TRACE: Evaluating Execution Efficiency of LLM-Based Code Translation

Anonymous ACL submission

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 degradations often overlooked by small-scale tests. Using TRACE, we conduct an extensive evaluation of 28 representative LLMs and highlight several key insights: 1) Correctness is not a reliable proxy for efficiency: the correctness leader *Claude-4-think* achieves only mid-level 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 exhibit pronounced inefficiency, distributed across algorithmic faults (11.9%), language construct mismatches (66.4%), and resource mismanagement (21.7%). 3) Inference-time prompt strategies bring only modest improvements, suggesting that current LLMs lack intrinsic efficiency awareness. Together, our results establish efficiency as an essential dimension of code translation and position TRACE as a principled foundation for efficiency-oriented evaluation.

1 Introduction

Automatic code translation aims to migrate 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 code translation has become more pressing.

The remarkable proficiency of Large Language Models (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.

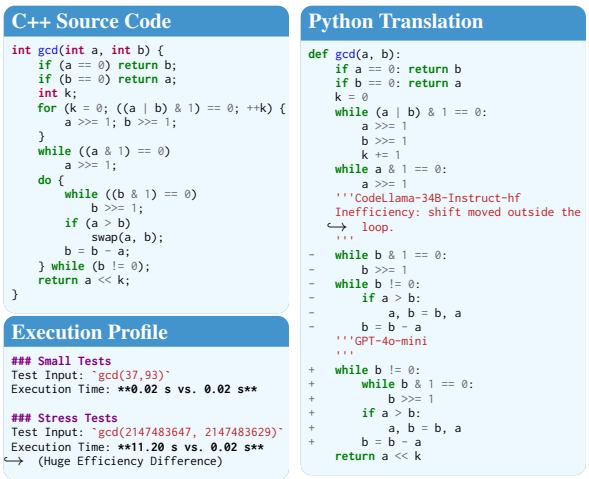

Figure 1: A motivating example illustrating the critical discrepancy between functional correctness and execution efficiency in LLM-based code translation.

Despite these advances in correctness, an equally critical dimension remains understudied: *execution efficiency* (Niu et al., 2024; Vartziotis et al., 2024). Efficiency dictates how a translated program performs in practice, directly influencing responsiveness, scalability, and operational costs. Crucially, even functionally correct translations may prove impractical when they employ suboptimal implementations that lead to severe runtime bottlenecks.

Figure 1 illustrates a code translation 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 translating 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 more demanding test inputs (e.g., a: 2147483647, b: 2147483629) drawn 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 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 that incur substantial execution-time or memory overheads*. However, evaluation protocols such as the small-scale tests in TRANSCODER-TEST mainly focus on correctness and fail to expose efficiency degradations under more demanding workloads. Hence, it is necessary to disentangle fidelity from mere correctness and explicitly account for efficiency.

To address the challenge, we present **TRACE** (**TRA**nslated **C**ode **E**fficiency), the first benchmark explicitly designed to evaluate efficiency in LLM-translated code. We focus on translation tasks in C++, Java, and Python, which consistently rank among the global top-3 most popular programming languages (Carbonnelle, 2025; TIOBE Software BV, 2025). Grounded in the well-established TRANSCODER-TEST benchmark, TRACE is constructed via 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 differentiate runtime efficiency. The final benchmark comprises 1,000 efficiency-critical tasks, each accompanied by 10 default correctness-check tests, 10 demanding stress tests, and 23 efficiency reference translations to support nuanced evaluation.

Using TRACE, we extensively evaluate 28 representative LLMs and derive key insights into efficiency in LLM-based code translation. First, correctness and efficiency are often misaligned. For example, the correctness leader *Claude-4-think* achieves a 95.5% pass rate but only mid-level time efficiency, outperformed by smaller open-source LLMs such as *Qwen2.5-Coder-14B-Instruct*. Besides, 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 substantially more memory-intensive than the most efficient counterparts. A manual study of 327 inefficient translation 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 but modest efficiency gains, suggesting that external guidance can mitigate but not eliminate the lack of intrinsic efficiency modeling in current LLMs. Taken together, our findings establish TRACE as the first benchmark that exposes efficiency as an orthogonal dimension to correctness 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 explicitly establish execution efficiency as a critical dimension in LLM-based code translation. 2) **Benchmark.** We present TRACE, the first benchmark specifically designed to expose and evaluate efficiency in LLM-translated code. 3) **Evaluation.** We perform a large-scale evaluation of 28 representative LLMs on TRACE, providing empirical evidence and broader insights into the interplay between correctness and efficiency in code translation. Anonymous code and data are available: `https://anonymous.4open.science/r/TRACE_ANONYMOUS-F133`.

2 Related Work

Code Translation and Benchmarks. 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 correctness through fine-tuning (He et al., 2025; Liu et al., 2023), reasoning (Macedo et al.,

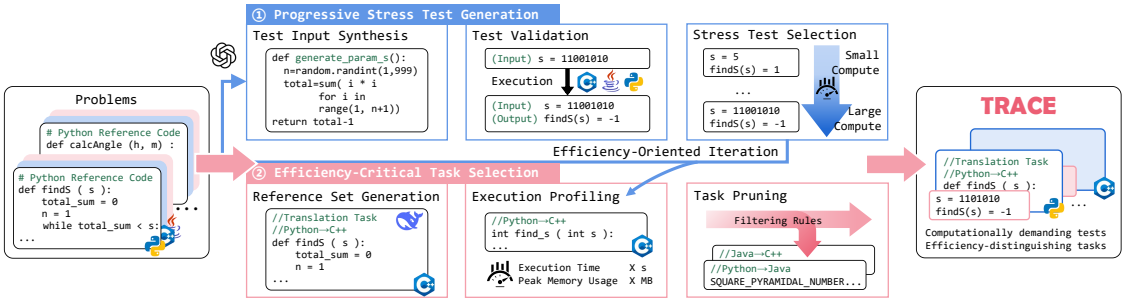

Figure 2: An overview of TRACE's LLM-driven two-stage construction pipeline.

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), ClassEval-T (Xue et al., 2025), and AlphaTrans (Ibrahimzada 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 also begun to assess efficiency in LLM-based code generation. Benchmarks such as EffiBench (Huang et al., 2024b) curate efficiency-critical tasks, EvalPerf (Liu et al., 2024) introduces stress-based differential performance evaluation, and Mercury (Du et al., 2024) formalizes runtime-aware metrics. Optimization methods such as PIE (Shypula et al., 2023), SwiftCoder (Huang et al., 2024c), and Afterburner (Du et al., 2025) adapt or fine-tune LLMs for efficient solutions.

This work specifically addresses efficiency in *code translation*, where LLMs must preserve semantic fidelity to the source code while adapting to efficient target-language constructs. Existing efficiency benchmarks, such as PIE, focus on *within-language* optimization (e.g., improving C++ implementations) and lack *aligned cross-language implementations*, making them unsuitable for assessing code translation efficiency. This gap motivates the development of a new benchmark explicitly dedicated to evaluating efficiency in code translation.

3 Benchmark Construction

Terminology Formulation. A *problem* (P) is a canonical programming challenge. For each problem, a *reference code / translation* (p) is a functionally correct solution. A *code translation task (task)* is the process 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 runtime indicators: *execution time* (ET) and *peak memory usage* (PM).

Methodology Overview. As presented in Figure 2, TRACE is constructed through an LLM-driven two-phase 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 builds upon TRANSCODER-TEST (Lachaux et al., 2020), 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 are 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 small-scale default tests.

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. Directly asking LLMs to output such test inputs is infeasible due to the context length limitation (e.g., tokenizing an array of 1M elements exceeds the 16K–32K context length limit of typical LLMs). We therefore

Algorithm 1 Progressive Stress Test Generation

1: **FUNCTION** GenerateStressTests($P, \mathcal{M}, I_{max}, K$)
2: $\mathcal{P} \leftarrow$ GetBasePrompt(P)
3: $\mathcal{T}_s, \mathcal{E} \leftarrow \emptyset, \emptyset$
4: **for** $i \leftarrow 1$ **to** I_{max} **do**
5: // — Stage 1.1: Test Input Synthesis —
6: $\mathcal{P}_{iter} \leftarrow$ EnrichPrompt($\mathcal{P}, \mathcal{E}$)
7: $\mathcal{S} \leftarrow \mathcal{M}(\mathcal{P}_{iter})$
8: // — Stage 1.2: Test Validation —
9: $\mathcal{V} \leftarrow$ ExecuteSynthesizers($\mathcal{S}$)
10: $\mathcal{T}_{cand} \leftarrow$ Validate($\mathcal{V}$)
11: // — Stage 1.3: Stress Test Selection —
12: $\mathcal{T}_{time} \leftarrow$ SelectTopK($\mathcal{T}_{cand}$, K, `time`)
13: $\mathcal{T}_{mem} \leftarrow$ SelectTopK($\mathcal{T}_{cand} \setminus \mathcal{T}_{time}$, K, `mem`)
14: $\mathcal{T}_{iter} \leftarrow \mathcal{T}_{time} \cup \mathcal{T}_{mem}$
15: // — Stage 1.4: Efficiency-Oriented Iteration —
16: $\mathcal{T}_{s,\text{prev}} \leftarrow \mathcal{T}_s$
17: $\mathcal{T}_s \leftarrow$ UpdateGlobalPool($\mathcal{T}_s, \mathcal{T}_{iter}, K$)
18: **if** $\mathcal{T}_s = \mathcal{T}_{s,\text{prev}}$ **then break**
19: **end if**
20: $\mathcal{E} \leftarrow$ ExtractExamples($\mathcal{T}_s$)
21: **end for**
22: **return** $\mathcal{T}_s$

utilize a *synthesis-based* approach (Lines 5-7) (Liu et al., 2024; Yang et al., 2025), where an advanced 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. Only the correct translations are retained and then 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 task-level 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

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

exhibit negligible performance variance. This is measured by requiring the coefficient of variation (CV = std/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* with stochastic decoding (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.

3.5 Benchmark Statistics

Table 1 summarizes the key characteristics of TRACE, and Appendix Table 12 further compares TRACE with existing code translation benchmarks.

Overall, the final TRACE benchmark contains 357 problems, yielding a total of 1,000 efficiency-critical translation tasks across six 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. In addition, 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, indicating heterogeneous efficiency regimes that enable fine-grained differentiation among translations.

Compared with existing benchmarks, TRACE introduces two key innovations. First, it integrates stress tests to progressively exercise computational complexity, thereby revealing inefficiencies overlooked by small-scale tests. Second, it selectively retains only efficiency-critical tasks via rigorous filtering, ensuring that every included task provides meaningful differentiation 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), Anthropic (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 11 illustrates more model details.

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

For correctness, we adopt the *Pass Rate* (Pass), defined as the proportion of tasks for which a model's translation passes all unit tests.

For efficiency, we compare each candidate translation c against TRACE's efficiency reference set $\mathcal{R}$. For each metric $X \in \{\text{ET}, \text{PM}\}$, we define

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
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-chat	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
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

the reference spectrum as $\mathcal{R}_X = \{X(r) : r \in \mathcal{R}\}$ and denote the candidate's measured performance by $x_c = X(c)$. Following (Du et al., 2024), efficiency is quantified using the *Beyond* score, which normalizes a translation's ET or PM relative to task-specific reference bounds:

$$B_X(c) = \frac{\max \mathcal{R}_X - \text{clip}\,(x_c, \min \mathcal{R}_X, \max \mathcal{R}_X)}{\max \mathcal{R}_X - \min \mathcal{R}_X} \times 100\%$$

where $\text{clip}(a, \ell, u)$ truncates a to $[\ell, u]$. Higher B_X indicates better efficiency. We report two variants: B_X, averaged over all translations with incorrect outputs scored as 0, and B_X^P, computed only over correct translations to isolate efficiency when correctness is ensured.

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 ET or 4096 MB PM) are marked as incorrect and excluded from evaluation. Correct translations are executed on stress tests to measure ET and PM, with each metric reported as the arithmetic mean of five runs. These measurements are then converted into Beyond scores. Further experimental details are provided in Appendix A.3.

5 Evaluation

We conduct extensive evaluation through three lenses: 1) end-to-end model performance, 2) a taxonomy-driven characterization of translation inefficiencies, and 3) an assessment 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 14.

Correctness is not a proxy for efficiency. From Table 2, we find that higher correctness does not necessarily lead to higher efficiency. For instance, *Claude-4-think* attains the highest Pass rate (95.5) but only mid-level 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.

Table 3: Taxonomy of inefficiencies 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***.	

Reasoning and scaling do not reliably improve efficiency. Across proprietary and open-source models, neither reasoning-oriented variants nor larger scale 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-chat* 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 *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 still 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, Table 14). However, their time efficiency differs sharply (B_T: 54.2 vs. 45.5), and this gap remains even when considering only correct translations

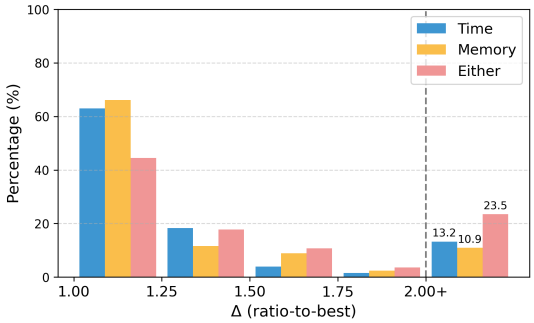

Figure 3: Execution profile distribution of translations.

(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

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

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 4 quantifies the severity of inefficiencies across categories. Algorithm-related errors 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: 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
DeepSeek-chat					
zero-shot	91.9	50.7	44.5	55.1	48.4
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	45.6	52.3	48.8
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 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., DeepSeek-chat, 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 show that prompt-based strategies can alleviate inefficiencies to a limited extent but cannot fundamentally eliminate them, highlighting the need for training methods that explicitly encode programming semantics and efficiency principles.

6 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 is a weak predictor of 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 training paradigms that endow LLMs with intrinsic efficiency awareness for code translation.

Limitations

The following limitations define the scope of our findings 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 computational efficiency, we acknowledge that they only partially capture real-world performance. Factors such as I/O latency, concurrency handling, compilation overhead, and energy consumption may also substantially influence efficiency in practice. Future extensions of our benchmark should therefore consider these broader and domain-specific aspects to provide a more comprehensive view of efficiency.

Scope of Programming Languages. Our study focuses on C++, Java, and Python, which consistently rank as the global top-3 most widely used languages according to established popularity indices (Carbonnelle, 2025; TIOBE Software BV, 2025). Concentrating on these programming languages provides strong representativeness and supports the significance of our findings. Meanwhile, we acknowledge that excluding other programming languages may pose risks to broader generalization. Efficiency characteristics may differ in languages with distinct paradigms (e.g., Go's goroutine model or Rust's ownership system). Extending TRACE to additional languages would strengthen our conclusions and further validate the applicability.

Scope of Programming Problems. The problems in TRACE are based on the TRANSCODER-TEST benchmark, whose tasks were originally collected from the GeeksForGeeks (GeeksforGeeks, 2025) platform. These problems primarily target method-level algorithmic and data-structure-oriented tasks. This evaluation setting differs from real-world scenarios, where file-level code generation or translation (Jimenez et al., 2023; Ibrahimzada et al., 2025; Xue et al., 2025) often involves classes and external dependencies such as APIs, databases, and I/O operations. We acknowledge that the current focus on method-level granularity may simplify the translation context. Nevertheless, since methods form the atomic units of higher-level modules, our benchmark still captures essential computational behaviors. Future work could incorporate larger-scale translation settings to improve external validity.

Validity of Internal Implementations. Our evaluation is also subject to internal risks. First, the efficiency spectrum of each translation task is constructed from a pool of LLM-generated solutions. While this pool offers a broad and representative sample of current model capabilities, it may not contain the theoretically optimal or human-expert implementation (Peng et al., 2025; Huang et al., 2024b). Therefore, our efficiency assessments are relative, which are measured against the best solution among LLMs rather than the optimum. Second, all models are evaluated using greedy decoding to ensure consistency and reproducibility. Although alternative strategies such as stochastic sampling could occasionally yield more efficient translations, exploring such variants lies beyond our current scope. Future research should investigate how decoding choices influence efficiency outcomes and incorporate expert-written implementations to mitigate potential risks and strengthen the robustness of the conclusions.

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 2016 ieee/acm 38th international conference on software engineering companion (icse-c). *IEEE, 756ś758*.

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. *arXiv preprint arXiv:2110.06773*.

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*.

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.

Jun Yang, Cheng-Chi Wang, Bogdan Alexandru Stoica, and Kexin Pei. 2025. Synthesizing performance constraints for evaluating and improving code efficiency. *arXiv preprint arXiv:2505.23471*.

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 pretrained 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 Details of TRACE's Stress Tests

Table 6 presents the average execution time (ET) and peak memory usage (PM) for the obtained top-10 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. We set the maximum number of iterations to five, as further iterations began to yield tests so computationally demanding that they caused some ground-truth implementations from TRANSCODER-TEST to exceed the predefined ET/PM limits (180 seconds/4096 MB), indicating diminishing returns.

Table 6: 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 7 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 tasks, the stress tests still demonstrate a significant performance 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 less demanding tests from the original benchmark.

A.2 Details of Task Pruning

Table 8 summarizes the task pruning process used to construct the final TRACE benchmark. 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 7: 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**

Table 8: 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

A.3 Experiment Environment

All the experiments were conducted in a controlled environment, as summarized in Table 9 and Table 10, to ensure consistency and reproducibility. The hardware platform consists of Intel Xeon Platinum 8468V processors (192 cores, 3.8 GHz), two NVIDIA H200 NVL GPUs (140 GB each), and 512 GB RAM, running Ubuntu 22.04 LTS with Linux kernel 6.8.0-60. Software dependencies included g++ 13.1.0, OpenJDK 17.0.15, and Python 3.12. 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

Table 9: Experiment environment details.

Component	Specification
Hardware	
Operating System	Ubuntu 22.0.5 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`

Table 10: Experiment settings.

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

loops. Each execution profile was averaged over five independent runs to mitigate noise from system load and nondeterministic scheduling. 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.

A.4 Model Choice

Table 11 provides details for the 28 Large Language Models (LLMs) used in our work. These models were utilized for both the curation of the benchmark and the evaluation of the work.

Table 11: Model details, including aliases used in this paper, names, sizes, and public links.

Alias	Name	Size	Public Link
		Proprietary Models	
Claude-4-think	Claude 4 (Thinking)	–	https://www.anthropic.com/news/claude-4
Claude-4	Claude 4 (Non-thinking)	–	https://www.anthropic.com/news/claude-4
Claude-3.5	Claude 3.5 Sonnet	–	https://www.anthropic.com/news/claude-3-5-sonnet
DS-reasoner	DeepSeek Reasoner	–	https://www.deepseek.com
DS-chat	DeepSeek Chat	–	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	7B	https://github.com/facebookresearch/codellama
CL-7B-Inst	CodeLlama-7B-Instruct	7B	https://github.com/facebookresearch/codellama
CL-13B	CodeLlama-13B	13B	https://github.com/facebookresearch/codellama
CL-13B-Inst	CodeLlama-13B-Instruct	13B	https://github.com/facebookresearch/codellama
CL-34B	CodeLlama-34B	34B	https://github.com/facebookresearch/codellama
CL-34B-Inst	CodeLlama-34B-Instruct	34B	https://github.com/facebookresearch/codellama
DSC-6.7B	deepseek-coder-6.7b	6.7B	https://github.com/deepseek-ai/DeepSeek-Coder
DSC-6.7B-Inst	deepseek-coder-6.7b-Instruct	6.7B	https://github.com/deepseek-ai/DeepSeek-Coder
DSC-33B	deepseek-coder-33b	33B	https://github.com/deepseek-ai/DeepSeek-Coder
DSC-33B-Inst	deepseek-coder-33b-Instruct	33B	https://github.com/deepseek-ai/DeepSeek-Coder
QC-7B	Qwen2.5-Coder-7B	7B	https://github.com/QwenLM/Qwen2.5-Coder
QC-7B-Inst	Qwen2.5-Coder-7B-Instruct	7B	https://github.com/QwenLM/Qwen2.5-Coder
QC-14B	Qwen2.5-Coder-14B	14B	https://github.com/QwenLM/Qwen2.5-Coder
QC-14B-Inst	Qwen2.5-Coder-14B-Instruct	14B	https://github.com/QwenLM/Qwen2.5-Coder
QC-32B	Qwen2.5-Coder-32B	32B	https://github.com/QwenLM/Qwen2.5-Coder
QC-32B-Inst	Qwen2.5-Coder-32B-Instruct	32B	https://github.com/QwenLM/Qwen2.5-Coder

B Extended Evaluation

B.1 Benchmark Comparison

Table 12 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 a handful of correctness-oriented tests and lack mechanisms to capture efficiency differences. Thus, they fail to reveal runtime degradations 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. With 1,000 efficiency-critical tasks spanning C++, Java, and Python, TRACE is the first benchmark to jointly assess correctness and efficiency, offering an explicit foundation for efficiency-aware evaluation of LLM-based code translation.

B.2 Granular Performance Comparison

To complement the overall results in the main paper, Table 14 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

Table 13: 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}	$0.64^\dagger$	$0.60^\dagger$

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 13 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.

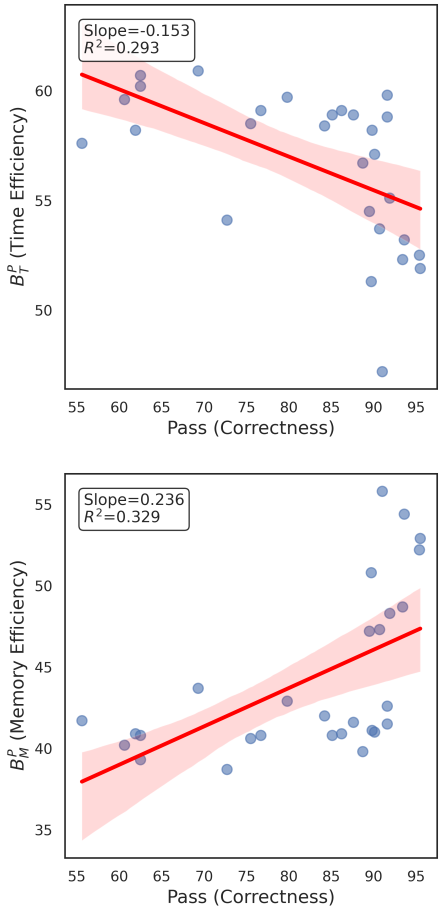

Figure 4: OLS regression of correctness-conditioned Beyond score (efficiency) on Pass rate (correctness).

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 12: 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, Python	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.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 14: Granular LLM efficiency performance across six translation directions. For each direction, we report **Pass** , $\boldsymbol{B_T}$, $\boldsymbol{B_M}$ $\boldsymbol{B_T^P}$, and $\boldsymbol{B_M^P}$ (%).

Model	**C++→Java**					**C++→Py**					**Java→C++**				
	Pass	$\boldsymbol{B_T}$	$\boldsymbol{B_M}$	$\boldsymbol{B_T^P}$	$\boldsymbol{B_M^P}$	**Pass**	$\boldsymbol{B_T}$	$\boldsymbol{B_M}$	$\boldsymbol{B_T^P}$	$\boldsymbol{B_M^P}$	**Pass**	$\boldsymbol{B_T}$	$\boldsymbol{B_M}$	$\boldsymbol{B_T^P}$	$\boldsymbol{B_M^P}$
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-chat	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	$\boldsymbol{B_T}$	$\boldsymbol{B_M}$	$\boldsymbol{B_T^P}$	$\boldsymbol{B_M^P}$	**Pass**	$\boldsymbol{B_T}$	$\boldsymbol{B_M}$	$\boldsymbol{B_T^P}$	$\boldsymbol{B_M^P}$	**Pass**	$\boldsymbol{B_T}$	$\boldsymbol{B_M}$	$\boldsymbol{B_T^P}$	$\boldsymbol{B_M^P}$
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-chat	96.6	33.0	14.9	34.2	15.5	66.3	30.4	31.7	45.8	47.7	95.3	39.7	54.0	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-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 Misalignment**. 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-Chat)

```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-Overhead 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 experiments. 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 (GPT-4o) 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 Test script Template

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

These test scripts 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 test script 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 computationally demanding stress test inputs that maximize execution time and peak memory usage.

C++ Test Script 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
//int addOne ( int x ) {...}

//TO_FILL_GOLD
//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

        // ** Example **
        // try {
        //    total_num++;
        //    int x_full_1005 = 715849402 ;
        //    auto output_1_1005 = addOne (x_full_1005);
        //    int x_gold_1005 = 715849402 ;
        //    auto output_2_1005 = f_gold(x_gold_1005);
        //    if (Test::AreEquivalent(output_1_1005, output_2_1005))
        //        pass_num++;
        //} catch (exception& e) {
        //}
        std::cout << "|OUTPUT| total " << total_num << " | passed " << pass_num << std::endl;
      }
};

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

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

Java Test Script 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

    // ** Example **
    //try {
    //    total_num++;
    //    int x_full_1005; x_full_1005 = readInt ( ... );
    //    Object output_1_1005 = addOne (x_full_1005);
    //    int x_gold_1005; x_gold_1005 = readInt ( ... );
    //    Object output_2_1005 = f_gold(x_gold_1005);
    //    if (areEquivalent(output_1_1005, output_2_1005))
    //        pass_num++;
    //} catch (Exception e) { }

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

Figure 14: The Java test script 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

    # ** Example **
    # try:
    #     total_num += 1
    #     x_full = 715849402
    #     output_1 = addOne (x_full)
    #     x_gold = 715849402
    #     output_2 = f_gold(x_gold)
    #     if are_equivalent(output_1, output_2):
    #         pass_num += 1
    # except Exception as e:
    #     pass

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

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

Figure 15: The Python test script template used in our work.