TRACE: Evaluating Execution Efficiency of LLM-Based Code Translation

Zhihao Gong[1], Zeyu Sun[3], Dong Huang[4], Qingyuan Liang[1], Jie M. Zhang[5], Dan Hao[1,2*]
[1]Key Lab of HCST (PKU), MOE; SCS, Peking University, China
[2]School of Electronic and Computer Engineering, PKU Shenzhen Graduate School, China
[3]Institute of Software, Chinese Academy of Sciences, Beijing, China
[4]National University of Singapore, Singapore
[5]King's College London, London, United Kingdom
{zhihaogong, liangqy}@stu.pku.edu.cn, haodan@pku.edu.cn
zeyu.zys@gmail.com, dong.huang@nus.edu.sg, jie.zhang@kcl.ac.uk

Abstract

While Large Language Models (LLMs) have substantially improved the functional correctness of code translation, the critical dimension of execution efficiency remains overlooked. We present **TRACE**, the first benchmark to explicitly assess efficiency in LLM-translated code. TRACE includes 1,000 efficiency-critical tasks across C++, Java, and Python, each augmented with stress tests that reveal efficiency disparities often overlooked by small-scale tests. Using TRACE, we conduct an extensive evaluation of 28 representative LLMs and highlight several key insights: 1) Correctness and efficiency are often misaligned: the correctness leader Claude-Sonnet-4-Think achieves only moderate time efficiency, outperformed by smaller open-source LLMs such as Qwen2.5-Coder-14B-Instruct. 2) Inefficiency is both prevalent and patterned: 23.5% of correct translations suffer from notable inefficiency, mainly arising from algorithm implementation discrepancy (11.9%), language construct mismatch (66.4%), and resource management inefficiency (21.7%). 3) Inference-time prompt strategies bring only modest improvements, indicating that simple prompting alone is insufficient to improve translation efficiency. Together, our results establish execution efficiency as an essential dimension of code translation and position TRACE as a principled foundation for efficiency-oriented evaluation. Our code and data are available at: https://github.com/Albert-Gong/TRACE.

1 Introduction

Code translation migrates programs across programming languages while preserving functionality. It supports critical tasks such as modernizing legacy systems, extending library interoperability, and consolidating disparate codebases into standardized stacks (Pan et al., 2024). As software grows in complexity, the demand for automatic code translation has become more pressing.

C++ Source Code

```cpp
int gcd(int a, int b) {
    if (a == 0) return b;
    if (b == 0) return a;
    int k;
    for (k = 0; ((a | b) & 1) == 0; ++k) {
        a >>= 1; b >>= 1;
    }
    while ((a & 1) == 0)
        a >>= 1;
    do {
        while ((b & 1) == 0)
            b >>= 1;
        if (a > b)
            swap(a, b);
        b = b - a;
    } while (b != 0);
    return a << k;
}
```

Execution Profile

```
### Small Tests
- Test Input: `37, 93`
- Execution Time: **0.02 s vs. 0.02 s**

### Stress Tests
- Test Input: `2147483647, 2147483629`
- Execution Time: **11.20 s vs. 0.02 s**
```

Python Translation

```python
def gcd(a, b):
    if a == 0: return b
    if b == 0: return a
    k = 0
    while (a | b) & 1 == 0:
        a >>= 1
        b >>= 1
        k += 1
    while a & 1 == 0:
        a >>= 1
    ''' CodeLlama-34B-Instruct-hf
    Inefficiency:
        shift moved outside the loop.
    '''
-   while b & 1 == 0:
-       b >>= 1
-   while b != 0:
-       if a > b:
-           a, b = b, a
-       b = b - a
    ''' GPT-4o-Mini
    '''
+   while b != 0:
+       while b & 1 == 0:
+           b >>= 1
+       if a > b:
+           a, b = b, a
+       b = b - a
    return a << k
```

Figure 1: A motivating example illustrating the critical discrepancy between functional correctness and execution efficiency in LLM-based code translation.

The remarkable proficiency of LLMs in multilingual code understanding and generation (Zhao et al., 2023; Zheng et al., 2023b,a; Jiang et al., 2024) has driven a paradigm shift in code translation. Prior work explores techniques such as back translation (Ahmad et al., 2022), fine-tuning (Zhu et al., 2024), chain-of-thought (Macedo et al., 2024; Nitin et al., 2024), and self-repair (Yang et al., 2024) to further enhance translation. Collectively, these efforts significantly improve the syntactic and functional fidelity of LLM-based code translation.

Despite advances in correctness, an equally critical dimension remains understudied: *execution efficiency* (Niu et al., 2024; Vartziotis et al., 2024). Execution efficiency determines how well a translated program performs, directly influencing responsiveness, scalability, and operational cost. Crucially, even functionally correct code translations must be efficient in practice; otherwise, excessive runtime overhead can make them impractical to use.

Figure 1 illustrates an example of Stein's GCD algorithm (Stein's GCD, 2025), a canonical problem from the widely used TRANSCODER-TEST benchmark (Lachaux et al., 2020). When trans-

*Corresponding author.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 3089–3117
July 2-7, 2026 ©2026 Association for Computational Linguistics

lating the C++ implementation into Python, solutions from CodeLlama-34B-Instruct-hf and GPT-4o-Mini appear indistinguishable under small-scale tests, both passing with nearly identical runtime (0.02 s). However, on a stress-test input ($a = 2147483647$, $b = 2147483629$) from our generated stress tests, their performance diverges sharply: CodeLlama's translation runs over 500× slower (11.20 s vs. 0.02 s). This failure stems from an algorithmic misinterpretation, where CodeLlama moves the bitwise-shift outside the loop, discarding the per-iteration optimization central to Stein's algorithm and thereby nullifying its efficiency.

This example underscores a key challenge in evaluating LLM-based code translation: *functional correctness does not guarantee efficiency equivalence*. As observed in our study, even when the source code embodies a highly optimized implementation, and the translation faithfully preserves functional correctness, LLMs may still introduce algorithmic degradations or adopt suboptimal idioms, incurring substantial runtime overheads. However, evaluation protocols such as the small-scale tests in TRANSCODER-TEST mainly focus on correctness and fail to expose such inefficiencies. This gap calls for treating execution efficiency as an explicit evaluation dimension.

To address the challenge, we present **TRACE** (**TRA**nslated **C**ode **E**fficiency), the first benchmark designed to evaluate execution efficiency in LLM-translated code. We focus on translation tasks across C++, Java, and Python, which consistently rank among the three most popular programming languages (Carbonnelle, 2025; TIOBE Software BV, 2025). TRACE studies efficiency in method-level code translation. It uses the well-established TRANSCODER-TEST task setting as a controlled basis for evaluation. We target this granularity as it represents a fundamental translation unit and underpins broader workflows, including class- and repository-level translation (Xue et al., 2025; Ibrahimzada et al., 2025).

TRACE is built through a two-stage process. First, we iteratively generate stress tests to uncover latent efficiency issues beyond the reach of small-scale tests. Second, we apply efficiency-oriented filtering to retain only tasks that meaningfully expose runtime differences. The final benchmark comprises 1,000 efficiency-critical tasks, each accompanied by 10 correctness tests, 10 stress tests, and an average of 23 efficiency reference translations to support nuanced evaluation.

With TRACE, we evaluate 28 representative LLMs and derive several key insights. First, correctness and efficiency are often misaligned. For example, the correctness leader Claude-Sonnet-4-Think achieves a 95.5% pass rate but only moderate time efficiency, outperformed by smaller open-source LLMs such as Qwen2.5-Coder-14B-Instruct. Moreover, reasoning-enhanced or larger-scale LLMs do not reliably improve efficiency. We also observe directional asymmetry: while Java→Python and Python→Java achieve comparable correctness, models are markedly more efficient in the former. Second, inefficiency is both widespread and patterned. Notably, 23.5% of functionally correct translations are over 2× slower or more memory-intensive than the most efficient counterparts. A manual study of 327 inefficient cases further reveals three major categories: algorithm implementation discrepancy (11.9%), language construct mismatch (66.4%), and resource management inefficiency (21.7%). Third, inference-time prompt strategies offer limited relief. Few-shot prompting provides the most consistent yet modest efficiency gains, suggesting that improving translation efficiency likely requires more than prompt-level intervention. Taken together, our findings establish TRACE as the first benchmark for jointly evaluating correctness and efficiency in LLM-based code translation, providing a principled foundation for future research.

This paper makes the following contributions: 1) **Dimension.** We are the first to identify execution efficiency as a critical dimension in LLM-based code translation. 2) **Benchmark.** We present TRACE, the first benchmark designed to expose efficiency issues in LLM-translated code. 3) **Analysis.** We conduct a large-scale empirical and diagnostic study of 28 representative LLMs on TRACE, revealing the misalignment between correctness and efficiency and deriving a code translation-specific taxonomy of inefficiencies.

2 Related Work

Automatic Code Translation. Automatic code translation has progressed from early rule-based (Immunant, 2025; GoTranspile, 2025) and statistical methods (Karaivanov et al., 2014; Nguyen et al., 2016) to neural approaches such as the Transcoder series (Lachaux et al., 2020; Roziere et al., 2021; Szafraniec et al., 2022). With the help of LLMs, research has mainly pursued cor-

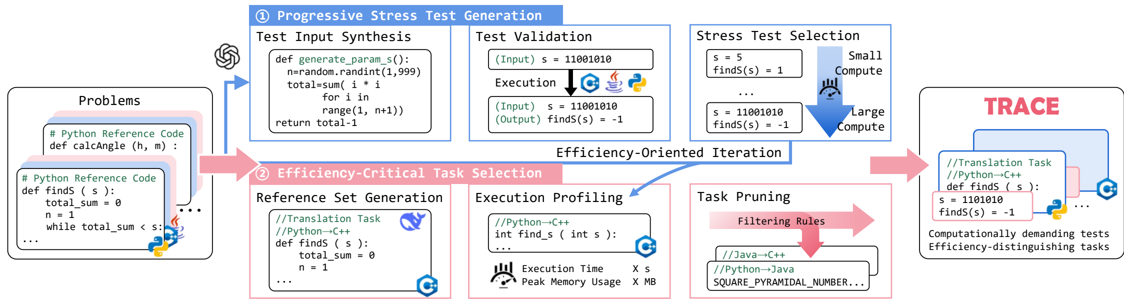

Figure 2: An overview of TRACE's LLM-driven two-stage construction pipeline.

rectness through fine-tuning (He et al., 2025; Liu et al., 2023), reasoning (Macedo et al., 2024; Nitin et al., 2024), and self-repair (Yang et al., 2024; Pan et al., 2024). Meanwhile, benchmarks such as Avatar (Ahmad et al., 2021), G-TransEval (Jiao et al., 2023), PolyHumanEval (Tao et al., 2024), and ClassEval-T (Xue et al., 2025) have progressively expanded coverage across both programming languages and evaluation granularities. However, current work remains centered on functional correctness and leaves efficiency underexplored.

Efficiency of LLM-generated Code. Recent work has begun to study the execution efficiency of LLM-generated code. Benchmarks such as EffiBench (Huang et al., 2024b) and EffiBench-X (Qing et al., 2025) construct efficiency-critical tasks, EvalPerf (Liu et al., 2024) proposes stress-based differential performance evaluation to reveal efficiency differences under stress-test inputs, and Mercury (Du et al., 2024) formalizes the Beyond score, a runtime-percentile-weighted metric for finer-grained performance assessment. On the optimization side, methods such as PIE (Shypula et al., 2023), SwiftCoder (Huang et al., 2024c), and Afterburner (Du et al., 2025) adapt or fine-tune LLMs to generate more efficient code.

TRACE connects and extends these two lines of research. First, it extends code translation evaluation beyond correctness by treating execution efficiency as an explicit quality dimension. Second, it studies efficiency in a *cross-language* setting that differs from prior work on code generation (Huang et al., 2024b; Qing et al., 2025; Liu et al., 2024; Du et al., 2024). Existing studies ask whether LLM-generated code is efficient for a given task within a single language. By contrast, TRACE examines whether LLM-translated code derived from the same source program preserves efficiency-critical semantics in the target language. This setting exposes translation-specific efficiency failures that prior work does not capture, such as mapping an efficient data structure or idiom to a suboptimal target-language counterpart.

3 Benchmark Construction

Terminology Formulation. A problem (P) is a canonical programming challenge. For each problem, a reference code (p) is a functionally correct solution. A code translation task (task) is an instance of translating a specific reference code p from a source language l_s to a target language l_t, denoted as $p_{l_s} \rightarrow p_{l_t}$. This work focuses on two efficiency-critical indicators: execution time (ET) and peak memory usage (PM).

Methodology Overview. As presented in Figure 2, TRACE is constructed through an LLM-driven two-stage pipeline. The construction begins with established programming challenges. The first stage, *Progressive Stress Test Generation*, iteratively synthesizes, validates, and selects stress tests through efficiency-oriented iteration, aiming to amplify latent efficiency disparities among translations. The second stage, *Efficiency-Critical Task Selection*, applies rigorous filtering rules to isolate tasks that meaningfully differentiate efficiency.

3.1 Problem Collection

TRACE uses problems from the TRANSCODER-TEST (Lachaux et al., 2020) benchmark, a widely used code translation benchmark containing diverse programming challenges. We select this foundation for two reasons: 1) It provides a high-quality set of recognized problems, ensuring relevance with prior work; 2) More importantly, it directly highlights the overlooked efficiency dimension within tasks that were previously evaluated only for correctness. We adopt the latest version of TRANSCODER-TEST (Yang et al., 2024), which contains 568 problems, each paired with one ground-truth implementation in C++, Java, and Python, along with 10 default correctness tests.

Algorithm 1 Progressive Stress Test Generation

```
1:  FUNCTION GenerateStressTests(P, M, I_max, K)
2:      P ← GetBasePrompt(P)
3:      T_s, E ← ∅, ∅
4:      for i ← 1 to I_max do
5:          // — Stage 1.1: Test Input Synthesis —
6:          P_iter ← EnrichPrompt(P, E)
7:          S ← M(P_iter)
8:          // — Stage 1.2: Test Validation —
9:          V ← ExecuteSynthesizers(S)
10:         T_cand ← Validate(V)
11:         // — Stage 1.3: Stress Test Selection —
12:         T_time ← SelectTopK(T_cand, K, time)
13:         T_mem ← SelectTopK(T_cand \ T_time, K, mem)
14:         T_iter ← T_time ∪ T_mem
15:         // — Stage 1.4: Efficiency-Oriented Iteration —
16:         T_s,prev ← T_s
17:         T_s ← UpdateGlobalPool(T_s, T_iter, K)
18:         if T_s = T_s,prev then break
19:         end if
20:         E ← ExtractExamples(T_s)
21:     end for
22:     return T_s
```

3.2 Progressive Stress Test Generation

We design an iterative stress test generation strategy, as detailed in Algorithm 1. This strategy progressively generates *stress tests* (tests demanding substantial computational resources during execution) to magnify hidden efficiency disparities.

Test Input Synthesis. We first synthesize large-scale test inputs that sufficiently exercise the runtime behavior of the translated code. However, directly generating such inputs with an LLM is often impractical, because their explicit representation may exceed the LLM context or output length limits. We therefore utilize the *synthesis-based* approach (Lines 5-7) (Liu et al., 2024; Shi et al., 2026; Wang et al., 2025), where the LLM ($\mathcal{M}$) is prompted to generate multiple test input *synthesizers* ($\mathcal{S}$) based on the ground-truth reference code. Each synthesizer is an independent Python program that programmatically outputs a test input when executed. To guide the LLM toward generating more resource-intensive inputs, we enrich the base prompt (Line 6) with synthesizer examples selected from the previous iteration. This enriched prompt ($\mathcal{E}$, Figure 12) incorporates: 1) two of the most effective synthesizers discovered so far; 2) the execution profiles of their outputs; and 3) chain-of-thought instructions. These modules jointly steer the LLM toward synthesizers that yield increasingly demanding test inputs.

Test Validation. After executing the synthesizers, the generated test inputs are rigorously validated to ensure soundness and cross-language consistency (Lines 8-10). A test input is retained only if it satisfies two mandatory runtime checks: 1) Execution Integrity, where it must execute successfully across ground-truth reference code (C++, Java, and Python) without runtime exceptions; and 2) Output Consistency, where it must produce identical outputs across reference code. This step eliminates test inputs susceptible to language-specific behaviors (e.g., integer overflow in C++ versus Python's arbitrary-precision integers), establishing reliable ground-truths for validated candidate tests ($\mathcal{T}_{cand}$).

Stress Test Selection. From the validated candidates $\mathcal{T}_{cand}$, we derive the stress test set $\mathcal{T}_{iter}$ for the current iteration (Lines 11-14). The targeted runtime indicators, ET and PM, are measured across reference code in C++, Java, and Python. For each indicator, we apply the *Borda Count* method (Emerson, 2013) to rank the tests. A test is ranked within each language, and its overall score is the sum of its three language-specific ranks. A lower Borda score thus reflects tests that consistently consume more resources across languages. Finally, for both ET and PM, we select the top-K tests with the lowest Borda scores to construct the stress test set $\mathcal{T}_{iter}$.

Efficiency-Oriented Iteration. The stress test generation proceeds as a feedback-driven loop to progressively discover computationally demanding tests (Lines 15-20). A global pool ($\mathcal{T}_s$) is maintained to store the most stressful tests identified so far. After each iteration, the newly selected tests ($\mathcal{T}_{iter}$) are merged into this pool. The pool is then re-ranked and pruned to retain only the overall top-K tests for both ET and PM (Line 17). If the updated pool $\mathcal{T}_s$ remains unchanged, the loop terminates early, indicating convergence. Synthesizers producing these top-performing tests are collected as guiding examples ($\mathcal{E}$) to enrich the synthesis prompt (Line 20) for the next iteration.

3.3 Efficiency-Critical Task Selection

Not all tasks are suitable for evaluating efficiency. For instance, a trivial translation of *def add(a, b): return a+b* is unlikely to produce variants with notable performance differences. To address this, we apply rigorous filtering to retain only tasks that meaningfully expose efficiency disparities.

For each task, we generate a diverse pool of reference translations by sampling from a representative set of LLMs. Each candidate translation is validated against the default tests from TRANSCODER-TEST to ensure functional correctness. The correct translations are retained and executed on the newly generated stress tests to measure ET and PM. The

resulting profiled translations form the task-level *efficiency reference set*.

Based on the efficiency reference set, we apply three filtering criteria: 1) Feasibility. A task is discarded if no evaluated LLM produces a functionally correct translation, as no valid samples remain for efficiency profiling. 2) Impactfulness. A task with trivial runtime cost is removed, since efficiency is not a practical concern. Concretely, at least one reference translation must exceed a predefined execution time threshold (ϵ_T) or peak memory threshold (ϵ_M) under stress tests. 3) Diversity. A task is excluded if correct translations exhibit negligible performance variance. This is measured by requiring the coefficient of variation ($\text{CV} = \text{std}/\text{mean}$) for either ET or PM to exceed a threshold (ϵ_D), ensuring that the task can meaningfully expose efficiency disparities.

3.4 Benchmark Implementation

For the *Progressive Stress Test Generation* stage, we employ GPT-4o (temperature $= 0.8$) to generate stress test synthesizers. We prompt GPT-4o to produce 3 distinct synthesizers for each ground-truth reference code across C++, Java, and Python, resulting in 9 synthesizers per problem. Each synthesizer is executed 3 times, yielding up to 27 candidate test inputs per iteration. The iterative loop (Algorithm 1) is capped at a maximum of $I_{\max} = 5$ iterations. In each iteration, we select the top $K = 5$ tests by ET and another $K = 5$ by PM, giving 10 stress tests per iteration. Aggregating across iterations, the final stress test suite for each problem contains the 10 most computationally demanding tests.

For the *Efficiency-Critical Task Selection* stage, we first construct a diverse pool of reference translations using 28 representative LLMs (see Section 4), decoded with sampling (temperature $= 0.8$). To select efficiency-critical tasks, we apply empirically derived thresholds: impactfulness ($\epsilon_T = 0.01$ seconds, $\epsilon_M = 1.5$ MB) and diversity ($\epsilon_D = 0.05$). Following prior work (Liu et al., 2024), these thresholds are estimated from the average resource consumption of a simple *Hello World* program across C++, Java, and Python, thereby excluding tasks with negligible runtime fluctuations.

Further benchmark construction details on the model choice and threshold selection are provided in Appendix A.1.

Table 1: Key statistics of the TRACE benchmark.

Benchmark Profile	Detail
Total Problems	357
Total Tasks	1,000
Default Tests (Per Task)	10
Execution Time (s)	0.09
Peak Memory (MB)	24.35
Stress Tests (Per Task)	10
Execution Time (s)	0.80 (8.9×)
Peak Memory (MB)	83.09 (3.4×)
Efficiency Reference Set (Per Task)	23.03
Performance Clusters	3.94

3.5 Benchmark Statistics

Table 1 summarizes the key characteristics of TRACE, and Appendix Table 13 further compares TRACE with existing code translation benchmarks.

Overall, TRACE contains 357 problems, yielding a total of 1,000 efficiency-critical tasks across six translation directions. Specifically, the distribution covers C++→Java (250), Java→C++ (127), Java→Python (148), Python→Java (236), C++→Python (141), and Python→C++ (98) tasks. Each task retains 10 default tests from TRANSCODER-TEST and is further equipped with 10 newly generated stress tests. On average, TRACE's stress tests increase ET and PM by $8.9\times$ and $3.4\times$, respectively, effectively exposing inefficiencies that otherwise remain invisible under small-scale tests. Each task is associated with an efficiency reference set averaging 23 correct translations. Clustering their ET/PM profiles with HDBSCAN (McInnes et al., 2017) yields 3.94 distinct performance clusters per task, revealing substantial heterogeneity in the efficiency behaviors of translations and enabling fine-grained evaluation.

Compared with existing benchmarks, TRACE introduces two key innovations. First, it explicitly integrates stress tests to exercise computational complexity, thereby revealing inefficiency overlooked by small-scale tests. Second, it selectively retains efficiency-critical tasks via rigorous filtering, ensuring that every included task provides meaningful variation in execution efficiency.

4 Experiment Setup

We first outline models, metrics, and the evaluation protocol used in our study.

Models. We evaluate 28 representative LLMs, covering both leading proprietary and prominent open-source code LLMs. For proprietary models, we include OpenAI (GPT-4o and O3 series), An-

Table 2: Overall performance of representative proprietary and open-source LLMs on TRACE.

Model	Summary					C++→Ja		C++→Py		Ja→C++		Ja→Py		Py→C++		Py→Ja	
	Pass	B_T	B_M	B_T^P	B_M^P	B_T	B_M	B_T	B_M	B_T	B_M	B_T	B_M	B_T	B_M	B_T	B_M
Proprietary LLMs																	
Claude-4-Think	**95.5**	49.6	50.5	51.9	52.9	37.1	54.0	61.3	48.4	58.8	50.7	65.4	46.9	52.0	44.4	39.9	52.6
Claude-4	93.4	48.8	45.5	52.3	48.7	66.8	62.9	25.7	13.2	45.5	56.6	27.0	16.1	43.4	47.5	61.1	58.1
DS-Reasoner	72.7	39.3	28.1	54.1	38.7	34.6	28.4	45.3	36.0	31.4	23.9	53.8	36.9	32.8	19.3	38.5	23.5
DS-V3.2	91.9	**50.7**	44.4	55.1	48.3	67.2	63.3	30.0	16.3	49.4	46.5	33.0	14.9	30.4	31.7	65.6	63.7
Gemini-Pro	62.5	37.9	24.5	**60.7**	39.3	35.9	20.7	44.8	34.7	29.3	22.2	47.6	35.8	27.3	10.0	38.9	22.8
Gemini-Flash	62.5	37.6	25.5	60.2	40.8	37.2	26.2	49.8	37.0	33.2	22.6	51.3	35.0	18.1	7.9	32.8	20.9
O3	85.1	50.2	34.8	58.9	40.8	53.5	34.3	63.6	49.0	39.9	25.1	61.0	46.5	26.9	17.2	46.9	31.8
O3-Mini	91.0	42.9	**50.7**	47.2	**55.8**	40.7	52.3	37.4	63.5	48.9	47.3	58.4	46.1	33.0	32.2	39.7	54.0
Open-Source LLMs																	
CL-7B-Inst	75.5	44.2	30.6	58.5	40.6	39.1	28.6	53.5	39.2	43.1	33.2	53.7	38.0	34.2	15.8	42.7	27.7
CL-13B-Inst	76.7	45.3	31.3	59.1	40.8	40.7	26.0	53.0	39.8	43.7	34.9	56.5	39.2	32.8	26.5	44.7	27.0
CL-34B-Inst	69.3	42.2	30.3	**60.9**	43.7	36.6	24.1	54.8	43.4	30.4	29.2	56.3	40.1	30.0	20.6	43.3	27.3
DSC-6.7B-Inst	86.2	51.0	35.3	59.1	40.9	50.9	33.7	58.4	44.9	42.4	34.2	62.1	43.6	36.4	22.5	50.3	31.9
DSC-33B-Inst	89.8	52.2	36.9	58.2	41.1	54.8	36.5	60.6	46.9	41.3	27.2	64.3	46.1	40.1	30.5	47.9	33.6
QC-7B-Inst	90.1	51.4	36.9	57.1	41.0	42.5	33.0	64.3	44.8	48.5	39.2	64.9	47.4	43.2	28.3	49.6	32.1
QC-14B-Inst	**91.6**	**54.8**	39.0	59.8	42.6	54.3	36.5	63.3	46.5	46.2	37.4	65.8	46.0	42.4	29.4	53.0	37.7
QC-32B-Inst	90.7	48.7	**42.9**	53.7	**47.3**	56.2	37.7	63.8	44.7	50.3	40.6	33.8	68.2	26.7	31.7	49.3	37.3
Average	84.9	45.8	37.7	54.0	43.6	45.8	37.7	52.6	39.7	41.4	34.6	54.2	39.2	35.7	27.5	45.5	37.3

thropic (Claude-4 and 3.5 series), Google (Gemini-2.5 series), and DeepSeek. For open-source models, we assess CodeLlama-hf (CL) (Roziere et al., 2023), DeepSeek-Coder (DSC) (Guo et al., 2024), and Qwen2.5-Coder (QC) series (Hui et al., 2024), including both base and instruction-tuned (Inst) models. Table 12 summarizes the model aliases and details.

Metrics. We evaluate model performance along two complementary dimensions: functional correctness and execution efficiency.

For correctness, we use *Pass Rate* (Pass), defined as the percentage of tasks whose translations pass all tests.

For efficiency, we adopt the *Beyond* score (Du et al., 2024). For a given efficiency indicator X (e.g., execution time), the score evaluates each candidate translation c against the task-level efficiency reference set $\mathcal{R}$. Specifically, let $\mathcal{R}_X = \{X(r) : r \in \mathcal{R}\}$ denote the task-specific reference values for indicator X, and let $x_c = X(c)$ denote the candidate's measured value. For brevity, let $r_X^{\min} = \min \mathcal{R}_X$ and $r_X^{\max} = \max \mathcal{R}_X$. Beyond is defined as:

$$B_X(c) = \frac{r_X^{\max} - \mathrm{clip}\big(x_c, r_X^{\min}, r_X^{\max}\big)}{r_X^{\max} - r_X^{\min}} \times 100\%.$$

where $\mathrm{clip}(a, \ell, u)$ truncates a to $[\ell, u]$. Intuitively, $B_X(c)$ reflects the normalized efficiency of candidate c relative to the task-specific reference range, where a higher score indicates better efficiency. In this work, we consider two indicators, ET and PM, and compute Beyond separately for each. We further report two variants: B_X, averaged over all translations with incorrect outputs assigned a score of 0, and B_X^P, computed only over correct translations to isolate efficiency under correctness.

Model Evaluation Protocol. For each task, the model generates a single translation using greedy decoding (temperature $= 0.0$) under a zero-shot prompt (Appendix C.1). Functional correctness is verified with the default tests from TRANSCODER-TEST. Translations that fail to compile, trigger runtime errors, or exceed resource limits (180 s or 4096 MB) are marked as incorrect and excluded from efficiency evaluation. Correct translations are then executed on stress tests to measure ET and PM, with each metric reported as the arithmetic mean of five runs. Further experimental details are provided in Appendix A.4.

5 Evaluation

We conduct extensive evaluation through three lenses: 1) an end-to-end model performance comparison, 2) a taxonomy-driven characterization of inefficient translations, and 3) an exploration of inference-time efficiency improvement strategies.

5.1 Overall Performance Landscape

Table 2 reports the performance of a representative set of LLMs to illustrate our key findings. The complete evaluation is reported in Table 15.

Correctness is not a proxy for efficiency. From Table 2, we find that higher correctness does not

Table 3: Taxonomy of inefficiency patterns observed in LLM-based code translation.

Category	Pct.(%)
1. Algorithm Implementation Discrepancy	**11.93**
1.1 Asymptotic Complexity Degradation	**3.36**
The translation adopts an alternative algorithm with worse asymptotic complexity than the source code.	
1.2 Omission of Target-Language Idioms	**8.56**
The translation ignores efficient idioms or optimized library routines available in the target language.	
Example: C++ in–place reversal **std::reverse(s.begin(), s.end())** ($O(N)$) is translated into Python repeated concatenation **res=""; for c in s: res=c+res** ($O(N^2)$), instead of the idiomatic target **s[::-1]**.	
2. Language Construct Mismatch	**66.36**
2.1 Suboptimal Data Structure Selection	**20.49**
The adopted data structure is asymptotically or practically inferior for the required operations.	
2.2 Inefficient API/Pattern Usage	**45.87**
The translation uses non-optimal APIs or coding patterns and introduces unnecessary runtime costs.	
Example: Java **HashMap** (amortized $O(1)$) is mapped to C++ **std::map** ($O(\log N)$) instead of **std::unordered_map**, which becomes inefficient when handling frequent lookups or insertions.	
3. Resource Management Inefficiency	**21.71**
3.1 Inefficient Data Representation	**17.13**
Primitive or compact source representations are replaced with memory-intensive object-based equivalents.	
3.2 Non-Idiomatic Memory Management	**4.59**
The translation adopts manual or unsafe memory practices instead of modern, idiomatic abstractions.	
Example: C++ **long** is translated into Java **BigInteger**, causing excessive allocation and memory footprint; when the value range permits, the optimal target is **long**.	

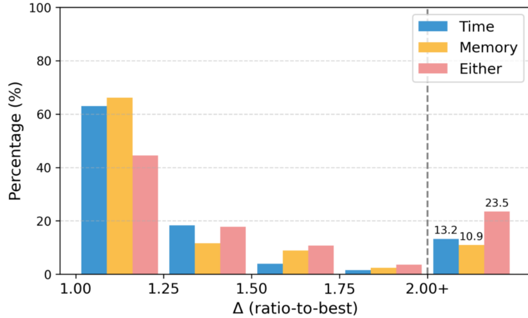

Figure 3: Execution profile distribution of translations.

necessarily lead to higher efficiency. For instance, Claude-4-Think attains the highest Pass rate (95.5) but only moderate time efficiency (B_T: 49.6), outperformed by smaller open-source LLMs such as QC-14B-Inst (B_T: 54.8). Conversely, CL-34B-Inst achieves the lowest correctness among open-source models (69.3), yet it delivers the most time-efficient solutions for its correct translations (B_T^P: 60.9).

To examine the relationship between correctness and efficiency, we conduct correlation analysis (Appendix B.3). Across LLMs, correctness shows a moderate negative association with time efficiency (Pearson $r = -0.54$, Spearman $\rho = -0.61$) and a moderate positive association with memory efficiency ($r = 0.57$, $\rho = 0.70$). Restricting the analysis to high-performing models (Pass rate $\geq 85\%$), evaluated on their 613 jointly solved tasks, yields similar trends ($r = -0.51$ for ET, $r = 0.64$ for PM). OLS regression further confirms that correctness explains only a limited portion of efficiency variance ($R^2 = 0.29$ for ET, $R^2 = 0.33$ for PM).

Our findings indicate that correctness and efficiency are distinct dimensions of performance, with correctness alone insufficient to explain the observed efficiency variation.

Reasoning and scaling do not reliably improve efficiency. Across proprietary and open-source models, neither reasoning-oriented variants nor larger models consistently improve efficiency. Reasoning models, trained with intermediate reasoning supervision, do not always outperform their standard counterparts: for instance, while Claude-4-Think achieves slightly higher time efficiency than Claude-4 (B_T: 49.6 vs. 48.8), the reverse is observed in the DeepSeek family, where the general-purpose DS-V3.2 surpasses DS-Reasoner in terms of both time (B_T: 50.7 vs. 39.3) and memory efficiency (B_M: 44.4 vs. 28.1). Second, scaling in model size likewise provides no monotonic improvements. Larger models generally exhibit stronger capabilities in code understanding and generation, yet these advantages often fail to yield corresponding efficiency gains. For instance, QC-14B-Inst is more time-efficient than its 32B counterpart (B_T: 54.8 vs. 48.7), and within the CodeLlama series, the 34B variant (B_T: 42.2) is surpassed by the 7B (44.2) and 13B (45.3) models.

These results suggest that existing reasoning and model scaling practices remain primarily correctness-driven, leaving efficiency as a secondary and unresolved challenge.

Efficiency exhibits strong directional asymmetry across languages. We observe a clear asymmetry across translation directions. For Java↔Python, Pass rates are relatively close (89.3 vs. 85.2, Ta-

Table 4: Statistics of manually classified inefficiency cases. $\Delta T/\Delta M$ denotes the ratio of ET/PM of each translation relative to the task-specific best.

Inefficiency Category	ΔT		ΔM	
	Avg.	**Med.**	**Avg.**	**Med.**
Algorithm Implementation	371.6	3.2	2.5	1.2
Language Construct	6.6	2.1	3.4	1.9
Resource Management	2.6	1.1	5.9	5.0

ble 15). However, their time efficiency differs sharply (B_T: 54.2 vs. 45.5), and this gap remains even when considering only correct translations (B_T^P: 61.2 vs. 53.5). These results suggest that LLMs internalize language-specific knowledge unevenly, producing more efficient translations into Python than into Java. This asymmetry shows that efficiency is shaped not only by model capacity but also by structural properties of programming languages, highlighting the challenge of building efficiency-aware multilingual models.

5.2 Inefficiency Profile and Taxonomy

We analyze the distribution and root causes of translation inefficiencies. Figure 3 shows the execution profile distribution, where each translation's ET/PM is measured relative to the most efficient within the task. Overall, 23.5% of translations exhibit substantial inefficiency, with ET/PM exceeding twice that of the task-specific best. This shows that inefficiency is not an isolated anomaly but a recurrent phenomenon even in correct translations.

To capture the root causes of inefficiency, we examined 327 representative cases (one case per problem) with an inefficiency ratio of over 2.0. Two authors independently classified the cases and resolved disagreements through discussion. The resulting taxonomy is summarized in Table 3.

The examination reveals three major inefficiency patterns in code translation. 1) Algorithm Implementation Discrepancy (11.93%). We classify a case as this type when the target algorithm diverges in computational complexity from the source or fails to employ efficient target-language idioms. It comprises two subtypes: asymptotic complexity degradation and omission of target-language idioms. 2) Language Construct Mismatch (66.36%). We classify a case as this type when the source specifies an explicit construct, yet the translation maps it to a correct but suboptimal alternative. It comprises two subtypes: suboptimal data structure selection and inefficient API/pattern usage. 3) Resource Management Inefficiency (21.71%). We classify a case as this type when the source specifies no representation, yet the translation introduces heavyweight abstractions or overhead. It comprises two subtypes: inefficient data representation and non-idiomatic memory management.

Table 5: Impact of prompt strategies on efficiency.

Prompt Strategy	**Pass**	B_T	B_M	B_T^P	B_M^P
GPT-4o					
zero-shot	89.5	48.8	42.2	54.5	47.2
perf-zero-shot	89.4	50.5	44.9	**56.5**	**50.2**
perf-few-shot	**93.9**	**51.6**	**46.3**	54.9	49.3
perf-self-refine	89.2	49.2	44.0	55.2	49.4
DS-V3.2					
zero-shot	91.9	50.7	44.4	55.1	48.3
perf-zero-shot	90.2	49.9	42.7	55.3	47.4
perf-few-shot	**94.4**	**52.9**	**47.5**	**56.1**	**50.3**
perf-self-refine	91.7	51.4	43.9	**56.1**	47.8
Claude-4					
zero-shot	93.4	48.8	45.5	52.3	48.7
perf-zero-shot	91.6	47.9	42.8	52.3	46.7
perf-few-shot	**94.3**	50.2	**48.7**	53.3	**51.7**
perf-self-refine	91.9	**52.3**	45.9	**56.9**	49.9

Table 4 quantifies the severity of inefficiencies across categories. Algorithm implementation discrepancies are relatively rare but induce catastrophic slowdowns in ET (avg. 371.6×, median 3.2×). Language construct mismatches dominate in frequency and introduce substantial overheads in both ET (avg. 6.6×, median 2.1×) and PM (avg. 3.4×, median 1.9×). Resource management inefficiencies impose the largest PM penalties, with an average increase of 5.9× and a median of 5.0×.

Our analysis shows that inefficiencies are both widespread and patterned, spanning algorithmic, construct-level, and resource-level dimensions.

5.3 Impact of Prompt Strategies on Efficiency

We further explore whether inference-time prompt strategies can improve code translation efficiency. Beyond the default *zero-shot* baseline, we experiment with three performance-aware prompts (Appendix C.1), inspired by recent code efficiency work (Shypula et al., 2023; Liu et al., 2024; Huang et al., 2024a): 1) *perf-zero-shot*, which adds an explicit system instruction on efficiency; 2) *perf-few-shot*, which provides two efficient translation examples; and 3) *perf-self-refine*, which refines the initial translation based on execution feedback.

Table 5 summarizes the results. Overall, prompting yields modest and model-dependent gains. The most consistent improvements arise with perf-few-shot, which increases both correctness and efficiency (e.g., DS-V3.2, Pass: 94.4; B_T: 52.9, B_M:

47.5). Other strategies are mixed: perf-self-refine, for instance, improves Claude-4's time efficiency (B_T: 48.8→52.3) while slightly reducing correctness (93.4→91.9). Taken together, the results suggest that prompt-based strategies can alleviate inefficiencies to a limited extent but fail to eliminate them, highlighting the need for more advanced methods to encode efficiency principles.

6 Discussion

6.1 Validity of the Efficiency Reference Set

A potential threat to construct validity lies in using an LLM-derived translation set (efficiency reference set) as the evaluation anchor. Since the Beyond score is defined relative to the observed reference spectrum, its validity depends on whether this set provides a sufficiently strong upper bound. If the reference set were weaker than human-optimized implementations, the resulting scores could overestimate the model performance.

To examine this risk, we conducted a human-baseline sanity check on 50 randomly sampled tasks (18 C++, 21 Java, and 11 Python). Two authors with over seven years of programming experience collaboratively implemented optimized solutions (*Human*), which we compared against the fastest LLM-produced reference translation for each task (*FastLLM*). Overall, *Human* achieves an average execution time of 2.61s, compared with 2.63s for *FastLLM* (a relative gap of about 0.8%). The same pattern holds across languages: 0.50s vs. 0.53s in C++, 0.56s vs. 0.55s in Java, and 10.00s vs. 10.03s in Python. These results suggest that the reference set reaches a comparable efficiency level to human-optimized solutions, supporting its use as a reasonable evaluation anchor. Future work should extend this validation with larger samples and broader expert-written baselines.

6.2 Towards Intrinsic Efficiency Awareness

Our findings suggest that improving LLM-based code translation efficiency requires more than simple inference-time prompt engineering. This work provides two foundations for future research on efficiency-aware code translation. First, we offer a controlled benchmark, TRACE, for evaluating whether new methods improve translation efficiency. Second, our diagnostic analysis identifies recurring inefficiency patterns. In particular, our taxonomy can guide the construction of targeted training data or optimization objectives. Future work can build on these signals through supervised fine-tuning or reinforcement learning to improve models' efficiency awareness in code translation.

7 Conclusion

This work foregrounds execution efficiency as a critical yet long-overlooked dimension in LLM-based code translation. We introduce TRACE, a benchmark that explicitly exposes efficiency gaps beyond correctness through progressive stress test generation and efficiency-critical task selection. From an evaluation of 28 models, we find that correctness does not necessarily imply efficiency; inefficiencies are both prevalent and patterned; and inference-time prompt strategies deliver limited and model-dependent gains. Ultimately, our work highlights the open challenge of developing advanced methods that equip LLMs with stronger efficiency awareness for code translation.

Acknowledgments

This work is supported by National Key Research and Development Program of China 2024YFE0204200, the National Natural Science Foundation of China under Grant No. 62372005 and 62402482. Jie M. Zhang is supported by the ITEA GreenCode Project under Project No. 23016 and the ITEA GENIUS Project under Project No. 23026.

Limitations

The following limitations define the scope of our work and highlight avenues for future research.

Scope of Efficiency. Our study focuses on two core dimensions of efficiency: execution time and peak memory usage, following prior work on LLM-generated code efficiency (Du et al., 2024; Huang et al., 2024b; Qing et al., 2025; Liu et al., 2024; Peng et al., 2025). While these metrics serve as fundamental indicators of execution efficiency, we acknowledge that they only partially capture real-world performance. Factors such as I/O latency, compilation overhead, and energy consumption may also substantially influence efficiency in practice. Future work should consider broader and domain-specific aspects to provide a more comprehensive view.

Scope of Programming Languages. Our study focuses on C++, Java, and Python, which consistently rank among the top three most popular languages in established popularity indices (Carbonnelle, 2025; TIOBE Software BV, 2025). Concentrating on these languages provides strong representativeness and supports the significance of our findings. Meanwhile, we acknowledge that excluding other languages may pose risks to broader generalization. Extending TRACE to additional programming languages would help evaluate the generalizability of our findings.

Scope of Translation Scenario. The problems in TRACE are drawn from TRANSCODER-TEST, whose tasks were originally collected from GeeksForGeeks (GeeksforGeeks, 2025). Accordingly, TRACE mainly studies method-level algorithmic and data-structure-oriented translation tasks. We acknowledge that this setting is cleaner than real-world translation scenarios, which often involve class-level or repository-level contexts (Jimenez et al., 2023; Ibrahimzada et al., 2025; Xue et al., 2025). Currently, we focus on method-level translation because it provides a controlled setting for efficiency evaluation, and it serves as a core operational unit in higher-level translation workflows. For example, prior class- and repository-level approaches often decompose larger translation units into methods (Xue et al., 2025; Ibrahimzada et al., 2025). As a result, inefficiencies introduced at the method level can carry over to higher-level settings. Future work should extend TRACE to class-level and repository-level settings to cover more software translation scenarios.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2022. Summarize and generate to back-translate: Unsupervised translation of programming languages. *arXiv preprint arXiv:2205.11116*.

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*.

Pierre Carbonnelle. 2025. PYPL: Popularity of programming language index. https://pypl.github.io/PYPL.html. Accessed: 2025-09-27.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A code efficiency benchmark for code large language models. *Advances in Neural Information Processing Systems*, 37:16601–16622.

Mingzhe Du, Luu Anh Tuan, Yue Liu, Yuhao Qing, Dong Huang, Xinyi He, Qian Liu, Zejun Ma, and See-kiong Ng. 2025. Afterburner: Reinforcement learning facilitates self-improving code efficiency optimization. *arXiv preprint arXiv:2505.23387*.

Peter Emerson. 2013. The original borda count and partial voting. *Social Choice and Welfare*, 40(2):353–358.

GeeksforGeeks. 2025. GeeksforGeeks | A computer science portal for geeks. https://www.geeksforgeeks.org/. Accessed: 2025-07-22.

GoTranspile. 2025. CxGo: Convert C code to Go. https://github.com/gotranspile/cxgo. Accessed: 2025-07-01.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Minghua He, Fangkai Yang, Pu Zhao, Wenjie Yin, Yu Kang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. 2025. Execoder: Empowering large language models with executability representation for code translation. *arXiv preprint arXiv:2501.18460*.

Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie Zhang. 2024a. Effilearner: Enhancing efficiency of generated code via self-optimization. *Advances in Neural Information Processing Systems*, 37:84482–84522.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M Zhang. 2024b. Effibench: Benchmarking the efficiency of automatically generated code. *arXiv preprint arXiv:2402.02037*.

Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M Zhang. 2024c. Swiftcoder: Enhancing code generation in large language models through efficiency-aware fine-tuning. *arXiv preprint arXiv:2410.10209*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2025. Alphatrans: A neuro-symbolic compositional approach for repository-level code translation and validation. *Proceedings of the ACM on Software Engineering*, 2(FSE):2454–2476.

Immunant. 2025. C2Rust: A tool for converting C to Rust. https://github.com/immunant/c2rust. Accessed: 2025-07-01.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the evaluation of neural code translation: Taxonomy and benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1529–1541. IEEE.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Svetoslav Karaivanov, 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*, pages 173–184.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*.

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)*, pages 755–767. IEEE.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. 2024. Evaluating language models for efficient code generation. *arXiv preprint arXiv:2408.06450*.

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*.

Leland McInnes, John Healy, Steve Astels, and 1 others. 2017. hdbscan: Hierarchical density based clustering. *J. Open Source Softw.*, 2(11):205.

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*, pages 756–758.

Vikram Nitin, Rahul Krishna, and Baishakhi Ray. 2024. Spectra: Enhancing the code translation ability of language models by generating multi-modal specifications. *arXiv preprint arXiv:2405.18574*.

Changan Niu, Ting Zhang, Chuanyi Li, Bin Luo, and Vincent Ng. 2024. On evaluating the efficiency of source code generated by llms. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pages 103–107.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. 2025. Coffe: A code efficiency benchmark for code generation. *Proceedings of the ACM on Software Engineering*, 2(FSE):242–265.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, and 1 others. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Yuhao Qing, Boyu Zhu, Mingzhe Du, Zhijiang Guo, Terry Yue Zhuo, Qianru Zhang, Jie M Zhang, Heming Cui, Siu-Ming Yiu, Dong Huang, and 1 others. 2025. Effibench-x: A multi-language benchmark for measuring efficiency of llm-generated code. *arXiv preprint arXiv:2505.13004*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. *ICLR*.

Jingwei Shi, Xinxiang Yin, Jing Huang, Jinman Zhao, and Shengyu Tao. 2026. Codehacker: Automated test case generation for detecting vulnerabilities in competitive programming solutions. *arXiv preprint arXiv:2602.20213*.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*.

Stein's GCD. 2025. Stein's Algorithm for Finding GCD. https://www.geeksforgeeks.org/dsa/steins-algorithm-for-finding-gcd/. Accessed: 2025-07-01.

Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578*.

Qingxiao Tao, Tingrui Yu, Xiaodong Gu, and Beijun Shen. 2024. Unraveling the potential of large language models in code translation: How far are we? In *2024 31st Asia-Pacific Software Engineering Conference (APSEC)*, pages 353–362. IEEE.

TIOBE Software BV. 2025. Tiobe index for september 2025. https://www.tiobe.com/tiobe-index/. Accessed: 2025-09-27.

Tina Vartziotis, Ippolyti Dellatolas, George Dasoulas, Maximilian Schmidt, Florian Schneider, Tim Hoffmann, Sotirios Kotsopoulos, and Michael Keckeisen. 2024. Learn to code sustainably: An empirical study on llm-based green code generation. *arXiv preprint arXiv:2403.03344*.

Zihan Wang, Siyao Liu, Yang Sun, Hongyan Li, and Kai Shen. 2025. Codecontests+: High-quality test case generation for competitive programming. *arXiv preprint arXiv:2506.05817*.

Pengyu Xue, Linhao Wu, Zhen Yang, Chengyi Wang, Xiang Li, Yuxiang Zhang, Jia Li, Ruikai Jin, Yifei Pei, Zhaoyan Shen, and 1 others. 2025. Classeval-t: Evaluating large language models in class-level code translation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1421–1444.

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):1585–1608.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, and 1 others. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2).

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, and 1 others. 2023a. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023b. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*.

Ming Zhu, Mohimenul Karim, Ismini Lourentzou, and Daphne Yao. 2024. Semi-supervised code translation overcoming the scarcity of parallel code data. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1545–1556.

A Implementation Details

A.1 Pipeline Design Ablations

We provide details to justify key design choices in our benchmark construction pipeline.

Model and Iteration Choice. We performed a pilot study on a 10% subset of TRANSCODER-TEST (56 problems), comparing three widely used LLMs, GPT-4o, Claude-4, and DS-V3.2, for stress test generation. Table 6 reports the average execution time (ET) and peak memory usage (PM) of the generated tests across iterations. For all three models, ET and PM increase substantially in the first five iterations, while the gains diminish beyond iteration 5. We therefore set the maximum number of iterations to $I_{\max} = 5$. Among the compared models, GPT-4o achieves performance comparable to Claude-4 at lower API cost, and consistently outperforms DS-V3.2. We therefore adopted GPT-4o.

Threshold Choice. We set $\epsilon_T = 0.01$ seconds and $\epsilon_M = 1.5$ MB following prior work (Liu et al., 2024), using the average resource consumption of a simple *Hello World* program across C++, Java, and Python as the baseline. We set $\epsilon_D = 0.05$, which is higher than both the fluctuation range reported in prior work (Peng et al., 2025) and the variance observed in our experiment environment (Appendix A.4).

Table 6: The average ET (seconds) and PM (MB) of the obtained stress tests across models and iterations.

LLM	Metric	Iter1	Iter2	Iter3	Iter4	Iter5	Iter6
Claude-4	ET	0.72	1.85	2.93	3.60	4.20	4.47
	PM	72.3	299.5	504.6	655.2	782.9	846.3
GPT-4o	ET	0.69	1.79	2.78	3.57	4.09	4.37
	PM	78.1	305.1	503.5	648.2	772.4	804.6
DS-V3.2	ET	0.63	1.59	2.38	3.02	3.51	3.74
	PM	69.2	256.2	426.2	552.0	661.8	718.3

A.2 Details of TRACE's Stress Tests

Table 7 presents the average ET and PM for the top-10 obtained stress tests across iterations. Both efficiency indicators consistently increase with each iteration, confirming that our iterative efficiency-oriented approach successfully guides the LLM to generate more resource-intensive tests.

Table 8 compares the effectiveness between TRANSCODER-TEST's default tests and TRACE's stress tests across LLM-generated reference translations. As shown, TRACE's stress tests significantly amplify the computational demands. On TRANSCODER-TEST's original tasks, stress tests increase the average ET by a factor of 84.57× (from 0.07 s to 5.92 s) and PM by 78.6× (from 15.02 MB to 1180.64 MB). On TRACE's selected tasks, the stress tests still demonstrate a significant superiority, increasing the ET by 8.9× (from 0.09 s to 0.80 s) and PM by 3.4× (from 24.35 MB to 83.09 MB). This substantial increase in resource consumption is critical for exposing latent efficiency differences that are otherwise undetectable with the default tests from the original benchmark.

Table 7: The average ET (seconds) and PM (MB) of the top-10 stress tests across iterations.

Language		Iter 1	Iter 2	Iter 3	Iter 4	Iter 5
C++	ET	0.14	0.24	0.33	0.42	**0.50**
	PM	19.01	22.43	25.85	28.70	**31.66**
Java	ET	0.50	0.69	0.88	1.08	**1.27**
	PM	131.10	173.69	216.28	252.24	**279.41**
Python	ET	1.45	3.93	6.41	8.29	**9.88**
	PM	54.13	585.00	1115.86	1496.83	**1897.97**
Avg.	ET	0.68	1.59	2.50	3.20	**3.80**
	PM	67.36	256.95	446.55	582.27	**721.09**

Table 8: The performance comparison between Transcoder-Test's default tests and TRACE's stress tests.

Language		Transcoder-Test		TRACE	
		Default	Stress	Default	Stress
C++	ET	<0.01	0.65	<0.01	0.28
	PM	1.51	32.50	1.52	32.04
Java	ET	0.18	2.16	0.18	0.54
	PM	45.86	497.04	46.09	140.90
Python	ET	0.02	14.85	0.01	1.57
	PM	7.72	3030.54	5.56	25.62
Avg.	ET	0.07	**5.92**	0.09	**0.80**
	PM	15.02	**1180.64**	24.35	**83.09**

A.3 Details of Task Pruning

Table 9 summarizes the task pruning process used to construct TRACE. Starting from the original 2,828 translation tasks in TRANSCODER-TEST, we first discard 362 tasks for which no valid stress test suite could be generated. We then apply three filtering rules introduced in the main paper: Rule-1 (Feasibility) removes 217 tasks unsolved by any model; Rule-2 (Impactfulness) excludes 309 tasks where efficiency is not practically relevant; and Rule-3 (Diversity) eliminates 940 tasks where correct solutions show negligible performance variation. After these stages, the benchmark retains 1,000 tasks that are demonstrably efficiency-critical.

Table 9: Statistics of the filtered tasks by each rule.

Direction	Orig.	w/o $\mathcal{T}_s$	Rule-1	Rule-2	Rule-3	TRACE
C++→Java	482	59	14	0	159	250
C++→Py	464	63	32	0	228	141
Java→C++	468	59	20	160	102	127
Java→Py	464	63	29	0	224	148
Py→C++	468	59	87	149	75	98
Py→Java	482	59	35	0	152	236
Summary	2828	-362	-217	-309	-940	1000

Table 10: Experiment environment details.

Component	Specification
Hardware	
Operating System	Ubuntu 22.04.5 LTS
CPU Type	Intel(R) Xeon(R) Platinum 8468V @ 3.8 GHz
CPU Cores	192
GPU Model	NVIDIA H200 NVL-140GB
GPU Quantity	2
System Memory (RAM)	512 GB
Software	
Linux Kernel	6.8.0-60-generic
C++ Compiler	g++ (Ubuntu 22.04)
Java Development Kit	openjdk 17.0.15
Python Interpreter	Python 3.12
Instruction	
C++ Compilation	`g++ -O2`
Java Compilation	`javac`
Time Profiling	`perf stat`
Memory Profiling	Custom script using `psutil`

A.4 Experiment Environment Setup

All the experiments were conducted in a controlled environment, as summarized in Table 10 and Table 11, to ensure consistency. To minimize interference, each profiled command was pinned to the least-utilized CPU core using `taskset -c`. ET was measured with `perf stat`, and PM was monitored by sampling the Resident Set Size (RSS) of the process tree via `psutil`.

We enforced a 180-second timeout and a 4096-MB peak memory limit, calibrated from reference translations to cover diverse valid implementations while excluding pathological cases such as infinite loops. Each execution profile was averaged over five independent runs to mitigate noise from system load. Under this controlled setup, measurement variance was minimal, with coefficients of variation for both ET and PM typically below 3.0%. Prior work (Peng et al., 2025) reported a broader fluctuation range of 2–5% in less controlled environments. To conservatively filter out noise, we set the diversity threshold to $\epsilon_D = 0.05$, which exceeds both our observed variance and the reported noise floor, thereby ensuring that retained tasks reflect genuine algorithmic or idiomatic efficiency differences rather than measurement errors.

Table 11: Experiment settings.

Setting	Value
Timeout Limit	180 seconds
Memory Limit	4096 MB peak RSS
Runs per Test	5 (arithmetic mean reported)

A.5 Evaluation Model Choice

Table 12 provides details for the 28 LLMs used in our work. These models were utilized for both the curation of the benchmark and the evaluation of the work.

Table 12: Model details, including aliases used in this paper, names, and public links.

Alias	**Name**	**Public Link**
	Proprietary Models	
Claude-4-Think	Claude-Sonnet-4-Think	https://www.anthropic.com/news/claude-4
Claude-4	Claude-Sonnet-4	https://www.anthropic.com/news/claude-4
Claude-3.5	Claude-Sonnet-3.5	https://www.anthropic.com/news/claude-3-5-sonnet
DS-Reasoner	DeepSeek-Reasoner	https://www.deepseek.com
DS-V3.2	DeepSeek-V3.2	https://www.deepseek.com
Gemini-Pro	Gemini-2.5-Pro	https://deepmind.google/models/gemini/pro
Gemini-Flash	Gemini-2.5-Flash	https://deepmind.google/models/gemini/flash
GPT-4o	GPT-4o	https://openai.com
GPT-4o-Mini	GPT-4o-Mini	https://openai.com
GPT-3.5-Turbo	GPT-3.5-Turbo	https://openai.com
O3	OpenAI-O3	https://openai.com
O3-Mini	OpenAI-O3-Mini	https://openai.com
	Open-Source Models	
CL-7B	CodeLlama-7B	https://github.com/facebookresearch/codellama
CL-7B-Inst	CodeLlama-7B-Instruct	https://github.com/facebookresearch/codellama
CL-13B	CodeLlama-13B	https://github.com/facebookresearch/codellama
CL-13B-Inst	CodeLlama-13B-Instruct	https://github.com/facebookresearch/codellama
CL-34B	CodeLlama-34B	https://github.com/facebookresearch/codellama
CL-34B-Inst	CodeLlama-34B-Instruct	https://github.com/facebookresearch/codellama
DSC-6.7B	deepseek-coder-6.7b	https://github.com/deepseek-ai/DeepSeek-Coder
DSC-6.7B-Inst	deepseek-coder-6.7b-Instruct	https://github.com/deepseek-ai/DeepSeek-Coder
DSC-33B	deepseek-coder-33b	https://github.com/deepseek-ai/DeepSeek-Coder
DSC-33B-Inst	deepseek-coder-33b-Instruct	https://github.com/deepseek-ai/DeepSeek-Coder
QC-7B	Qwen2.5-Coder-7B	https://github.com/QwenLM/Qwen2.5-Coder
QC-7B-Inst	Qwen2.5-Coder-7B-Instruct	https://github.com/QwenLM/Qwen2.5-Coder
QC-14B	Qwen2.5-Coder-14B	https://github.com/QwenLM/Qwen2.5-Coder
QC-14B-Inst	Qwen2.5-Coder-14B-Instruct	https://github.com/QwenLM/Qwen2.5-Coder
QC-32B	Qwen2.5-Coder-32B	https://github.com/QwenLM/Qwen2.5-Coder
QC-32B-Inst	Qwen2.5-Coder-32B-Instruct	https://github.com/QwenLM/Qwen2.5-Coder

Table 13: Comparison of TRACE with representative code translation benchmarks. Corr. and Stress. denote the average number of correctness and stress tests, respectively.

Benchmark	**Source**	**Prob.**	**Programming Languages**	**Granularity**	**Corr.**	**Stress.**	**Evaluation**
CodeNet	Contests	200	C++, C, Go, Java, Py	stmt./Method	1	✗	Correctness
Avatar	Contests	250	Java, Py	stmt./Method	25	✗	Correctness
Transcoder-Test	Geeks4Geeks	568	C++, Java, Py	Method	10	✗	Correctness
G-TransEval	Multiple	400	C++, C#, Java, Py, Js	Method	5	✗	Correctness
PolyHumanEval	HumanEval	164	C++, C#, Java, Py (+10 others)	Method	7	✗	Correctness
ClassEval-T	Multiple	94	C++, Java, Py	Class	34	✗	Correctness
TRACE	Geeks4Geeks	357	C++, Java, Py	Method	10	10	**Efficiency** Correctness

B Extended Evaluation

B.1 Benchmark Comparison

Table 13 compares TRACE with representative code translation benchmarks, which have traditionally centered on functional correctness. Benchmarks such as CodeNet (Puri et al., 2021), Avatar (Ahmad et al., 2021), Transcoder-Test (Lachaux et al., 2020), G-TransEval (Jiao et al., 2023), PolyHumanEval (Tao et al., 2024), and ClassEval-T (Xue et al., 2025) provide only correctness-oriented tests and lack mechanisms to capture efficiency differences. Thus, they may fail to reveal runtime efficiency differences that often remain hidden under small-scale evaluation. In contrast, TRACE augments each task with stress tests that exercise computational complexity, and selectively retains efficiency-critical tasks to ensure meaningful differentiation.

B.2 Granular Performance Comparison

To complement the overall results in the main paper, Table 15 reports a per-direction breakdown of correctness and efficiency across six translation directions, offering a fine-grained view of model behavior beyond aggregate performance.

B.3 Statistical Relationship Between Correctness and Efficiency

To understand whether functional correctness aligns with efficiency, we conduct correlation and regression analysis across all evaluated LLMs. Since incorrect translations trivially receive zero efficiency, we report conditioned efficiency scores (B_T^P, B_M^P) that only consider successful cases.

Table 14 summarizes the correlation results. Across the full set of models, correctness is moderately negatively correlated with time efficiency (Pearson $r = -0.54$, Spearman $\rho = -0.61$) and moderately positively correlated with memory efficiency ($r = 0.57$, $\rho = 0.70$). Similar trends hold for the high-performing subset (Pass $\geq 85\%$, 613 tasks in common), where correctness again shows a negative association with time efficiency ($r = -0.51$) and a positive association with memory efficiency ($r = 0.64$). These findings demonstrate that higher correctness does not necessarily imply higher efficiency.

To further quantify explanatory power, we fit Ordinary Least Squares (OLS) regressions with efficiency as the dependent variable and correctness as the predictor. The fitted models yield a negative slope for time efficiency ($\beta_1 = -0.153$, $R^2 = 0.29$) and a positive slope for memory efficiency ($\beta_1 = 0.236$, $R^2 = 0.33$). The relatively low R^2 values indicate that correctness accounts for only a limited portion of efficiency variance, confirming that efficiency reflects distinct factors beyond functional accuracy and should be evaluated as an independent dimension.

Table 14: Correlation analysis between correctness (Pass) and efficiency (Beyond); † indicates $p < 0.05$.

Correctness v.s. Efficiency	**Pearson r**	**Spearman ρ**
Full Set (28 models, 1000 tasks)		
Pass vs B_T^P	$-0.54^\dagger$	$-0.61^\dagger$
Pass vs B_M^P	$0.57^\dagger$	$0.70^\dagger$
High-Pass Set (16 models, 613 tasks)		
Pass vs B_T^{com}	$-0.51^\dagger$	$-0.43^\dagger$
Pass vs B_M^{com}	$0.64^\dagger$	$0.60^\dagger$

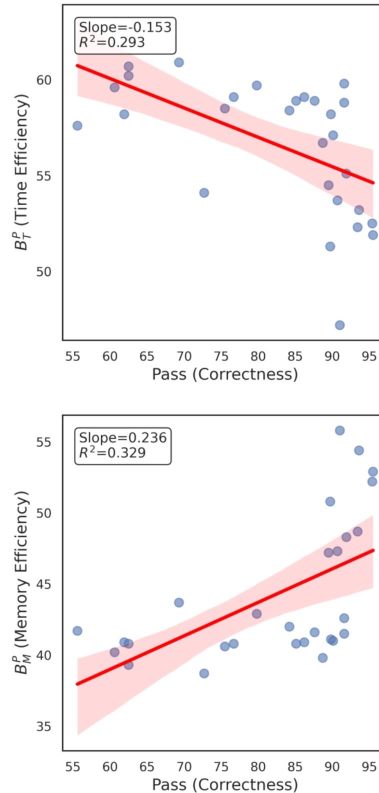

Figure 4: OLS regression of correctness-conditioned Beyond score (efficiency) on Pass rate (correctness).

B.4 Additional Case Study

To provide concrete illustrations of the inefficiency patterns in our taxonomy, we further present three representative cases. Each case corresponds to one of the high-level categories and demonstrates dramatic performance degradation under stress testing. The first case study (Figure 5) highlights a critical

algorithmic degradation. The second (Figure 6) shows the impact of suboptimal non-idiomatic data structure. The final case (Figure 7) illustrates how unnecessary resource overhead can severely harm performance.

Table 15: Granular LLM efficiency performance across six translation directions. For each direction, we report **Pass** , B_T, B_M B_T^P, and B_M^P (%).

Model	C++→Java					C++→Py					Java→C++				
	Pass	B_T	B_M	B_T^P	B_M^P	Pass	B_T	B_M	B_T^P	B_M^P	Pass	B_T	B_M	B_T^P	B_M^P
Claude-4-Think	95.6	37.1	54.0	38.8	56.5	97.2	61.3	48.4	63.1	49.8	98.4	58.8	50.7	59.8	51.6
Claude-4	97.2	66.8	62.9	68.8	64.7	97.2	25.7	13.2	26.5	13.6	91.3	45.5	56.6	49.9	62.0
Claude-3.5	98.0	45.2	51.9	46.1	52.9	97.2	64.5	50.5	66.4	51.9	92.1	53.6	46.2	58.2	50.2
DS-Reasoner	81.2	34.6	28.4	42.6	35.0	78.0	45.3	36.0	58.1	46.1	61.4	31.4	23.9	51.2	39.0
DS-V3.2	97.2	67.2	63.3	69.2	65.1	95.7	30.0	16.3	31.3	17.0	85.0	49.4	46.5	58.1	54.7
Gemini-Pro	58.8	35.9	20.7	61.1	35.2	70.2	44.8	34.7	63.8	49.4	56.7	29.3	22.2	51.6	39.1
Gemini-Flash	68.4	37.2	26.2	54.3	38.2	73.0	49.8	37.0	68.1	50.6	55.1	33.2	22.6	60.2	41.0
GPT-4o	95.2	63.9	63.0	67.1	66.2	97.2	31.6	9.7	32.5	10.0	66.9	36.1	35.7	53.9	53.3
GPT-4o-Mini	91.2	37.7	50.6	41.3	55.5	95.0	59.7	43.2	62.9	45.4	81.9	47.7	42.3	58.2	51.7
O3	94.0	53.5	34.3	57.0	36.5	95.0	63.6	49.0	66.9	51.5	68.5	39.9	25.1	58.3	36.7
O3-Mini	95.2	40.7	52.3	42.8	55.0	95.7	37.4	63.5	39.0	66.3	89.0	48.9	47.3	54.9	53.1
GPT-3.5-Turbo	95.6	43.6	54.6	45.6	57.2	92.2	61.0	45.5	66.2	49.3	91.3	56.5	52.1	61.9	57.0
CL-7B	57.2	33.8	22.0	59.0	38.5	67.4	44.2	30.9	65.7	45.8	47.2	24.9	19.9	52.7	42.0
CL-7B-Inst	75.2	39.1	28.6	51.9	38.0	82.3	53.5	39.2	65.1	47.6	70.9	43.1	33.2	60.8	46.9
CL-13B	62.8	33.7	22.1	53.7	35.2	68.1	43.0	32.9	63.1	48.4	36.2	19.8	18.4	54.8	50.9
CL-13B-Inst	73.2	40.7	26.0	55.7	35.6	80.9	53.0	39.8	65.6	49.3	77.2	43.7	34.9	56.7	45.2
CL-34B	66.4	34.9	25.4	52.6	38.3	62.4	42.6	30.7	68.3	49.2	14.2	9.5	6.8	67.2	47.9
CL-34B-Inst	65.6	36.6	24.1	55.8	36.8	83.0	54.8	43.4	66.0	52.2	50.4	30.4	29.2	60.4	58.0
DSC-6.7B	76.8	40.8	30.1	53.1	39.2	87.9	56.6	41.3	64.4	47.0	75.6	45.2	33.2	59.8	43.9
DSC-6.7B-Inst	90.4	50.9	33.7	56.3	37.3	89.4	58.4	44.9	65.4	50.2	74.0	42.4	34.2	57.3	46.1
DSC-33B	88.8	47.9	37.1	53.9	41.8	87.2	56.8	42.3	65.1	48.5	70.1	37.1	28.2	53.0	40.2
DSC-33B-Inst	95.2	54.8	36.5	57.5	38.3	93.6	60.6	46.9	64.7	50.1	76.4	41.3	27.2	54.0	35.6
QC-7B	89.6	48.1	31.9	53.7	35.6	92.9	58.3	46.0	62.8	49.5	84.3	48.0	36.5	57.0	43.3
QC-7B-Inst	94.0	42.5	33.0	45.2	35.1	92.2	64.3	44.8	69.7	48.6	85.8	48.5	39.2	56.6	45.7
QC-14B	91.6	52.0	33.3	56.7	36.3	94.3	64.4	46.4	68.2	49.1	88.2	51.6	45.6	58.5	51.7
QC-14B-Inst	96.0	54.3	36.5	56.5	38.1	97.2	63.3	46.5	65.1	47.9	81.9	46.2	37.4	56.5	45.7
QC-32B	92.0	52.1	34.9	56.6	37.9	91.5	59.5	43.9	65.0	48.0	84.3	46.3	33.7	54.9	40.0
QC-32B-Inst	95.2	56.2	37.7	59.0	39.6	93.6	63.8	44.7	68.1	47.7	83.5	50.3	40.6	60.3	48.7
Average	84.9	45.8	37.7	54.0	43.6	87.4	52.6	39.7	60.6	45.7	72.8	41.4	34.6	57.0	47.2

Model	Java→Py					Py→C++					Py→Java				
	Pass	B_T	B_M	B_T^P	B_M^P	Pass	B_T	B_M	B_T^P	B_M^P	Pass	B_T	B_M	B_T^P	B_M^P
Claude-4-Think	98.6	65.4	46.9	66.3	47.6	86.7	52.0	44.4	59.9	51.2	94.5	39.9	52.6	42.2	55.6
Claude-4	98.0	27.0	16.1	27.6	16.4	81.6	43.4	47.5	53.2	58.1	90.3	61.1	58.1	67.7	64.3
Claude-3.5	96.6	61.2	45.0	63.3	46.6	85.7	46.1	44.1	53.8	51.4	96.6	39.5	54.6	40.9	56.6
DS-Reasoner	80.4	53.8	36.9	67.0	45.9	57.1	32.8	19.3	57.5	33.7	68.2	38.5	23.5	56.4	34.4
DS-V3.2	96.6	33.0	14.9	34.2	15.5	66.3	30.4	31.7	45.8	47.7	95.3	65.6	63.7	68.8	66.8
Gemini-Pro	73.0	47.6	35.8	65.3	49.0	42.9	27.3	10.0	63.8	23.3	66.5	38.9	22.8	58.4	34.3
Gemini-Flash	79.1	51.3	35.0	64.8	44.2	29.6	18.1	7.9	61.0	26.5	57.2	32.8	20.9	57.3	36.5
GPT-4o	97.3	31.0	15.4	31.9	15.9	64.3	35.2	32.5	54.8	50.6	96.6	66.7	64.0	69.1	66.3
GPT-4o-Mini	97.3	63.7	41.7	65.5	42.9	66.3	35.4	30.8	53.3	46.5	94.1	39.2	51.8	41.7	55.1
O3	93.9	61.0	46.5	65.0	49.5	52.0	26.9	17.2	51.7	33.0	86.9	46.9	31.8	54.0	36.6
O3-Mini	95.3	58.4	46.1	61.3	48.4	62.2	33.0	32.2	53.0	51.7	94.1	39.7	54.0	42.2	57.4
GPT-3.5-Turbo	99.3	58.8	45.7	59.2	46.0	83.7	47.4	49.5	56.7	59.2	94.1	41.4	53.5	44.0	56.8
CL-7B	70.9	46.7	32.4	65.8	45.6	54.1	32.1	22.1	59.4	40.8	69.1	34.6	25.3	50.1	36.6
CL-7B-Inst	81.8	53.7	38.0	65.7	46.5	53.1	34.2	15.8	64.4	29.8	79.7	42.7	27.7	53.6	34.8
CL-13B	70.3	46.6	31.8	66.4	45.2	33.7	20.5	11.5	61.0	34.3	72.0	43.1	25.5	59.8	35.5
CL-13B-Inst	84.5	56.5	39.2	66.9	46.4	58.2	32.8	26.5	56.4	45.5	80.5	44.7	27.0	55.5	33.5
CL-34B	75.7	50.7	33.5	67.0	44.3	19.4	13.9	9.6	71.6	49.5	64.8	30.6	24.4	47.2	37.7
CL-34B-Inst	83.1	56.3	40.1	67.7	48.3	45.9	30.0	20.6	65.3	44.8	76.3	43.3	27.3	56.8	35.8
DSC-6.7B	85.8	58.7	40.1	68.4	46.7	74.5	50.0	37.7	67.1	50.6	78.8	42.8	29.6	54.2	37.6
DSC-6.7B-Inst	93.9	62.1	43.6	66.1	46.4	60.2	36.4	22.5	60.5	37.5	92.4	50.3	31.9	54.5	34.5
DSC-33B	86.5	58.2	39.0	67.3	45.1	75.5	48.6	30.5	64.3	40.3	87.3	47.0	32.9	53.8	37.7
DSC-33B-Inst	96.6	64.3	46.1	66.6	47.7	73.5	40.1	30.5	54.6	41.6	91.5	47.9	33.6	52.3	36.7
QC-7B	93.9	63.5	40.2	67.7	42.8	68.4	38.4	26.1	56.2	38.2	92.8	45.8	32.5	49.3	35.0
QC-7B-Inst	92.6	64.9	47.4	70.1	51.2	74.5	43.2	28.3	58.0	37.9	91.9	49.6	32.1	54.0	34.9
QC-14B	95.9	63.9	44.9	66.6	46.7	77.6	44.6	30.6	57.6	39.4	94.9	48.2	32.8	50.8	34.6
QC-14B-Inst	97.3	65.8	46.0	67.6	47.3	70.4	42.4	29.4	60.2	41.7	94.1	53.0	37.7	56.4	40.1
QC-32B	89.9	59.3	41.6	66.0	46.3	63.3	37.2	30.0	58.8	47.4	91.1	50.4	34.5	55.4	37.8
QC-32B-Inst	96.6	33.8	68.2	35.0	70.6	64.3	26.7	31.7	41.5	49.3	95.3	49.3	37.3	51.7	39.1
Average	89.3	54.2	39.2	61.2	44.1	62.3	35.7	27.5	57.9	42.9	85.2	45.5	37.3	53.5	43.0

Java Source Code

```java
static int findS ( int s ) {
  int sum = 0 ;
  for ( int n = 1 ; sum < s ; n ++ ) {
    sum += n * n ;
    if ( sum == s )
        return n ;
  }
  return - 1 ;
}
```

Efficient Python Translation (Claude-Sonnet-4)

```python
# Execution Time  0.02 s
# Peak Memory     9.00 MB
def findS(s):
    sum = 0
    n = 1
    while sum < s:
        sum += n * n
        if sum == s:
            return n
        n += 1
    return -1
```

Inefficient Python Translation (CodeLlama-13B-Instruct-hf)

```python
# Execution Time  115.86 s
# Peak Memory       9.00 MB
def findS(s):
    sum = 0
    for n in range(1, s + 1): # Algorithm Discrepancy
        sum += n * n
        if sum == s:
            return n
    return -1
```

Figure 5: A case study of Algorithm Implementation Discrepancy. The inefficient translation mistakenly converts an early-exit loop into one that iterates over the entire potential range, which degrades the original algorithm's time complexity and causes a catastrophic time slowdown of over 6400×.

Python Source Code

```python
def findSubarraySum ( arr , n ) :
    res = 0
    m = dict ( )
    for i in range ( n ) :
        Sum = 0
        for j in range ( i , n ) :
            Sum += arr [ j ]
            m [ Sum ] = m.get ( Sum , 0 ) + 1
    for x in m :
        if m [ x ] == 1 :
            res += x
    return res
```

Efficient C++ Translation (Qwen2.5-Coder-7B-Instruct)

```cpp
// Execution Time  2.09 s
// Peak Memory    14.60 MB
int findSubarraySum(int arr[], int n) {
    int res = 0;
    unordered_map<int, int> m;
    for (int i = 0; i < n; i++) {
        int Sum = 0;
        for (int j = i; j < n; j++) {
            Sum += arr[j];
            m[Sum]++;
        }
    }
    for (auto x : m) {
        if (x.second == 1) {
            res += x.first;
        }
    }
    return res;
}
```

Inefficient C++ Translation (CodeLlama-34B-hf)

```cpp
// Execution Time  14.40 s
// Peak Memory     16.50 MB
int findSubarraySum ( int arr [ ] , int n ) {
    int res = 0;
    map < int , int > m; // Language-Construct Misalignment
    for ( int i = 0 ; i < n ; i ++ ) {
        int Sum = 0;
        for ( int j = i ; j < n ; j ++ ) {
            Sum += arr [ j ];
            m [ Sum ] ++;
        }
    }
    for ( auto x : m ) {
        if ( x.second == 1 ) {
            res += x.first;
        }
    }
    return res;
}
```

Figure 6: A case study of Language Construct Mismatch. The inefficient translation fails to adopt the hash-based map in C++, relying on the sub-optimal tree-based map instead. This idiomatic misalignment degrades the expected constant-time lookups to logarithmic-time operations, causing a significant slowdown of over 6.9×.

C++ Source Code

```cpp
bool isPrime ( int p ) {
  long long checkNumber = pow ( 2, p ) - 1;
  long long nextval = 4 % checkNumber;
  for ( int i = 1; i < p - 1; i ++ )
    nextval = ( nextval * nextval - 2 ) % checkNumber;
  return ( nextval == 0 );
}
```

Efficient Java Translation (Deepseek-V3.2)

```java
// Execution Time  0.16 s
// Peak Memory    46.50 MB
public static boolean isPrime(int p) {
    long checkNumber = (long) (Math.pow(2, p) - 1);
    long nextval = 4 % checkNumber;
    for (int i = 1; i < p - 1; i++) {
        nextval = (nextval * nextval - 2) % checkNumber;
    }
    return (nextval == 0);
}
```

Inefficient Java Translation (Deepseek-Coder-6.7B-Instruct)

```java
// Execution Time  8.35 s
// Peak Memory   1403.50 MB
public static boolean isPrime(int p) {
    BigInteger checkNumber = BigInteger.valueOf(2).pow(p).subtract(BigInteger.ONE); //
    ↪  Resource-Overhead Inefficiency
    BigInteger nextval = BigInteger.valueOf(4).mod(checkNumber);
    for (int i = 1; i < p - 1; i ++) {
        nextval = nextval.multiply(nextval).subtract(BigInteger.valueOf(2)).mod(checkNumber);
    }
    return nextval.compareTo(BigInteger.ZERO) == 0;
}
```

Figure 7: A case study of Resource Management Inefficiency. The inefficient translation replaces primitive `long`-based arithmetic with Java's heavyweight `BigInteger` class. While functionally correct, this introduces substantial object creation and garbage-collection overhead, resulting in a severe slowdown of over 53.9× and a memory consumption increase of over 30.2×.

C Prompt and Script Design

C.1 Prompt for LLM-based Code Translation

We present the prompt templates used in our study. Four settings are considered: 1) *zero-shot*, which directly asks the LLM to translate code without additional guidance (Figure 8); 2) *perf-zero-shot*, which augments the zero-shot setting with explicit efficiency-oriented system instructions (Figure 9); 3) *perf-few-shot*, which provides the LLM with two efficient code translation examples, which are drawn from other problems in TRACE following the same translation direction (Figure 10); and 4) *perf-self-refine*, which includes the execution feedback of the translation produced in the previous zero-shot round and asks the LLM for further refinement or optimization (Figure 11).

C.2 Prompt for Stress Test Generation

Figure 12 presents the prompt template used to guide the LLM in generating stress test input synthesizers. The prompt is structured with five key sections: it defines the `Task` by providing the source code, specifies `Requirements` for creating large and worst-case inputs, sets `Output Constraints` for both type and performance, and enforces a strict `Output Format`. A key component is the `Efficiency Example` section, which provides few-shot examples (two in our work) of the top-performing synthesizers and the execution profiles of the sampled test inputs. This feedback-driven approach is crucial for progressively synthesizing diverse and challenging test inputs to expose latent efficiency bottlenecks in LLM-translated code.

C.3 Testscript Template

Figures 13, Figure 14, and Figure 15 demonstrate the designed testscript templates for C++, Java, and Python, which are engineered for automatic evaluation of code translations.

These testscripts are structured with specific placeholders to programmatically embed external code snippets during evaluation. Specifically, the `TO_FILL_FUNC` marker is replaced with the LLM-generated translation; the `TO_FILL_GOLD` marker is replaced with the ground-truth implementation from the benchmark; and the `TEST_SUITE_FILL` section is injected with the full suite of tests. We implemented a helper function, `areEquivalent`, to perform comparison by serializing the outputs to validate functional equivalence. During evaluation, the testscript executes and compares each test on both the LLM-generated and ground-truth translations. Finally, the script outputs a parsable summary of execution results for further analysis.

System Prompt

````
You are a code translation expert proficient in C++, Java, and Python. You translate code
↪  accurately while preserving its functionality.
````

User Prompt

````
### Task Description
Given the SRC_LANG code:
```SRC_LANG
SRC_CODE_FILL
```
Please translate the above SRC_LANG code to TGT_LANG code.

### Output Format
Wrap your output in a Markdown code block using ``` backticks, and end your response with <|END|>,
↪  e.g.:
```TGT_LANG
...
```
<|END|>
````

Figure 8: The prompt template for the zero-shot setting.

System Prompt

````
You are a code translation expert proficient in C++, Java, and Python. You translate code
↪  accurately while preserving its functionality.
You are also a code optimization expert, ensuring the translated code is efficient and adheres to
↪  best practices.
````

User Prompt

````
### Task Description
Given the SRC_LANG code:
```SRC_LANG
SRC_CODE_FILL
```
Please translate the above SRC_LANG code to TGT_LANG code.

### Output Format
Wrap your output in a Markdown code block using ``` backticks, and end your response with <|END|>,
↪  e.g.:
```TGT_LANG
...
```
<|END|>
````

Figure 9: The prompt template for the perf-zero-shot setting.

System Prompt

```
You are a code translation expert proficient in C++, Java, and Python. You translate code
↪  accurately while preserving its functionality.
You are also a code optimization expert, ensuring the translated code is efficient and adheres to
↪  best practices.
```

User Prompt

````
### Task Description
Given the SRC_LANG code:
```SRC_LANG
SRC_CODE_FILL
```
Please translate the above SRC_LANG code to TGT_LANG code.

### Translation Examples
Below are some examples of efficient code translations from SRC_LANG to TGT_LANG.

#### Example-1
Given the SRC_LANG code:
```SRC_LANG
SRC_CODE_FILL_EXAMPLE_1
```
The correct and efficient TGT_LANG code translation is as below:
```TGT_LANG
EFFI_TGT_CODE_FILL_1
```
<|END|>

#### Example-2
Given the SRC_LANG code:
```SRC_LANG
SRC_CODE_FILL_EXAMPLE_2
```
The correct and efficient TGT_LANG code translation is as below:
```TGT_LANG
EFFI_TGT_CODE_FILL_2
```
<|END|>

### Output Format
Wrap your output in a Markdown code block using ``` backticks, and end your response with <|END|>,
↪  e.g.:
```TGT_LANG
...
```
<|END|>
````

Figure 10: The prompt template for the perf-few-shot setting.

System Prompt

````
You are a code translation expert proficient in C++, Java, and Python. You translate code
↪  accurately while preserving its functionality.
You are also a code optimization expert, ensuring the translated code is efficient and adheres to
↪  best practices.
````

User Prompt

````
### Task
Given the SRC_LANG code:
```SRC_LANG
SRC_CODE_FILL
```
Please translate the above SRC_LANG code to TGT_LANG code.

### Reference Translation
Below is a reference TGT_LANG translation for the given SRC_LANG code:
```TGT_LANG
REF_TGT_CODE_FILL
```
<|END|>

The execution profile of the reference TGT_LANG translation is as below:
```markdown
**Test Result** (0.0 means all tests failed, and 1.0 means all tests passed)
TEST_RESULT_FILL

**Execution Feedback**
EXEC_FEEDBACK_FILL

**Execution Time (seconds)**
EXEC_TIME_FILL

**Execution Memory (MB)**
EXEC_MEM_FILL
```

If you think the reference TGT_LANG translation is not correct or efficient, please try to improve
↪  it. Otherwise, output the original reference TGT_LANG translation without any changes.

### Output Format
Wrap your output in a Markdown code block using ``` backticks, and end your response with <|END|>,
↪  e.g.:
```TGT_LANG
...
```
<|END|>
````

Figure 11: The prompt template for the perf-self-refine setting.

Prompt for Stress Test Input Generation

```
### Task
You are tasked with generating a challenging test input for the following `SRC_LANG` function:
```SRC_LANG
SRC_CODE_FILL
```

Requirements
Please carefully analyze the logic and complexity of the above source code. For each parameter in
↪ the function signature (`PARAM`):
1. Implement a Python function `generate_param_X()` for each `X`.
2. Each `generate_param_X()` must return one valid value.
3. The joint input should expose computational limits, trigger worst-case behaviour, and fully
↪ stress the function logic.

Output Constraint
Ensure the generated test input runs within 10 seconds and 4 GB peak memory. All values must
↪ satisfy the type constraints in `SRC_LANG`.
1. Each input matches the expected type (e.g. `int`, `float`, `double`, `str`, `List[int]`,
↪ `Map<String,Integer>`,...).
2. For composite types (e.g. `vector<vector<T>>`, `pair<T1,T2>`), ensure all sub-elements conform
↪ to their declared types.

Output Format
Implement each `generate_param_X()` and print them in order with a separator `|SEP|`. The
↪ implementation should be runnable and enclosed in a ```python...``` code block, e.g.:

```Python
import ...
def generate_param_X():
    ...
def generate_param_Y():
    ...

print(generate_param_X())
print("|SEP|")  # separator
print(generate_param_Y())
```

Efficiency Example
Below are two validated generators and their execution profiles from their sampled values. Study
↪ their patterns and create more stressful test generators by mutating size, distribution,
↪ ordering, and edge-case choices while keeping the same output format.

- Example of Test Input Generator
```Python
GENERATOR_FILL
```

- Sampled Test Input Value
```
VALUE_FILL
```
- Efficiency Profile on SRC_LANG Code
    [Elapsed Time] TIME_FILL Seconds
    [Peak Memory] MEM_FILL MB

...(One More Example)
```

Figure 12: The prompt template used to elicit stress test inputs.

C++ Testscript Template

```cpp
#include <iostream>
#include <cstdlib>
#include <string>
#include <vector>
#include <fstream>
#include <iomanip>
#include <bits/stdc++.h>
//TO_FILL_IMPORT

using namespace std;
//TO_FILL_FUNC
// e.g., int addOne ( int x ) {...}

//TO_FILL_GOLD
// e.g., int f_gold ( int x ) {...}

class Test {
    // Additional helper methods here ...

    public:
      template <typename T1, typename T2> static bool AreEquivalent(T1 o1, T2 o2) {
        return serialize_obj_(o1) == serialize_obj_(o2);
      }
    public:
      static void Start() {
        int total_num = 0;
        int pass_num = 0;
        //TEST_SUITE_FILL

        std::cout << "|OUTPUT| total " << total_num << " | passed " << pass_num << std::endl;
      }
};

int main() {
  Test::Start();
  return 0;
}
```

Figure 13: The C++ testscript template used in our work.

Java Testscript Template

```java
import java.util.*;
import java.util.stream.*;
import java.lang.*;
import java.io.*;
import java.lang.reflect.Array;
//TO_FILL_IMPORT

class MAIN {
    // Additional helper methods here ...

  public static boolean areEquivalent(Object o1, Object o2) {
    return serializeObj(o1).equals(serializeObj(o2));
  }

//TO_FILL_FUNC
// e.g. public static int addOne(int x) { ... }

//TO_FILL_GOLD
// e.g. static int f_gold ( int x ) { ... }

  public static void start() {
    int total_num = 0;
    int pass_num = 0;
    //TEST_SUITE_FILL

    System.out.println("|OUTPUT| total " + total_num +  " | passed " + pass_num );
  }
  public static void main(String[] args) {
    start();
  }
}
```

Figure 14: The Java testscript template used in our work.

Python Testscript Template

```python
from collections import *
from typing import *
#TO_FILL_IMPORT

# Additional helper methods here ...

def are_equivalent(o1, o2) -> bool:
    return serialize_obj(o1) == serialize_obj(o2)

#TO_FILL_FUNC
# e.g. def addOne(x): ...

#TO_FILL_GOLD
# e.g. def f_gold ( x ): ...

def start():
    total_num = 0
    pass_num = 0
    #TEST_SUITE_FILL

    print(f"|OUTPUT| total {total_num} | passed {pass_num}")

if __name__ == "__main__":
    start()
```

Figure 15: The Python testscript template used in our work.