14.6%} | 37.91 ^{↑7.5%} | 1265.65 ^{↓14.7%} | 44.40 ^{↓20.9%} |
| | EffiReasonTrans | 64.22 ^{↑8.2%} | 70.58 ^{↑18.6%} | 38.59 ^{↑9.4%} | 1231.41 ^{↓17.0%} | 42.07 ^{↓25.1%} |
| Python → Java | Base | 33.20 | 37.22 | 36.50 | 1347.86 | 44.57 |
| | RL-Only | 35.89 ^{↑8.1%} | 40.21 ^{↑8.0%} | 36.75 ^{↑0.7%} | 1363.22 ^{↑1.1%} | 76.99 ^{↑72.7%} |
| | SFT-Only | 42.74 ^{↑28.7%} | 46.41 ^{↑24.7%} | 40.08 ^{↑9.8%} | 1083.93 ^{↓19.6%} | 38.21 ^{↓14.3%} |
| | EffiReasonTrans | 45.85 ^{↑38.1%} | 49.07 ^{↑31.8%} | 40.51 ^{↑11.0%} | 1199.68 ^{↓11.0%} | 34.89 ^{↓21.7%} |
| Java → C++ | Base | 41.97 | 44.56 | 38.42 | 1187.64 | 46.16 |
| | RL-Only | 41.97 ^{±0.0%} | 45.18 ^{↑1.4%} | 37.36 ^{↓2.8%} | 1230.71 ^{↑3.6%} | 72.36 ^{↑56.8%} |
| | SFT-Only | 44.97 ^{↑7.1%} | 48.20 ^{↑8.2%} | 42.16 ^{↑9.7%} | 985.86 ^{↓17.0%} | 49.95 ^{↑8.2%} |
| | EffiReasonTrans | 50.54 ^{↑20.4%} | 53.04 ^{↑19.0%} | 43.34 ^{↑12.8%} | 967.63 ^{↓18.5%} | 37.27 ^{↓19.3%} |

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. After supervised fine-tuning, reinforcement learning further refines and enhances the model’s behavior.

4.2.3 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 [23]. According to the results in DeepSeek-R1 report [23], DeepSeek-R1-Distill-Llama-8B consistently outperforms DeepSeek-R1-Distill-Qwen-1.5B across several benchmarks, including AIME 2024 [54], MATH-500 [23], GPQA Diamond[68], LiveCode [35], CodeForces [10], LiveCodeBench [35].

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 enable small models (1.5B) to achieve comparable or superior performance to larger models (8B), with 1.5B_{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.

4.2.4 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 using our proposed EffiReasonTrans on single-translation-pair data. Specifically, we collect 1,089 samples for the Java-to-Python translation pair based on the data synthesis procedure described earlier.
- **EffiReasonTrans-multi:** the model trained using EffiReasonTrans on multilingual data covering multiple translation pairs.

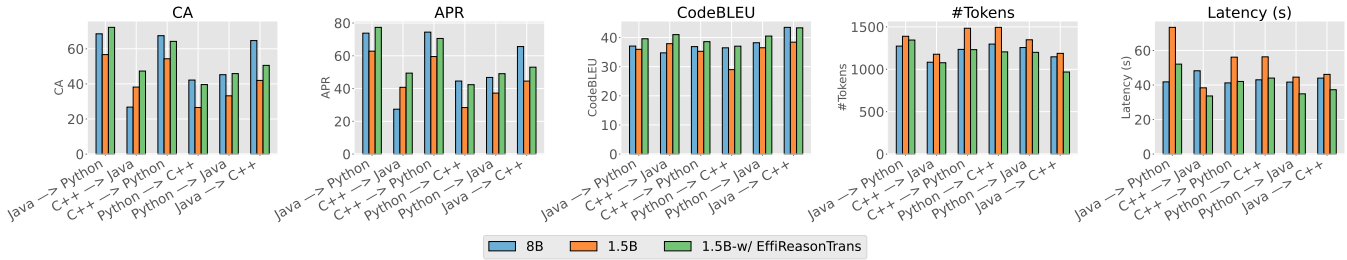


Figure 5: Performance of EffiReasonTrans-enhanced 1.5B model vs. 8B model across six translation pairs.

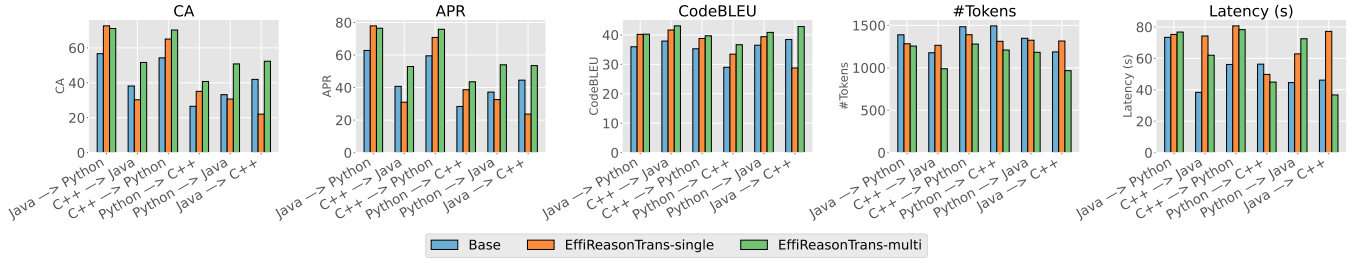


Figure 6: Performance comparison of different compositions of training data.

We collect 1,089 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 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 average 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.

4.2.5 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 [33, 95]. In this subsection, we adopt

the UniTrans framework [95] to assess the effectiveness of EffiReasonTrans in the agent-based setting. UniTrans generates a set of test cases using LLM and incorporates these test cases into a prompt that instructs the LLM to translate the source code into target code aiming to pass the given test cases. Subsequently, through execution feedback, the LLM is guided to fix errors in the translated code. Referring to the original configuration, we generate three test cases per example and conduct experiments over two interaction rounds on six translation pairs.

Table 4 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 achieve smaller number of 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

Table 4: 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 ^{↑32.2%} | 79.98 ^{↑26.3%} | 39.82 ^{↑15.2%} | 1235.12 ^{↑26.8%} | 69.98 ^{↑89.3%} |
| C++ → Java | Base | 42.36 | 44.73 | 36.52 | 1030.29 | 46.31 |
| | EffiReasonTrans | 45.44 ^{↑6.8%} | 48.05 ^{↑6.9%} | 40.77 ^{↑10.4%} | 1464.05 ^{↑42.1%} | 72.00 ^{↑55.4%} |
| Python → C++ | Base | 21.20 | 23.40 | 28.19 | 1050.44 | 51.69 |
| | EffiReasonTrans | 27.84 ^{↑31.3%} | 30.02 ^{↑28.3%} | 32.22 ^{↑14.3%} | 1360.57 ^{↑29.5%} | 33.22 ^{↓35.7%} |
| C++ → Python | Base | 54.09 | 61.12 | 34.00 | 1089.06 | 50.59 |
| | EffiReasonTrans | 72.63 ^{↑34.3%} | 79.07 ^{↑29.4%} | 39.27 ^{↑15.5%} | 1112.02 ^{↑2.1%} | 68.81 ^{↑36.0%} |
| Python → Java | Base | 36.50 | 39.00 | 35.70 | 1300.00 | 48.00 |
| | EffiReasonTrans | 45.23 ^{↑23.9%} | 47.82 ^{↑22.6%} | 39.82 ^{↑11.5%} | 1458.43 ^{↑12.2%} | 61.67 ^{↑28.5%} |
| Java → C++ | Base | 36.53 | 38.71 | 34.53 | 1074.42 | 40.80 |
| | EffiReasonTrans | 40.04 ^{↑9.6%} | 41.88 ^{↑8.2%} | 36.67 ^{↑6.2%} | 1069.00 ^{↓0.5%} | 62.52 ^{↑53.2%} |

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.

Overall, these observations reveal that while EffiReasonTrans improves functional correctness, it may introduce an inference-time cost, particularly in multi-round agent-based workflows. This trade-off highlights the challenge of balancing accuracy and efficiency in practical deployments.

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.

5 Threats to Validity

Despite our efforts to conduct a comprehensive evaluation, several factors may pose threats to the validity of our findings, including the reliability of synthesized data, model selection, and programming language coverage.

One potential threat lies in the reliance on reasoning-augmented data automatically generated during the data synthesis stage. EffiReasonTrans relies on automatically generated reasoning-augmented data in the data synthesis stage to train models. We utilize a stronger language model to generate output with reasoning and verify them based on execution, which aiming to filter the invalid outputs and make sure the reasoning-augmented data more reliable. However, the generated reasoning process may still introduce hallucinated steps and guaranteeing the correctness of all reasoning steps remains challenging, due to the large amount of text information. Future work could incorporate stronger verification method to further detect the deeper reliability of the detailed reasoning.

Another potential threat to the validity is that all experiments are conducted using a single base model, DeepSeek-R1-Distill-Qwen-1.5B. We choose the model for two main reasons. First, it demonstrates reasonable reasoning ability without additional pretraining, making it a meaningful and comparable baseline for evaluating our training framework. Second, experiments involving reinforcement learning and supervised fine-tuning are computationally expensive; focusing on a single lightweight yet capable model allowed us to conduct comprehensive evaluations across multiple translation pairs and perform detailed ablation studies within our available resources. While our current experiments are based on a single model, we acknowledge that a more comprehensive evaluation of our method requires testing it on diverse model architectures. Therefore, we expect future work to explore this direction to more fully assess the effectiveness and generalizability 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 and facilitates comparison with prior work. However, we acknowledge that this choice may limit the generalizability of our conclusions to other languages, particularly those low resource programming languages (e.g., ArkTS). Extending our 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.

In summary, while we have addressed several potential concerns, some aspects such as data quality, reliance on a single base model, and the range of programming languages evaluated remain areas for further exploration. These factors highlight important avenues for future research to enhance the robustness, generalizability, and practical applicability of EffiReasonTrans. We encourage further investigations incorporating stronger verification techniques, diverse model architectures, and a wider variety of programming languages to build upon our findings and drive advancements in reasoning-enhanced code translation.

6 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.

References

- [1] Simon A Aytes, Jinheon Baek, and Sung Ju Hwang. 2025. Sketch-of-thought: Efficient llm reasoning with adaptive cognitive-inspired sketching. *arXiv preprint arXiv:2503.05179* (2025).
- [2] Manish Bhattarai, Javier E Santos, Shawn Jones, Ayan Biswas, Boian Alexandrov, and Daniel O'Malley. 2024. Enhancing code translation in language models with few-shot learning via retrieval-augmented generation. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.
- [3] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Jianxing Yu, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2025. When chatgpt meets smart contract vulnerability detection: How far are we? *ACM Transactions on Software Engineering and Methodology* 34, 4 (2025), 1–30.
- [4] Jiachi Chen, Chong Chen, Jiang Hu, John Grundy, Yanlin Wang, Ting Chen, and Zibin Zheng. 2024. Identifying smart contract security issues in code snippets from stack overflow. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1198–1210.
- [5] Jiachi Chen, Yiming Shen, Jiashuo Zhang, Zihao Li, John Grundy, Zhenzhe Shao, Yanlin Wang, Jiashui Wang, Ting Chen, and Zibin Zheng. 2025. FORGE: An LLM-driven Framework for Large-Scale Smart Contract Vulnerability Dataset Construction. *arXiv preprint arXiv:2506.18795* (2025).
- [6] Jiachi Chen, Qingyuan Zhong, Yanlin Wang, Kaiwen Ning, Yongkun Liu, Zenan Xu, Zhe Zhao, Ting Chen, and Zibin Zheng. 2024. Rmcbench: Benchmarking large language models' resistance to malicious code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 995–1006.
- [7] Qiguang Chen, Libo Qin, Jinhao Liu, Dengyun Peng, Jiannan Guan, Peng Wang, Mengkang Hu, Yuhang Zhou, Te Gao, and Wanxiang Che. 2025. Towards reasoning era: A survey of long chain-of-thought for reasoning large language models. *arXiv preprint arXiv:2503.09567* (2025).
- [8] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems* 31 (2018).
- [9] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2024. Scaling instruction-finetuned language models. *Journal of Machine Learning Research* 25, 70 (2024), 1–53.
- [10] Codeforces Community. [n. d.]. Codeforces Programming Contest Platform. <https://codeforces.com/>. Accessed: July 2025.
- [11] Matthew T Dearing, Yiheng Tao, Xingfu Wu, Zhiling Lan, and Valerie Taylor. 2024. Lassi: An llm-based automated self-correcting pipeline for translating parallel scientific codes. In *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*. IEEE, 136–143.
- [12] Yuntian Deng, Yejin Choi, and Stuart Shieber. 2024. From explicit cot to implicit cot: Learning to internalize cot step by step. *arXiv preprint arXiv:2405.14838* (2024).
- [13] Yuntian Deng, Kiran Prasad, Roland Fernandez, Paul Smolensky, Vishrav Chaudhary, and Stuart Shieber. 2023. Implicit chain of thought reasoning via knowledge distillation. *arXiv preprint arXiv:2311.01460* (2023).
- [14] Hande Dong, Jiayi Lin, Yanlin Wang, Yichong Leng, Jiawei Chen, and Yutao Xie. 2024. Improving Code Search with Hard Negative Sampling Based on Fine-tuning. In *2024 31st Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 221–230.
- [15] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [16] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards translating real-world code with llms: A study of translating to rust. *arXiv preprint arXiv:2405.11514* (2024).
- [17] Sicheng Feng, Gongfan Fang, Xinyin Ma, and Xinchao Wang. 2025. Efficient reasoning models: A survey. *arXiv preprint arXiv:2504.10903* (2025).
- [18] GeeksforGeeks. n.d.. GeeksforGeeks. <https://www.geeksforgeeks.org/>. Accessed on 05/06/2024.
- [19] Jing Gong, Yanghui Wu, Linxi Liang, Zibin Zheng, and Yanlin Wang. 2024. CoSQA+: Enhancing code search dataset with matching code. *arXiv e-prints* (2024), arXiv–2406.
- [20] Linyuan Gong, Jiayi Wang, and Alvin Cheung. 2023. ADELTA: Transpilation between deep learning frameworks. *arXiv preprint arXiv:2303.03593* (2023).
- [21] gotranspile. n.d.. cxgo: Tool for transpiling C to Go. <https://github.com/gotranspile/cxgo>. Accessed on 2025-07-10.
- [22] Wenchao Gu, Juntao Chen, Yanlin Wang, Tianyue Jiang, Xingzhe Li, Mingwei Liu, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. What to Retrieve for Effective Retrieval-Augmented Code Generation? An Empirical Study and Beyond. *arXiv preprint arXiv:2503.20589* (2025).
- [23] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).
- [24] Lianghong Guo, Wei Tao, Runhan Jiang, Yanlin Wang, Jiachi Chen, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2025. Omnigirl: A multilingual and multimodal benchmark for github issue resolution. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 24–46.
- [25] Lianghong Guo, Yanlin Wang, Ensheng Shi, Wanjun Zhong, Hongyu Zhang, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. When to stop? towards efficient code generation in llms with excess token prevention. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1073–1085.
- [26] Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. 2024. Token-budget-aware llm reasoning. *arXiv preprint arXiv:2412.18547* (2024).
- [27] Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. 2024. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769* (2024).
- [28] Kaifeng He, Mingwei Liu, Chong Wang, Zike Li, Yanlin Wang, Xin Peng, and Zibin Zheng. 2025. AdaDec: Uncertainty-Guided Adaptive Decoding for LLM-based Code Generation. *arXiv preprint arXiv:2506.08980* (2025).
- [29] Namgyu Ho, Laura Schmid, and Se-Young Yun. 2022. Large language models are reasoning teachers. *arXiv preprint arXiv:2212.10071* (2022).
- [30] Fan Hu, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Xirong Li. 2022. Tackling long code search with splitting, encoding, and aggregating. *arXiv preprint arXiv:2208.11271* (2022).
- [31] Hugging Face. 2023. Transformers Documentation. <https://huggingface.co/docs/transformers>. Accessed: 2025-07-13.
- [32] Hugging Face. 2024. GRPO Trainer – TRL. https://huggingface.co/docs/trl/main/grpo_trainer. Accessed: 2025-07-15.
- [33] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation. *arXiv preprint arXiv:2410.24117* (2024).
- [34] immunant. n.d.. c2rust: Migrate C code to Rust. <https://github.com/immunant/c2rust>. Accessed on 2025-07-10.
- [35] Varun Jain, Caleb Choquette-Choo, Colin White, Xueqian Ma, Anish Abid, Aida Salehi, Yujia Zhu, Colin Raffel, and Sara Hooker. 2024. LiveCodeBench: Holistic and contamination-free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974* (2024). doi:10.48550/arXiv.2403.07974
- [36] Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2024. Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution. In *ECAI 2024*. IOS Press, 4011–4018.
- [37] Yu Kang, Xianghui Sun, Liangyu Chen, and Wei Zou. 2025. C3ot: Generating shorter chain-of-thought without compromising effectiveness. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 24312–24320.
- [38] Svetoslav Karaiyanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM international symposium on new ideas, new paradigms, and reflections on programming & software*. 173–184.
- [39] Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. 2023. Defiater: Detecting price manipulation vulnerabilities in defi protocols. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1144–1156.
- [40] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint*

- arXiv:2006.03511* (2020).
- [41] Yifan Li, Ensheng Shi, Dewu Zheng, Kefeng Duan, Jiachi Chen, and Yanlin Wang. 2024. Repomincoder: Improving repository-level code generation based on information loss screening. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*. 229–238.
- [42] Linxi Liang, Jing Gong, Mingwei Liu, Chong Wang, Guangsheng Ou, Yanlin Wang, Xin Peng, and Zibin Zheng. 2025. RustEvo²: An Evolving Benchmark for API Evolution in LLM-based Rust Code Generation. *arXiv preprint arXiv:2503.16922* (2025).
- [43] Fang Liu, Jia Li, and Li Zhang. 2023. Syntax and domain aware model for unsupervised program translation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 755–767.
- [44] Junwei Liu, Yixuan Chen, Mingwei Liu, Xin Peng, and Yiling Lou. 2024. Stall+: Boosting llm-based repository-level code completion with static analysis. *arXiv preprint arXiv:2406.10018* (2024).
- [45] Jiaqi Liu, Fengming Zhang, Xin Zhang, Zhiwen Yu, Liang Wang, Yao Zhang, and Bin Guo. 2024. hmCodeTrans: Human-Machine Interactive Code Translation. *IEEE Transactions on Software Engineering* (2024).
- [46] Mingwei Liu, Juntao Li, Ying Wang, Xueying Du, Zuoyu Ou, Qiuyuan Chen, Bingxu An, Zhao Wei, Yong Xu, Fangming Zou, et al. 2025. Code Copycat Conundrum: Demystifying Repetition in LLM-based Code Generation. *arXiv preprint arXiv:2504.12608* (2025).
- [47] Mingwei Liu, Tianyong Yang, Yiling Lou, Xueying Du, Ying Wang, and Xin Peng. 2023. Codegen4libs: A two-stage approach for library-oriented code generation. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 434–445.
- [48] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).
- [49] Wenqiang Luo, Jacky Wai Keung, Boyang Yang, Jacques Klein, Tegawende F Bissyande, Haoye Tian, and Bach Le. 2025. Unlocking LLM Repair Capabilities in Low-Resource Programming Languages Through Cross-Language Translation and Multi-Agent Refinement. *arXiv preprint arXiv:2503.22512* (2025).
- [50] Yang Luo, Richard Yu, Fajun Zhang, Ling Liang, and Yongqiang Xiong. 2024. 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*. 2448–2449.
- [51] Xinyin Ma, Guangnian Wan, Rumpeng Yu, Gongfan Fang, and Xinchao Wang. 2025. Cot-valve: Length-compressible chain-of-thought tuning. *arXiv preprint arXiv:2502.09601* (2025).
- [52] Marcos Macedo, Yuan Tian, Filipe Cogo, and Bram Adams. 2024. Exploring the impact of the output format on the evaluation of large language models for code translation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*. 57–68.
- [53] Marcos Macedo, Yuan Tian, Pengyu Nie, Filipe R Cogo, and Bram Adams. 2024. InterTrans: Leveraging transitive intermediate translations to enhance LLM-based code translation. *arXiv preprint arXiv:2411.01063* (2024).
- [54] Mathematical Association of America. 2024. American Invitational Mathematics Examination (AIME) 2024. <https://maa.org/math-competitions/american-invitational-mathematics-examination-aime>. Accessed: July 2025.
- [55] Yifan Mo, Jiachi Chen, Yanlin Wang, and Zibin Zheng. 2023. Toward automated detecting unanticipated price feed in smart contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1257–1268.
- [56] Sania Nayab, Giulio Rossolini, Marco Simoni, Andrea Saracino, Giorgio Buttazzo, Nicolamaria Manes, and Fabrizio Giacomelli. 2024. Concise thoughts: Impact of output length on llm reasoning and cost. *arXiv preprint arXiv:2407.19825* (2024).
- [57] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 651–654.
- [58] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 585–596.
- [59] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2016. Mapping API elements for code migration with vector representations. In *Proceedings of the 38th international conference on software engineering companion*. 756–758.
- [60] Vikram Nitin, Rahul Krishna, and Baisakhhi Ray. 2024. Spectra: Enhancing the code translation ability of language models by generating multi-modal specifications. *arXiv preprint arXiv:2405.18574* (2024).
- [61] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.
- [62] Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Xueying Du, Shengbo Wang, Zekai Zhang, Xin Peng, and Zibin Zheng. 2025. Enhancing llm-based code translation in repository context via triple knowledge-augmented. *arXiv preprint arXiv:2503.18305* (2025).
- [63] Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Xin Peng, and Zibin Zheng. 2024. Repository-level code translation benchmark targeting rust. *arXiv preprint arXiv:2411.13990* (2024).
- [64] Jialing Pan, Adrien Sadé, Jin Kim, Eric Soriano, Guillem Sole, and Sylvain Flammant. 2023. SteloCoder: a decoder-only LLM for multi-language to Python code translation. *arXiv preprint arXiv:2310.15539* (2023).
- [65] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pougum Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the effectiveness of large language models in code translation. *CoRR* (2023).
- [66] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pougum Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [67] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.
- [68] Dori Rein, Matthew Tworkowski, Xinyun Zhang, Tushar Khot, Oyvind Tafjord, Antoine Bosselut, and Hannaneh Hajishirzi. 2023. GPQA: A graduate-level google-proof Q&A benchmark. *arXiv preprint arXiv:2311.12022* (2023). <https://arxiv.org/abs/2311.12022>
- [69] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).
- [70] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems* 33 (2020), 20601–20611.
- [71] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773* (2021).
- [72] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).
- [73] Xuan Shen, Yizhou Wang, Xiangxi Shi, Yanzhi Wang, Pu Zhao, and Jiuxiang Gu. 2025. Efficient reasoning with hidden thinking. *arXiv preprint arXiv:2501.19201* (2025).
- [74] Zhenyi Shen, Hanqi Yan, Linhai Zhang, Zhanghao Hu, Yali Du, and Yulan He. 2025. Codi: Compressing chain-of-thought into continuous space via self-distillation. *arXiv preprint arXiv:2502.21074* (2025).
- [75] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th international conference on software engineering*. 1597–1608.
- [76] Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. *arXiv preprint arXiv:2108.12987* (2021).
- [77] Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. CoCoSoDa: Effective Contrastive Learning for Code Search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2198–2210. doi:10.1109/ICSE48619.2023.00185
- [78] Yang Sui, Yu-Neng Chuang, Guancho Wang, Jiamu Zhang, Tianyi Zhang, Jiayi Yuan, Hongyi Liu, Andrew Wen, Shaochen Zhong, Hanjie Chen, et al. 2025. Stop overthinking: A survey on efficient reasoning for large language models. *arXiv preprint arXiv:2503.16419* (2025).
- [79] Qiushi Sun, Nuo Chen, Jianing Wang, Xiang Li, and Ming Gao. 2023. TransCoder: Towards unified transferable code representation learning inspired by human skills. *arXiv preprint arXiv:2306.07285* (2023).
- [80] Qingxiao Tao, Tingrui Yu, Xiaodong Gu, and Beijun Shen. 2024. Unraveling the Potential of Large Language Models in Code Translation: How Far Are We? *arXiv preprint arXiv:2410.09812* (2024).
- [81] Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. 2024. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems* 37 (2024), 51963–51993.
- [82] Bo Wang, Ruishi Li, Mingkai Li, and Prateek Saxena. 2023. Transmap: Pinpointing mistakes in neural code translation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 999–1011.
- [83] Yanlin Wang, Lianghong Guo, Ensheng Shi, Wenqing Chen, Jiachi Chen, Wanjuan Zhong, Menghan Wang, Hui Li, Hongyu Zhang, Ziyu Lyu, et al. 2023. You augment me: Exploring chatgpt-based data augmentation for semantic code search. In *2023 IEEE International Conference on Software Maintenance and*

- Evolution (ICSME)*. IEEE, 14–25.
- [84] Yanlin Wang, Yanxian Huang, Daya Guo, Hongyu Zhang, and Zibin Zheng. 2024. Sparsecoder: Identifier-aware sparse transformer for file-level code summarization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 614–625.
- [85] Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, Mingzhi Mao, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. Beyond functional correctness: Investigating coding style inconsistencies in large language models. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 690–712.
- [86] Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Yanli Wang, Daya Guo, Shi Han, Hongyu Zhang, and Dongmei Zhang. 2025. Context-aware code summarization with multi-relational graph neural network. *Automated Software Engineering* 32, 1 (2025), 1–26.
- [87] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
- [88] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487* (2024).
- [89] Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, et al. 2024. Repotransbench: A real-world benchmark for repository-level code translation. *arXiv preprint arXiv:2412.17744* (2024).
- [90] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [91] Justin D Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection not required? Human-AI partnerships in code translation. In *Proceedings of the 26th International Conference on Intelligent User Interfaces*. 402–412.
- [92] Heming Xia, Chak Tou Leong, Wenjie Wang, Yongqi Li, and Wenjie Li. 2025. Tokenskip: Controllable chain-of-thought compression in llms. *arXiv preprint arXiv:2502.12067* (2025).
- [93] Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. *arXiv preprint arXiv:2501.05040* (2025).
- [94] Silei Xu, Wenhao Xie, Lingxiao Zhao, and Pengcheng He. 2025. Chain of draft: Thinking faster by writing less. *arXiv preprint arXiv:2502.18600* (2025).
- [95] 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. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1585–1608.
- [96] Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code translation with corrector via llms. *arXiv preprint arXiv:2407.07472* (2024).
- [97] Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Kai Yu, Xin Peng, and Yiling Lou. 2024. Transagent: An llm-based multi-agent system for code translation. *arXiv preprint arXiv:2409.19894* (2024).
- [98] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240* (2023).
- [99] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726.
- [100] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, et al. 2024. Swe-bench-java: A github issue resolving benchmark for java. *arXiv preprint arXiv:2408.14354* (2024).
- [101] Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. 2025. Scalable, validated code translation of entire projects using large language models. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1616–1641.
- [102] Jiashuo Zhang, Yiming Shen, Jiachi Chen, Jianzhong Su, Yanlin Wang, Ting Chen, Jianbo Gao, and Zhong Chen. 2024. Demystifying and detecting cryptographic defects in ethereum smart contracts. *arXiv preprint arXiv:2408.04939* (2024).
- [103] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 481–503.
- [104] Dewu Zheng, Yanlin Wang, Wenqing Chen, Jiachi Chen, and Zibin Zheng. 2024. CoSTV: Accelerating Code Search with Two-Stage Paradigm and Vector Retrieval. In *2024 31st Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 383–392.
- [105] Dewu Zheng, Yanlin Wang, Ensheng Shi, Hongyu Zhang, and Zibin Zheng. 2024. How well do llms generate code for different application domains? benchmark and evaluation. *arXiv preprint arXiv:2412.18573* (2024).
- [106] Dewu Zheng, Yanlin Wang, Ensheng Shi, Ruikai Zhang, Yuchi Ma, Hongyu Zhang, and Zibin Zheng. 2024. Humanevo: An evolution-aware benchmark for more realistic evaluation of repository-level code generation. *arXiv preprint arXiv:2406.06918* (2024).
- [107] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. 2025. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering* 30, 2 (2025), 50.
- [108] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 195–204.
- [109] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625* (2022).