EFFIBENCH-X: A Multi-Language Benchmark for Measuring Efficiency of LLM-Generated Code

Yuhao Qing[1*] **Boyu Zhu**[2*] **Mingzhe Du**[3, 4*] **Zhijiang Guo**[5, 6†]
Terry Yue Zhuo[7, 8] **Qianru Zhang**[1] **Jie M. Zhang**[9] **Heming Cui** [1]
Siu-Ming Yiu[1] **Dong Huang**[1, 4†] **See-Kiong Ng**[4] **Luu Anh Tuan**[3]

[1]HKU [2]UCL [3]NTU [4]NUS [5]HKUST (GZ) [6]HKUST
[7]Monash University [8]CSIRO's Data61 [9]KCL

Abstract

Existing code generation benchmarks primarily evaluate functional correctness, with limited focus on code efficiency and often restricted to a single language like Python. To address this gap, we introduce EFFIBENCH-X, the first multi-language benchmark designed to measure the efficiency of LLM-generated code. EFFIBENCH-X supports Python, C++, Java, JavaScript, Ruby, and Golang. It comprises competitive programming tasks with human-expert solutions as efficiency baselines. Evaluating state-of-the-art LLMs on EFFIBENCH-X reveals that while models generate functionally correct code, they consistently underperform human experts in efficiency. Even the most efficient LLM-generated solutions (Qwen3-32B) achieve only around **62%** of human efficiency on average, with significant language-specific variations. LLMs show better efficiency in Python, Ruby, and JavaScript than in Java, C++, and Golang. For instance, DeepSeek-R1's Python code is significantly more efficient than its Java code. These results highlight the critical need for research into LLM optimization techniques to improve code efficiency across diverse languages. The dataset and evaluation infrastructure are submitted and available at `https://github.com/EffiBench/EffiBench-X.git` and `https://huggingface.co/datasets/EffiBench/effibench-x`.

1 Introduction

Code efficiency is becoming a critical concern for LLM-generated code as these models are more widely adopted. While many LLMs and agent frameworks successfully produce correct solutions [16, 36, 75, 39, 30], their outputs often incur substantial resource overhead, compromising performance and feasibility in settings demanding high efficiency, such as mobile phones, embedded systems, and latency-sensitive cloud environments [48, 32, 66, 31, 33]. This underscores the need for benchmarks that measure not just correctness but also runtime efficiency [56, 66, 48].

Responding to this critical need, several benchmarks have recently emerged to measure the efficiency of LLM-generated code. EffiBench [32] and Mercury [23] use LeetCode problems and their solutions to evaluate Python code based on runtime and memory. Moving beyond LeetCode, EvalPerf [48] and ENAMEL [56] assess efficiency using subsets of existing benchmarks HumanEval [16] and MBPP [11], while PIE [62] benchmarks efficiency by compiling performance-improving edits across various CodeNet challenges. Despite these valuable contributions, a closer examination reveals limitations in current benchmarks, which hinder their effectiveness in accurately evaluating efficiency.

*Equal Contribution.
†Corresponding Authors.

Specifically, current code efficiency benchmarks exhibit three key limitations. Firstly, they suffer from **Language Diversity** issues, primarily focusing on single-language measurement [32, 23, 66, 56, 48], often Python, despite its relatively small share (25%) of the overall programming landscape[†]. This narrow focus overlooks crucial language-specific factors like compiler optimizations and memory management prevalent in languages like C++, Java, and Golang, necessitating a multi-language benchmark. Secondly, **Data Contamination** is a significant problem, as many benchmarks, such as EffiBench and Mercury (using LeetCode), and ENAMEL and EvalPerf (using HumanEval and MBPP), rely on dated and widely circulated problem sets. These have often been "seen" by models during training [12, 40, 71, 22], leading to performance metrics that reflect memorization rather than genuine reasoning or optimization, thereby reducing their representativeness for evaluating performance on novel challenges. Lastly, current benchmarks are hampered by **Limited Complexity**, featuring tasks that are often straightforward and solvable without advanced algorithms. Benchmarks like HumanEval and MBPP, used by EvalPerf and ENAMEL, include simple prompts where models have already achieved high pass rates (e.g., >95% pass@1 on HumanEval)[†]. Such trivial tasks fail to reveal critical performance differentials or require substantial computational resources, making them unsuitable for measuring efficiency and highlighting the need for benchmarks incorporating more complex, computationally intensive problems to better assess an LLM's ability to handle large inputs or implement optimized solutions for real-world applications.

To address critical limitations in evaluating LLM code efficiency, we introduce EFFIBENCH-X, the first multi-language benchmark specifically designed for this purpose. EFFIBENCH-X evaluates efficiency across six diverse programming languages: Python, C++, Java, JavaScript, Ruby, and Golang, directly tackling the language diversity problem. It comprises recently released competitive programming tasks from various platforms, paired with canonical human expert solutions, effectively mitigating data contamination. By focusing on complex problems requiring advanced algorithms and data structures, EFFIBENCH-X overcomes the issue of limited complexity, providing a more accurate assessment of LLMs' efficiency in challenging scenarios. Furthermore, our comprehensive evaluation framework with high-resolution profiling ensures robust and reliable measurement of LLM-generated code efficiency within a controlled environment.

Leveraging EFFIBENCH-X and its comprehensive evaluation framework, we conducted a comprehensive study evaluating the performance of a wide range of LLMs against expert-written baselines. Our experiments revealed that while most LLMs are capable of generating functionally correct solutions, they consistently exhibit shortcomings in efficiency. Specifically, even the most efficient code produced by an LLM achieved only approximately **62%** of the runtime performance demonstrated by expert-crafted code. These findings highlight a significant gap and underscore the critical need for further research into optimization techniques. Such advancements are essential to empower LLMs to generate code that is not only correct but also highly efficient across diverse programming languages.

2 Related Work

Early efforts on code generation with LLMs gravitated toward assessing correctness and functionality, exemplified by HumanEval [16] and MBPP [11], which challenge models to produce valid code snippets from natural language docstrings. Building on this foundation, subsequent works have aimed to mitigate issues of limited diversity in tasks. For instance, HumanEval-X [74], MultiPLe [14], and MBXP [10] expand these benchmarks to multiple programming languages, while DS-1000 [42], ARCADE [72], and NumpyEval [73] target data science–oriented tasks. Beyond these, efforts like APIBench [55], BigCodeBench [75], BiasBench [34], and RepoBench [49] delve into broader software engineering subtasks, including library usage and repository-level code completion. While these benchmarks have been instrumental in gauging a model's correctness, they rarely address performance, potentially overlooking inefficiencies that hinder real-world adoption. Recognizing this significant limitation, a new wave of evaluations has emerged. As shown in Table 1), benchmarks such as EffiBench [32], Mercury [23], EvalPerf [48], ENAMEL [56], and PIE [62] specifically target performance metrics like runtime and memory usage. While these efforts are valuable, they commonly face limitations. They are largely confined to single-language settings, predominantly Python. Furthermore, their reliance on popular tasks (e.g., HumanEval, MBPP) increases the risk of data contamination from LLM training data. Moreover, the often-simple nature of these tasks does

[†]https://www.tiobe.com/tiobe-index/

[†]HumanEval Leaderboard: `https://paperswithcode.com/sota/code-generation-on-humaneval.`

Table 1: Comparison of EFFIBENCH-X to other code efficiency benchmarks. EFFIBENCH-X covers six programming languages, with one human-written solution for each language per task.

Dataset	Tasks	Test Cases	Solutions	Metrics	Language	Source From
EvalPerf	121	1/284.78	7.6	DPS/DPS_{norm}	Python	EvalPlus/HumanEval/MBPP
ENAMEL	142	20	1	Eff@k	Python	HumanEval
COFFE (Function)	398	10.71	1	Efficiency@k	Python	HumanEval/MBPP/APPS/CodeContests
COFFE (File)	358	48.63	66.93	Efficiency@k	Python	HumanEval/MBPP/APPS/CodeContests
PIE	41	104	23.8	%Opt/Speedup/%Correct	C++	CodeNet
ECCO	48	20.0	16.5	Time/Memory	Python	CodeNet
Mercury	256	$+\infty$	18.4	Pass/Beyond	Python	LeetCode
EffiBench	1000	100	1	ET/MU/TMU	Python	LeetCode
EffiBench+	160	100	10.7	GET/ECC	Python	LeetCode
EFFIBENCH-X	623	100	1 (6)	ET/MP/MI	Multiple	Competitions

not adequately expose nuanced performance differences critical in complex scenarios. To overcome these shortcomings, EFFIBENCH-X offers a multi-language evaluation across *Python, C++, Java, JavaScript, Ruby, and Golang*. By pairing solutions with expert-optimized versions from competitive programming sites and utilizing diverse, complex tasks, EFFIBENCH-X allows for a more thorough assessment.

3 EFFIBENCH-X

3.1 Efficiency-Critical Problem Collection

The foundation of EFFIBENCH-X lies in a curated collection of programming problems sourced from a wide array of competitive programming platforms, which are commonly used for evaluate human developers' abilities in writing efficient algorithms. Sourcing from multiple platforms ensures diversity in problem style, constraints, and required algorithmic techniques. These platforms include Aizu [4], AtCoder [9], CodeChef [17], Codeforces [18], and LeetCode [43]. Problems are categorized into two types based on their input/output handling requirements:

(1) Functional Problems require the implementation of a specific function or class, typically receiving input via parameters and returning output directly. LeetCode problems often fall into this category. The benchmark infrastructure handles I/O serialization and deserialization via test templates.

(2) Standard I/O (stdio) Problems require the implementation of a complete program that reads input from standard input (stdin) and writes output to standard output (stdout). This format is common on platforms like Codeforces.

Our two problem types allow EFFIBENCH-X to evaluate LLM capabilities in both library-like function generation and standalone program creation. A critical aspect of our collection process is *mitigating data contamination*, where LLMs might have encountered benchmark problems during their pre-training phase. To address this, we meticulously collect the original release date for each problem. We prioritize and filter for problems released after October 2023 [†], significantly reducing the likelihood that contemporary LLMs have been trained on them. This focus on recent problems ensures that EFFIBENCH-X primarily evaluates the models' generalization and reasoning capabilities rather than memorization. Furthermore, the competitive programming origin of these tasks inherently selects for problems demanding non-trivial algorithmic thinking and efficient data structure usage, addressing the limitation of benchmarks based on overly simplistic tasks.

3.2 Canonical Solution Construction

To establish a reliable baseline for efficiency comparison, each problem in EFFIBENCH-X is paired with a few canonical solutions for each target language. These canonical solutions aim to represent high-quality, efficient code typically authored by human experts. We gather potential canonical solutions through multiple channels: (1) Publicly available solutions from platform discussion forums (e.g., LeetCode Discussion). (2) Publicly available accepted submissions retrieved via platform APIs. (3) Solutions curated from open-source repositories on GitHub and datasets on Hugging Face.

[†]October 2023 represents a common knowledge cutoff date for many contemporary LLMs, including GPT-4o and o1 models, reducing the likelihood that these problems appeared in their training data.

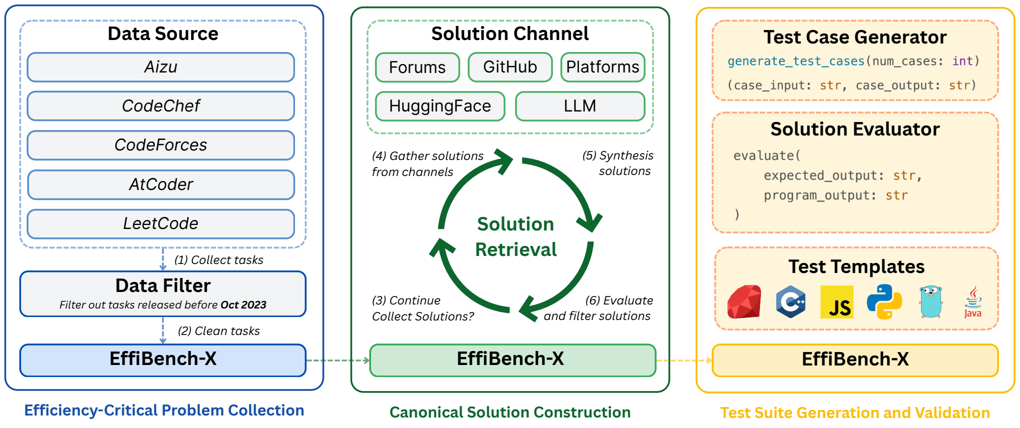

Figure 1: Overview of the construction pipeline. The process begins with collecting efficiency-critical problems from competitive programming platforms, followed by constructing canonical solutions from expert programmers, and then generating test suites with test case generators and solution evaluators.

For functional problems, we strive to collect canonical solutions in all six target languages. This comprehensive collection is crucial for the subsequent validation of language-specific test templates. For stdio problems, collecting a canonical solution in at least one language is sufficient for validating the test case generator and solution evaluator. When canonical solutions for specific languages are missing, we employ a careful translation process using state-of-the-art LLMs (o3-mini). The key objective during translation is to preserve the original algorithm, logic, and time/space complexity of the source human-written solution. We explicitly instruct the LLM to perform a direct translation without introducing optimizations or degradations. This ensures that the translated solutions maintain the efficiency characteristics of the original expert code, serving as a fair baseline. All collected and translated solutions undergo a rigorous verification process. We submit them back to their original platforms whenever possible. Only solutions that are accepted (i.e., pass all internal tests for correctness and efficiency within the limits) are retained as canonical solutions in EffiBench-X. Invalid or inefficient submissions are discarded. This step guarantees that our baseline represents demonstrably correct and performant code. We also collect multiple accepted solutions per problem-language pair where available, enabling future studies on the distribution of human-level performance.

3.3 Test Suite Generation and Validation

A robust test suite [23, 32, 35, 56] is essential for reliably evaluating both correctness and efficiency. For each problem, we generate a comprehensive test suite comprising three following components:

3.3.1 Core Components

Test Case Generator. Instead of directly generating many test cases with LLMs, which costs excessive tokens and hardly scalable, we prompt LLMs to generate a test case generator. `generate_test_cases(num_cases: int, seed: int)` is a Python function responsible for producing a list of test cases (including `input` and `output`). The generator is designed to create diverse test cases covering various scenarios, including boundary values, typical inputs, stress tests (large inputs), edge cases, and potential performance traps, guided by the problem constraints, to sufficiently evaluate a program's efficiency. One canonical solution is provided for the generator to produce the expected output for each test input. By default, EffiBench-X generates 100 test cases per problem using a fixed seed for reproducibility. The serialization format is kept simple (e.g., comma-separated values, space-separated values) to ensure easy parsing across all target languages.

Solution Evaluator. `evaluate(expected_output: str, program_output: str)` is a Python function that determines the correctness of a solution's output for a given test case. Crucially, the evaluator first deserializes both the expected and program outputs using logic consistent with the test case generator. It then performs a logical comparison based on the problem's requirements, rather than a simple string comparison. This allows for flexibility in output formatting (e.g., handling different ordering in lists if permissible).

Test Templates (Functional Problems Only). For each target language, a code template is generated for functional problems. This template provides the necessary boilerplate code (e.g., main function, I/O handling) to create a runnable program. It includes a specific placeholder string, `==Code Submission==` for injecting the functional solution (typically a function or method implementation). The template is responsible for reading the serialized input string from stdin, deserializing it into the appropriate data types expected by the solution function, calling the solution function, serializing the returned value, and printing the serialized result to stdout.

3.3.2 Component Validation

The test suites undergo an initial generation followed by a rigorous, multi-stage validation process to ensure their integrity. This comprehensive workflow guarantees their reliability, cross-language consistency, and accuracy, providing a solid basis for evaluating correctness and efficiency.

Test Case Validation. We execute the generated `generate_test_cases` function to produce 100 test cases. If the count is incorrect, the generator is deemed invalid, and the entire test suite for that problem is regenerated.

Test Template Validation. We check if the LLM generated test templates for all six target languages. If any templates are missing, we prompt the LLM specifically to generate the missing ones based on the context of the existing components.

Test Suite Validation. For *functional problems*, we combine its canonical solution with its corresponding test template for each target language. This complete program is then executed against all 100 generated test cases. The program's stdout is captured and compared against the expected output using the `evaluate` function. If the canonical solution fails any test case for a given language, the test template for that language is considered potentially faulty. We attempt to automatically repair the template by providing the LLM (o3-mini) with the template code, the canonical solution, the failing test case, and the error message (or incorrect output). We allow up to 3 repair attempts per template. If repair fails, the problem is flagged for manual inspection. This process ensures that the test templates correctly interface with known-good solutions. For *stdio problems*, we execute the canonical solution (in at least one available language) against all generated test cases, using the `evaluate` function to verify the output. If the canonical solution passes all test cases, the test case generator and solution evaluator are considered valid for this problem.

3.4 Sandboxed Execution Environment

To ensure fair and accurate performance measurements, all solution code executions are performed within a controlled sandbox environment for a reproducible execution environment. Our approach builds upon the concepts of [65, 24], but incorporates substantial proprietary enhancements including a high-resolution profiler, standard I/O support, and numerous performance improvements to meet the rigorous demands of our benchmark. This environment isolates executions from the host system and from each other, minimizing interference and variability. We leverage Docker containers as our sandboxing mechanism. Each execution runs within a dedicated container based on a pre-built image specific to the programming language (e.g., official Python, GCC, OpenJDK images), as detailed in Appendix C.5. Crucially, to mitigate interference from other processes and ensure consistent access to computational resources, we pin each worker's container to specific physical CPU cores using the `cpuset-cpus` Docker option. Our infrastructure detects the system's CPU topology (mapping logical cores to physical cores) and assigns containers to distinct physical cores, preventing multiple benchmark executions from contending for the same core resources simultaneously. This sandboxed setup provides a consistent and isolated environment, critical for obtaining reliable runtime and memory usage measurements.

3.5 High-Resolution Performance Profiling

Accurate efficiency assessment necessitates precise measurement of resource consumption. To this end, we developed a custom profiler that integrates with a sandboxed execution environment. This profiler captures high-resolution data by periodically sampling the execution time and memory usage of the solution. The sampling occurs at a high frequency (0.1 milliseconds, or 10 kHz) to accurately capture peak resource usage and rapid fluctuations. The profiler also enforces specified memory limits, terminating processes that exceed them and logging Out-Of-Memory events. For each execution on a test case, the profiler outputs time-series data consisting of timestamps and corresponding memory usage readings, which forms the basis for the efficiency metrics (Section 4).

4 Evaluation Metrics

To quantify the efficiency of LLM-generated solutions relative to the human-expert-written solutions, we follow existing works [32, 56, 48, 23, 62, 66] to define the following three key metrics:

Execution Time (ET) measures the runtime performance of an LLM-generated solution compared to the human-expert-written solution. For each problem i, let T_i^{H} be the execution time of the human-expert-written solution required to pass all test cases for the task, and T_i^{L} be the execution time of the LLM-generated solution required to pass all test cases for the task. If the LLM-generated solution for problem i fails to pass all test cases or encounters a runtime error (e.g., timeout, crash) [23, 56], its ET score s_i^T for that problem is defined as zero. Otherwise, the score is computed as the ratio $\frac{T_i^{\text{H}}}{T_i^{\text{L}}}$, which is then clipped to the interval $[0, 1]$:

$$s_i^T = \text{clip}\left(\frac{T_i^{\text{H}}}{T_i^{\text{L}}}, 0, 1\right)$$

This clipping ensures that an LLM-generated solution performing faster than the human-expert-written solution is considered equally efficient (score of 1) for this metric [23, 56, 48], preventing disproportionate influence from exceptionally fast outliers. The overall ET is then calculated as the average of these individual scores across all N evaluated problems, expressed as a percentage:

$$\text{ET (\%)} = \left(\frac{1}{N}\sum_{i=1}^{N} s_i^T\right) \times 100\%$$

A higher ET percentage indicates that, on average, LLM-generated solutions achieve runtime performance closer to, or as good as, the human-expert-written solutions (where 100% signifies performance equivalent to or better than the human-expert-written solution).

Memory Peak (MP) indicates the minimum memory required for the server, such as mobile devices, to execute the code for the test cases. MP evaluates the memory peak of LLM-generated code relative to the human-expert-written solution. For each problem i, M_i^{H} denotes the memory peak of the expert-written solution, and M_i^{L} denote that of the LLM-generated solution. Similar to ET, if the solution fails, its individual memory score s_i^M for that problem is defined as zero. Otherwise, the score is computed as the ratio $\frac{M_i^{\text{H}}}{M_i^{\text{L}}}$, which is then clipped to $[0, 1]$:

$$s_i^M = \text{clip}\left(\frac{M_i^{\text{H}}}{M_i^{\text{L}}}, 0, 1\right)$$

Similar to ET, the overall MP can be expressed as a percentage:

$$\text{MP (\%)} = \left(\frac{1}{N}\sum_{i=1}^{N} s_i^M\right) \times 100\%$$

A higher MP percentage suggests that LLM-generated solutions, on average, exhibit peak memory footprints comparable to or as good as human-expert-written solutions.

Memory Integral (MI) measures the overall memory consumption throughout a solution's execution by comparing the area under the memory-time curve for LLM-generated solutions against human-expert-written ones. The memory integral A for a single execution is defined as $A = \int_0^{T_{\text{total}}} M(t)\,dt$, where $M(t)$ is the memory usage at time t over the solution's total execution time T_{total}. This integral is numerically approximated based on the high-resolution profiling data (Section 3.5). Let A_i^{H} be the memory integral of the expert-written solution for problem i, and A_i^{L} be that of the LLM-generated solution. Similar to ET, if the solution fails, its individual MI score s_i^A for that problem is defined as zero. Otherwise, the score is computed as the ratio $\frac{A_i^{\mathrm{H}}}{A_i^{\mathrm{L}}}$, which is then clipped to $[0, 1]$:

$$s_i^A = \mathrm{clip}\left(\frac{A_i^{\mathrm{H}}}{A_i^{\mathrm{L}}}, 0, 1\right)$$

This clipping ensures that solutions with a smaller memory integral (i.e., $A_i^{\mathrm{L}} < A_i^{\mathrm{H}}$, resulting in a ratio $\frac{A_i^{\mathrm{H}}}{A_i^{\mathrm{L}}} > 1$) than the human-expert solution are considered equivalently efficient (score of 1) for this metric, preventing disproportionate influence from exceptionally memory-frugal solutions. The overall MI can be expressed as a percentage:

$$\text{MI (\%)} = \left(\frac{1}{N}\sum_{i=1}^{N} s_i^A\right) \times 100\%$$

A higher MI percentage indicates that, on average, LLM-generated solutions exhibit overall memory consumption (integrated over time) comparable to or as good as human-expert solutions.

5 Evaluation

Testbed: Efficiency evaluations are conducted on AWS i7ie.metal-48xl instances. These instances feature 4th generation Intel Xeon Scalable (Sapphire Rapids) processors with 96 physical cores (192 vCPUs) and 384 GiB of RAM. All code execution occurs within our sandboxed environment, detailed in Section 3.4 and implemented using Docker containers, thereby ensuring reliable efficiency evaluations by isolating processes from host machine variations and preventing inter-task interference. Each code execution is constrained by a 10-second timeout and a 1024 MiB memory limit.

Models and Metrics: As detailed in Table 5, various LLMs are selected for evaluation. Open-source LLMs comprise DeepSeek-R1 [26] and DeepSeek-V3-0324 [47]; Llama-4-Scout-17B-16E-Instruct and Llama-4-Maverick-17B-128E-Instruct [51]; Qwen3 (8B, 14B, 32B) [57], Qwen2.5-Coder-Instruct (14B, 32B) [37], and QwQ-32B [58]; Gemma-3-It (4B, 12B, 27B) [63]; and Phi-4, Phi-4-Reasoning, and Phi-4-Reasoning-Plus [1]. Proprietary LLMs include GPT-4o and GPT-4o-mini [38]; Claude-3.5-Haiku and Claude-3.7-Sonnet [8, 7]; Gemini-2.0 (Flash, Flash-Lite, Flash-Thinking) and Gemini-2.5 (Flash, Pro) [6]. For evaluation metrics, We use Execution Time (ET), Memory Peak (MP), Memory Integral (MI), and Pass@1 [15] to measure the correctness of the generated code.

5.1 Main Results

Open-source LLMs: As shown in Table 2, among the evaluated open-source models, DeepSeek-R1 demonstrates exceptional performance, achieving the highest scores among open-source models with an ET of 61.33%, MP of 69.41%, MI of 60.06%, and a Pass@1 rate of 72.79%. The Qwen3 series also shows strong performance, particularly Qwen3-32B, which slightly outperforms DeepSeek-R1 in execution time efficiency with an ET of 62.21% and MI of 61.48%, while achieving a solid Pass@1 of 70.41%. The Phi-4 family shows interesting variations, with Phi-4-Reasoning achieving notably better performance (ET 48.37%, Pass@1 50.54%) compared to the base Phi-4 model (ET 28.42%, Pass@1 30.60%). Other models like the Llama-4 series generally exhibit lower performance in both correctness and efficiency metrics, with Llama-4-Maverick-17B-128E-Instruct scoring an ET of just 16.28% and Pass@1 of 37.32%. This highlights a significant variance in capabilities among currently available open-source LLMs for generating efficient and correct code.

Proprietary LLMs: The proprietary LLMs generally show strong performance. Gemini-2.5-Pro emerges as the top-performing model overall, with the highest MP of 75.60% and Pass@1 rate of 79.43%, though its ET (47.82%) and MI (45.08%) fall short of the best open-source models. Gemini-2.5-Flash also delivers robust results (ET 40.42%, MP 65.13%, MI 38.22%, Pass@1 68.08%). Models like Claude-3.7-Sonnet (ET 47.79%, Pass@1 56.23%) and GPT-4o (ET 24.53%, Pass@1 43.61%) offer competitive, albeit lower, performance levels. This suggests that while leading proprietary models often set the benchmark for correctness and memory peak efficiency, some open-source models are now achieving superior execution time efficiency.

Impact of Model Size: Model size within a family generally correlates positively with performance. Examining the Qwen3 series, Qwen3-8B achieves an ET of 45.44% and Pass@1 of 53.50%. This

Table 2: Results are averaged across six programming languages on EFFIBENCH-X.

Model Name	**Execution Time**	**Memory Peak**	**Memory Integral**	**Pass@1**
DeepSeek-V3-0324	40.46%	51.52%	39.38%	53.29%
DeepSeek-R1	61.33%	69.41%	60.06%	72.79%
Llama-4-Scout-17B-16E-Instruct	23.16%	28.09%	22.61%	28.44%
Llama-4-Maverick-17B-128E-Instruct	16.28%	36.47%	15.52%	37.32%
Qwen3-8B	45.44%	51.64%	45.11%	53.50%
Qwen3-14B	59.75%	60.79%	58.58%	63.30%
Qwen3-32B	**62.21%**	67.26%	**61.48%**	70.41%
Qwen2.5-Coder-7B-Instruct	24.37%	25.40%	24.01%	25.74%
Qwen2.5-Coder-14B-Instruct	32.06%	34.19%	31.23%	34.88%
Qwen2.5-Coder-32B-Instruct	36.66%	39.12%	36.05%	39.94%
QwQ-32B	31.60%	35.51%	31.38%	36.78%
Gemma-3-4B-It	9.69%	16.74%	9.21%	17.15%
Gemma-3-12B-It	15.53%	27.64%	14.26%	28.25%
Gemma-3-27B-It	16.62%	32.52%	15.21%	33.49%
Phi-4	28.42%	29.81%	27.34%	30.60%
Phi-4-Reasoning	48.37%	48.83%	46.98%	50.54%
Phi-4-Reasoning-Plus	36.62%	38.22%	35.86%	39.27%
GPT-4o-mini	16.96%	35.12%	16.19%	36.06%
GPT-4o	24.53%	42.61%	24.00%	43.61%
Claude-3.5-Haiku	36.33%	44.06%	35.07%	45.24%
Claude-3.7-Sonnet	47.79%	54.60%	46.98%	56.23%
Gemini-2.0-Flash	30.61%	47.15%	28.57%	48.56%
Gemini-2.0-Flash-Lite	27.86%	38.61%	26.28%	39.89%
Gemini-2.0-Flash-Thinking	38.82%	55.56%	36.83%	57.38%
Gemini-2.5-Flash	40.42%	65.13%	38.22%	68.08%
Gemini-2.5-Pro	47.82%	**75.60%**	45.08%	**79.43%**

Table 3: Results on EFFIBENCH-X-C++ and EFFIBENCH-X-Python.

Model Name	**ET**	**MP**	**MI**	**Pass@1**	**ET**	**MP**	**MI**	**Pass@1**
	C++				Python			
DeepSeek-V3-0324	39.02%	52.54%	32.23%	54.41%	48.02%	53.22%	48.59%	55.38%
DeepSeek-R1	60.89%	71.57%	51.89%	75.12%	67.30%	69.66%	66.27%	74.64%
Qwen3-14B	61.22%	63.84%	**57.03%**	66.77%	65.28%	64.36%	63.64%	68.38%
Qwen3-32B	**63.89%**	70.30%	56.48%	74.80%	**69.28%**	70.94%	**68.34%**	75.44%
Qwen2.5-Coder-14B-Instruct	32.02%	35.09%	28.17%	35.63%	31.89%	34.26%	31.89%	34.99%
Qwen2.5-Coder-32B-Instruct	35.57%	38.43%	32.12%	39.33%	37.13%	39.29%	36.89%	40.45%
QwQ-32B	34.33%	38.60%	30.29%	40.45%	38.82%	40.10%	38.35%	42.05%
Gemma-3-4B-It	9.51%	20.77%	6.48%	20.87%	14.54%	18.95%	15.42%	19.26%
Gemma-3-12B-It	13.76%	32.44%	7.46%	32.58%	18.72%	23.34%	19.95%	23.60%
Gemma-3-27B-It	12.93%	38.32%	5.77%	38.68%	25.39%	32.78%	27.13%	33.87%
Phi-4	28.60%	30.56%	24.08%	31.46%	30.14%	31.71%	29.63%	32.74%
Phi-4-Reasoning	42.63%	42.77%	36.59%	45.10%	50.86%	50.76%	49.92%	53.45%
GPT-4o-mini	10.28%	35.32%	6.07%	36.12%	27.12%	36.21%	28.26%	37.40%
GPT-4o	12.62%	37.27%	7.58%	37.88%	38.05%	48.13%	40.24%	48.96%
Claude-3.5-Haiku	35.17%	45.84%	31.00%	47.03%	34.46%	40.41%	34.86%	41.41%
Claude-3.7-Sonnet	45.45%	55.26%	40.68%	56.98%	51.90%	55.88%	52.03%	57.78%
Gemini-2.5-Flash	35.84%	68.26%	23.52%	71.91%	56.08%	66.14%	58.30%	69.82%
Gemini-2.5-Pro	43.33%	**75.51%**	26.60%	**81.06%**	66.12%	**75.08%**	67.72%	**79.94%**

Table 4: DeepSeek-R1 and Claude-3.7-Sonnet on EFFIBENCH-X across different languages.

Language	ET	MP	MI	Pass@1	ET	MP	MI	Pass@1
	DeepSeek-R1				Claude-3.7-Sonnet			
JavaScript	63.34%	69.19%	63.03%	71.43%	50.15%	**57.33%**	49.48%	**58.59%**
Ruby	64.01%	66.11%	63.65%	69.02%	49.44%	52.48%	49.59%	53.61%
Python	**67.30%**	69.66%	**66.27%**	74.64%	**51.90%**	55.88%	**52.03%**	57.78%
Java	52.23%	69.36%	54.98%	73.19%	46.33%	55.11%	46.41%	56.66%
C++	60.89%	**71.57%**	51.89%	**75.12%**	45.45%	55.26%	40.68%	56.98%
Go	60.24%	70.57%	60.57%	73.35%	43.48%	51.52%	43.67%	53.77%

scales up with Qwen3-14B (ET 59.75%, Pass@1 63.30%), and further with Qwen3-32B (ET 62.21%, Pass@1 70.41%). A similar trend is observed in the Qwen2.5-Coder series, with the 32B variant (ET 36.66%, Pass@1 39.94%) outperforming the 14B counterpart (ET 32.06%, Pass@1 34.88%). The Gemma-3 series follows this pattern as well, with performance increasing from Gemma-3-4B-It (ET 9.69%, Pass@1 17.15%) to Gemma-3-27B-It (ET 16.62%, Pass@1 33.49%). This indicates that larger models tend to have better capabilities for generating both correct and more efficient code.

Impact of Model Specialization: Newer versions and specialized variants of models often demonstrate considerable performance improvements. The Gemini family illustrates this clearly: Gemini-2.0-Flash shows an ET of 30.61% and Pass@1 of 48.56%, while the specialized Gemini-2.0-Flash-Thinking variant improves upon this with an ET of 38.82% and Pass@1 of 57.38%. Further advancements are seen with Gemini-2.5-Flash (ET 40.42%, Pass@1 68.08%), and Gemini-2.5-Pro leads in correctness (Pass@1 79.43%). Similar benefits can be observed in the Phi-4 series, where the reasoning-enhanced variant (ET 48.37%, Pass@1 50.54%) significantly outperforms the base model (ET 28.42%, Pass@1 30.60%). This trend highlights the rapid evolution in LLM capabilities and the benefits of targeted model enhancements.

Overall, while several state-of-the-art LLMs can generate functionally correct code at high rates, there remains a significant gap in achieving efficiency comparable to human expert solutions. Even the best-performing models achieve around 60-62% relative execution time on average, indicating substantial room for improvement in generating truly optimized code.

5.2 Comparison on Different Language Subsets

We verify the consistency of model performance across programming languages by analyzing results on the C++ and Python subsets (Table 3). This analysis confirms that the relative ranking of models observed in the end-to-end results remains largely consistent across individual language subsets, despite variations in absolute performance values. In both C++ and Python, the top-performing models maintain their leadership positions: Qwen3-32B and DeepSeek-R1 excel in execution time efficiency across both languages (C++: 63.89% and 60.89%; Python: 69.28% and 67.30%, respectively), while Gemini-2.5-Pro consistently leads in Pass@1 (C++: 81.06%; Python: 79.94%) and memory peak metrics (C++: 75.51%; Python: 75.08%). Similarly, mid-tier models like Claude-3.7-Sonnet and Phi-4-Reasoning maintain their relative positions, as do lower-performing models like the Gemma-3 series. While absolute performance tends to be higher on Python than C++ across most models, these variations affect all models similarly, preserving their relative standings. This cross-language consistency validates the robustness of EFFIBENCH-X, confirming that a model's general code efficiency capabilities transfer across diverse programming contexts.

5.3 Language-Specific Performance

We provide the evaluation results of DeepSeek-R1 and Claude-3.7-Sonnet on EFFIBENCH-X across different programming languages in Table 4. Both models demonstrate strong functional correctness across languages, with DeepSeek-R1 achieving Pass@1 rates from 69.02% (Ruby) to 75.12% (C++), and Claude-3.7-Sonnet ranging from 53.61% (Ruby) to 58.59% (JavaScript). However, we observe significant variations in efficiency metrics across language types. Dynamically-typed languages (Python, Ruby, JavaScript) consistently show higher execution time efficiency, with Python leading at 67.30% ET for DeepSeek-R1 and 51.90% ET for Claude-3.7-Sonnet, while statically-typed languages like Java show comparatively lower ET scores. This language-based efficiency gap persists across both models despite strong functional correctness, suggesting that LLMs may have developed better optimization strategies for widely used scripting languages. The contrast is particularly notable with C++, where DeepSeek-R1 achieves its highest Pass@1 (75.12%) and strong memory performance (MP: 71.57%), yet shows only 60.89% execution time efficiency compared to Python's 67.30%. These findings indicate that while current LLMs can generate syntactically correct and functionally working code across languages, their ability to produce optimized code that matches human efficiency varies substantially by language type, highlighting an area for future research.

6 Conclusion

In this paper, we propose EFFIBENCH-X, the first multi-language benchmark designed specifically to evaluate code efficiency across six programming languages. Our comprehensive assessment of 26 SOTA LLMs reveals significant efficiency gaps between LLM-generated and human expert code, with even the best-performing model (Qwen3-32B) achieving only about 62% of human-level efficiency on average, and the performance varies considerably by language. These findings underscore the critical need to enhance LLMs' optimization capabilities across diverse languages, particularly where correctness and efficiency are vital. EFFIBENCH-X could serve as a resource for future research aimed at improving multilingual code generation.

References

[1] Marah Abdin, Sahaj Agarwal, Ahmed Awadallah, Vidhisha Balachandran, Harkirat Behl, Lingjiao Chen, Gustavo de Rosa, Suriya Gunasekar, Mojan Javaheripi, Neel Joshi, Piero Kauffmann, Yash Lara, Caio César Teodoro Mendes, Arindam Mitra, Besmira Nushi, Dimitris Papailiopoulos, Olli Saarikivi, Shital Shah, Vaishnavi Shrivastava, Vibhav Vineet, Yue Wu, Safoora Yousefi, and Guoqing Zheng. Phi-4-reasoning technical report. *arXiv preprint arXiv:2504.21318*, 2025.

[2] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. AVATAR: A parallel corpus for java-python program translation. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 2268–2281. Association for Computational Linguistics, 2023.

[3] Toufique Ahmed and Premkumar T. Devanbu. Few-shot training llms for project-specific code-summarization. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 177:1–177:5. ACM, 2022.

[4] Aizu Online Judge. Aizu Online Judge. `https://onlinejudge.u-aizu.ac.jp/`, 2025.

[5] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy-Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. Santacoder: don't reach for the stars! *CoRR*, abs/2301.03988, 2023.

[6] Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Slav Petrov, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy P. Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul Ronald Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski,

and et al. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805, 2023.

[7] Anthropic. Claude 3.7 sonnet, February 2025. Released February 2025.

[8] Anthropic. Model card addendum: Claude 3.5 haiku and upgraded claude 3.5 sonnet. Technical report, Anthropic, 2025.

[9] AtCoder. AtCoder. `https://atcoder.jp/`, 2025.

[10] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, and Ramesh Nallapati. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[11] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021.

[12] BigCode. Self-oss-instruct-sc2-exec-filter-50k. `https://huggingface.co/datasets/bigcode/self-oss-instruct-sc2-exec-filter-50k`, 2023.

[13] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[14] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Trans. Software Eng.*, 49(7):3675–3691, 2023.

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[17] CodeChef. CodeChef. `https://www.codechef.com/`, 2025.

[18] Codeforces. Codeforces. `https://codeforces.com/`, 2025.

[19] Jianbo Dai, Jianqiao Lu, Yunlong Feng, Rongju Ruan, Ming Cheng, Haochen Tan, and Zhijiang Guo. MHPP: exploring the capabilities and limitations of language models beyond basic code generation. *CoRR*, abs/2405.11430, 2024.

[20] DeepSeekAI. Deepseek coder: Let the code write itself, 2023.

[21] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *CoRR*, abs/2304.02014, 2023.

[22] Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. Generalization or memorization: Data contamination and trustworthy evaluation for large language models. *arXiv preprint arXiv:2402.15938*, 2024.

[23] Mingzhe Du, Anh Tuan Luu, Bin Ji, and See-Kiong Ng. Mercury: An efficiency benchmark for LLM code synthesis. *CoRR*, abs/2402.07844, 2024.

[24] Mingzhe Du, Anh Tuan Luu, Bin Ji, Xiaobao Wu, Dong Huang, Terry Yue Zhuo, Qian Liu, and See-Kiong Ng. Codearena: A collective evaluation platform for llm code generation. *arXiv preprint arXiv:2503.01295*, 2025.

[25] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[26] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[27] Md. Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. Fixeval: Execution-based evaluation of program fixes for competitive programming problems. *CoRR*, abs/2206.07796, 2022.

[28] Masum Hasan, Tanveer Muttaqueen, Abdullah Al Ishtiaq, Kazi Sajeed Mehrab, Md. Mahim Anjum Haque, Tahmid Hasan, Wasi Uddin Ahmad, Anindya Iqbal, and Rifat Shahriyar. Codesc: A large code-description parallel dataset. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*, volume ACL/IJCNLP 2021 of *Findings of ACL*, pages 210–218. Association for Computational Linguistics, 2021.

[29] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

[30] Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. Codecot: Tackling code syntax errors in cot reasoning for code generation, 2024.

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

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

[33] Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M. Zhang. Swiftcoder: Enhancing code generation in large language models through efficiency-aware fine-tuning, 2025.

[34] Dong Huang, Jie M Zhang, Qingwen Bu, Xiaofei Xie, Junjie Chen, and Heming Cui. Bias testing and mitigation in llm-based code generation. *ACM Transactions on Software Engineering and Methodology*, 2024.

[35] Dong Huang, Jie M. Zhang, Mark Harman, Mingzhe Du, and Heming Cui. Measuring the influence of incorrect code on test generation, 2025.

[36] Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

[37] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. 2024.

[38] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[39] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.

[40] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[41] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1430–1442. IEEE, 2023.

[42] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. DS-1000: A natural and reliable benchmark for data science code generation. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR, 2023.

[43] LeetCode. LeetCode. `https://leetcode.com/`, 2025.

[44] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 919–931. IEEE, 2023.

[45] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! *CoRR*, abs/2305.06161, 2023.

[46] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Réêmi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022.

[47] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

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

[49] Tianyang Liu, Canwen Xu, and Julian J. McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *CoRR*, abs/2306.03091, 2023.

[50] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[51] MetaAI. Introducing llama 4: Advancing multimodal intelligence, 2024.

[52] Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py: Practical deep similarity learning-based type inference for python. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2241–2252. ACM, 2022.

[53] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[54] Changan Niu, Ting Zhang, Chuanyi Li, Bin Luo, and Vincent Ng. On evaluating the efficiency of source code generated by llms. *arXiv preprint arXiv:2404.06041*, 2024.

[55] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *CoRR*, abs/2305.15334, 2023.

[56] Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. How efficient is llm-generated code? a rigorous & high-standard benchmark. *arXiv preprint arXiv:2406.06647*, 2024.

[57] QwenTeam. Qwen3, April 2025.

[58] QwenTeam. Qwq-32b: Embracing the power of reinforcement learning, March 2025.

[59] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jé̊emy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023.

[60] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[61] Jieke Shi, Zhou Yang, and David Lo. Efficient and green large language models for software engineering: Vision and the road ahead. *arXiv preprint arXiv:2404.04566*, 2024.

[62] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning Performance-Improving Code Edits. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[63] Gemma Team and Google DeepMind. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

[64] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[65] vndee. llm-sandbox: A sandbox for large language models, 2023.

[66] Siddhant Waghjale, Vishruth Veerendranath, Zora Zhiruo Wang, and Daniel Fried. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness? *arXiv preprint arXiv:2407.14044*, 2024.

[67] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics, 2021.

[68] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[69] Jiayi Wei, Greg Durrett, and Isil Dillig. Typet5: Seq2seq type inference using static analysis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[70] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.

[71] Shuo Yang, Wei-Lin Chiang, Lianmin Zheng, Joseph E Gonzalez, and Ion Stoica. Rethinking benchmark and contamination for language models with rephrased samples. *arXiv preprint arXiv:2311.04850*, 2023.

[72] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Oleksandr Polozov, and Charles Sutton. Natural language to code generation in interactive data science notebooks. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 126–173. Association for Computational Linguistics, 2023.

[73] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. CERT: continual pre-training on sketches for library-oriented code generation. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 2369–2375. ijcai.org, 2022.

[74] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *CoRR*, abs/2303.17568, 2023.

[75] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

A Limitations

Despite the significant contributions of EffiBench-X in establishing a multi-language benchmark for code efficiency using novel and complex competitive programming tasks, our study and benchmark have several limitations that warrant discussion. Firstly, while competitive programming problems effectively address data contamination and complexity, this focus primarily evaluates algorithmic and data structure efficiency. It may not fully encompass the spectrum of efficiency considerations critical in other software development domains, such as low-level system programming, optimizing for specific hardware architectures, or managing large-scale data processing pipelines, thus potentially limiting the generalizability of findings to all efficiency-critical code generation contexts. Our current analysis primarily quantifies the efficiency gap between LLM-generated and expert code; it does not delve deeply into the root causes of LLM inefficiency at a granular code level (e.g., identifying specific code constructs or algorithmic choices that are suboptimal), which is useful for providing targeted feedback for model improvement. Finally, conducting extensive efficiency evaluations across multiple languages, numerous models, and complex problems with high-resolution profiling is computationally demanding, which could present a practical barrier for researchers with limited resources wishing to replicate or significantly extend this evaluation on a large scale.

B Broader Impacts

Positive Societal Impacts The development and adoption of benchmarks like EffiBench-X are crucial for driving progress in LLM-generated code efficiency. This progress promises significant positive societal impacts, including reduced energy consumption and lower operational costs for computing resources at scale, as inefficient code requires more energy and infrastructure. Furthermore, enabling LLMs to generate more efficient code is critical for deploying these models and the applications they help create in resource-constrained environments such as mobile devices and embedded systems, as well as in latency-sensitive cloud applications where performance is paramount. Benchmarks like ours also provide the research community with a clearer understanding of current LLM capabilities and limitations regarding code efficiency across diverse languages, fostering targeted research towards more effective and beneficial LLM-driven development workflows.

Negative Societal Impacts However, improvements in LLM capabilities, including enhanced code efficiency facilitated by benchmarks like EffiBench-X, also carry potential negative societal impacts. As LLMs become more adept at generating highly optimized code, there is a risk that these powerful capabilities could be exploited for malicious purposes, such as creating more efficient and evasive malware or facilitating sophisticated cyberattacks. This potential advancement in LLM-generated harmful code could inadvertently lower the technical barrier for malicious actors. Additionally, while efficiency is a goal, relying heavily on LLM-generated code necessitates robust validation processes; even efficient code could contain subtle bugs or security vulnerabilities that might be overlooked, potentially introducing new risks at scale. Addressing these concerns requires responsible development and deployment practices for LLMs, coupled with continued research into security, interpretability, and comprehensive validation techniques.

C Technical Appendices and Supplementary Material

C.1 Model List

C.2 Reliability of Efficiency Measurement

To assess the reliability of our efficiency metrics and ensure robustness against potential execution variability, we conducted multiple runs of our evaluation pipeline. Table 6 presents the results of executing the code generated by DeepSeek-R1 and Claude-3.7-Sonnet three times, reporting the mean, minimum, and maximum values for each efficiency metric. The results demonstrate high consistency across repeated executions. For execution time (ET), which might be expected to show the most variability due to system load fluctuations, we observe relatively small variations. DeepSeek-R1 shows an overall mean ET of 62.45% with a range from 61.33% to 63.55% (a variation of approximately ±1.7%). Similarly, Claude-3.7-Sonnet achieves a mean ET of 48.63% with a range from 47.79% to 49.31% (a variation of about ±1.5 percentage points). This narrow range confirms

Table 5: Model list and URL.

Model Name	URL
Open Source Models	
DeepSeek-V3	`https://huggingface.co/deepseek-ai/DeepSeek-V3-0324`
DeepSeek-R1	`https://huggingface.co/deepseek-ai/DeepSeek-R1`
Llama-4-Scout-17B-16E-Instruct	`https://huggingface.co/meta-llama/Llama-4-Scout-17B-16E-Instruct`
Llama-4-Maverick-17B-128E-Instruct	`https://huggingface.co/meta-llama/Llama-4-Maverick-17B-128E-Instruct`
Qwen3-8B	`https://huggingface.co/Qwen/Qwen3-8B`
Qwen3-14B	`https://huggingface.co/Qwen/Qwen3-14B`
Qwen3-32B	`https://huggingface.co/Qwen/Qwen3-32B`
Qwen2.5-Coder-7B-Instruct	`https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct`
Qwen2.5-Coder-14B-Instruct	`https://huggingface.co/Qwen/Qwen2.5-Coder-14B-Instruct`
Qwen2.5-Coder-32B-Instruct	`https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct`
QwQ-32B	`https://huggingface.co/Qwen/QwQ-32B`
Gemma-3-4B-It	`https://huggingface.co/google/gemma-3-4b-it`
Gemma-3-12B-It	`https://huggingface.co/google/gemma-3-12b-it`
Gemma-3-27B-It	`https://huggingface.co/google/gemma-3-27b-it`
Phi-4	`https://huggingface.co/microsoft/phi-4`
Phi-4-Reasoning	`https://huggingface.co/microsoft/Phi-4-reasoning`
Phi-4-Reasoning-Plus	`https://huggingface.co/microsoft/Phi-4-reasoning-plus`
Proprietary Models	
GPT-4o-mini	`https://platform.openai.com/docs/models/gpt-4o-mini`
GPT-4o	`https://platform.openai.com/docs/models/gpt-4o`
Claude-3.5-Haiku	`https://www.anthropic.com/claude/haiku`
Claude-3.7-Sonnet	`https://www.anthropic.com/claude/sonnet`
Gemini-2.0-Flash	`https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash`
Gemini-2.0-Flash-Lite	`https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash-lite`
Gemini-2.0-Flash-Thinking	`https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-flash`
Gemini-2.5-Flash	`https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-flash`
Gemini-2.5-Pro	`https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-pro`

the reliability of our execution time measurements. Memory metrics show even greater stability across runs. For memory peak (MP), both models demonstrate extremely consistent results, with variations of less than 0.1 percentage points (DeepSeek-R1: 69.42% with range 69.41%-69.44%; Claude-3.7-Sonnet: 54.63% with range 54.60%-54.65%). Memory integral (MI) shows slightly more variation but remains highly stable (DeepSeek-R1: 61.15% with range 60.06%-62.19%; Claude-3.7-Sonnet: 47.78% with range 46.98%-48.40%). Language-specific results follow similar patterns of consistency. For instance, Python shows the highest stability with ET ranges of 67.30%-67.67% for DeepSeek-R1 and 51.90%-52.06% for Claude-3.7-Sonnet. Java exhibits slightly wider variations (DeepSeek-R1: 52.23%-57.56%; Claude-3.7-Sonnet: 46.33%-49.01%), potentially reflecting the additional variability introduced by JVM optimization and garbage collection processes. However, even these wider ranges remain relatively narrow, preserving the relative performance relationships between models.

These results validate the robustness of our efficiency measurement approach. The sandboxed execution environment, combined with our methodology of executing code multiple times, effectively controls for transient system variations while capturing meaningful differences in code efficiency. The consistent results across repeated runs confirm that the efficiency metrics reported throughout this paper reliably represent the true performance characteristics of LLM-generated code, providing a solid foundation for comparative analysis across models and languages.

C.3 Additional Language Subsets

Tables 7, 8, 9, and 10 present detailed evaluation results for the Java, JavaScript, Ruby, and Go subsets of EFFIBENCH-X. These findings reinforce the performance patterns observed in our primary analysis while revealing language-specific insights. For JavaScript, Qwen3-32B leads open-source models with 70.29% ET and 69.87% MI, slightly outperforming DeepSeek-R1, while Gemini-2.5-Pro achieves the highest Pass@1 (80.90%) and MP (78.22%). In Java, Qwen3-14B unexpectedly achieves the highest efficiency (62.32% ET, 62.02% MI) among all models, outperforming larger variants, while Gemini-2.5-Pro leads in correctness (80.42% Pass@1). For Ruby, Gemini-2.5-Pro demonstrates the best overall performance (65.66% ET, 74.32% Pass@1), with DeepSeek-R1 leading open-source models (64.01% ET, 69.02% Pass@1). In Go, Qwen3-32B shows the highest efficiency among open-source models (62.13% ET, 62.60% MI), while Gemini-2.5-Pro achieves the highest correctness (79.94% Pass@1) despite lower execution efficiency (31.50% ET). Several patterns emerge consistently across languages: model rankings remain relatively stable despite variations in

Table 6: Robustness evaluation of code efficiency metrics through triple execution of DeepSeek-R1 and Claude-3.7-Sonnet generated solutions. Results are reported in Mean (min, max) format to demonstrate consistency across multiple runs.

Model Name	ET	MP	MI	Pass@1
		All		
DeepSeek-R1	62.45% (61.33%, 63.55%)	69.42% (69.41%, 69.44%)	61.15% (60.06%, 62.19%)	72.79%
Claude-3.7-Sonnet	48.63% (47.79%, 49.31%)	54.63% (54.60%, 54.65%)	47.78% (46.98%, 48.40%)	56.21%
		C++		
DeepSeek-R1	61.79% (60.89%, 62.77%)	71.54% (71.44%, 71.61%)	52.99% (51.89%, 54.10%)	75.12%
Claude-3.7-Sonnet	46.36% (45.45%, 47.18%)	55.12% (55.03%, 55.26%)	41.58% (40.68%, 42.05%)	56.82%
		Java		
DeepSeek-R1	55.01% (52.23%, 57.56%)	69.34% (69.31%, 69.36%)	57.54% (54.98%, 59.82%)	73.19%
Claude-3.7-Sonnet	47.95% (46.33%, 49.01%)	55.09% (55.06%, 55.11%)	47.92% (46.41%, 49.04%)	56.66%
		JavaScript		
DeepSeek-R1	64.09% (63.34%, 64.84%)	69.20% (69.19%, 69.21%)	63.71% (63.03%, 64.38%)	71.43%
Claude-3.7-Sonnet	50.86% (50.15%, 51.44%)	57.31% (57.30%, 57.33%)	50.19% (49.48%, 50.76%)	58.59%
		Ruby		
DeepSeek-R1	64.14% (64.01%, 64.27%)	66.14% (66.11%, 66.16%)	63.78% (63.65%, 63.91%)	69.02%
Claude-3.7-Sonnet	49.61% (49.44%, 49.76%)	52.48% (52.48%, 52.49%)	49.73% (49.59%, 49.86%)	53.61%
		Golang		
DeepSeek-R1	62.20% (60.24%, 64.16%)	70.65% (70.57%, 70.71%)	62.58% (60.57%, 64.59%)	73.35%
Claude-3.7-Sonnet	45.01% (43.48%, 46.41%)	51.89% (51.52%, 52.11%)	45.23% (43.67%, 46.66%)	53.77%
		Python		
DeepSeek-R1	67.49% (67.30%, 67.67%)	69.66% (69.66%, 69.67%)	66.32% (66.27%, 66.36%)	74.64%
Claude-3.7-Sonnet	51.99% (51.90%, 52.06%)	55.89% (55.88%, 55.90%)	52.04% (52.00%, 52.09%)	57.78%

Table 7: Evaluation of code generated by various LLMs on EFFIBENCH-X-Java.

Model Name ET	MP	MI	Pass@1	
DeepSeek-V3-0324	34.46%	52.11%	34.35%	53.93%
DeepSeek-R1	52.23%	69.36%	54.98%	73.19%
Llama-4-Scout-17B-16E-Instruct	20.92%	26.16%	20.46%	26.48%
Llama-4-Maverick-17B-128E-Instruct	11.24%	38.80%	8.45%	39.81%
Qwen3-8B	32.63%	53.83%	37.54%	56.18%
Qwen3-14B	**62.32%**	65.04%	**62.02%**	68.70%
Qwen3-32B	52.18%	68.92%	56.32%	72.71%
Qwen2.5-Coder-7B-Instruct	25.14%	25.96%	24.99%	26.32%
Qwen2.5-Coder-14B-Instruct	33.76%	34.24%	32.88%	35.47%
Qwen2.5-Coder-32B-Instruct	38.83%	40.64%	39.19%	41.41%
QwQ-32B	23.90%	36.84%	27.17%	38.20%
Gemma-3-4B-It	6.01%	12.84%	4.89%	13.32%
Gemma-3-12B-It	14.71%	27.28%	11.62%	28.25%
Gemma-3-27B-It	13.24%	28.08%	9.61%	29.37%
Phi-4	30.31%	30.14%	29.29%	31.46%
Phi-4-Reasoning	54.94%	54.68%	54.39%	56.66%
Phi-4-Reasoning-Plus	40.69%	44.02%	41.38%	45.10%
GPT-4o-mini	12.40%	35.23%	9.84%	35.96%
GPT-4o	18.21%	37.84%	14.99%	39.00%
Claude-3.5-Haiku	42.46%	45.10%	38.64%	46.71%
Claude-3.7-Sonnet	46.33%	55.11%	46.41%	56.66%
Gemini-2.0-Flash	33.66%	48.21%	27.35%	50.24%
Gemini-2.0-Flash-Lite	24.47%	29.72%	20.66%	30.98%
Gemini-2.0-Flash-Thinking	42.30%	56.61%	35.51%	58.59%
Gemini-2.5-Flash	32.40%	65.65%	28.20%	68.70%
Gemini-2.5-Pro	33.05%	**76.51%**	30.50%	**80.42%**

Table 8: Evaluation of code generated by various LLMs on EFFIBENCH-X-JavaScript.

Model Name ET	**MP**	**MI**	**Pass@1**	
DeepSeek-V3-0324	40.18%	51.22%	39.75%	52.17%
DeepSeek-R1	63.34%	69.19%	63.03%	71.43%
Llama-4-Scout-17B-16E-Instruct	24.70%	31.20%	24.61%	31.46%
Llama-4-Maverick-17B-128E-Instruct	14.10%	44.75%	14.45%	45.10%
Qwen3-8B	52.38%	54.11%	52.37%	55.22%
Qwen3-14B	60.65%	61.65%	60.05%	63.08%
Qwen3-32B	**70.29%**	71.77%	**69.87%**	73.84%
Qwen2.5-Coder-7B-Instruct	27.25%	28.78%	26.88%	29.05%
Qwen2.5-Coder-14B-Instruct	33.51%	36.62%	33.08%	37.08%
Qwen2.5-Coder-32B-Instruct	36.98%	40.55%	36.48%	41.09%
QwQ-32B	32.79%	34.61%	32.42%	35.47%
Gemma-3-4B-It	10.23%	18.86%	10.27%	19.58%
Gemma-3-12B-It	15.48%	28.93%	15.74%	29.37%
Gemma-3-27B-It	13.60%	31.67%	13.80%	32.26%
Phi-4	29.37%	31.44%	28.90%	31.78%
Phi-4-Reasoning	52.94%	53.99%	52.27%	55.38%
Phi-4-Reasoning-Plus	42.53%	44.08%	42.25%	44.94%
GPT-4o-mini	12.81%	34.48%	13.04%	34.83%
GPT-4o	24.91%	49.66%	26.62%	50.08%
Claude-3.5-Haiku	38.36%	48.62%	38.10%	49.44%
Claude-3.7-Sonnet	50.15%	57.33%	49.48%	58.59%
Gemini-2.0-Flash	31.97%	52.82%	32.10%	53.61%
Gemini-2.0-Flash-Lite	28.06%	40.92%	27.66%	42.22%
Gemini-2.0-Flash-Thinking	40.30%	60.92%	40.29%	62.44%
Gemini-2.5-Flash	37.09%	66.03%	37.25%	68.22%
Gemini-2.5-Pro	47.26%	**78.22%**	47.93%	**80.90%**

Table 9: Evaluation of code generated by various LLMs on EFFIBENCH-X-Ruby.

Model Name ET	**MP**	**MI**	**Pass@1**	
Open-source LLMs				
DeepSeek-V3-0324	45.61%	49.28%	45.65%	50.88%
DeepSeek-R1	64.01%	66.11%	63.65%	69.02%
Llama-4-Scout-17B-16E-Instruct	21.30%	23.19%	21.51%	23.60%
Llama-4-Maverick-17B-128E-Instruct	28.05%	32.59%	28.93%	33.23%
Qwen3-8B	44.44%	45.99%	44.35%	47.35%
Qwen3-14B	52.76%	53.15%	52.45%	54.41%
Qwen3-32B	55.47%	56.32%	55.25%	58.27%
Qwen2.5-Coder-7B-Instruct	21.77%	22.69%	21.76%	22.95%
Qwen2.5-Coder-14B-Instruct	30.32%	32.53%	30.43%	32.91%
Qwen2.5-Coder-32B-Instruct	34.35%	36.67%	34.40%	37.24%
QwQ-32B	28.66%	29.65%	28.65%	30.18%
Gemma-3-4B-It	11.72%	13.76%	12.04%	13.80%
Gemma-3-12B-It	21.37%	26.26%	21.68%	26.81%
Gemma-3-27B-It	25.48%	31.84%	25.86%	32.42%
Phi-4	26.66%	28.34%	26.65%	28.89%
Phi-4-Reasoning	47.53%	48.61%	47.22%	49.92%
Phi-4-Reasoning-Plus	38.32%	39.49%	38.24%	40.61%
GPT-4o-mini	30.72%	36.52%	31.48%	37.24%
GPT-4o	41.34%	47.64%	42.39%	48.48%
Claude-3.5-Haiku	35.47%	41.13%	35.66%	42.05%
Claude-3.7-Sonnet	49.44%	52.48%	49.59%	53.61%
Gemini-2.0-Flash	35.75%	44.32%	35.87%	45.43%
Gemini-2.0-Flash-Lite	30.32%	37.20%	30.27%	37.88%
Gemini-2.0-Flash-Thinking	43.40%	52.69%	43.63%	53.77%
Gemini-2.5-Flash	57.06%	63.60%	57.88%	65.65%
Gemini-2.5-Pro	**65.66%**	**72.16%**	**66.12%**	**74.32%**

Table 10: Evaluation of code generated by various LLMs on EFFIBENCH-X-Go.

Model Name ET	MP	MI	Pass@1	
Open-source LLMs				
DeepSeek-V3-0324	35.49%	50.75%	35.71%	52.97%
DeepSeek-R1	<u>60.24%</u>	<u>70.57%</u>	<u>60.57%</u>	<u>73.35%</u>
Llama-4-Scout-17B-16E-Instruct	20.84%	24.89%	20.89%	25.52%
Llama-4-Maverick-17B-128E-Instruct	7.12%	30.20%	7.12%	32.10%
Qwen3-8B	42.88%	46.34%	43.33%	47.83%
Qwen3-14B	56.25%	56.69%	56.28%	58.43%
Qwen3-32B	**62.13%**	65.28%	**62.60%**	67.42%
Qwen2.5-Coder-7B-Instruct	20.59%	20.72%	20.60%	21.35%
Qwen2.5-Coder-14B-Instruct	30.86%	32.40%	30.95%	33.23%
Qwen2.5-Coder-32B-Instruct	37.08%	39.13%	37.19%	40.13%
QwQ-32B	31.13%	33.25%	31.41%	34.35%
Gemma-3-4B-It	6.15%	15.24%	6.17%	16.05%
Gemma-3-12B-It	9.11%	27.56%	9.12%	28.89%
Gemma-3-27B-It	9.06%	32.41%	9.08%	34.35%
Phi-4	25.45%	26.68%	25.50%	27.29%
Phi-4-Reasoning	41.35%	42.15%	41.48%	42.70%
Phi-4-Reasoning-Plus	31.64%	32.35%	31.81%	33.07%
GPT-4o-mini	8.40%	32.96%	8.44%	34.83%
GPT-4o	12.05%	35.11%	12.20%	37.24%
Claude-3.5-Haiku	32.07%	43.23%	32.18%	44.78%
Claude-3.7-Sonnet	43.48%	51.52%	43.67%	53.77%
Gemini-2.0-Flash	18.18%	43.94%	18.19%	46.07%
Gemini-2.0-Flash-Lite	22.79%	41.05%	22.78%	42.70%
Gemini-2.0-Flash-Thinking	26.21%	52.72%	26.23%	55.22%
Gemini-2.5-Flash	24.08%	61.09%	24.17%	64.21%
Gemini-2.5-Pro	31.50%	**76.11%**	31.61%	**79.94%**

absolute scores; dynamically-typed languages generally show higher efficiency scores than statically-typed languages; memory efficiency and correctness appear more closely correlated than execution time efficiency and correctness; and reasoning-enhanced models consistently outperform their base counterparts. These additional evaluations validate EFFIBENCH-X's robustness as a comprehensive benchmark for assessing code efficiency generation across diverse programming contexts.

C.4 Problem Types

Tables 11 and 12 present detailed evaluation results across two problem categories in our benchmark: functional problems and standard I/O problems. For functional problems, DeepSeek-R1 achieves the highest execution efficiency among open-source models (74.53% ET, 73.26% MI), followed closely by Qwen3-32B (73.14% ET, 72.85% MI) and Qwen3-14B (72.92% ET, 72.37% MI). Gemini-2.5-Pro excels in correctness and memory peak efficiency (90.04% MP, 92.50% Pass@1) but shows lower execution efficiency (57.41% ET) than the top open-source alternatives. Phi-4-Reasoning delivers strong performance (56.73% ET, 58.28% Pass@1), significantly outperforming its base model Phi-4 (30.54% ET, 32.79% Pass@1).

For standard I/O problems, performance drops substantially across all models. Qwen3-32B leads in execution efficiency (49.25% ET, 48.00% MI), followed by DeepSeek-R1 (45.68% ET, 44.42% MI) and Qwen3-14B (44.12% ET, 42.22% MI). Gemini-2.5-Pro maintains its leadership in correctness and memory metrics (58.47% MP, 63.92% Pass@1), though with much lower absolute scores than on functional problems. Claude-3.7-Sonnet performs well (39.35% ET, 45.96% Pass@1), outperforming many other closed-source alternatives.

Across all models, performance on functional problems significantly exceeds that on standard I/O problems, with most models showing 15-30 percentage point higher scores on the functional subset for both efficiency and correctness metrics. This pattern holds regardless of model size or architecture. For instance, DeepSeek-R1 achieves 74.53% ET and 85.60% Pass@1 on functional problems but only

Table 11: Evaluation of code generated by various LLMs on the EFFIBENCH-X-functional subset.

Model Name	**ET**	**MP**	**MI**	**Pass@1**
Open-source LLMs				
DeepSeek-V3-0324	45.95%	58.46%	45.01%	59.52%
DeepSeek-R1	**74.53%**	83.26%	**73.26%**	85.60%
Llama-4-Scout-17B-16E-Instruct	29.47%	32.48%	28.80%	32.79%
Llama-4-Maverick-17B-128E-Instruct	20.71%	42.62%	19.91%	43.29%
Qwen3-8B	55.88%	63.69%	55.83%	65.09%
Qwen3-14B	72.92%	74.78%	72.37%	76.53%
Qwen3-32B	73.14%	80.68%	72.85%	82.74%
Qwen2.5-Coder-7B-Instruct	28.02%	28.58%	27.74%	28.85%
Qwen2.5-Coder-14B-Instruct	37.78%	40.15%	37.15%	40.53%
Qwen2.5-Coder-32B-Instruct	45.49%	48.06%	44.93%	48.67%
QwQ-32B	34.21%	37.55%	34.17%	38.12%
Gemma-3-4B-It	11.99%	20.18%	11.43%	20.56%
Gemma-3-12B-It	20.37%	35.25%	18.80%	36.19%
Gemma-3-27B-It	21.94%	41.74%	20.21%	42.95%
Phi-4	30.54%	32.32%	29.80%	32.79%
Phi-4-Reasoning	56.73%	57.21%	55.48%	58.28%
Phi-4-Reasoning-Plus	41.18%	42.52%	40.40%	43.15%
Proprietary LLMs				
GPT-4o-mini	18.01%	37.46%	17.08%	38.12%
GPT-4o	24.33%	44.20%	23.25%	45.02%
Claude-3.5-Haiku	44.72%	54.04%	43.38%	55.13%
Claude-3.7-Sonnet	54.91%	63.57%	54.42%	64.89%
Gemini-2.0-Flash	36.46%	55.27%	33.90%	56.56%
Gemini-2.0-Flash-Lite	35.60%	47.93%	33.74%	48.96%
Gemini-2.0-Flash-Thinking	47.60%	66.95%	45.23%	68.39%
Gemini-2.5-Flash	46.42%	74.12%	43.79%	76.48%
Gemini-2.5-Pro	57.41%	**90.04%**	53.78%	**92.50%**

Table 12: Evaluation of code generated by various LLMs on the EFFIBENCH-X-standard I/O subset.

Model Name	**ET**	**MP**	**MI**	**Pass@1**
DeepSeek-V3-0324	33.95%	43.29%	32.70%	45.91%
DeepSeek-R1	45.68%	52.99%	44.42%	57.60%
Llama-4-Scout-17B-16E-Instruct	15.66%	22.89%	15.28%	23.27%
Llama-4-Maverick-17B-128E-Instruct	11.03%	29.18%	10.31%	30.23%
Qwen3-8B	33.06%	37.35%	32.40%	39.77%
Qwen3-14B	44.12%	44.19%	42.22%	47.60%
Qwen3-32B	**49.25%**	51.33%	**48.00%**	55.79%
Qwen2.5-Coder-7B-Instruct	20.05%	21.63%	19.60%	22.05%
Qwen2.5-Coder-14B-Instruct	25.28%	27.12%	24.22%	28.19%
Qwen2.5-Coder-32B-Instruct	26.17%	28.51%	25.51%	29.59%
QwQ-32B	28.51%	33.09%	28.07%	35.20%
Gemma-3-4B-It	6.97%	12.66%	6.58%	13.10%
Gemma-3-12B-It	9.79%	18.61%	8.88%	18.83%
Gemma-3-27B-It	10.30%	21.58%	9.27%	22.28%
Phi-4	25.91%	26.84%	24.42%	28.01%
Phi-4-Reasoning	38.46%	38.89%	36.89%	41.35%
Phi-4-Reasoning-Plus	31.22%	33.12%	30.48%	34.68%
GPT-4o-mini	15.70%	32.34%	15.13%	33.63%
GPT-4o	24.77%	40.73%	24.90%	41.93%
Claude-3.5-Haiku	26.38%	32.22%	25.22%	33.51%
Claude-3.7-Sonnet	39.35%	43.95%	38.15%	45.96%
Gemini-2.0-Flash	23.69%	37.52%	22.25%	39.06%
Gemini-2.0-Flash-Lite	18.68%	27.55%	17.43%	29.12%
Gemini-2.0-Flash-Thinking	28.41%	42.05%	26.88%	44.33%
Gemini-2.5-Flash	33.31%	54.46%	31.62%	58.13%
Gemini-2.5-Pro	36.45%	**58.47%**	34.76%	**63.92%**

45.68% ET and 57.60% Pass@1 on standard I/O problems. Similarly, Gemini-2.5-Pro shows 57.41% ET and 92.50% Pass@1 versus 36.45% ET and 63.92% Pass@1 on these respective categories.

Despite these absolute differences, the relative performance hierarchy remains consistent across both problem types. DeepSeek-R1, Qwen3-32B, and Gemini-2.5-Pro maintain their leadership positions in their respective metrics across both categories, while smaller models like Gemma-3-4B-It (11.99% ET on functional, 6.97% on standard I/O) consistently rank lower. Both problem types show similar scaling patterns with model size and generation, with larger and newer models generally outperforming their smaller and older counterparts.

The performance gap between functional and standard I/O problems reveals that current LLMs struggle more with optimizing I/O operations, buffer management, and input parsing compared to core algorithmic logic. Standard I/O problems involve more complex implementation aspects, creating additional opportunities for inefficiencies. This finding suggests that enhanced training on I/O patterns and memory management could substantially improve overall code generation capabilities, particularly for solving end-to-end programming problems that require both algorithmic reasoning and efficient implementation details.

C.5 Language Runtime Specifications

Table 13 details the specific Docker images and compilation/execution flags used for each programming language in our sandboxed execution environment. For C++, we utilize gcc:14.2.0-bookworm with optimization level -O2 and address sanitization enabled (-fsanitize=address) to ensure memory safety while maintaining good performance. Java code is executed using openjdk:21-jdk-bookworm without additional flags, leveraging the JVM's default optimization capabilities. For JavaScript, we employ node:22.14.0-bookworm with the –harmony flag to enable the latest ECMAScript features. Ruby evaluations use ruby:3.2.7-bookworm, while Go utilizes golang:1.23.7-bookworm, both with their respective default runtime configurations. Python code runs on python:3.11.11-bookworm, which provides a balanced combination of performance and compatibility with modern Python libraries. All these environments are consistent across evaluations, ensuring fair comparisons between human reference solutions and LLM-generated code while capturing real-world performance characteristics for each language.

Table 13: Language Runtime Specifications and Flags

Language	Docker Image	Compilation/Execution Flags
C++	gcc:14.2.0-bookworm	-O2, -fsanitize=address
Java	openjdk:21-jdk-bookworm	-
JavaScript	node:22.14.0-bookworm	–harmony
Ruby	ruby:3.2.7-bookworm	-
Go	golang:1.23.7-bookworm	-
Python3	python:3.11.11-bookworm	-

C.6 Unexpected Regression in Reasoning-Enhanced Models:

An intriguing observation in our results is the performance regression seen in Phi-4-Reasoning-Plus compared to Phi-4-Reasoning. While Phi-4-Reasoning achieves respectable performance metrics across languages (ET 48.37%, Pass@1 50.54%), its enhanced counterpart Phi-4-Reasoning-Plus shows notably lower performance (ET 37.55%, Pass@1 38.65%). As shown in Table 14, this

Table 14: Rate of LLM-generated responses that do not extract code, presented per model.

Model Name	**all**	**C++**	**Java**	**JavaScript**	**Ruby**	**Go**	**Python**
DeepSeek-V3	32/3738 (0.86%)	5/623 (0.80%)	4/623 (0.64%)	5/623 (0.80%)	8/623 (1.28%)	4/623 (0.64%)	6/623 (0.96%)
DeepSeek-R1	6/3738 (0.16%)	2/623 (0.32%)	1/623 (0.16%)	1/623 (0.16%)	1/623 (0.16%)	1/623 (0.16%)	0/623 (0.00%)
Gemini-2.0-Flash	53/3738 (1.42%)	2/623 (0.32%)	2/623 (0.32%)	5/623 (0.80%)	10/623 (1.61%)	13/623 (2.09%)	21/623 (3.37%)
Gemini-2.0-Flash-Thinking	32/3738 (0.86%)	2/623 (0.32%)	3/623 (0.48%)	7/623 (1.12%)	7/623 (1.12%)	8/623 (1.28%)	5/623 (0.80%)
Gemini-2.5-Flash	78/3738 (2.09%)	12/623 (1.93%)	17/623 (2.73%)	13/623 (2.09%)	11/623 (1.77%)	15/623 (2.41%)	10/623 (1.61%)
Gemini-2.5-Pro	2/3738 (0.05%)	0/623 (0.00%)	0/623 (0.00%)	0/623 (0.00%)	0/623 (0.00%)	2/623 (0.32%)	0/623 (0.00%)
Phi-4-reasoning	1151/3738 (30.79%)	254/623 (40.77%)	161/623 (25.84%)	176/623 (28.25%)	183/623 (29.37%)	180/623 (28.89%)	197/623 (31.62%)
Phi-4-reasoning-plus	1957/3738 (52.35%)	371/623 (59.55%)	297/623 (47.67%)	299/623 (47.99%)	315/623 (50.56%)	327/623 (52.49%)	348/623 (55.86%)

Table 15: Evaluation of code generated by various LLMs on EffiBench-X-Aizu.

Model Name	Execution Time (ET) (%)	Memory Peak (MP) (%)	Memory Integral (MI) (%)	Pass@1 (%)
DeepSeek-V3-0324	25.81%	31.38%	24.61%	34.34%
DeepSeek-R1	36.09%	42.02%	34.96%	45.96%
Llama-4-Scout-17B-16E-Instruct	6.83%	10.73%	7.01%	11.11%
Llama-4-Maverick-17B-128E-Instruct	5.05%	12.68%	4.66%	14.14%
Qwen3-8B	18.69%	21.02%	18.07%	22.22%
Qwen3-14B	24.87%	24.70%	23.30%	27.27%
Qwen3-32B	33.96%	34.82%	32.62%	38.89%
Qwen2.5-Coder-7B-Instruct	6.18%	6.95%	6.03%	7.07%
Qwen2.5-Coder-14B-Instruct	10.82%	11.20%	10.41%	12.12%
Qwen2.5-Coder-32B-Instruct	9.78%	10.37%	9.15%	11.62%
QwQ-32B	27.44%	30.92%	26.53%	33.33%
Gemma-3-4B-It	2.70%	4.37%	2.61%	4.55%
Gemma-3-12B-It	2.57%	4.50%	2.58%	4.55%
Gemma-3-27B-It	4.52%	8.32%	4.52%	8.59%
Phi-4	12.03%	12.28%	10.95%	13.64%
Phi-4-Reasoning	23.78%	24.26%	23.27%	25.76%
Phi-4-Reasoning-Plus	14.39%	14.86%	13.90%	15.66%
GPT-4o-mini	8.96%	18.02%	8.90%	18.69%
GPT-4o	14.35%	25.06%	14.75%	25.76%
Claude-3.5-Haiku	14.64%	17.71%	14.06%	18.69%
Claude-3.7-Sonnet	26.14%	28.83%	25.53%	30.81%
Gemini-2.0-Flash	13.87%	19.73%	12.75%	21.72%
Gemini-2.0-Flash-Lite	10.74%	15.65%	10.62%	16.67%
Gemini-2.0-Flash-Thinking	32.08%	46.42%	30.86%	50.00%
Gemini-2.5-Flash	20.74%	34.54%	19.83%	36.36%
Gemini-2.5-Pro	**37.03%**	**58.70%**	**36.94%**	**65.15%**

regression can be largely attributed to the inability of Phi-4-Reasoning-Plus to properly generate extractable code in many cases, with a concerning 52.35% of responses lacking proper code extraction across all languages, compared to just 30.79% for Phi-4-Reasoning. This problem is particularly pronounced in C++ (59.55% non-extractable responses) and Python (55.86%). In contrast, high-performing models like DeepSeek-R1 and Gemini-2.5-Pro have minimal code extraction issues (0.16% and 0.05%, respectively), demonstrating their superior ability to produce well-structured outputs. These findings suggest that for intermediate model sizes like Phi-4 (16B), adding more complex reasoning capabilities might actually interfere with the model's ability to focus on the core task of generating functional code. The model appears to over-index on explanation and verbosity at the expense of concise, executable solutions. This stands in contrast to reasoning enhancements in larger models, where reasoning capabilities complement rather than detract from code generation performance. Our findings indicate that effective reasoning for code generation may have a model size threshold below which the added complexity becomes detrimental rather than beneficial, aligning with findings on Chain-of-Thought reasoning documented by Wei et al. [68], which confirm that model size significantly impacts the reasoning abilities of LLMs.

C.7 Website-Level Results

We provide the website-level code efficiency results in Tables 15 to 19.

C.8 Additional Related Work

The increasing popularity of LLMs for code generation has coincided with the growing availability of open-source code repositories and the need to boost developer productivity. Initial efforts focused on training models specifically for coding tasks, such as CodeT5 [67], AlphaCode [46], CodeGen [53], InCoder [25], StarCoder [45], SantaCoder [5], QwenCoder [37], and DeepSeek-Coder [20]. Contrastingly, models such as Codex [16], CodeLlama [59], Magicoder [70], and WizardCoder [50] represent a subsequent stride, being fine-tuned from foundation models [13, 64]. These code LLMs have been applied to various tasks, including code generation [16, 19, 11, 29], program repair [27, 41], automated testing [44, 21], code translation [60, 2], type prediction [52, 69], and code summarization [28, 3]. While LLMs have achieved impressive results in code generation tasks like HumanEval [16] and MBPP [11], their efficiency has received less attention. Recent studies [61, 32, 54] have shown that LLM-generated code exhibits lower efficiency in terms of execution time and memory usage compared to canonical solutions. These findings highlight the need for further research and development to improve the efficiency of LLM-generated code. In this work, we propose the first multi-language code efficiency benchmarks to measure the efficiency of LLM-generated code for different programming languages.

Table 16: Evaluation of code generated by various LLMs on EFFIBENCH-X-AtCoder.

Model Name	**Execution Time (ET) (%)**	**Memory Peak (MP) (%)**	**Memory Integral (MI) (%)**	**Pass@1 (%)**
DeepSeek-V3-0324	29.97%	37.61%	28.06%	41.16%
DeepSeek-R1	<u>37.36%</u>	42.25%	<u>35.33%</u>	48.21%
Llama-4-Scout-17B-16E-Instruct	10.78%	19.41%	10.41%	19.80%
Llama-4-Maverick-17B-128E-Instruct	9.69%	26.13%	9.00%	27.29%
Qwen3-8B	25.09%	28.46%	23.77%	31.32%
Qwen3-14B	35.39%	35.32%	32.87%	39.60%
Qwen3-32B	**42.85%**	42.77%	**40.72%**	48.77%
Qwen2.5-Coder-7B-Instruct	17.18%	19.00%	16.55%	19.57%
Qwen2.5-Coder-14B-Instruct	23.85%	25.90%	22.33%	27.40%
Qwen2.5-Coder-32B-Instruct	24.64%	27.08%	23.63%	28.52%
QwQ-32B	30.65%	35.81%	29.86%	39.15%
Gemma-3-4B-It	6.21%	11.23%	5.87%	11.52%
Gemma-3-12B-It	8.74%	16.67%	7.85%	16.89%
Gemma-3-27B-It	9.05%	19.35%	7.92%	20.13%
Phi-4	21.97%	22.93%	20.44%	24.50%
Phi-4-Reasoning	31.30%	31.15%	28.98%	34.45%
Phi-4-Reasoning-Plus	22.55%	23.81%	21.69%	25.73%
GPT-4o-mini	14.11%	28.71%	13.41%	30.54%
GPT-4o	24.02%	38.21%	23.75%	39.93%
Claude-3.5-Haiku	23.87%	29.35%	22.31%	31.21%
Claude-3.7-Sonnet	35.20%	39.39%	33.24%	42.17%
Gemini-2.0-Flash	21.73%	34.41%	20.17%	36.47%
Gemini-2.0-Flash-Lite	15.94%	23.47%	14.63%	25.28%
Gemini-2.0-Flash-Thinking	24.17%	35.83%	22.43%	38.81%
Gemini-2.5-Flash	27.56%	<u>44.72%</u>	25.57%	<u>49.78%</u>
Gemini-2.5-Pro	35.24%	**55.78%**	32.17%	**63.42%**

Table 17: Evaluation of code generated by various LLMs on EFFIBENCH-X-CodeChef.

Model Name	**Execution Time (ET) (%)**	**Memory Peak (MP) (%)**	**Memory Integral (MI) (%)**	**Pass@1 (%)**
DeepSeek-V3-0324	42.74%	55.97%	42.48%	57.17%
DeepSeek-R1	62.31%	<u>73.85%</u>	62.22%	<u>76.88%</u>
Llama-4-Scout-17B-16E-Instruct	26.57%	33.12%	25.91%	33.51%
Llama-4-Maverick-17B-128E-Instruct	15.46%	40.29%	14.56%	41.04%
Qwen3-8B	50.15%	56.55%	50.55%	58.78%
Qwen3-14B	<u>64.57%</u>	65.03%	<u>63.54%</u>	67.56%
Qwen3-32B	**64.76%**	70.47%	**64.92%**	72.94%
Qwen2.5-Coder-7B-Instruct	30.68%	32.29%	30.39%	32.62%
Qwen2.5-Coder-14B-Instruct	33.08%	35.16%	32.63%	35.66%
Qwen2.5-Coder-32B-Instruct	35.24%	38.19%	35.08%	38.71%
QwQ-32B	26.63%	30.93%	26.89%	31.18%
Gemma-3-4B-It	10.01%	18.39%	9.43%	19.18%
Gemma-3-12B-It	14.46%	27.28%	13.19%	27.60%
Gemma-3-27B-It	14.39%	30.07%	13.09%	30.82%
Phi-4	38.42%	39.60%	36.88%	40.14%
Phi-4-Reasoning	54.20%	55.46%	53.32%	56.99%
Phi-4-Reasoning-Plus	50.18%	53.55%	49.49%	54.84%
GPT-4o-mini	21.15%	43.90%	20.54%	44.62%
GPT-4o	30.17%	51.51%	30.67%	52.15%
Claude-3.5-Haiku	36.49%	44.34%	35.70%	44.98%
Claude-3.7-Sonnet	51.29%	57.10%	50.99%	58.06%
Gemini-2.0-Flash	30.72%	49.32%	29.30%	50.00%
Gemini-2.0-Flash-Lite	25.37%	37.61%	23.88%	39.07%
Gemini-2.0-Flash-Thinking	34.40%	51.09%	33.00%	51.97%
Gemini-2.5-Flash	46.88%	**76.87%**	45.26%	**79.21%**
Gemini-2.5-Pro	38.35%	62.43%	38.05%	64.52%

Table 18: Evaluation of code generated by various LLMs on EFFIBENCH-X-CodeForces.

Model Name	**Execution Time (ET) (%)**	**Memory Peak (MP) (%)**	**Memory Integral (MI) (%)**	**Pass@1 (%)**
DeepSeek-V3-0324	38.47%	49.32%	37.71%	50.00%
DeepSeek-R1	46.62%	55.19%	45.52%	56.67%
Llama-4-Scout-17B-16E-Instruct	16.13%	19.72%	16.23%	20.00%
Llama-4-Maverick-17B-128E-Instruct	9.50%	25.88%	9.02%	26.67%
Qwen3-8B	40.19%	45.15%	39.54%	46.67%
Qwen3-14B	47.63%	46.83%	45.74%	48.33%
Qwen3-32B	**50.69%**	55.40%	**49.73%**	56.67%
Qwen2.5-Coder-7B-Instruct	9.72%	10.00%	9.32%	10.00%
Qwen2.5-Coder-14B-Instruct	21.80%	23.12%	19.72%	23.33%
Qwen2.5-Coder-32B-Instruct	18.88%	19.67%	18.56%	20.00%
QwQ-32B	17.70%	19.94%	17.53%	20.00%
Gemma-3-4B-It	4.13%	8.08%	3.92%	8.33%
Gemma-3-12B-It	5.93%	13.33%	4.90%	13.33%
Gemma-3-27B-It	10.03%	19.67%	9.63%	20.00%
Phi-4	14.12%	14.55%	12.28%	15.00%
Phi-4-Reasoning	47.28%	48.47%	46.81%	50.00%
Phi-4-Reasoning-Plus	39.58%	42.22%	39.44%	43.33%
GPT-4o-mini	11.06%	26.26%	11.14%	26.67%
GPT-4o	20.15%	29.67%	21.94%	30.00%
Claude-3.5-Haiku	8.52%	10.00%	8.04%	10.00%
Claude-3.7-Sonnet	33.83%	39.57%	33.63%	40.00%
Gemini-2.0-Flash	19.79%	33.00%	19.10%	33.33%
Gemini-2.0-Flash-Lite	23.41%	34.11%	21.65%	35.00%
Gemini-2.0-Flash-Thinking	23.84%	36.28%	23.01%	36.67%
Gemini-2.5-Flash	34.35%	57.09%	33.78%	58.33%
Gemini-2.5-Pro	34.91%	**60.99%**	35.66%	**61.67%**

Table 19: Evaluation of code generated by various LLMs on EFFIBENCH-X-LeetCode.

Model Name	**Execution Time (ET) (%)**	**Memory Peak (MP) (%)**	**Memory Integral (MI) (%)**	**Pass@1 (%)**
Open-source LLMs				
DeepSeek-V3-0324	45.95%	58.46%	45.01%	59.52%
DeepSeek-R1	**74.53%**	83.26%	**73.26%**	85.60%
Llama-4-Scout-17B-16E-Instruct	29.47%	32.48%	28.80%	32.79%
Llama-4-Maverick-17B-128E-Instruct	20.71%	42.62%	19.91%	43.29%
Qwen3-8B	55.88%	63.69%	55.83%	65.09%
Qwen3-14B	72.92%	74.78%	72.37%	76.53%
Qwen3-32B	73.14%	80.68%	72.85%	82.74%
Qwen2.5-Coder-7B-Instruct	28.02%	28.58%	27.74%	28.85%
Qwen2.5-Coder-14B-Instruct	37.78%	40.15%	37.15%	40.53%
Qwen2.5-Coder-32B-Instruct	45.49%	48.06%	44.93%	48.67%
QwQ-32B	34.21%	37.55%	34.17%	38.12%
Gemma-3-4B-It	11.99%	20.18%	11.43%	20.56%
Gemma-3-12B-It	20.37%	35.25%	18.80%	36.19%
Gemma-3-27B-It	21.94%	41.74%	20.21%	42.95%
Phi-4	30.54%	32.32%	29.80%	32.79%
Phi-4-Reasoning	56.73%	57.21%	55.48%	58.28%
Phi-4-Reasoning-Plus	41.18%	42.52%	40.40%	43.15%
GPT-4o-mini	18.01%	37.46%	17.08%	38.12%
GPT-4o	24.33%	44.20%	23.25%	45.02%
Claude-3.5-Haiku	44.72%	54.04%	43.38%	55.13%
Claude-3.7-Sonnet	54.91%	63.57%	54.42%	64.89%
Gemini-2.0-Flash	36.46%	55.27%	33.90%	56.56%
Gemini-2.0-Flash-Lite	35.60%	47.93%	33.74%	48.96%
Gemini-2.0-Flash-Thinking	47.60%	66.95%	45.23%	68.39%
Gemini-2.5-Flash	46.42%	74.12%	43.79%	76.48%
Gemini-2.5-Pro	57.41%	**90.04%**	53.78%	**92.50%**

D Case Studies

Efficient Example 1: As shown in Figure 2, we provide a Python case example where the solution generated by DeepSeek-R1 closely matches the human-written canonical solution, both in structure and runtime efficiency (ratio = 0.985). The problem requires computing the sum of absolute differences between adjacent characters' ASCII values in a given string. The DeepSeek-R1 implementation uses a forward loop from index 1, calculating abs(ord(s[i]) - ord(s[i-1])), while the canonical solution performs the same calculation from index 0 using abs(ord(s[i]) - ord(s[i+1])). These two approaches are functionally equivalent due to the symmetric nature of the absolute difference, and both operate in linear time complexity $\mathcal{O}(n)$ with constant space usage. This convergence highlights the LLM's ability to infer optimal solutions not only from problem descriptions but also from general coding patterns. Unlike in worst-case examples where LLMs diverge in algorithm design or introduce unnecessary overhead, this case demonstrates a high-quality, generalized solution that mirrors expert-level efficiency and readability.

Efficient Example 2: As shown in Figure 3, we provide a C++ best-case example in which the code generated by Claude-3-7-Sonnet almost perfectly replicates the canonical solution. The task is to return the XOR of all numbers that appear exactly twice in an array. Both implementations adopt the same high-level strategy: (1) Count the frequency of each element in the input array; (2) Iterate through the frequency map and apply the XOR operation only to elements with a frequency of two. The canonical solution employs a map, which maintains ordering but results in slightly higher overhead due to its O(log N) access and insertion times. In contrast, the LLM-generated code optimizes performance by utilizing an unordered_map, which allows for average-case O(1) access and insertion, demonstrating not only correctness but also superior performance awareness. This case exemplifies a high-efficiency general solution, where the LLM demonstrates an understanding of optimal data structures and adheres to clean, human-like code organization. It highlights Claude's capability to generate competitive solutions that are not only correct but also efficient, often rivaling or even surpassing human-written implementations in specific scenarios.

Worst Case Example 1: As shown in Tables 4 and 5, Claude-3.7-Sonnet generated solution for counting the number of good strings—which can be rearranged to contain ''leet'' as a substring—exemplifies a clear inefficiency in both memory and runtime performance. The task requires counting strings of length n that contain at least one character l, at least two e's, and one t.

Claude-3.7-Sonnet approaches this problem using a 4-dimensional dynamic programming (DP) table. Specifically, the state dp[i][l][e][t] records the number of strings of length i with l occurrences of l, e of e, and t of t, capped respectively at 1, 2, and 1. This results in a state space of size $O(n \times 2 \times 3 \times 2)$, and the model updates each state at every iteration. While the solution is logically correct, it suffers from redundant state expansions and excessive memory usage, especially for large n. The model does not employ pruning strategies, prefix-counting optimizations, or algebraic transformations, and instead falls back on a simulation-based enumeration of state transitions.

In contrast, the human expert adopts a concise and efficient analytical approach using the *Inclusion-Exclusion Principle*. Rather than enumerating each valid string, it directly subtracts the number of invalid strings (e.g., those missing required characters) from the total 26^n. By leveraging combinatorial identities and fast modular exponentiation, it computes the final result in $O(\log n)$ time with constant space.

This example clearly illustrates a broader pattern observed in LLM-generated code: while often logically valid, such solutions tend to lack mathematical abstraction and optimization, opting instead for generic enumeration techniques. In this case, the Claude 3 Sonnet solution was approximately **33$\times$ slower** and significantly more memory-intensive compared to the canonical one, underscoring its limitations in recognizing mathematically structured problem-solving strategies.

Task Description

~~~
## Instructions
You are a competitive programming expert skilled in multiple programming languages.
Your task is to solve the following problem in python3:

## Problem Description
You are given a string s. The **score** of a string is defined as the sum of the absolute
↪  difference between the **ASCII** values of adjacent characters.

Return the **score** ofs.

**Example 1:**

**Input:** s = "hello"

**Output:** 13

**Explanation:**
The **ASCII** values of the characters in s are: 'h' = 104, 'e' = 101, 'l' = 108, 'o' = 111.
↪  So, the score of s would be |104 - 101| + |101 - 108| + |108 - 108| + |108 - 111| = 3 +
↪  7 + 0 + 3 = 13.

**Example 2:**

**Input:** s = "zaz"

**Output:** 50

**Explanation:**
The **ASCII** values of the characters in s are: 'z' = 122, 'a' = 97. So, the score of s
↪  would be |122 - 97| + |97 - 122| = 25 + 25 = 50.

**Constraints:**

- 2 <= s.length <= 100
- s consists only of lowercase English letters.

## Starter Code
```python
class Solution:
    def scoreOfString(self, s: str) -> int:
```

## Output Format
- Provide the complete solution code in **one markdown code block** with appropriate
↪  language identifier. If your response has multiple code blocks, only the last one will
↪  be used.
- Implement the function with the exact signature (name, parameters, etc.) specified in the
↪  starter code.
- EXCLUDE ALL explanations, code comments, import/package/library statements, additional
↪  classes or functions outside of the starter code scope, or starting code like if
↪  __name__ == "__main__": or func main() or package main or using namespace std;.
- Use but do not redefine any helper data structures provided in the starter code (even if
↪  commented out).
~~~

DeepSeek-R1

```python
class Solution:
    def scoreOfString(self, s: str) ->
    ↪  int:
        score = 0
        for i in range(1, len(s)):
            score += abs(ord(s[i]) -
            ↪  ord(s[i-1]))
        return score
```

Human Expert

```python
class Solution:
    def scoreOfString(self, s: str) ->
    ↪  int:
        ans = 0
        n = len(s)
        for i in range(n - 1):
            ans += abs(ord(s[i]) -
            ↪  ord(s[i+1]))
        return ans
```

Figure 2: This figure shows a Python problem where DeepSeek-R1 generated code with efficiency nearly matching the human expert solution (ratio = 0.985). Both calculate adjacent character ASCII differences using equivalent linear-time algorithms with constant space complexity, demonstrating that well-trained LLMs can identify optimal approaches matching expert solutions.

Task Description

~~~
## Instructions
You are a competitive programming expert skilled in multiple programming languages.
Your task is to solve the following problem in cpp:

## Problem Description
You are given an array `nums`, where each number in the array appears **either** once or
↪  twice.

Return the bitwise `XOR` of all the numbers that appear twice in the array, or 0 if no
↪  number appears twice.

**Example 1:**

**Input:** nums = [1,2,1,3]
**Output:** 1
**Explanation:**
The only number that appears twice in `nums` is 1.

**Example 2:**

**Input:** nums = [1,2,3]
**Output:** 0
**Explanation:**
No number appears twice in `nums`.

**Constraints:**

- `1 <= nums.length <= 50`
- `1 <= nums[i] <= 50`
- Each number in `nums` appears either once or twice.

## Starter Code
```cpp
class Solution {
public:
    int duplicateNumbersXOR(vector<int>& nums) {

    }
};
```

## Output Format
- Provide the complete solution code in **one markdown code block** with appropriate
↪  language identifier. If your response has multiple code blocks, only the last one will
↪  be used.
- Implement the function with the exact signature (name, parameters, etc.) specified in the
↪  starter code.
- EXCLUDE ALL explanations, code comments, import/package/library statements, additional
↪  classes or functions outside of the starter code scope, or starting code like if
↪  __name__ == "__main__": or func main() or package main or using namespace std;.
- Use but do not redefine any helper data structures provided in the starter code (even if
↪  commented out).
~~~

Claude-3.7-Sonnet

```cpp
class Solution {
public:
    int duplicateNumbersXOR(vector<int>&
    ↪  nums) {
        unordered_map<int, int> frequency;
        for (int num : nums) {
            frequency[num]++;
        }
        int result = 0;
        for (const auto& pair : frequency)
        ↪  {
            if (pair.second == 2) {
                result ^= pair.first;
            }
        }
        return result;
    }
};
```

Human Expert

```cpp
class Solution {
public:
    int duplicateNumbersXOR(vector<int>&
    ↪  nums) {
        map<int,int> mp;
        for(int i : nums) {
            mp[i]++;
        }

        int ans = 0;
        for(auto i : mp) {
            if(i.second == 2) {
                ans ^= i.first;
            }
        }
        return ans;
    }
};
```

Figure 3: This figure shows a C++ problem where Claude-3-7-Sonnet's solution achieves exceptional efficiency by choosing an optimal data structure.

Task Description

~~~
## Instructions
You are a competitive programming expert skilled in multiple programming languages.
Your task is to solve the following problem in python3:

## Problem Description
You are given an integer `n`.

A string `s` is called **good** if it contains only lowercase English characters **and** it is possible to rearrange the characters of `s` such that the new string contains `"leet"` as a **substring**.

For example:

- The string `"lteer"` is good because we can rearrange it to form `"leetr"` .
- `"letl"` is not good because we cannot rearrange it to contain `"leet"` as a substring.

Return *the **total** number of good strings of length* `n`.

Since the answer may be large, return it **modulo** `10^9 + 7`.

A **substring** is a contiguous sequence of characters within a string.

**Constraints:**
- `1 <= n <= 10^5`
## Starter Code
```python
class Solution:
    def stringCount(self, n: int) -> int:
```

## Output Format
- Provide the complete solution code in **one markdown code block** with appropriate language identifier. If your response has multiple code blocks, only the last one will be used.
- Implement the function with the exact signature (name, parameters, etc.) specified in the starter code.
- EXCLUDE ALL explanations, code comments, import/package/library statements, additional classes or functions outside of the starter code scope, or starting code like if __name__ == "__main__": or func main() or package main or using namespace std;.
- Use but do not redefine any helper data structures provided in the starter code (even if commented out).
~~~

Claude-3-Sonnet

```python
class Solution:
    def stringCount(self, n: int) -> int:
        MOD = 10**9 + 7
        dp = [[[[0 for _ in range(2)] for _ in range(3)] for _ in range(2)] for _ in range(n+1)]
        dp[0][0][0][0] = 1
        for i in range(n):
            for l in range(2):
                for e in range(3):
                    for t in range(2):
                        count = dp[i][l][e][t]
                        if count == 0:
                            continue
                        dp[i+1][min(l+1, 1)][e][t] = (dp[i+1][min(l+1, 1)][e][t] + count) % MOD
                        dp[i+1][l][min(e+1, 2)][t] = (dp[i+1][l][min(e+1, 2)][t] + count) % MOD
                        dp[i+1][l][e][min(t+1, 1)] = (dp[i+1][l][e][min(t+1, 1)] + count) % MOD
                        dp[i+1][l][e][t] = (dp[i+1][l][e][t] + count * 23) % MOD
        total = pow(26, n, MOD)
        bad = 0
        for l in range(2):
            for e in range(3):
                for t in range(2):
                    if l == 1 and e == 2 and t == 1:
                        continue
                    bad = (bad + dp[n][l][e][t]) % MOD
        return (total - bad) % MOD
```

Figure 4: This figure shows Claude-3.7-Sonnet's inefficient solution using 4D dynamic programming to count strings containing "leet."

Human Expert

```python
class Solution:
    def stringCount(self, n: int) -> int:
        if n < 4:
            return 0  # Cannot fit 'leet' with fewer than 4 characters

        M = 10**9 + 7
        total = pow(26, n, M)  # All possible lowercase strings

        # Inclusion-Exclusion: Subtract cases that fail to meet 'leet' criteria

        # No 'l'
        sub = pow(25, n, M)
        # No 't'
        sub = (sub + pow(25, n, M)) % M
        # 0 or 1 'e'
        sub = (sub + pow(25, n, M) + n * pow(25, n - 1, M)) % M

        # No 'l' and no 't'
        sub = (sub - pow(24, n, M)) % M
        # No 'l' and <=1 'e'
        sub = (sub - pow(24, n, M) - n * pow(24, n - 1, M)) % M
        sub = (sub - pow(24, n, M) - n * pow(24, n - 1, M)) % M
        sub = (sub + pow(23, n, M) + n * pow(23, n - 1, M)) % M
        return (total - sub) % M
```

Figure 5: This figure presents the human expert's elegant solution using the Inclusion-Exclusion Principle, which directly calculates the answer by subtracting invalid configurations from the total possible strings.