Improving the Efficiency of LLM-Generated Code

Dong HUANG

Introduction

- Large language models (LLMs) plays an important role in software development.
- Efficiency of the software affects the validity of the code to be deployed in efficiency-critical environments.
    - Execution time.
    - Memory usage.

Outline

- EffiBench: Benchmarking the Efficiency of Automatically Generated Code.
- EffiLearner: Enhancing Efficiency of Generated Code via Self-Optimization.
  - Execution time profile
  - Memory usage profile

EffiBench: Benchmarking the Efficiency of Automatically Generated Code

NeurIPS 2024

[1] https://arxiv.org/abs/2402.02037

Motivation

- Efficiency of LLM-generated code are crucial.
- Existing datasets mainly focus on the correctness of LLM-generated code, while the efficiency results have been ignored.
- Using existing datasets to measure the efficiency of LLM-generated code exists some problems:
  - Short code snippets
  - Not efficiency-critical tasks
  - Small number of test cases

Dataset Construction

- Select efficiency-critical tasks from LeetCode
- Construct canonical solution
- Construct test case generator to generated 100-1000 test cases
- Dataset statistics

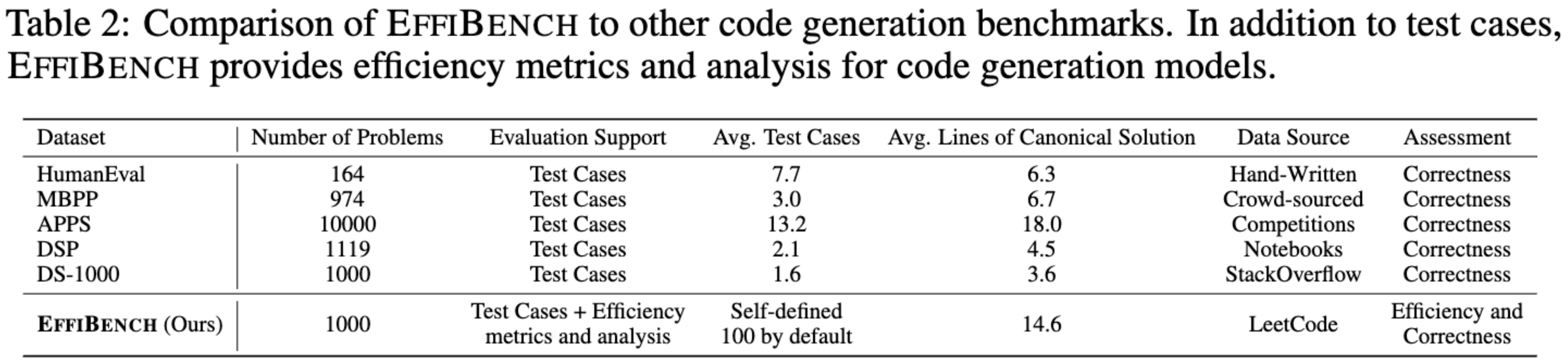


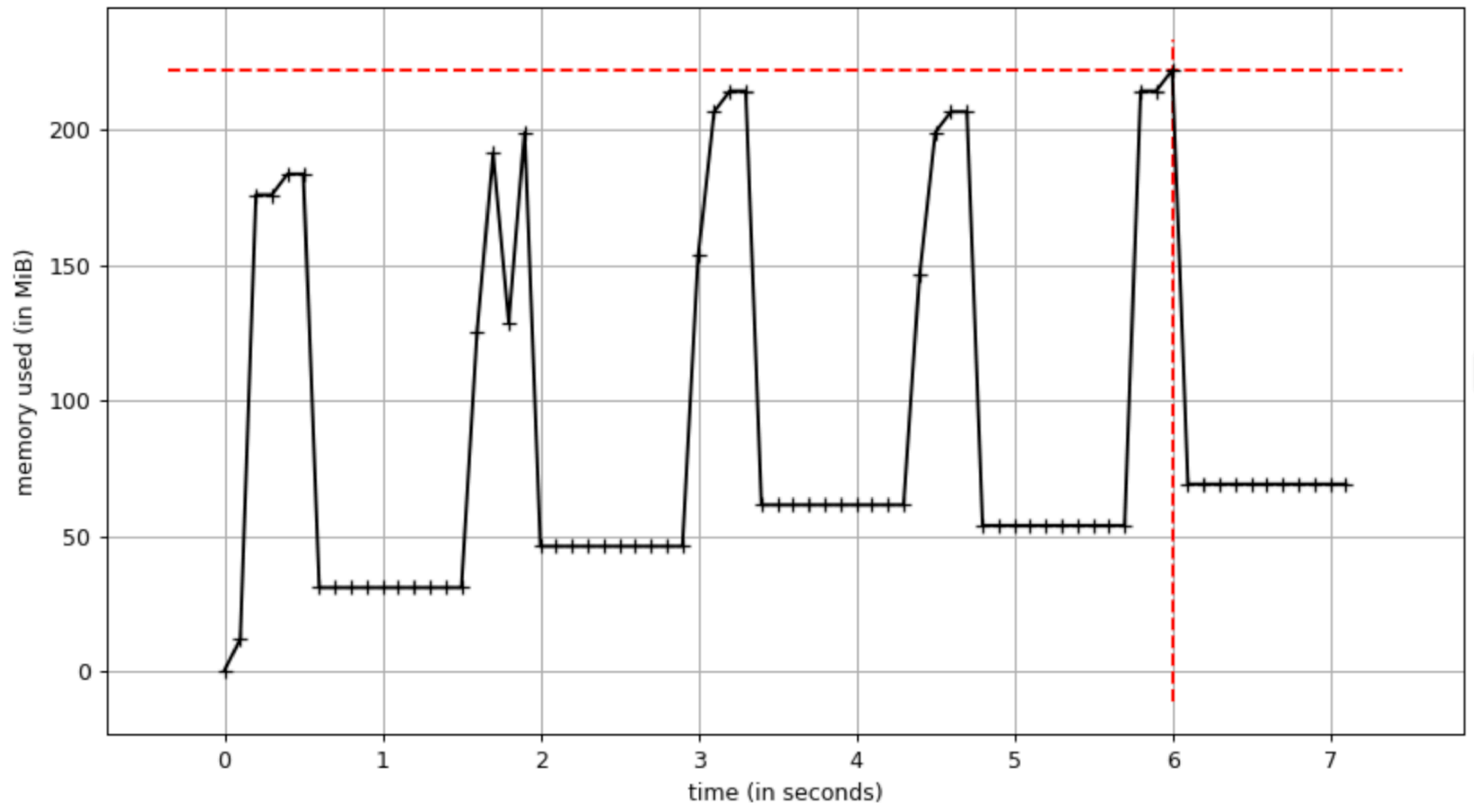
Table 2: Comparison of EffiBench to other code generation benchmarks. In addition to test cases, EffiBench provides efficiency metrics and analysis for code generation models.

Dataset	Number of Problems	Evaluation Support	Avg. Test Cases	Avg. Lines of Canonical Solution	Data Source	Assessment
HumanEval	164	Test Cases	7.7	6.3	Hand-Written	Correctness
MBPP	974	Test Cases	3.0	6.7	Crowd-sourced	Correctness
APPS	10000	Test Cases	13.2	18.0	Competitions	Correctness
DSP	1119	Test Cases	2.1	4.5	Notebooks	Correctness
DS-1000	1000	Test Cases	1.6	3.6	StackOverflow	Correctness
EffiBench (Ours)	1000	Test Cases + Efficiency metrics and analysis	Self-defined 100 by default	14.6	LeetCode	Efficiency and Correctness

Efficiency Metrics

- Execution Time (ET)
- Memory Peak (MU)
- Total Memory Usage (TMU)

- Normalised Metrics:
    - NET
    - NMU
    - NTMU



Evaluation Setup

- Models:
  - CodeLlama, DeepSeek-Coder, OpenCodeInterpreter, StarCoder, WizardCoder , Phind-CodeLlama, XwinCoder, Yi, GPT, Claude
- Machine Setup:
  - Intel Xeon Platinum 8336C CPU with 128 cores, 8 * NVIDIA A100-SXM GPUs, and a total memory capacity of 2.0TB.
- Prompt:
  - Few-shot prompting

End2End

- LLM-generated code are less efficient than canonical solution.
- GPT-4 requires 3.12x average execution time (ET) compared to canonical solution.
- In worst case, GPT-4 generated code requires 13.89x ET than canonical solution.

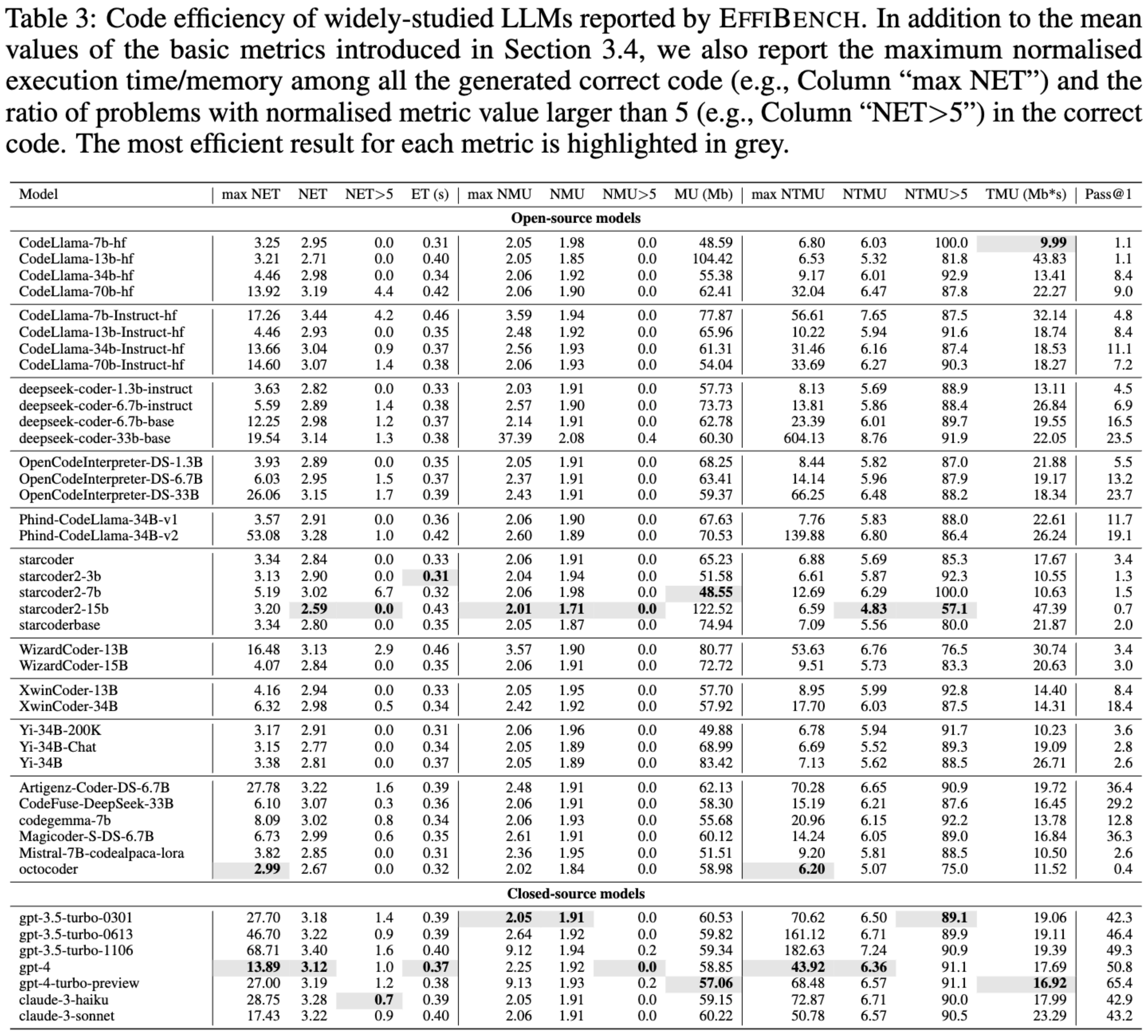


Table 3: Code efficiency of widely-studied LLMs reported by EFFIBENCH. In addition to the mean values of the basic metrics introduced in Section 3.4, we also report the maximum normalised execution time/memory among all the generated correct code (e.g., Column "max NET") and the ratio of problems with normalised metric value larger than 5 (e.g., Column "NET>5") in the correct code. The most efficient result for each metric is highlighted in grey.

Model	max NET	NET	NET>5	ET (s)	max NMU	NMU	NMU>5	MU (Mb)	max NTMU	NTMU	NTMU>5	TMU (Mb*s)	Pass@1
Open-source models													
CodeLlama-7b-hf	3.25	2.95	0.0	0.31	2.05	1.98	0.0	48.59	6.80	6.03	100.0	**9.99**	1.1
CodeLlama-13b-hf	3.21	2.71	0.0	0.40	2.05	1.85	0.0	104.42	6.53	5.32	81.8	43.83	1.1
CodeLlama-34b-hf	4.46	2.98	0.0	0.34	2.06	1.92	0.0	55.38	9.17	6.01	92.9	13.41	8.4
CodeLlama-70b-hf	13.92	3.19	4.4	0.42	2.06	1.90	0.0	62.41	32.04	6.47	87.8	22.27	9.0
CodeLlama-7b-Instruct-hf	17.26	3.44	4.2	0.46	3.59	1.94	0.0	77.87	56.61	7.65	87.5	32.14	4.8
CodeLlama-13b-Instruct-hf	4.46	2.93	0.0	0.35	2.48	1.92	0.0	65.96	10.22	5.94	91.6	18.74	8.4
CodeLlama-34b-Instruct-hf	13.66	3.04	0.9	0.37	2.56	1.93	0.0	61.31	31.46	6.16	87.4	18.53	11.1
CodeLlama-70b-Instruct-hf	14.60	3.07	1.4	0.38	2.06	1.93	0.0	54.04	33.69	6.27	90.3	18.27	7.2
deepseek-coder-1.3b-instruct	3.63	2.82	0.0	0.33	2.03	1.91	0.0	57.73	8.13	5.69	88.9	13.11	4.5
deepseek-coder-6.7b-instruct	5.59	2.89	1.4	0.38	2.57	1.90	0.0	73.73	13.81	5.86	88.4	26.84	6.9
deepseek-coder-6.7b-base	12.25	2.98	1.2	0.37	2.14	1.91	0.0	62.78	23.39	6.01	89.7	19.55	16.5
deepseek-coder-33b-base	19.54	3.14	1.3	0.38	37.39	2.08	0.4	60.30	604.13	8.76	91.9	22.05	23.5
OpenCodeInterpreter-DS-1.3B	3.93	2.89	0.0	0.35	2.05	1.91	0.0	68.25	8.44	5.82	87.0	21.88	5.5
OpenCodeInterpreter-DS-6.7B	6.03	2.95	1.5	0.37	2.37	1.91	0.0	63.41	14.14	5.96	87.9	19.17	13.2
OpenCodeInterpreter-DS-33B	26.06	3.15	1.7	0.39	2.43	1.91	0.0	59.37	66.25	6.48	88.2	18.34	23.7
Phind-CodeLlama-34B-v1	3.57	2.91	0.0	0.36	2.06	1.90	0.0	67.63	7.76	5.83	88.0	22.61	11.7
Phind-CodeLlama-34B-v2	53.08	3.28	1.0	0.42	2.60	1.89	0.0	70.53	139.88	6.80	86.4	26.24	19.1
starcoder	3.34	2.84	0.0	0.33	2.06	1.91	0.0	65.23	6.88	5.69	85.3	17.67	3.4
starcoder2-3b	3.13	2.90	0.0	**0.31**	2.04	1.94	0.0	51.58	6.61	5.87	92.3	10.55	1.3
starcoder2-7b	5.19	3.02	6.7	0.32	2.06	1.98	0.0	**48.55**	12.69	6.29	100.0	10.63	1.5
starcoder2-15b	3.20	**2.59**	**0.0**	0.43	**2.01**	**1.71**	**0.0**	122.52	6.59	**4.83**	**57.1**	47.39	0.7
starcoderbase	3.34	2.80	0.0	0.35	2.05	1.87	0.0	74.94	7.09	5.56	80.0	21.87	2.0
WizardCoder-13B	16.48	3.13	2.9	0.46	3.57	1.90	0.0	80.77	53.63	6.76	76.5	30.74	3.4
WizardCoder-15B	4.07	2.84	0.0	0.35	2.06	1.91	0.0	72.72	9.51	5.73	83.3	20.63	3.0
XwinCoder-13B	4.16	2.94	0.0	0.33	2.05	1.95	0.0	57.70	8.95	5.99	92.8	14.40	8.4
XwinCoder-34B	6.32	2.98	0.5	0.34	2.42	1.92	0.0	57.92	17.70	6.03	87.5	14.31	18.4
Yi-34B-200K	3.17	2.91	0.0	0.31	2.06	1.96	0.0	49.88	6.78	5.94	91.7	10.23	3.6
Yi-34B-Chat	3.15	2.77	0.0	0.34	2.05	1.89	0.0	68.99	6.69	5.52	89.3	19.09	2.8
Yi-34B	3.38	2.81	0.0	0.37	2.05	1.89	0.0	83.42	7.13	5.62	88.5	26.71	2.6
Artigenz-Coder-DS-6.7B	27.78	3.22	1.6	0.39	2.48	1.91	0.0	62.13	70.28	6.65	90.9	19.72	36.4
CodeFuse-DeepSeek-33B	6.10	3.07	0.3	0.36	2.06	1.91	0.0	58.30	15.19	6.21	87.6	16.45	29.2
codegemma-7b	8.09	3.02	0.8	0.34	2.06	1.93	0.0	55.68	20.96	6.15	92.2	13.78	12.8
Magicoder-S-DS-6.7B	6.73	2.99	0.6	0.35	2.61	1.91	0.0	60.12	14.24	6.05	89.0	16.84	36.3
Mistral-7B-codealpaca-lora	3.82	2.85	0.0	0.31	2.36	1.95	0.0	51.51	9.20	5.81	88.5	10.50	2.6
octocoder	**2.99**	2.67	0.0	0.32	2.02	1.84	0.0	58.98	**6.20**	5.07	75.0	11.52	0.4
Closed-source models													
gpt-3.5-turbo-0301	27.70	3.18	1.4	0.39	**2.05**	**1.91**	0.0	60.53	70.62	6.50	**89.1**	19.06	42.3
gpt-3.5-turbo-0613	46.70	3.22	0.9	0.39	2.64	1.92	0.0	59.82	161.12	6.71	89.9	19.11	46.4
gpt-3.5-turbo-1106	68.71	3.40	1.6	0.40	9.12	1.94	0.2	59.34	182.63	7.24	90.9	19.39	49.3
gpt-4	**13.89**	**3.12**	1.0	**0.37**	2.25	1.92	**0.0**	58.85	**43.92**	**6.36**	91.1	17.69	50.8
gpt-4-turbo-preview	27.00	3.19	1.2	0.38	9.13	1.93	0.2	**57.06**	68.48	6.57	91.1	**16.92**	65.4
claude-3-haiku	28.75	3.28	**0.7**	0.39	2.05	1.91	0.0	59.15	72.87	6.71	90.0	17.99	42.9
claude-3-sonnet	17.43	3.22	0.9	0.40	2.06	1.91	0.0	60.22	50.78	6.57	90.5	23.29	43.2

Results with Identical Problems

- Similar to the results in all problems, GPT-4 generated code achieve most efficient results.
- However, GPT-4 generated code still less efficient than canonical solution

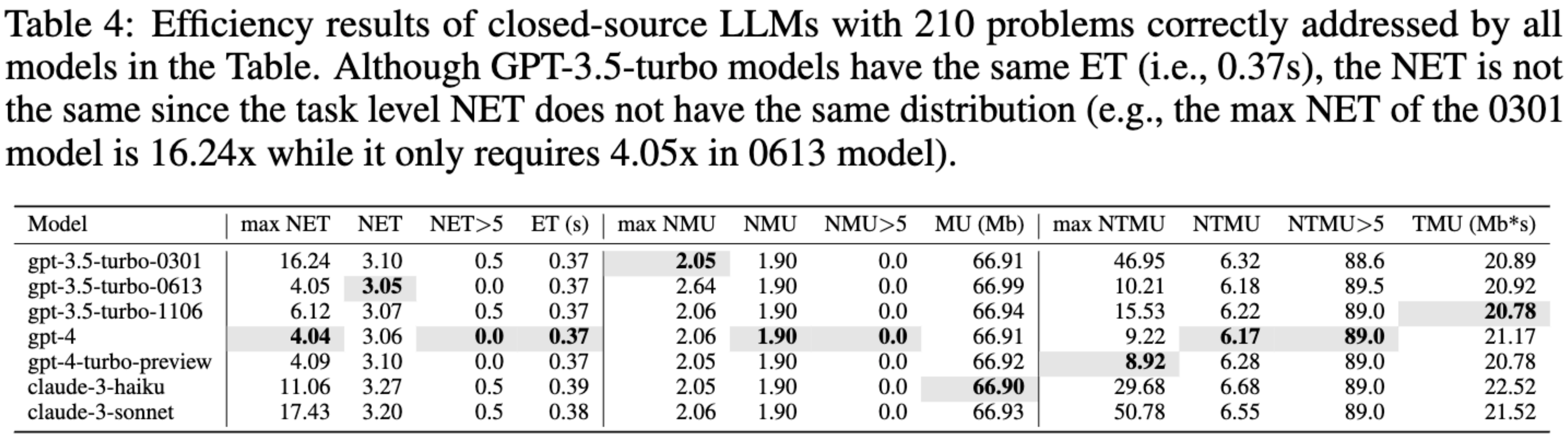
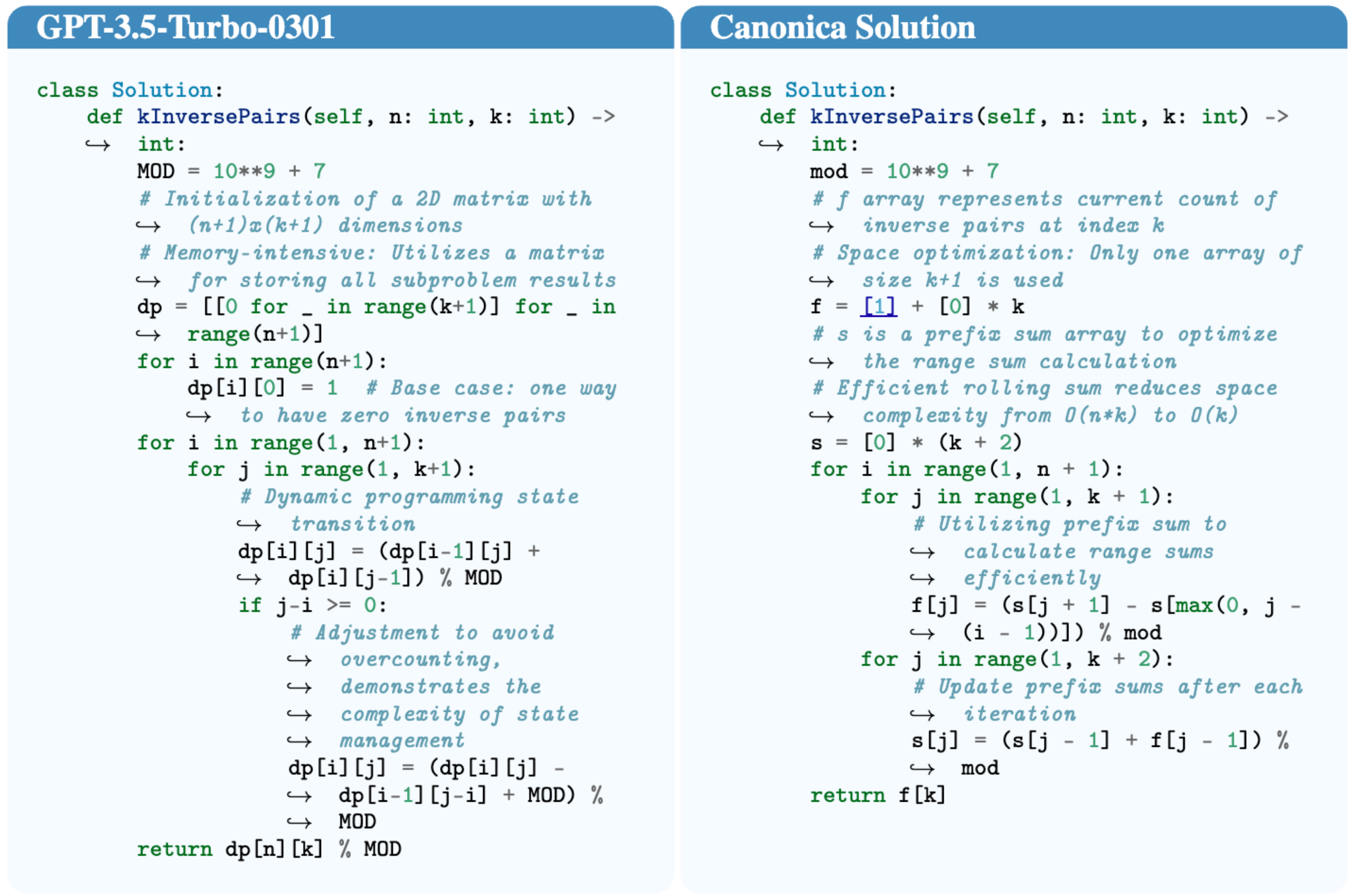


Table 4: Efficiency results of closed-source LLMs with 210 problems correctly addressed by all models in the Table. Although GPT-3.5-turbo models have the same ET (i.e., 0.37s), the NET is not the same since the task level NET does not have the same distribution (e.g., the max NET of the 0301 model is 16.24x while it only requires 4.05x in 0613 model).

Model	max NET	NET	NET>5	ET (s)	max NMU	NMU	NMU>5	MU (Mb)	max NTMU	NTMU	NTMU>5	TMU (Mb*s)
gpt-3.5-turbo-0301	16.24	3.10	0.5	0.37	**2.05**	1.90	0.0	66.91	46.95	6.32	88.6	20.89
gpt-3.5-turbo-0613	4.05	**3.05**	0.0	0.37	2.64	1.90	0.0	66.99	10.21	6.18	89.5	20.92
gpt-3.5-turbo-1106	6.12	3.07	0.5	0.37	2.06	1.90	0.0	66.94	15.53	6.22	89.0	**20.78**
gpt-4	**4.04**	3.06	**0.0**	**0.37**	2.06	**1.90**	**0.0**	66.91	9.22	**6.17**	**89.0**	21.17
gpt-4-turbo-preview	4.09	3.10	0.0	0.37	2.05	1.90	0.0	66.92	**8.92**	6.28	89.0	20.78
claude-3-haiku	11.06	3.27	0.5	0.39	2.05	1.90	0.0	**66.90**	29.68	6.68	89.0	22.52
claude-3-sonnet	17.43	3.20	0.5	0.38	2.06	1.90	0.0	66.93	50.78	6.55	89.0	21.52

Worst Case Analysis

- The code generated by GPT-3.5-Turbo-0301 uses a two-dimensional array to store intermediate results.
- In contrast, the canonical solution employs two one-dimensional arrays.



GPT-3.5-Turbo-0301

```
class Solution:
    def kInversePairs(self, n: int, k: int) ->
    ↪   int:
        MOD = 10**9 + 7
        # Initialization of a 2D matrix with
        ↪  (n+1)x(k+1) dimensions
        # Memory-intensive: Utilizes a matrix
        ↪  for storing all subproblem results
        dp = [[0 for _ in range(k+1)] for _ in
        ↪  range(n+1)]
        for i in range(n+1):
            dp[i][0] = 1  # Base case: one way
            ↪  to have zero inverse pairs
        for i in range(1, n+1):
            for j in range(1, k+1):
                # Dynamic programming state
                ↪  transition
                dp[i][j] = (dp[i-1][j] +
                ↪  dp[i][j-1]) % MOD
                if j-i >= 0:
                    # Adjustment to avoid
                    ↪  overcounting,
                    ↪  demonstrates the
                    ↪  complexity of state
                    ↪  management
                    dp[i][j] = (dp[i][j] -
                    ↪  dp[i-1][j-i] + MOD) %
                    ↪  MOD
        return dp[n][k] % MOD
```

Canonica Solution

```
class Solution:
    def kInversePairs(self, n: int, k: int) ->
    ↪   int:
        mod = 10**9 + 7
        # f array represents current count of
        ↪  inverse pairs at index k
        # Space optimization: Only one array of
        ↪  size k+1 is used
        f = [1] + [0] * k
        # s is a prefix sum array to optimize
        ↪  the range sum calculation
        # Efficient rolling sum reduces space
        ↪  complexity from O(n*k) to O(k)
        s = [0] * (k + 2)
        for i in range(1, n + 1):
            for j in range(1, k + 1):
                # Utilizing prefix sum to
                ↪  calculate range sums
                ↪  efficiently
                f[j] = (s[j + 1] - s[max(0, j -
                ↪  (i - 1))]) % mod
            for j in range(1, k + 2):
                # Update prefix sums after each
                ↪  iteration
                s[j] = (s[j - 1] + f[j - 1]) %
                ↪  mod
        return f[k]
```

Figure 2: A case illustration of GPT-3.5-turbo-0301 and *canonica_solution*. GPT-3.5-turbo-0301 generated code requires 70.62x memory usage compared with *canonical_solution*. GPT-3.5-turbo-0301 generated code employs a 2-dimensional matrix to manage state transitions, leading to substantial memory overhead, particularly evident when the parameters n and k are large. In contrast, the *canonical_solution* optimizes memory usage by utilizing a rolling sum technique and a single-dimensional dynamic array, significantly reducing the space complexity from $O(n \times k)$ to $O(k)$.

Summary

- We propose the first dataset used to measure the efficiency of LLM-generated code
- We conduct an extensive evaluation of 42 LLMs on EffiBench, revealing that even state-of-the-art LLMs (e.g. GPT-4) exhibit significant inefficiencies compared to optimal human-written solutions

EffiLearner: Enhancing Efficiency of Generated Code via Self-Optimization

NeurIPS 2024

[2] https://arxiv.org/abs/2405.15189

Motivation

- The correctness of LLM-generated code are achieving near optimal results
- However, existing works demonstrate that LLM-generated code are inefficient

Self-Optimization with Overhead Profile

- Generating source code with task description
- Overhead profiling:
  - Execution Time Profiling: We profile the execution time of each line in LLM-generated code with line_profiler
  - Memory Usage Profiling: We profile the memory usage of each line with memory_profile
- Code Refinement
- Prompt Construction

Code Refinement

- Execute LLM-generated code to obtain the overhead profiles.
- Feed overhead profiles into LLM for code efficiency optimisation

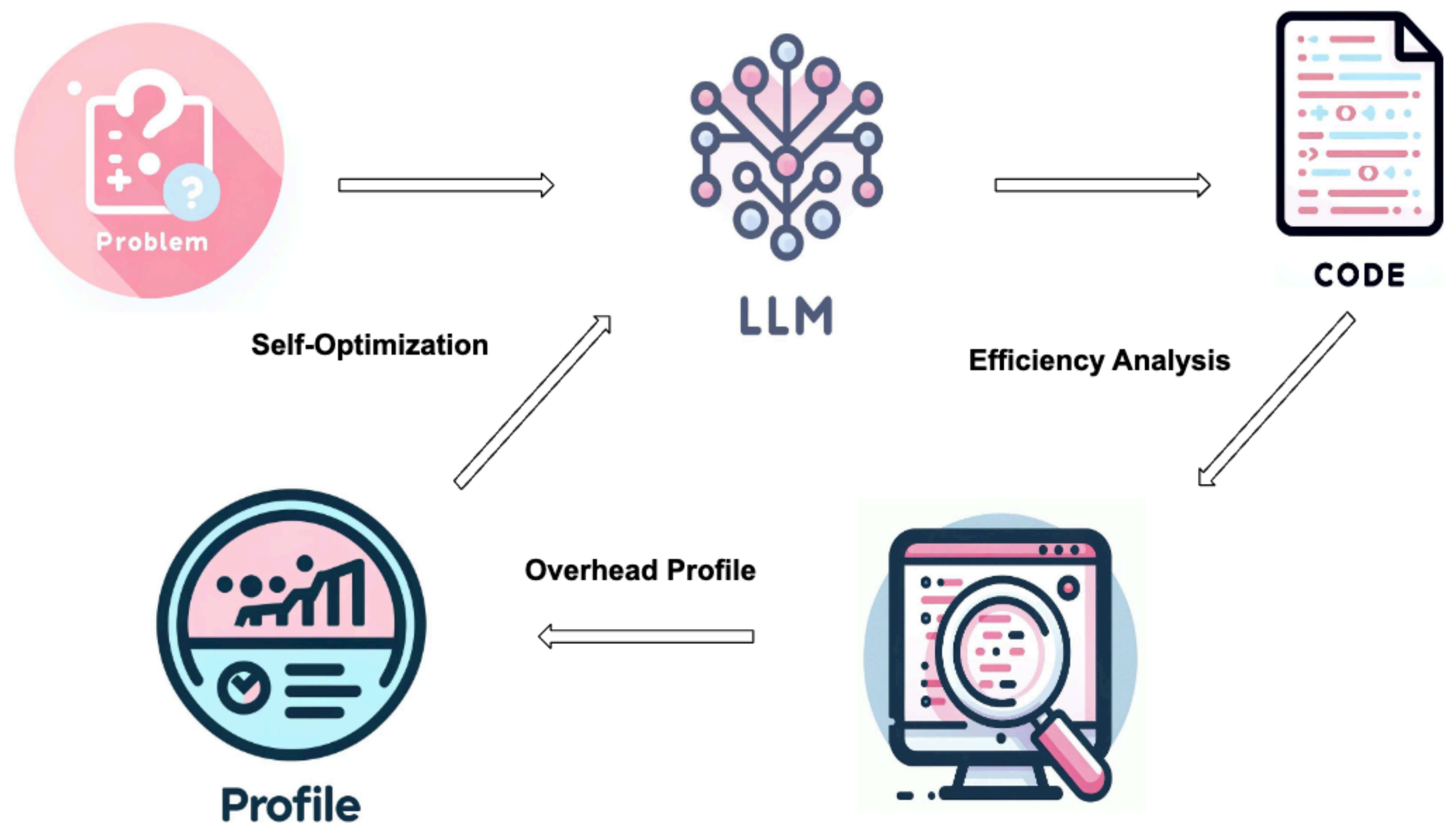


Figure 2: Pipeline of EFFI-LEARNER. LLMs first generate code for the given problem. This code is then executed locally to gather overhead profiles. These profiles are subsequently utilized by the LLMs to optimize the code in successive iterations, thereby enhancing the overall efficiency of the generated code. A comprehensive illustration is provided in the Appendix Figure 4-Figure 11.

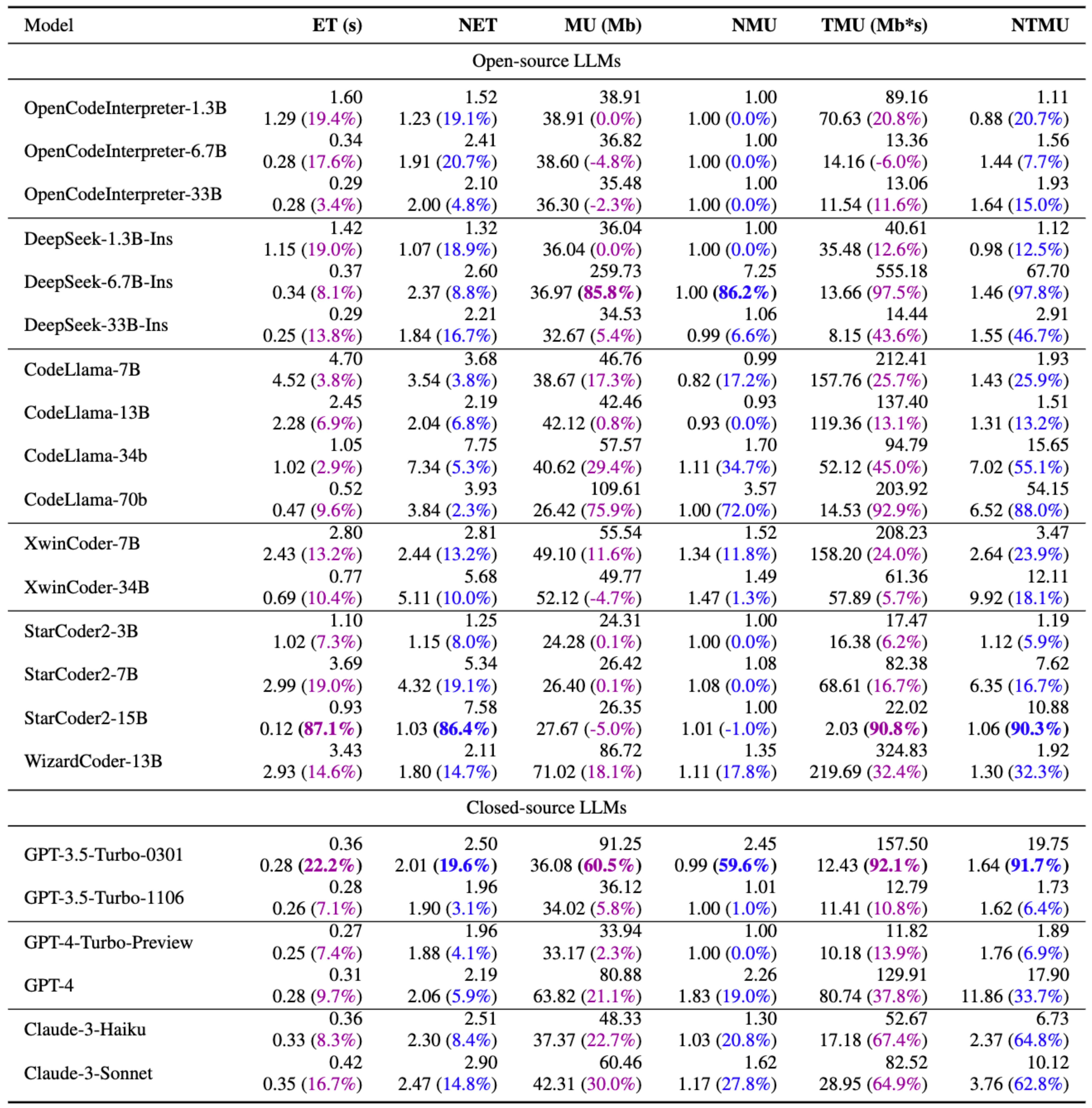
Evaluation Setup

- Models:
  - CodeLlama, DeepSeek-Coder, OpenCodeInterpreter, StarCoder, WizardCoder, XwinCoder, GPT, Claude
- Machine Setup:
  - Intel Xeon Platinum 8336C CPU with 128 cores, 8 * NVIDIA A100-SXM GPUs, and a total memory capacity of 2.0TB.
- Prompt:
  - Few-shot prompting
- Dataset:
  - EffiBench

End2End

- EffiLearner improve the efficiency of LLM-generated code.
- In OpenCodeInterpreter-1.3B, the execution time for its generated code decreases fin 1.60 (s) to 1.29 (s), a reduction of 19.4% execution time.
- .The memory peak of GPT-3.5-Turbo-0301 also decrease from 91.25 (Mb) to 36.08 (Mb), which reduces 60.5% memory peak.

Table 1: Code efficiency of LLMs with EFFI-LEARNER on EffiBench. The percentage in the brackets indicates the extent of the reduction for each respective item. Top performing LLMs are highlighted.



Model	ET (s)	NET	MU (Mb)	NMU	TMU (Mb*s)	NTMU
Open-source LLMs						
OpenCodeInterpreter-1.3B	1.60	1.52	38.91	1.00	89.16	1.11
	1.29 (19.4%)	1.23 (19.1%)	38.91 (0.0%)	1.00 (0.0%)	70.63 (20.8%)	0.88 (20.7%)
OpenCodeInterpreter-6.7B	0.34	2.41	36.82	1.00	13.36	1.56
	0.28 (17.6%)	1.91 (20.7%)	38.60 (-4.8%)	1.00 (0.0%)	14.16 (-6.0%)	1.44 (7.7%)
OpenCodeInterpreter-33B	0.29	2.10	35.48	1.00	13.06	1.93
	0.28 (3.4%)	2.00 (4.8%)	36.30 (-2.3%)	1.00 (0.0%)	11.54 (11.6%)	1.64 (15.0%)
DeepSeek-1.3B-Ins	1.42	1.32	36.04	1.00	40.61	1.12
	1.15 (19.0%)	1.07 (18.9%)	36.04 (0.0%)	1.00 (0.0%)	35.48 (12.6%)	0.98 (12.5%)
DeepSeek-6.7B-Ins	0.37	2.60	259.73	7.25	555.18	67.70
	0.34 (8.1%)	2.37 (8.8%)	36.97 (**85.8%**)	1.00 (**86.2%**)	13.66 (97.5%)	1.46 (97.8%)
DeepSeek-33B-Ins	0.29	2.21	34.53	1.06	14.44	2.91
	0.25 (13.8%)	1.84 (16.7%)	32.67 (5.4%)	0.99 (6.6%)	8.15 (43.6%)	1.55 (46.7%)
CodeLlama-7B	4.70	3.68	46.76	0.99	212.41	1.93
	4.52 (3.8%)	3.54 (3.8%)	38.67 (17.3%)	0.82 (17.2%)	157.76 (25.7%)	1.43 (25.9%)
CodeLlama-13B	2.45	2.19	42.46	0.93	137.40	1.51
	2.28 (6.9%)	2.04 (6.8%)	42.12 (0.8%)	0.93 (0.0%)	119.36 (13.1%)	1.31 (13.2%)
CodeLlama-34b	1.05	7.75	57.57	1.70	94.79	15.65
	1.02 (2.9%)	7.34 (5.3%)	40.62 (29.4%)	1.11 (34.7%)	52.12 (45.0%)	7.02 (55.1%)
CodeLlama-70b	0.52	3.93	109.61	3.57	203.92	54.15
	0.47 (9.6%)	3.84 (2.3%)	26.42 (75.9%)	1.00 (72.0%)	14.53 (92.9%)	6.52 (88.0%)
XwinCoder-7B	2.80	2.81	55.54	1.52	208.23	3.47
	2.43 (13.2%)	2.44 (13.2%)	49.10 (11.6%)	1.34 (11.8%)	158.20 (24.0%)	2.64 (23.9%)
XwinCoder-34B	0.77	5.68	49.77	1.49	61.36	12.11
	0.69 (10.4%)	5.11 (10.0%)	52.12 (-4.7%)	1.47 (1.3%)	57.89 (5.7%)	9.92 (18.1%)
StarCoder2-3B	1.10	1.25	24.31	1.00	17.47	1.19
	1.02 (7.3%)	1.15 (8.0%)	24.28 (0.1%)	1.00 (0.0%)	16.38 (6.2%)	1.12 (5.9%)
StarCoder2-7B	3.69	5.34	26.42	1.08	82.38	7.62
	2.99 (19.0%)	4.32 (19.1%)	26.40 (0.1%)	1.08 (0.0%)	68.61 (16.7%)	6.35 (16.7%)
StarCoder2-15B	0.93	7.58	26.35	1.00	22.02	10.88
	0.12 (**87.1%**)	1.03 (**86.4%**)	27.67 (-5.0%)	1.01 (-1.0%)	2.03 (**90.8%**)	1.06 (**90.3%**)
WizardCoder-13B	3.43	2.11	86.72	1.35	324.83	1.92
	2.93 (14.6%)	1.80 (14.7%)	71.02 (18.1%)	1.11 (17.8%)	219.69 (32.4%)	1.30 (32.3%)
Closed-source LLMs						
GPT-3.5-Turbo-0301	0.36	2.50	91.25	2.45	157.50	19.75
	0.28 (**22.2%**)	2.01 (**19.6%**)	36.08 (**60.5%**)	0.99 (**59.6%**)	12.43 (**92.1%**)	1.64 (**91.7%**)
GPT-3.5-Turbo-1106	0.28	1.96	36.12	1.01	12.79	1.73
	0.26 (7.1%)	1.90 (3.1%)	34.02 (5.8%)	1.00 (1.0%)	11.41 (10.8%)	1.62 (6.4%)
GPT-4-Turbo-Preview	0.27	1.96	33.94	1.00	11.82	1.89
	0.25 (7.4%)	1.88 (4.1%)	33.17 (2.3%)	1.00 (0.0%)	10.18 (13.9%)	1.76 (6.9%)
GPT-4	0.31	2.19	80.88	2.26	129.91	17.90
	0.28 (9.7%)	2.06 (5.9%)	63.82 (21.1%)	1.83 (19.0%)	80.74 (37.8%)	11.86 (33.7%)
Claude-3-Haiku	0.36	2.51	48.33	1.30	52.67	6.73
	0.33 (8.3%)	2.30 (8.4%)	37.37 (22.7%)	1.03 (20.8%)	17.18 (67.4%)	2.37 (64.8%)
Claude-3-Sonnet	0.42	2.90	60.46	1.62	82.52	10.12
	0.35 (16.7%)	2.47 (14.8%)	42.31 (30.0%)	1.17 (27.8%)	28.95 (64.9%)	3.76 (62.8%)

Feedback of Overhead Profile

- EffiLearner achieves SOTA efficiency results when we provide both execution time and memory usage profile

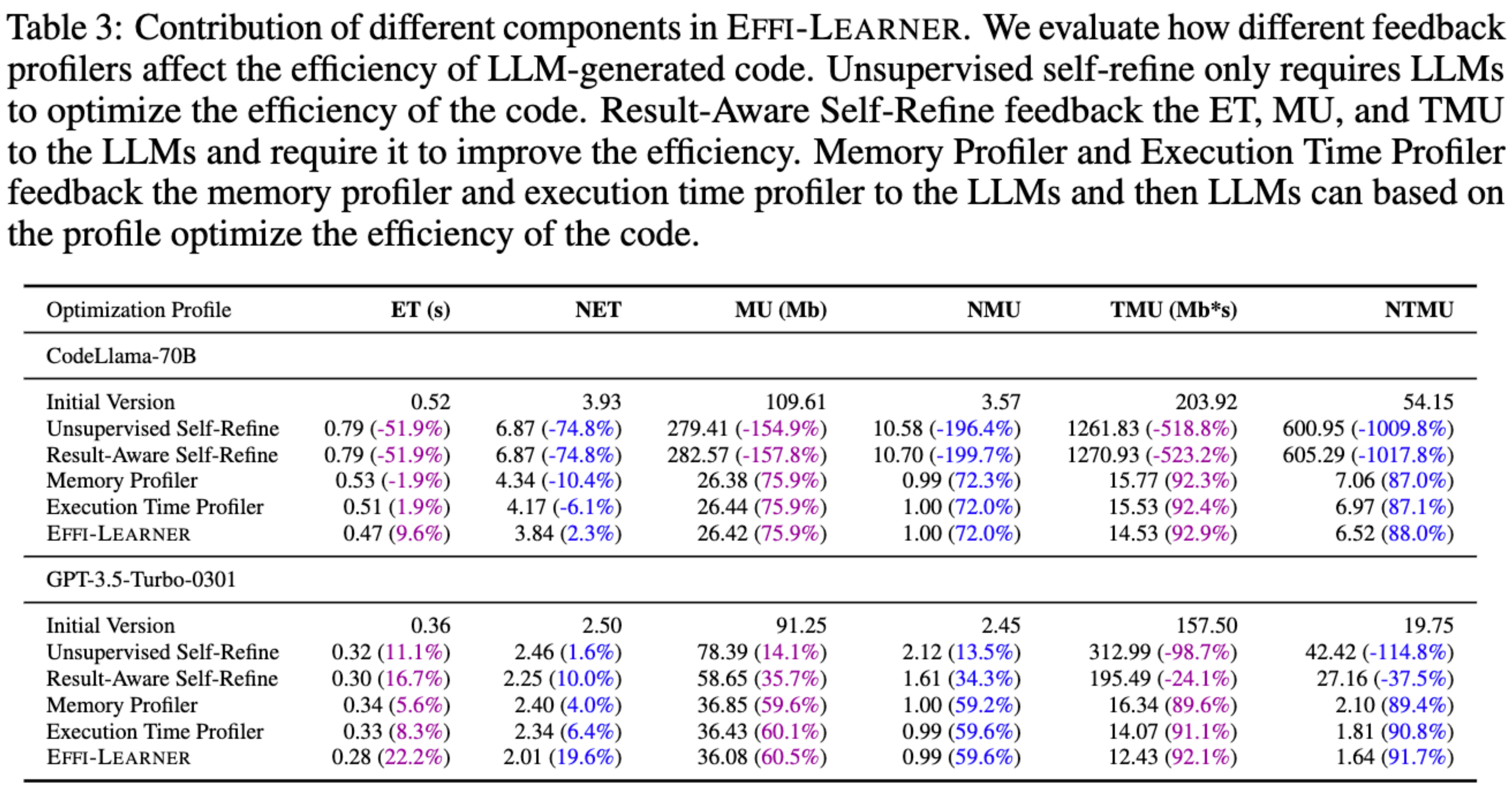


Table 3: Contribution of different components in EFFI-LEARNER. We evaluate how different feedback profilers affect the efficiency of LLM-generated code. Unsupervised self-refine only requires LLMs to optimize the efficiency of the code. Result-Aware Self-Refine feedback the ET, MU, and TMU to the LLMs and require it to improve the efficiency. Memory Profiler and Execution Time Profiler feedback the memory profiler and execution time profiler to the LLMs and then LLMs can based on the profile optimize the efficiency of the code.

Optimization Profile	ET (s)	NET	MU (Mb)	NMU	TMU (Mb*s)	NTMU
CodeLlama-70B						
Initial Version	0.52	3.93	109.61	3.57	203.92	54.15
Unsupervised Self-Refine	0.79 (-51.9%)	6.87 (-74.8%)	279.41 (-154.9%)	10.58 (-196.4%)	1261.83 (-518.8%)	600.95 (-1009.8%)
Result-Aware Self-Refine	0.79 (-51.9%)	6.87 (-74.8%)	282.57 (-157.8%)	10.70 (-199.7%)	1270.93 (-523.2%)	605.29 (-1017.8%)
Memory Profiler	0.53 (-1.9%)	4.34 (-10.4%)	26.38 (75.9%)	0.99 (72.3%)	15.77 (92.3%)	7.06 (87.0%)
Execution Time Profiler	0.51 (1.9%)	4.17 (-6.1%)	26.44 (75.9%)	1.00 (72.0%)	15.53 (92.4%)	6.97 (87.1%)
EFFI-LEARNER	0.47 (9.6%)	3.84 (2.3%)	26.42 (75.9%)	1.00 (72.0%)	14.53 (92.9%)	6.52 (88.0%)
GPT-3.5-Turbo-0301						
Initial Version	0.36	2.50	91.25	2.45	157.50	19.75
Unsupervised Self-Refine	0.32 (11.1%)	2.46 (1.6%)	78.39 (14.1%)	2.12 (13.5%)	312.99 (-98.7%)	42.42 (-114.8%)
Result-Aware Self-Refine	0.30 (16.7%)	2.25 (10.0%)	58.65 (35.7%)	1.61 (34.3%)	195.49 (-24.1%)	27.16 (-37.5%)
Memory Profiler	0.34 (5.6%)	2.40 (4.0%)	36.85 (59.6%)	1.00 (59.2%)	16.34 (89.6%)	2.10 (89.4%)
Execution Time Profiler	0.33 (8.3%)	2.34 (6.4%)	36.43 (60.1%)	0.99 (59.6%)	14.07 (91.1%)	1.81 (90.8%)
EFFI-LEARNER	0.28 (22.2%)	2.01 (19.6%)	36.08 (60.5%)	0.99 (59.6%)	12.43 (92.1%)	1.64 (91.7%)

Summary

- We propose EffiLearner, a novel self-optimization framework that leverages execution overhead profiles to guide LLMs in improving code efficiency.
- Extensive experiments demonstrate that EffiLearner significantly enhances the efficiency of LLM-generated code, achieving substantial reductions in execution time and memory usage.

Thanks for listening

Introduction

- Large language models (LLMs) plays an important role in software development.
- Efficiency of the software affects the validity of the code to be deployed in efficiency-critical environments.
  - Execution time.
  - Memory usage.
- We benchmark and improve the efficiency of LLM-generated code with overhead profiles.
  - Execution time profile
  - Memory usage profile