EffiBench: Benchmarking the Efficiency of Automatically Generated Code

Anonymous Authors[1]

Abstract

Code generation models have increasingly become integral to aiding software development, offering assistance in tasks such as code completion, debugging, and code translation. Although current research has thoroughly examined the correctness of the code produced by code generation models, a vital aspect — the efficiency of the generated code — has often been neglected. This paper presents EffiBench, a benchmark with 1,000 efficiency-critical Python coding problems for assessing the efficiency of code generated by code generation models. EffiBench contains a diverse set of LeetCode coding problems. Each problem is paired with an executable human-written canonical solution, which obtains SOTA efficiency in LeetCode solution leaderboard. With EffiBench, we empirically examine the ability of 21 large language models (13 open-source and 8 closed-source) in generating efficient code. The results demonstrate that GPT-4-turbo generates the most efficient code, outperforming Palm-2-chat-bison, Claude-instant-1, Gemini-pro, GPT-4, and GPT-3.5. Nevertheless, its code efficiency is still worse than the efficiency of human-written canonical solutions. In particular, the average / worst execution time of GPT-4-turbo generated code is 1.69 / 45.49 times that of the canonical solutions. The source code of EffiBench is released on https://anonymous.4open.science/r/EffiBench-0F60.

1. Introduction

Code generation models have increasingly been adopted in aiding software developers across various coding tasks, such as code completion, debugging, and code translation. Notable examples of such models are StarCoder (Li et al., 2023a), CodeLlama (Rozière et al., 2023), ChatGPT, WizardCoder (Luo et al., 2023), CodeGen (Nijkamp et al., 2023), Copilot, and CodeGeeX (Zheng et al., 2023) which have been seamlessly integrated into popular development environments like Visual Studio Code (VSCode). These code generation models generate code snippets based on coding instructions and offer intelligent recommendations, which significantly augments developer productivity.

Recent researchers have developed a variety of datasets and benchmarks to evaluate the **correctness** of the code generated by these code generation models. For example, Chen et al. (2021b) introduced HumanEval with 164 basic programming tasks in Python language, Austin et al. (2021) created MBPP (Mostly Basic Programming Problems), Lai et al. (2022) developed D-1000 which is specifically curated for data science tasks, and Hendrycks et al. (2021) proposed APPS with code collected from competition websites such as CodeForces and Codewars. These datasets have been widely recognized and employed in benchmarking code generation models, in particular large language models (Chen et al., 2021b; Austin et al., 2021; Wang et al., 2022; Lai et al., 2022).

Nevertheless, there is a notable oversight in benchmarking the efficiency of code produced by code generation models in the literature. Efficient code executes faster and consumes less memory and processing power, which is especially important in resource-constrained environments like mobile devices or embedded systems. It also means lower operational costs for applications that run on servers. Furthermore, efficient code contributes to reduced energy consumption, aligning with environmental sustainability goals. Therefore, the efficiency of code is crucial for building **scalable and sustainable** software to meet the growing demands of the digital world.

The efficiency of two correctly generated code snippets can be significantly different. Figure 1 illustrates an example of such efficiency disparity with code generated by Copilot and GPT-4, respectively when tasked with merging two sorted arrays. Copilot generates a function that first concatenates the arrays and then applies a basic Bubble Sort algorithm. The code is technically correct but has a sub-optimal time complexity of $O((n+m)^2)$ and space complexity of $O(n+m)$, where n and m are the array lengths. In contrast, GPT-4

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

Problem

Write a function to merge two sorted arrays into a single sorted array.
Input: Two arrays, array1 and array2, each sorted in non-decreasing order.
Output: A single array that combines the elements of array1 and array2, sorted in non-decreasing order.
Example: Given array1 = [2, 7, 11, 15], array2 = [5, 8, 15, 17], return [2, 5, 7, 8, 11, 15, 15, 17].

VS Code Copilot v1.156.683

```
def mergeArrays(array1, array2):
    # Manually concatenating the two arrays
    merged_array = []
    for element in array1:
        merged_array.append(element)
    for element in array2:
        merged_array.append(element)

    n = len(merged_array)
    for i in range(n):
        for j in range(0, n-i-1):
            if merged_array[j] >
            ↪  merged_array[j+1]:
                merged_array[j],
                ↪  merged_array[j+1] =
                ↪  merged_array[j+1],
                ↪  merged_array[j]

    return merged_array
```

GPT-4

```
def mergeArrays(array1, array2):
    # Merging two arrays efficiently
    result = []
    i, j = 0, 0
    while i < len(array1) and j < len(array2):
        if array1[i] < array2[j]:
            result.append(array1[i])
            i += 1
        else:
            result.append(array2[j])
            j += 1
    # Append remaining elements of array1
    while i < len(array1):
        result.append(array1[i])
        i += 1
    # Append remaining elements of array2
    while j < len(array2):
        result.append(array2[j])
        j += 1
    return result
```

Figure 1. Copilot and GPT-4 generate code with distinct time complexity. Code accessed on January 15, 2024.

generates a function that merges the arrays more efficiently, by systematically comparing and appending elements from each array, effectively combining them in a single pass. The time complexity of this method is $O(n+m)$, reflecting a linear relationship with the sum of the lengths of the two arrays. Furthermore, its space complexity remains $O(n+m)$. The disparity in efficiency demonstrated by Figure 1 highlights the necessity to benchmark code generation models from the perspective of code efficiency.

Intuitively, one might consider employing existing code generation correctness benchmarks such as HumanEval, MBPP, and APPS to benchmark code efficiency. However, these datasets have several limitations in assessing efficiency. First, the coding tasks are often simple tasks that can be completed with short code snippets. This simplicity often leads to indistinguishable efficiency across different code generation models. Second, many tasks in these datasets are not efficiency-critical, and the reported efficiency issues would be less significant than with efficiency-critical tasks. Third, these datasets lack extensive and varied test cases that can thoroughly assess code efficiency under varied and substantial computational loads.

This paper introduces EffiBench, a benchmark specifically designed for evaluating code efficiency for code generation models. EffiBench comprises 1,000 efficiency-critical code generation problems selected from LeetCode. Each coding problem is paired with an executable manually-written canonical solution which has been awarded the highest rating on LeetCode for its optimal time and space efficiency. We also developed a test case generator to produce a vast number of test cases for each problem to allow for an in-depth and comprehensive analysis of the code efficiency. Moreover, EffiBench integrates a comprehensive set of efficiency metrics, such as execution time (ET), normalised execution time (NET), maximum memory usage (MU), and total memory usage during execution time (TMU).

With EffiBench, we conduct an empirical study to assess the efficiency of code generated by 13 open-source and 8 closed-source Large Language Models (LLMs). Our results reveal that among the evaluated closed-source models, GPT-4-turbo yields higher code efficiency than Palm-2-chat-bison, Claude-instant-1, Gemini-pro, GPT-4, and GPT-3.5, in terms of both execution time and memory usage. Nevertheless, the efficiency of the code generated by GPT-4-turbo is still worse than the efficiency of human-written canonical solutions. In particular, the average execution time of GPT-4-turbo code is **1.69** times that of the canonical solutions. In the worse scenario, the execution time / memory usage is **45.49 / 142.89** times that of the canonical solution.

We also observed that code generation models exhibit varying efficiencies across different algorithm subsets. For example, in the divide and conquer subset, GPT-3.5-turbo-0301 requires only **1.30x** the execution time and **1.53x** the total memory usage compared to the canonical solution. However, for the DFS and BFS subsets, it needs 3.56x and 3.38x more execution time, and 501.64x and 412.37x more total

memory usage, respectively, compared with the canonical solution.

To conclude, this paper makes the following contributions:

- We introduce EffiBench, a code generation benchmark designed for assessing the efficiency of code for code generation models. As far as we know, EffiBench is the first benchmark for assessing the capability of code generation models in generating efficient code.
- We conduct an empirical study to compare 13 open-source LLMs and 8 closed-source LLMs in generating efficient code. Our evaluation results illustrate that as the SOTA model in code efficiency, GPT-4-turbo still requires **3.18x** total memory usage and **1.69x** execution time compared to canonical solution.

2. Related Work

2.1. Large Language Model for Code

Large language models (LLMs) are widely used for code generation tasks. For example, CodeBERT (Feng et al., 2020), PLBART (Ahmad et al., 2021), and CodeGPT (Zan et al., 2022) explore the BERT, BART, and GPT architectures. For example, GraphCodeBERT (Guo et al., 2020) incorporates graph-based representations, while CodeT5+ (Wang et al., 2023) combines the encoder-decoder paradigm with the structural essence of code. These enhancements aim to give the models a more fine-grained understanding of code relationships and dependencies beyond just syntactic patterns. A current trend is the construction of large-scale models, e.g., Codex (Chen et al., 2021a), CodeGen (Nijkamp et al., 2023), PanGu Coder (Christopoulou et al., 2022), WizardCoder (Luo et al., 2023), CodeLlama (Rozière et al., 2023), with billions of parameters, which have illustrated the performance of SOTA in code generation tasks. Another way is to directly use foundation models (e.g., GPT-3.5-turbo, GPT-4 (OpenAI, 2023), Claude, PaLM Coder (Chowdhery et al., 2022)) to generate code functions, which have been evaluated for their effectiveness in generating functional code.

2.2. Benchmark for Code Generation Models

Quantifying the effectiveness of recent state-of-the-art (SOTA) code generation models necessitates specialized benchmarks. To this end, several benchmarks (Chen et al., 2021a; Wang et al., 2022; Lai et al., 2022; Yu et al., 2023; Cassano et al., 2022; Hao et al., 2022; Austin et al., 2021; Liu et al., 2023) have been introduced. For instance, Chen et al. (2021a) proposed HumanEval, comprising 164 varied code generation tasks, to evaluate effectiveness by analyzing the pass@k metric for generated code. Building on this, Zheng et al. (2023) introduced HumanEval-X, a multilingual program synthesis benchmark featuring 820 high-quality, human-crafted data samples in languages like Python, C++, Java, JavaScript, and Go, suitable for diverse tasks. Liu et al. (2023) developed HumanEval-Plus and MBPP-Plus, augmenting the HumanEval and MBPP datasets with additional test cases for each problem. Recognizing a gap in the evaluation of data science applications, Lai et al. (2022) proposed DS-1000, a benchmark focused on data science problems and spanning seven Python libraries, including Numpy and Pandas. Recently, Wang et al. (2022) presented ReCode, a robustness evaluation framework for code, designed to provide a thorough assessment of code generation models' robustness, particularly in perturbation scenarios.

Noting the predominance of low-difficulty tasks in current code generation datasets, Hendrycks et al. (2021) introduced APPS, a collection of manually curated problems from open-access programming websites like Codewars, AtCoder, Kattis, and Codeforces. In a similar vein, Li et al. (2023b) developed TACO, sourced from a range of platforms including Aizu AtCoder, CodeChef, Codeforces, CodeWars, GeeksforGeeks, HackerEarth, HackerRank, Katti, and LeetCode. While these benchmarks have significantly contributed to evaluating the correctness of code generation, they have largely overlooked the efficiency of the generated code. Addressing this gap, our work introduces EffiBench, a benchmark comprising 1,000 efficiency-critical problems, specifically designed to evaluate the efficiency of code produced by code generation models.

3. Benchmark Construction

3.1. Efficiency-critical Problem Collection

Coding Problem Collection Inspired by the common practice (Bell, 2023; Harper, 2022; Behroozi et al., 2019) of using LeetCode problems to evaluate human developers' abilities in writing efficient algorithms, we collect the coding problems that appear on LeetCode. Specifically, we collect all problems tagged with "LeetCode" on the HuggingFace platform. We remove duplicate problems with identical problem ID (each project has a unique ID in LeetCode). We also remove problems whose interview frequencies are lower than 40% at LeetCode. At the end we obtain 2,605 problems as initial problem candidates.

Efficiency-Critical Problem Filtering This step selects efficiency-critical problems from the initial 2,605 problem candidates. The problems collected from HuggingFace are not tagged with algorithm topics, therefore, we map each problem in LeetCode and label the problem with the "Topic" tag provided by LeetCode. We then choose typical algorithms (Table 1) that are introduced in common algorithm text books (Shyamasundar, 1996), which are also the most widely covered in Leetcode. This yields 1,146 problems all

together.

3.2. Canonical Solution Construction

For each coding problem, EffiBench provides an executable canonical solution so as to serve as a baseline to calculate the normalised efficiency. Drawing inspiration from DS-1000 (Lai et al., 2022), which collects canonical solutions based on the most starred responses on Stack Overflow, we begin with collecting the top-starred solutions for each problem from the LeetCode Discussion Forum. For each collected solution, we need to guarantee that they are executable in non-Leetcode environment. To this end, we manually fix the solutions that need to import extra classes such as TreeNode and ListNode as well as extra packages such as List and Bisect. We also remove the solutions that require specialized packages implemented only by LeetCode. In the end, we manage to map executable canonical solutions for 1,000 coding problems, which will be regarded as our final coding efficiency dataset.

3.3. Test Case Generation

It is essential to have adequate and diverse test cases to evaluate a program's efficiency across various scenarios.Since directly generate test cases with LLMs (e.g., GPT-3.5) have low accuracy (See Appendix), we develop a test case generator for each coding problem as an integral part of our benchmark construction. In particular, we require GPT-3.5-turbo to produce the test case generator, which is prompted to generate massive test cases with different input sizes, data distribution, and edge cases (We provide a test case generator example in Appendix Figure 4). Users can decide how many tests they would like to generate for each problem. We also provide 100 tests within EffiBench for users to use directly, which also serve as the tests in our evaluation in this paper (Results with 10 tests and 1,000 tests are shown in Appendix Table 12).

3.4. Efficiency Metrics

Efficiency metrics are crucial for benchmarking code generation models automatically. Following LeetCode, we design automatic efficiency metrics from two aspects: execution time and memory usage. Specifically, we use the following metrics: Execution Time (ET), Normalized Execution Time (NET), Max Memory Usage (MU), Normalized Max Memory Usage (NMU), Total Memory Usage (TMU), and Normalized Total Memory Usage (NTMU) to measure the overall capability of a code generation model in generating efficient code. Following HumanEval and MBPP pass@1 calculation, we will only calculate the efficiency metrics with code completion that can pass all test cases.

Execution Time (ET) Execution time (ET) measures the average time taken for code execution. Mathematically, ET is defined as:

$$ET = \frac{1}{N}\sum^{N} T_{\text{code}}$$

where ET is the execution time metric, T_{code} is the execution time of the code (with all the test cases), and N is the number of codes generated by code generation models used for evaluation.

Normalized Execution Time (NET) Normalized Execution Time (NET) measures the execution time required by generated code relative to that of a canonical solution. We define NET as:

$$NET = \frac{1}{N}\sum^{N} \frac{T_{\text{code}}}{T_{\text{canonical}}}$$

where T_{code} is the execution time of the generated code, and $T_{\text{canonical}}$ is the execution time of the canonical solution. A NET value greater than 1 indicates that the generated code is slower than the canonical solution, while a value less than 1 suggests the generated code is faster.

Max Memory Usage (MU) Max Memory Usage (MU) measures the average max memory consumption during code execution. Mathematically, MU is defined as:

$$MU = \frac{1}{N}\sum^{N} M_{\text{code}}$$

where MU is the memory usage metric, M_{code} is the max memory consumption of the generated code among all the test cases, and N is the number of code instances generated by code generation models used for evaluation. This metric is critical for assessing the resource efficiency of generated code, particularly in environments with limited maximum memory capacity.

Normalized Max Memory Usage (NMU) Normalized Max Memory Usage (NMU) quantifies how the max memory efficiency of the generated code compares to the canonical solution. We define NMU as:

$$NMU = \frac{1}{N}\sum^{N} \frac{M_{\text{code}}}{M_{\text{canonical}}}$$

where NMU is the normalized max memory usage metric, M_{code} is the max memory usage of the generated code, and $M_{\text{canonical}}$ is the max memory usage of the canonical solution. An NMU value less than 1 indicates that the generated code is more memory-efficient than the canonical solution, whereas a value greater than 1 suggests it is less efficient in terms of memory usage. This metric provides a relative measure of the memory optimization in the generated code in comparison to a standard baseline.

Table 1. Statistics of EffiBench with different algorithms.

Algorithm	Greedy	DP	Backtracking	Divide and Conquer	DFS	BFS	Binary Search	Two Pointers	Sliding Window	Bit Manipulation	Sorting	Total/Avg.
Number of problems	243	277	48	21	108	86	148	105	70	102	238	1000
Number of Easy problems	32	8	1	4	18	8	23	39	9	26	63	171
Number of Medium problems	170	151	37	8	72	52	75	59	47	58	133	589
Number of Hard problems	41	118	10	9	18	26	50	7	14	18	42	240
Avg. length of problem description	224.8	216.4	162.0	205.1	218.9	239.7	216.4	198.6	188.7	195.0	220.7	212.0
Avg. lines of Canonical Solution	12.6	15.1	19.3	18.2	20.8	22.7	14.4	13.0	14.6	12.8	12.0	14.6

Total Memory Usage (TMU) Total Memory Usage (TMU) assesses the efficiency of memory usage throughout the execution of code, taking into account both the magnitude and duration of memory utilization. To calculate TMU, first, monitor and record the memory usage at discrete time intervals during the execution, resulting in a memory usage profile $M(t)$, where t represents time. Then, compute the area under the curve of $M(t)$ over the total execution time, T_{total}, using numerical integration methods such as the trapezoidal rule:

$$TMU = \frac{1}{N}\sum^{N}\int_{0}^{T_{\text{total}}} M(t)\,dt$$

A lower TMU value indicates higher memory efficiency, reflecting an optimized balance between the amount of memory used and the duration of its usage.

Normalized Total Memory Usage (NTMU) The Normalized Total Memory Usage (NTMU) offers a comparison of the dynamic memory efficiency between the generated code and the canonical solution. To determine NTMU, calculate the TMU for both the generated code and the canonical solution. Normalize the TMU of the generated code by dividing it by the TMU of the canonical solution:

$$NTMU = \frac{1}{N}\sum^{N}\frac{TMU_{\text{code}}}{TMU_{\text{canonical}}}$$

where TMU_{code} is the TMU of the generated code and $TMU_{\text{canonical}}$ is the TMU of the canonical solution. An NTMU value less than 1 signifies that the generated code manages dynamic memory more efficiently compared to the canonical solution, while a value greater than 1 indicates a less efficient management of dynamic memory. This metric provides insight into the relative use of dynamic memory of generated code compared to an established benchmark.

4. Benchmark Statistics

We provide the detailed statistics of the dataset in Table 1. The coding problems in EffiBench have three difficulty levels (171 easy-level, 589 medium-level, and 240 hard-level problems), where the difficulty of each problem is defined by LeetCode (Lee).

The table lists the number of problems for each algorithm. Specifically, EffiBench contains 243 problems for greedy algorithm, 277 for dynamic programming (DP), 48 for backtracking, 21 for divide and conquer, 108 for depth-first search (DFS), 86 for breadth-first search (BFS), 148 for binary search, 105 for two pointers, 70 for sliding window, 102 for bit manipulation and 238 for sorting algorithm. The sum of problems in different algorithms can be larger than the number of total problems because one problem in our dataset may belong to two algorithm classes. On average, the problem description in EffiBench has 212.0 words. The canonical solutions, which represent the baseline code against which the generated code is compared, have 14.6 lines on average.

We provide the comparison of EffiBench and other code generation datasets in Table 2. Specifically, we compare EffiBench with the five most widely used code-related datasets (i.e., HumanEval, MBPP, APPS, DSP, and DS-1000). Different from the previous dataset that focuses on analyzing whether the code passes all test cases, EffiBench also analyzes the efficiency during the code execution procedure. Although EffiBench is primarily designed to assess the efficiency of generated code, it can also serve to evaluate code correctness, akin to other code generation datasets.

5. Evaluation

Models: We study both open- and closed-source large language models (LLMs) in code generation. For open-source models, we evaluate EffiBench with Incoder-1B/6B (Fried et al., 2022), StartCoder (Li et al., 2023a), CodeGen-2B/6B-mono (Nijkamp et al., 2023), Magicoder-S-CL-7B, Magicoder-S-DS-6.7B (Wei et al., 2023), WizardCoder-15B-V1.0 (Luo et al., 2023), InstructCodeT5p-16b (Wang et al., 2023), Mistral-7B-v0.1, Mistral-7B-Instruct-v0.2 (Jiang et al., 2023), CodeLlama-7b-Python-hf, CodeLlama-13b-Python-hf (Rozière et al., 2023) since these open-source models have obtained SOTA pass@1 in the HumanEval and MBPP datasets. For closed-source models, we evaluated EffiBench with GPT-3.5, GPT-4 (OpenAI, 2023), PaLM-2-chat-bison (Chowdhery et al., 2022), claude-instant-1, and gemini-pro (Google, 2023) since we observe that these models obtain high pass@1 in code generation datasets (e.g., HumanEval (Chen et al., 2021a), MBPP (Austin et al., 2021)). For GPT-3.5 models, we experiment with GPT-3.5-turbo-0301, GPT-3.5-turbo-0613, and GPT-3.5-turbo-1106 which represent three different versions of the GPT-3.5. For GPT-4 models, we experi-

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

Prompt

Problem: Given a string s, find the length of the longest substring without repeating characters.

Example 1:
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.

Example 2:
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Note: The answer must be a substring, "pwke" is a subsequence and not a substring.

```
solution = Solution()
assert
↪  solution.lengthOfLongestSubstring('abcabcbb')
↪  == 3
assert
↪  solution.lengthOfLongestSubstring('pwwkew')
↪  == 3
```

Generated Code

```
class Solution:
    def lengthOfLongestSubstring(self, s: str)
    ↪  -> int:
        ss = set()
        i = ans = 0
        for j, c in enumerate(s):
            while c in ss:
                ss.remove(s[i])
                i += 1
            ss.add(c)
            ans = max(ans, j - i + 1)
        return ans
```

Prompt

Problem: Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

Example 1:
Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: Merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.

```
solution = Solution()
assert solution.findMedianSortedArrays([1, 2],
↪  [3, 4]) == 2.50000
```

Generated Code

```
class Solution:
    def findMedianSortedArrays(self, nums1:
    ↪  List[int], nums2: List[int]) -> float:
        def f(i: int, j: int, k: int) -> int:
            if i >= m:
                return nums2[j + k - 1]
            if j >= n:
                return nums1[i + k - 1]
            if k == 1:
                return min(nums1[i], nums2[j])
            p = k // 2
            x = nums1[i + p - 1] if i + p - 1 <
            ↪  m else inf
            y = nums2[j + p - 1] if j + p - 1 <
            ↪  n else inf
            return f(i + p, j, k - p) if x < y
            ↪  else f(i, j + p, k - p)

        m, n = len(nums1), len(nums2)
        a = f(0, 0, (m + n + 1) // 2)
        b = f(0, 0, (m + n + 2) // 2)
        return (a + b) / 2
```

Figure 2. Example problems synthesized (few-shot) by GPT-4-0613. The prompt is shown in purple, and the model response is shown in blue. The prompt also typically contains several few-shot examples in the same format, which are not shown here.

ment with GPT-4-turbo and GPT-4 (GPT-4-0613). For each LLM, we first collect the code that is correctly generated for each coding problem, then execute these correct code and calculate the efficiency metrics we introduce in Section 3.4.

Prompt: As shown in Figure 2, our prompt design follows the MBPP code generation prompt, where the prompt first provides the task description and then provides a few examples with input and output pairs. Each example has an explanation of the rationality of the output. The prompt also has the assertion part which intends to constrain the function signature with a specific format of input and output.

5.1. End2End Results

Table 3 illustrates the efficiency of code generation models with EffiBench. All experiments are executed on a single server running Ubuntu 20.04, equipped with an Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz, and with 32GB RAM and Python 3.8[1].

[1]We report the average results as well as the standard deviation of five executions for all evaluation results Appendix Table 20.

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

Model	**ET (s)**	**NET**	max NET	NET>5	**MU (Mb)**	**NMU**	max NMU	NMU>5	**TMU (Mb*s)**	**NTMU**	max NTMU	NTMU>5	Pass@1
Open-source models													
incoder-1B	2.41	1.71	7.12	11.1	1363.87	1.71	7.21	11.1	3419.35	7.82	62.03	11.1	0.9
incoder-6B	2.65	12.27	57.88	29.4	656.32	6.65	71.11	17.6	1681.10	52.50	456.16	35.3	1.7
starcoder	1.28	5.68	63.39	12.4	434.87	1.79	85.57	4.1	1006.86	30.72	3574.68	14.7	17.0
codegen-2B-mono	1.58	5.90	57.32	13.6	681.58	3.06	151.60	6.8	1675.81	88.64	6772.04	15.9	8.8
codegen-6B-mono	1.56	7.84	57.77	15.9	525.32	1.21	6.65	2.4	1226.53	12.30	90.57	15.9	8.2
Magicoder-S-CL-7B	1.65	7.75	57.71	18.2	610.73	2.30	50.40	9.1	1431.75	50.67	2056.16	18.2	7.7
Magicoder-S-DS-6.7B	1.04	5.77	57.83	12.4	265.73	1.51	67.31	2.9	590.40	22.25	2908.75	13.6	24.2
WizardCoder-15B-V1.0	0.53	6.29	16.78	33.3	23.11	1.00	1.01	0.0	11.60	8.94	24.78	33.3	0.3
instructcodet5p-16b	0.10	1.12	1.12	0.0	25.06	1.09	1.09	0.0	1.48	1.11	1.11	0.0	0.1
Mistral-7B-Instruct-v0.2	3.14	19.67	57.98	41.0	968.02	9.21	453.87	8.2	2383.72	254.81	12252.50	42.6	6.1
Mistral-7B-v0.1	0.82	7.17	63.36	14.2	103.37	1.33	10.36	3.7	197.53	16.90	507.68	14.9	13.4
CodeLlama-7b-Python-hf	1.54	7.37	57.30	16.7	542.48	1.35	7.27	5.6	1234.69	13.05	90.73	18.1	7.2
CodeLlama-13b-Python-hf	0.91	3.99	57.52	9.6	288.90	2.20	156.45	3.9	645.02	45.98	6548.75	11.2	17.8
Closed-source models													
gpt-3.5-turbo-0301	0.27	2.62	57.97	4.6	52.57	1.92	299.62	1.8	50.97	30.61	11479.88	6.8	43.9
gpt-3.5-turbo-0613	0.30	2.77	57.07	6.2	37.15	1.25	8.04	2.0	19.17	5.26	187.09	7.9	45.5
gpt-3.5-turbo-1106	0.32	2.91	63.34	6.2	41.41	1.38	68.70	2.1	25.41	11.61	3359.71	8.1	53.1
gpt-4-turbo	0.20	1.69	45.49	3.5	37.06	1.24	12.68	1.8	14.36	3.18	142.89	5.5	60.2
gpt-4	0.24	2.27	58.08	4.8	59.09	2.19	278.63	2.4	87.54	58.21	14818.60	6.8	50.1
palm-2-chat-bison	0.34	3.25	57.74	6.2	91.90	3.35	399.96	3.1	179.21	89.71	12555.34	8.2	38.8
claude-instant-1	0.38	3.80	57.79	8.3	47.61	1.52	23.64	3.6	29.66	13.33	875.31	10.7	25.3
gemini-pro	0.43	4.27	57.85	8.2	71.95	2.72	335.74	3.1	131.65	88.71	17268.88	10.3	31.9

Table 4. Efficiency results of closed-source LLMs with a common set of problems correctly addressed by all models.

Model	**ET (s)**	**NET**	max NET	NET>5	**MU (Mb)**	**NMU**	max NMU	NMU>5	**TMU (Mb*s)**	**NTMU**	max NTMU	NTMU>5	Problem Ratio
gpt-3.5-turbo-0301	0.28	2.85	57.90	2.7	36.38	1.28	8.04	0.0	12.55	5.55	135.43	6.2	11.3
gpt-3.5-turbo-0613	0.23	2.23	56.27	1.8	36.18	1.27	8.04	0.0	11.89	4.71	187.09	5.3	11.3
gpt-3.5-turbo-1106	0.20	1.86	56.09	0.9	36.18	1.27	8.04	0.0	9.21	3.12	84.20	5.3	11.3
gpt-4-turbo	0.14	1.34	6.73	0.0	36.19	1.27	8.04	0.0	7.94	2.34	63.49	3.5	11.3
gpt-4	0.19	1.92	56.19	0.9	93.36	3.74	278.63	0.9	184.33	133.61	14818.60	5.3	11.3
palm-2-chat-bison	0.19	1.84	56.16	0.9	36.22	1.27	8.04	0.0	9.01	3.16	84.39	5.3	11.3
claude-instant-1	0.32	3.26	57.79	3.5	40.72	1.47	23.64	0.9	22.89	13.20	875.31	7.1	11.3
gemini-pro	0.23	2.26	56.21	1.8	38.02	1.33	9.43	0.0	17.81	8.02	555.16	5.3	11.3

As shown in Table 3, we can observe that first, most open-source code generation models (e.g., Incoder, and StarCoder) have lower pass@1, e.g., most of their pass@1 are lower than 10%, which indicates that open-source models may need to consider the code generation correctness rather than efficiency since their generated code can not address the problem description tasks. We can also observe that the generated code is less efficient for the open-source models compared with our `canonical_solution` for most of the models, so we will then focus on the closed-source models in other discussions for ease of discussion.

Then, for the closed-source models, we can observe that first, the code efficiency results of code generation models are different for different model versions and model architectures. We can also observe that the most efficient code generation model GPT-4-turbo obtains SOTA in all metrics but is still far from perfect efficiency compared to our *canonical_solution*. For example, it still has **1.69x** NET compared to our *canonical_solution* results, and it obtains competitive max memory usage (**1.24x**) compared to *canonical_solution*. The total memory usage during the code execution procedure has **3.18x** compared with *canonical_solution*.

We also examined all the coding problems to check whether there are cases where LLMs yield more efficient code than the canonical solutions. The evaluation results are shown in Appendix Table 9 to Table 11, where we can observe that few code generated by LLMs are more efficient than the canonical solutions. The minimum NET is only 0.93 with GPT-4-turbo, indicating that for cases where the code from LLMs is more efficient, the superiority is very minor.

We suspect that the overall inefficiency the code produced by of LLMs, when compared to canonical solutions, may be attributed to the distribution of the training data. Typically, these datasets prioritize the correctness of code and collect code from repositories like GitHub where code is often correct but not necessarily optimized for efficiency. Focusing primarily on correctness without adequate attention to efficiency could result in neglecting efficiency in the code generated by LLMs.

As to other closed-source LLMs, GPT-3.5 models have worse efficiency than GPT-4 models. For example, GPT-3.5-turbo-0301 has an NTMU of 30.61, its code also requires more max memory usage (1.92x) and execution time (2.62x). We can also observe that palm-2-chat-bison and Gemini-pro require more NTMU than other models, e.g., they require 89.71x and 88.71x total memory usage compared to the canonical solutions.

Consistency of different metrics: When we compare the benchmarking results from different efficiency metrics, we observe that the rankings of different LLMs from the basic metrics (highlighted in bold in the head row) maintain a general consistency. This consistency across metrics reinforces their credibility in assessing a model's capability in

Table 5. Efficiency results for different algorithm subset with GPT-3.5-turbo-0301 .

Model	ET (s)	NET	max NET	NET>5	MU (Mb)	NMU	max NMU	NMU>5	TMU (Mb*s)	NTMU	max NTMU	NTMU>5	Pass@1
greedy	0.43	4.14	57.97	7.4	34.43	1.19	6.43	1.1	15.61	6.95	135.43	8.4	35.7
dynamic_programming	0.29	2.81	56.48	4.9	27.86	1.10	4.99	0.0	9.34	4.29	133.84	6.5	35.7
backtracking	0.27	2.12	10.21	5.6	92.21	1.98	12.68	5.6	69.58	10.66	148.16	16.7	25.4
divide_and_conquer	0.11	1.30	2.03	0.0	28.83	1.21	2.41	0.0	2.11	1.53	4.15	0.0	45.5
dfs	0.34	3.56	51.09	4.3	349.18	14.38	299.62	4.3	728.65	501.64	11479.88	13.0	18.4
bfs	0.38	3.38	51.09	7.1	290.27	12.02	299.62	3.6	599.88	412.37	11479.88	14.3	28.9
binary_search	0.16	1.46	6.95	4.5	54.36	1.46	8.04	6.1	19.52	3.87	65.84	6.1	40.5
two_pointers	0.22	2.16	48.77	5.3	43.34	1.26	8.04	3.5	18.31	4.35	76.69	5.3	48.3
sliding_window	0.14	1.35	6.95	3.0	44.25	1.30	8.04	3.0	16.96	3.22	65.84	6.1	42.3
bit_manipulation	0.25	2.60	57.30	4.4	48.51	1.33	12.68	2.2	30.35	6.41	148.16	4.4	33.6
sorting	0.17	1.80	57.97	3.4	39.35	1.28	8.04	2.5	12.53	3.38	86.37	4.2	45.9

generating efficient code.

Correctness: Although EffiBench is designed to focus on benchmarking code efficiency, it can also be adopted to benchmark code correctness, as shown by Pass@1 in the last column of Table 3. For open-sourced LLMs, we can observe that they have low pass@1: many of their pass@1 are lower than 10%. This indicates that open-source models still need to put a lot of effort into improving code generation correctness. For closed-sourced LLMs, gpt-4-turbo has the highest Pass@1 of 60.2%.

5.2. Results with Identical Coding Problems

The efficiency metrics are calculated based on all the correctly generated code in Table 3. However, different LLMs may have different correctness regarding the same coding problem. As a result, the results for different LLMs in Table 3 are based on different coding problems. This part investigates such threat by analysing the efficiency results with identical coding problems. In other words, we focus on analyzing problems that are correctly addressed by all LLMs. Since we observe no tasks addressed by all open- and closed-source models, and even no tasks addressed by all open-source models, in this section we focus on closed-source models. Table 4 shows the results with 113 problems that are correctly addressed by all the 8 closed-sourced LLMs. We observe that GPT-4-turbo is still the best model in code efficiency, similar to the distribution observed in Table 3. For instance, in the filtered set of problems, GPT-4-turbo requires only **2.34x** the total memory and **1.34x** the execution time compared to the canonical solution. It is also observed that GPT-4 demands significantly more total memory usage (**133.61x**) compared to other models. This increased requirement could be attributed to one or a few problems where GPT-4 requires more total memory usage (e.g., requires **14818.60** MB*s for the max NTMU code).

5.3. Results for Different Algorithms

As shown in Table 1, EffiBench is constructed with 11 different algorithms. This part explores whether the code generation model have different code efficiency across different algorithm subset. Table 5 reports the results of GPT-3.5-turbo-0301 (results of other LLMs are in Appendix Table 13 to Table 19) We can observe that GPT-3.5-turbo-0301 has different code efficiency for different algorithm subsets. For example, GPT-3.5-turbo-0301 is less efficient for the DFS and BFS subsets, which requires 501.64x and 412.37x total memory usage during the code execution procedure. In contrast, GPT-3.5-turbo-0301 demonstrated exceptional efficiency in the divide_and_conquer subset, which only requires 1.53x NTMU compared with canonical solution. We suspect that the observed differences come from the availability of training data. Specifically, models tend to perform better on tasks for which their training corpus contains abundant and varied examples with efficient solutions.

6. Conclusion and Future work

In this paper, we introduce EffiBench, a benchmark designed to evaluate the efficiency of Python code generated by various code generation models. EffiBench encompasses 1,000 problems and consists of 11 distinct algorithmic subsets. Unlike previous benchmarks that primarily emphasize the correctness of code generation, EffiBench extends the evaluation criteria to include both execution time analysis and memory usage analysis. This dual-focus approach allows for a more nuanced quantification of efficiency, addressing a critical aspect of code generation that has been less explored in the past. By incorporating these metrics, EffiBench aims to inspire the research community's focus towards not only the correctness but also the efficiency of code generated by code generation models. We believe that EffiBench will serve as a valuable tool in guiding future developments in building scalable and sustainable software, encouraging the creation of code generation models that not only target generating correct code, but also optimized for efficiency and resource utilization. In the future, we will consider extending EffiBench with other different programming languages such as C++, Java, JS, and Go, and more resources, e.g., APPS, CodeParrot, and CodeContest. Besides, we will also provide the evaluation server (we will provide docker and AWS server in the future) to allow researchers to evaluate their methods with the same hardware and software.

Limitations First, similar to recently constructed datasets, e.g., TACO (Li et al., 2023b), EffiBench may have some data samples that might have been exposed to the pretraining corpus of LLMs. However, after checking the GPT4 report (OpenAI, 2023), we believe that our benchmark tasks are not used to train LLMs as currently, LLMs such as GPT4 still use it as the test set to quantify LLMs code generation effectiveness (See (OpenAI, 2023) Tables 1, 9 and 10).

References

Leetcode. https://leetcode.com/. Accessed: January 31, 2024.

Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. Unified pre-training for program understanding and generation. *ArXiv*, abs/2103.06333, 2021. URL https://api.semanticscholar.org/CorpusID:232185260.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C. J., Terry, M., Le, Q. V., and Sutton, C. Program synthesis with large language models. *ArXiv*, abs/2108.07732, 2021. URL https://api.semanticscholar.org/CorpusID:237142385.

Behroozi, M., Parnin, C., and Barik, T. Hiring is broken: What do developers say about technical interviews? In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–9. IEEE, 2019.

Bell, B. A. *Understanding the Preparation Phase of Technical Interviews*. PhD thesis, Virginia Tech, 2023.

Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S. D., Phipps-Costin, L., Pinckney, D., Yee, M.-H., Zi, Y., Anderson, C. J., Feldman, M. Q., Guha, A., Greenberg, M., and Jangda, A. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. 2022. URL https://api.semanticscholar.org/CorpusID:254854172.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021a.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D. W., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Babuschkin, I., Balaji, S. A., Jain, S., Carr, A., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M. M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021b. URL https://api.semanticscholar.org/CorpusID:235755472.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N. M., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B. C., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., García, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Díaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K. S., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311, 2022. URL https://api.semanticscholar.org/CorpusID:247951931.

Christopoulou, F., Lampouras, G., Gritta, M., Zhang, G., Guo, Y., yi Li, Z., Zhang, Q., Xiao, M., Shen, B., Li, L., Yu, H., yu Yan, L., Zhou, P., Wang, X., Ma, Y., Iacobacci, I., Wang, Y., Liang, G., Wei, J., Jiang, X., Wang, Q., and Liu, Q. Pangu-coder: Program synthesis with function-level language modeling. *ArXiv*, abs/2207.11280, 2022. URL https://api.semanticscholar.org/CorpusID:251040785.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

Fried, D., Aghajanyan, A., Lin, J., Wang, S. I., Wallace, E., Shi, F., Zhong, R., tau Yih, W., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. *ArXiv*, abs/2204.05999, 2022. URL https://api.semanticscholar.org/CorpusID:248157108.

Google, G. T. Gemini: A family of highly capable multimodal models. *ArXiv*, abs/2312.11805, 2023. URL https://api.semanticscholar.org/CorpusID:266361876.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Yin, J., Jiang, D., and Zhou, M. Graphcodebert: Pre-training code representations with data flow. *ArXiv*, abs/2009.08366, 2020.

Hao, Y., Li, G., Liu, Y., Miao, X., Zong, H., Jiang, S., Liu, Y., and Wei, H. Aixbench: A code generation benchmark dataset. *ArXiv*, abs/2206.13179,

2022. URL https://api.semanticscholar.org/CorpusID:250072468.

Harper, J. Interview insight: How to get the job. In *A Software Engineer's Guide to Seniority: A Guide to Technical Leadership*, pp. 19–28. Springer, 2022.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de Las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mistral 7b. *ArXiv*, abs/2310.06825, 2023. URL https://api.semanticscholar.org/CorpusID:263830494.

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, S., Fried, D., yi Wang, S., and Yu, T. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501, 2022. URL https://api.semanticscholar.org/CorpusID:253734939.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M.-H., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C. J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C. M., Hughes, S. M., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder: may the source be with you! *ArXiv*, abs/2305.06161, 2023a. URL https://api.semanticscholar.org/CorpusID:258588247.

Li, R., Fu, J., Zhang, B.-W., Huang, T., Sun, Z., Lyu, C., Liu, G., Jin, Z., and Li, G. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*, 2023b.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. Wizardcoder: Empowering code large language models with evol-instruct. *ArXiv*, abs/2306.08568, 2023. URL https://api.semanticscholar.org/CorpusID:259164815.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*, 2023.

OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023. URL https://api.semanticscholar.org/CorpusID:257532815.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M. P., Ferrer, C. C., Grattafiori, A., Xiong, W., D'efossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code. *ArXiv*, abs/2308.12950, 2023. URL https://api.semanticscholar.org/CorpusID:261100919.

Shyamasundar, R. K. Introduction to algorithms. *Resonance*, 1:14–24, 1996. URL https://api.semanticscholar.org/CorpusID:123556377.

Wang, S., Li, Z., Qian, H., Yang, C., Wang, Z., Shang, M., Kumar, V., Tan, S., Ray, B., Bhatia, P., Nallapati, R., Ramanathan, M. K., Roth, D., and Xiang, B. Recode: Robustness evaluation of code generation models. *ArXiv*, abs/2212.10264, 2022. URL https://api.semanticscholar.org/CorpusID:254877229.

Wang, Y., Le, H., Gotmare, A. D., Bui, N. D., Li, J., and Hoi, S. C. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

Wei, Y., Wang, Z., Liu, J., Ding, Y., and Zhang, L. Magicoder: Source code is all you need. *ArXiv*, abs/2312.02120, 2023. URL https://api.semanticscholar.org/CorpusID:265609970.

Yu, H., Shen, B., Ran, D., Zhang, J., Zhang, Q., Ma, Y., Liang, G., Li, Y., Xie, T., and Wang, Q. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. *ArXiv*, abs/2302.00288, 2023. URL https://api.semanticscholar.org/CorpusID:256459413.

Zan, D., Chen, B., Yang, D., Lin, Z., Kim, M., Guan, B., Wang, Y., Chen, W., and Lou, J.-G. CERT: Continual pre-training on sketches for library-oriented code generation. In *The 2022 International Joint Conference on Artificial Intelligence*, 2022.

Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Wang, Z., Shen, L., Wang, A., Li, Y., Su, T., Yang, Z., and Tang, J. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *ArXiv*, abs/2303.17568, 2023. URL https://api.semanticscholar.org/CorpusID:257834177.

Model	ET	NET	NET*	MU	NMU	NMU*	TMU	NTMU	NTMU*	pass@1
gpt-3.5-turbo-0301	0.27	2.62	2.65	52.57	1.92	2.2	50.97	30.61	29.23	43.9
gpt-3.5-turbo-0613	0.30	2.77	3.19	37.15	1.25	1.5	19.17	5.26	12.06	45.5
gpt-3.5-turbo-1106	0.32	2.91	3.10	41.41	1.38	1.7	25.41	11.61	14.01	53.1
gpt-4-1106-preview	0.20	1.69	1.95	37.06	1.24	1.5	14.36	3.18	8.33	60.2
gpt-4	0.24	2.27	2.35	59.09	2.19	2.4	87.54	58.21	50.24	50.1
palm-2-chat-bison	0.34	3.25	3.71	91.90	3.35	3.8	179.21	89.71	119.66	38.8
claude-instant-1	0.38	3.80	4.10	47.61	1.52	1.9	29.66	13.33	18.70	25.3
gemini-pro	0.43	4.27	4.62	71.95	2.72	3.0	131.65	88.71	86.30	31.9

Table 6. Evaluation results of different LLMs efficiency results for EffiBench. We use "*" to represent the results with new calculation type.

Model	accuracy
GPT-3.5-turbo-0301	5.9
GPT-3.5-turbo-0613	8.2
GPT-4-turbo	14.3
GPT-4	13.7

Table 7. Evaluation results of test case accuracy for canonical solution. For each test case generated by LLMs, we will analyze whether the test case is accurate for the canonical solution. Then, we will calculate the accuracy based on the total correct test cases/total generated test cases.

A. Appendix

we provide the evaluation results for two types of metrics calculation results, i.e., original metrics calculation and calculation with the equation of Total_model/Total_baseline (columns with *) in Table 6, where we can observe that with two different metric calculation strategies, the overall results only have minor changes, and GPT-4-turbo still have SOTA performance for efficiency metrics.

In this section, we provide an empirical study to discuss the accuracy of LLMs generated test cases in canonical solutions. As shown in Table 7, we can observe that the accuracy of test cases generated by four LLMs is lower than 20%.

We provide a deep analysis result of EffiBench in the Appendix. Specifically, we first provide the evaluation result index for each dataset and metrics in Table 8. Specifically, for GPT-3.5-turbo-0301 model, we first provide the algorithm subset evaluation results in Table 5, we also illustrate the distribution results for each metric, where we divided each metric results into ten columns based on the metrics value range.

A.1. Deep analysis for GPT-3.5-turbo-0301

As shown in Figure 5, we illustrate the distribution of the number of codes generated by GPT-3.5-turbo-0301 for different metrics. Here we can find that in Figure 5 (b), (d), and (f), some of the code generated by GPT-3.5-turbo-0301 requires 10^0

Table 8. Evaluation result index for closed-source models.

Model	Subset Evaluation	metrics distribution
GPT-3.5-turbo-0301	Table 5	Figure 5
GPT-3.5-turbo-0613	Table 13	Figure 6
GPT-3.5-turbo-1106	Table 14	Figure 7
GPT-4-turbo (GPT-4-1106-preview)	Table 15	Figure 8
GPT-4	Table 16	Figure 9
palm-2-chat-bison	Table 17	Figure 10
claude-instant-1	Table 18	Figure 11
gemini-pro	Table 19	Figure 12

Table 9. Distribution of NET. Column "<0.95" shows the ratio of problems with NET smaller than 0.95. For best-performing gpt-4-turbo, it yields slightly more efficient code than canonical solutions for only 0.33% of the problems .

Model	Min NET	<0.95 (%)	<0.97(%)	<0.99(%)	<1(%)	>1(%)	>10(%)	>50(%)	>100(%)	Max NET
gpt-3.5-turbo-0301	0.94	0.23	0.23	0.23	0.91	98.41	4.56	2.73	0.00	57.97
gpt-3.5-turbo-0613	0.95	0.22	0.66	0.66	1.32	97.58	6.15	3.74	0.00	57.07
gpt-3.5-turbo-1106	0.94	0.19	0.75	0.94	2.82	96.61	6.21	3.39	0.00	63.34
gpt-4-turbo	0.93	0.33	1.33	1.83	3.32	94.19	3.49	1.50	0.00	45.49
gpt-4	0.96	0.00	0.40	0.80	1.80	97.41	4.79	1.80	0.00	58.08
palm-2-chat-bison	0.95	0.26	0.52	0.77	2.32	97.42	6.19	4.38	0.00	57.74
claude-instant-1	0.96	0.00	0.79	1.19	1.98	97.63	8.30	4.74	0.00	57.79
gemini-pro	0.95	0.00	0.63	0.94	0.94	99.06	8.15	6.27	0.00	57.85

Table 10. Distribution of NMU. Column "<0.95" shows the ratio of problems with NMU smaller than 0.95.

Model	Min NMU	<0.95	<0.97	<0.99	<1	>1	>10	>50	>100	Max NMU
gpt-3.5-turbo-0301	0.98	0.00	0.00	0.23	9.57	90.21	1.82	0.46	0.23	299.62
gpt-3.5-turbo-0613	0.84	0.22	0.22	1.10	8.35	90.11	1.98	0.00	0.00	8.04
gpt-3.5-turbo-1106	0.84	0.19	0.19	0.75	8.47	90.58	2.07	0.38	0.00	68.70
gpt-4-turbo	0.98	0.00	0.00	0.50	8.97	87.87	1.83	0.17	0.00	12.68
gpt-4	0.97	0.00	0.00	0.80	6.99	92.02	2.40	0.60	0.40	278.63
palm-2-chat-bison	0.98	0.00	0.00	0.77	12.11	87.63	3.09	1.55	0.77	399.96
claude-instant-1	0.99	0.00	0.00	0.40	10.67	88.93	3.56	1.19	0.00	23.64
gemini-pro	0.99	0.00	0.00	1.57	9.09	89.97	3.13	1.25	0.31	335.74

to 10^2 execution time and 10^0 to 10^3 memory usage. For example, one code generated by GPT-3.5-turbo-0301 requires **11422.48x** total memory usage compared with canonical solution. However, other codes require about **58.40x** total memory usage compared with canonical solution. We can also observe that the code generated by GPT-3.5-turbo-0301 requires about **1.23x** to **57.68x** execution time compared with canonical solution. Where most of the codes (384 out of 439) will require about **1.23x** execution time compared with canonical solution.

A.2. Deep analysis for execution time

As illustrated in Figure 5, the efficiency distribution for Normalized Execution Time (NET) approximates **1.23x** for GPT-3.5-turbo-0301. This observation raises a compelling question: Are some codes generated by code generation models more efficient compared to the canonical solution? To explore this, we delve into the evaluation results presented in Table 9, where we report the number of codes generated by closed-source models that are faster than the canonical solution. It is noticeable that only a limited number of codes outperform the canonical solution in terms of execution time. Then, we also illustrate the max memory usage and total memory usage distribution in Table 10 and Table 11, where we can also observe that there is no code generated by the code generation model requires less total memory usage compared with canonical solution.

Table 11. Distribution of NTMU. Column "<0.95" shows the ratio of problems with NTMU smaller than 0.95.

Model	Min NTMU	<0.95	<0.97	<0.99	<1	>1	>10	>50	>100	Max NTMU
gpt-3.5-turbo-0301	1.00	0.00	0.00	0.00	0.00	100.00	6.83	5.01	0.91	11479.88
gpt-3.5-turbo-0613	1.00	0.00	0.00	0.00	0.00	98.90	7.91	6.59	0.66	187.09
gpt-3.5-turbo-1106	1.00	0.00	0.00	0.00	0.00	99.44	8.10	6.97	0.94	3359.71
gpt-4-turbo	1.00	0.00	0.00	0.00	0.00	97.51	5.48	3.99	0.33	142.89
gpt-4	1.00	0.00	0.00	0.00	0.00	99.40	6.79	5.39	0.60	14818.60
palm-2-chat-bison	1.00	0.00	0.00	0.00	0.00	99.74	8.25	6.96	2.06	12555.34
claude-instant-1	1.00	0.00	0.00	0.00	0.00	99.60	10.67	8.70	1.58	875.31
gemini-pro	1.02	0.00	0.00	0.00	0.00	100.00	10.34	8.78	2.19	17268.88

A.3. Case Illustration for GPT-3.5-turbo-0301

To illustrate why some code requires more execution time and memory usage during the code execution procedure, we provide a case illustration in Figure 3. We can observe that the code completed by GPT-3.5-turbo-0301 is less efficient in terms of memory usage compared to our *canonical_solution*. Specifically, GPT-3.5-turbo-0301's code employs a standard BFS with a list-based queue, alongside a set for tracking visited states and deadends. The space complexity for this solution includes O(N) for deadends and visited states, and potentially $O(10^4)$ for the queue, as it may store all possible lock combinations in the worst-case scenario. The breadth of the search linearly expands with the number of steps, as each step introduces multiple neighbors into the queue. Conversely, *canonical_solution* adopts a more sophisticated approach with a two-way BFS, utilizing two dictionaries for tracking the search from both ends and two deques for managing the queues. The space complexity remains O(N) for deadends, similar to Solution 1, but each dictionary and deque can grow up to $O(10^4)$ in the worst-case scenario. However, the two-way BFS approach significantly condenses the search breadth by converging from both ends, reducing the overall memory consumption.

A.4. Efficiency of Code with different number of tests

Our experiments in Table 3 only consider 100 tests for each problem, which inspires us to consider how different numbers of tests affect the efficiency of code generated by code generation models. To answer this question, we investigate how does different number of tests affects the efficiency score for each metric. The evaluation results are shown in Table 12, where we can observe that once we increase the tests from 10 to 1000, the efficiency score for NET, NMU, and NTMU will increase for each code generation model. For example, the GPT-3.5-turbo-0301's NTMU increases from 18.1 to 244.8. We indicate that the key reason is once we increase the number of tests, more edge cases will be covered (e.g., more length, data distribution). However, we can also observe that, even though the efficiency score has changed for each model with a different number of tests, the relative efficiency ranking among the models remains consistent. This consistency indicates that while the absolute efficiency scores are sensitive to the number of test cases, the relative performance of different models in handling efficiency remains largely unchanged.

Table 12. Evaluation result of and closed-source models in EffiBench with the different number of tests. "_10" means the evaluation results are obtained with 10 tests.

Model (_number of tests)	**ET (s)**	**NET**	max NET	NET>5	**MU (Mb)**	**NMU**	max NMU	NMU>5	**TMU (Mb*s)**	**NTMU**	max NTMU	NTMU>5	Pass@1
gpt-3.5-turbo-0301_10	0.22	2.24	57.78	2.2	38.15	1.58	265.73	0.2	37.97	26.03	10646.54	2.2	45.7
gpt-3.5-turbo-0301_100	0.27	2.62	57.97	4.6	52.57	1.92	299.62	1.8	50.97	30.61	11479.88	6.8	43.9
gpt-3.5-turbo-0301_1000	0.54	4.50	57.86	13.6	167.97	4.05	288.16	9.7	300.30	72.17	10455.70	20.0	43.4
gpt-3.5-turbo-0613_10	0.24	2.34	57.03	2.7	24.26	1.00	1.50	0.0	5.08	2.93	86.04	2.9	47.8
gpt-3.5-turbo-0613_100	0.30	2.77	57.07	6.2	37.15	1.25	8.04	2.0	19.17	5.26	187.09	7.9	45.5
gpt-3.5-turbo-0613_1000	0.57	4.89	57.78	13.3	137.50	3.30	54.41	8.8	244.81	51.18	2309.21	20.7	44.4
gpt-3.5-turbo-1106_10	0.24	2.29	63.27	2.6	27.29	1.12	68.69	0.2	13.11	8.90	3310.99	2.7	54.8
gpt-3.5-turbo-1106_100	0.32	2.91	63.34	6.2	41.41	1.38	68.70	2.1	25.41	11.61	3359.71	8.1	53.1
gpt-3.5-turbo-1106_1000	0.58	4.83	63.22	12.8	144.54	3.37	68.71	8.2	261.43	55.35	3320.50	20.7	52.2
gpt-4-turbo_10	0.14	1.31	45.49	1.0	24.48	1.01	7.65	0.2	3.26	1.39	63.06	1.1	62.2
gpt-4-turbo_100	0.20	1.69	45.49	3.5	37.06	1.24	12.68	1.8	14.36	3.18	142.89	5.5	60.2
gpt-4-turbo_1000	0.45	3.67	55.99	10.6	129.84	3.12	54.42	8.0	219.86	41.54	2328.68	17.8	59.7
gpt-4_10	0.18	1.88	58.02	1.7	45.12	1.90	278.61	0.6	73.68	53.74	14861.00	2.1	52.6
gpt-4_100	0.24	2.27	58.08	4.8	59.09	2.19	278.63	2.4	87.54	58.21	14818.60	6.8	50.1
gpt-4_1000	0.52	4.30	58.13	12.8	170.02	4.31	278.87	9.5	335.65	100.77	14853.95	21.1	49.3
palm-2-chat-bison_10	0.27	2.71	57.69	3.6	75.34	2.93	390.52	1.4	155.69	80.99	12457.68	3.6	42.0
palm-2-chat-bison_100	0.34	3.25	57.74	6.2	91.90	3.35	399.96	3.1	179.21	89.71	12555.34	8.2	38.8
palm-2-chat-bison_1000	0.62	5.25	57.63	14.5	212.33	5.57	389.66	10.3	449.81	134.84	12466.20	21.9	37.9
claude-instant-1_10	0.29	3.08	57.89	4.3	26.65	1.09	23.63	0.4	10.10	6.97	869.83	4.3	27.6
claude-instant-1_100	0.38	3.80	57.79	8.3	47.61	1.52	23.64	3.6	29.66	13.33	875.31	10.7	25.3
claude-instant-1_1000	0.78	6.38	57.97	19.6	239.43	4.67	54.41	13.6	487.63	87.65	2315.37	25.2	25.0
gemini-pro_10	0.33	3.38	57.62	5.0	55.73	2.37	335.67	1.2	111.47	80.07	17220.89	5.0	34.2
gemini-pro_100	0.43	4.27	57.85	8.2	71.95	2.72	335.74	3.1	131.65	88.71	17268.88	10.3	31.9
gemini-pro_1000	0.71	6.19	57.87	16.9	200.18	4.97	336.30	9.9	428.00	138.72	16989.13	22.7	31.3

A.5. Discussion on Time and Space Complexity

In our experiment, we aim to quantify the efficiency of code generated by code generation models using metrics such as Execution Time (ET), Normalized Execution Time (NET), Maximum Memory Usage (MU), Normalized Maximum Memory Usage (NMU), Total Memory Usage (TMU), and Normalized Total Memory Usage (NTMU). While time and space complexity are conventional metrics in software development for assessing code efficiency, we opted not to rely

Problem

Problem: You have a lock in front of you with 4 circular wheels. Each wheel has 10 slots: ’0’, ’1’, ’2’, ’3’, ’4’, ’5’, ’6’, ’7’, ’8’, ’9’. The wheels can rotate freely and wrap around: for example we can turn ’9’ to be ’0’, or ’0’ to be ’9’. Each move consists of turning one wheel one slot.
The lock initially starts at ’0000’, a string representing the state of the 4 wheels.
You are given a list of deadends dead ends, meaning if the lock displays any of these codes, the wheels of the lock will stop turning and you will be unable to open it.
Given a target representing the value of the wheels that will unlock the lock, return the minimum total number of turns required to open the lock, or -1 if it is impossible.

Example 1:
Input: deadends = [”0201”,”0101”,”0102”,”1212”,”2002”], target = ”0202”
Output: 6
Explanation: A sequence of valid moves would be ”0000” − > ”1000” − > ”1100” − > ”1200” − > ”1201” − > ”1202” − > ”0202”. Note that a sequence like ”0000” − > ”0001” − > ”0002” − > ”0102” − > ”0202” would be invalid, because the wheels of the lock become stuck after the display becomes the dead end ”0102”.

Example 2:
Input: deadends = [”8888”], target = ”0009”
Output: 1
Explanation: We can turn the last wheel in reverse to move from ”0000” − > ”0009”.

Example 3:
Input: deadends = [”8887”,”8889”,”8878”,”8898”,”8788”,”8988”,”7888”,”9888”], target = ”8888”
Output: -1
Explanation: We cannot reach the target without getting stuck.

```
solution = Solution()
assert solution.openLock(["0201","0101","0102","1212","2002"], "0202") == 6
assert solution.openLock(["8888"], "0009") == 1
assert solution.openLock(["8887","8889","8878","8898","8788","8988","7888","9888"], "8888") == -1
```

GPT-3.5-Turbo-0301

```
class Solution:
    def openLock(self, deadends: List[str], target: str)
    ↪  -> int:
        deadends = set(deadends)#For faster search
        head = '0000'
        if head in deadends:
            return -1

        def _gen_neighbors(num):
            for i in range(4):
                x = int(num[i])
                for d in [-1, 1]:
                    y = (x + d) % 10
                    yield num[:i] + str(y) + num[i+1:]

        stack = [head]
        visited = set(stack)
        steps = 0
        while len(stack) > 0:
            size = len(stack)
            for i in range(size):
                node = stack.pop(0)
                if node == target:
                    return steps
                for neighbor in _gen_neighbors(node):
                    if neighbor in deadends:
                        continue
                    if neighbor in visited:
                        continue
                    visited.add(neighbor)
                    stack.append(neighbor)
            steps += 1
        return -1
```

Canonica Solution

```
def mergeArrays(array1, array2):
    # Merging two arrays efficiently
    result = []
    i, j = 0, 0
    while i < len(array1) and j <
    ↪  len(array2):
        if array1[i] < array2[j]:
            result.append(array1[i])
            i += 1
        else:
            result.append(array2[j])
            j += 1
    # Append remaining elements of
    ↪  array1
    while i < len(array1):
        result.append(array1[i])
        i += 1
    # Append remaining elements of
    ↪  array2
    while j < len(array2):
        result.append(array2[j])
        j += 1
    return result
```


Figure 3. A case illustration of GPT-3.5-turbo-0301 and *canonica_solution*. The left code is completed by GPT-3.5-turbo-0301, which requires 50.1 MB*seconds, while the right result is our *canonica_solution*, which requires 7.5 MB*seconds.

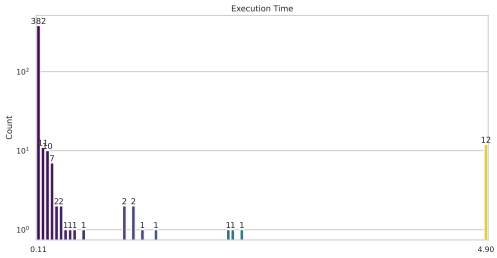
Test Case Generation

```python
import random

class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        ss = set()
        i = ans = 0
        for j, c in enumerate(s):
            while c in ss:
                ss.remove(s[i])
                i += 1
            ss.add(c)
            ans = max(ans, j - i + 1)
        return ans

def generate_test_case():
    solution = Solution()

    # Generate a random string
    s = ''.join(random.choices('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789',
    ↪  k=random.randint(0, 10)))

    # Calculate the expected result using the provided Solution class
    expected_result = solution.lengthOfLongestSubstring(s)

    return (s, ), expected_result

def test_generated_test_cases(num_tests):
    test_case_generator_results = []
    for i in range(num_tests):
        inputs, expected_result = generate_test_case()
        solution = Solution()
        assert solution.lengthOfLongestSubstring(*inputs) == expected_result

        test_case_generator_results.append(f"assert solution.lengthOfLongestSubstring({',
        ↪  '.join(map(repr, inputs))}) == {expected_result}")
    return test_case_generator_results

if __name__ == '__main__':
    num_tests = 100
    test_case_generator_results = test_generated_test_cases(num_tests)

    with open("./full_tmp/0.txt", "w") as f:
        f.write("\n".join(test_case_generator_results))
    print(len(test_case_generator_results))
```

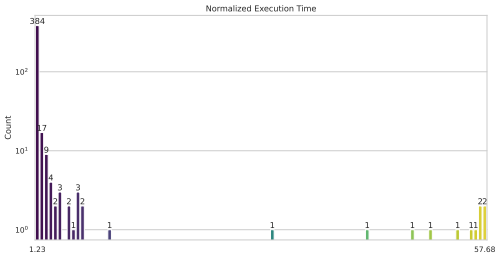
Figure 4. A case illustration of the test case generation process for the LeetCode task. The test case generator (function generate_test_case) will generate 100 tests for the solution.

Table 13. Evaluation result of GPT-3.5-turbo-0613 for different algorithm subsets.

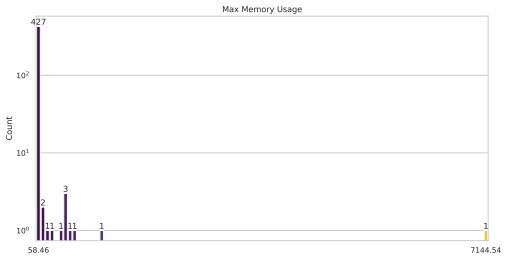
Model	**ET (s)**	**NET**	max NET	NET>5	**MU (Mb)**	**NMU**	max NMU	NMU>5	**TMU (Mb*s)**	**NTMU**	max NTMU	NTMU>5	Pass@1
greedy	0.40	4.08	57.07	7.0	36.87	1.19	7.14	2.0	37.42	7.75	161.90	8.0	37.6
dynamic_programming	0.32	2.56	56.38	6.2	26.94	1.11	4.99	0.0	10.15	3.74	83.77	7.0	37.1
backtracking	0.20	2.13	11.17	10.5	29.94	1.23	2.97	0.0	4.87	3.16	20.82	15.8	26.8
divide_and_conquer	0.13	1.38	2.38	0.0	29.73	1.25	2.41	0.0	2.60	1.79	4.67	0.0	54.5
dfs	0.13	1.34	3.69	0.0	39.51	1.41	4.73	0.0	5.40	2.53	16.10	9.7	24.8
bfs	0.53	4.88	46.76	13.9	37.76	1.41	4.73	0.0	22.39	12.22	187.09	19.4	37.1
binary_search	0.18	1.51	6.81	6.3	62.41	1.53	8.04	7.9	25.15	4.53	64.45	7.9	38.7
two_pointers	0.14	1.32	6.81	3.8	45.42	1.29	8.04	3.8	17.00	3.09	64.45	3.8	44.1
sliding_window	0.13	1.30	6.81	2.8	41.65	1.24	8.04	2.8	15.28	2.90	64.45	2.8	46.2
bit_manipulation	0.49	4.84	54.31	8.2	24.26	1.05	1.72	0.0	11.09	6.88	84.33	8.2	36.6
sorting	0.33	3.05	57.07	8.0	51.54	1.41	8.04	4.5	43.86	7.35	161.90	9.8	43.6



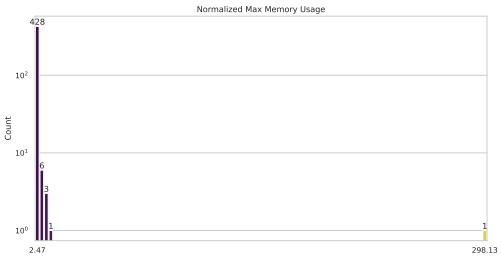
(a) Execution time distribution for GPT-3.5 Turbo.



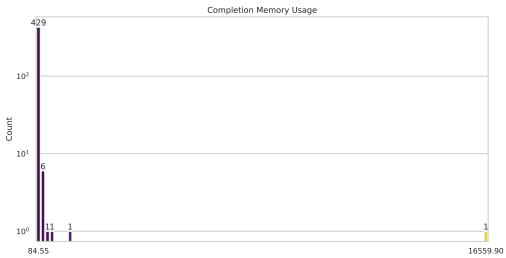
(b) Normalized execution time distribution.



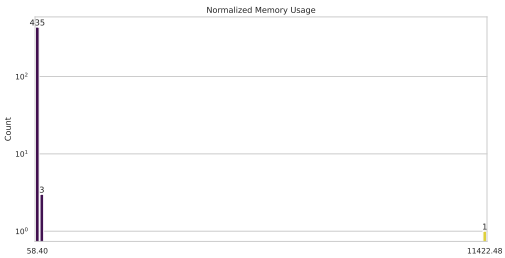
(c) Maximum memory usage distribution.



(d) Normalized maximum memory usage distribution.



(e) Completion memory usage distribution.

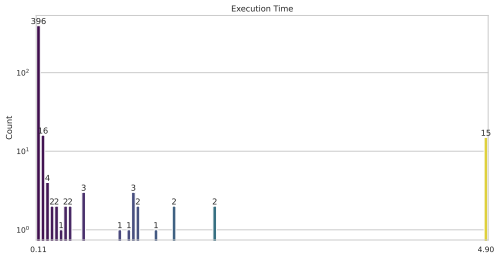


(f) Normalized memory usage distribution.

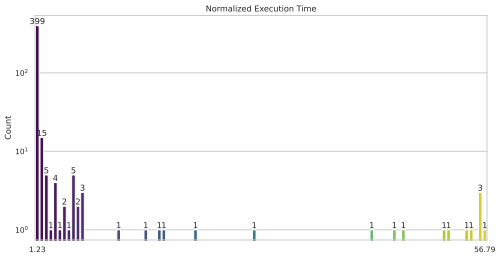
Figure 5. Various distributions of computational resources used by GPT-3.5 Turbo 0301 version. We divided the metric value range into ten columns based on the minimum and maximum values for each metric.

Table 14. Evaluation result of GPT-3.5-turbo-1106 for different algorithm subsets.

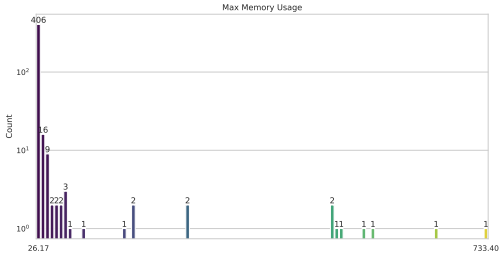
Model	**ET (s)**	**NET**	max NET	NET>5	**MU (Mb)**	**NMU**	max NMU	NMU>5	**TMU (Mb*s)**	**NTMU**	max NTMU	NTMU>5	Pass@1
greedy	0.42	4.08	56.78	6.9	37.51	1.21	7.13	1.5	19.49	6.77	103.13	7.7	48.9
dynamic_programming	0.45	3.84	56.46	7.6	38.96	1.56	68.70	0.6	42.57	27.09	3359.71	10.2	45.5
backtracking	0.34	2.53	9.95	21.7	101.37	1.86	12.68	8.7	80.61	10.51	143.58	26.1	32.4
divide_and_conquer	0.12	1.37	3.01	0.0	24.83	1.07	1.30	0.0	2.14	1.55	4.24	0.0	40.9
dfs	0.15	1.57	8.24	3.1	39.25	1.40	4.73	0.0	5.85	2.89	16.02	12.5	25.6
bfs	0.23	1.73	9.27	5.6	37.75	1.38	4.73	0.0	7.12	2.91	16.02	13.9	37.1
binary_search	0.17	1.56	8.24	6.5	56.41	1.48	8.04	6.5	21.09	4.16	63.85	7.8	47.2
two_pointers	0.13	1.25	6.78	3.2	41.57	1.25	8.04	3.2	14.43	2.74	63.85	3.2	53.4
sliding_window	0.39	4.22	63.34	8.1	42.23	1.28	8.04	2.7	25.95	11.64	248.75	8.1	47.4
bit_manipulation	0.44	3.89	54.27	9.3	56.17	1.36	12.68	3.7	40.57	8.23	143.58	11.1	40.3
sorting	0.14	1.31	6.78	3.1	43.74	1.32	8.04	3.1	16.83	2.85	63.85	3.8	50.6



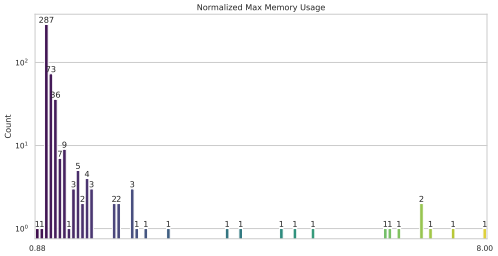
(a) Execution time distribution for GPT-3.5 Turbo.



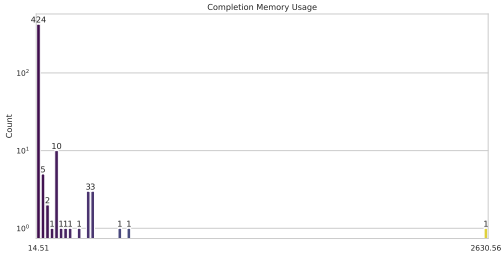
(b) Normalized execution time distribution.



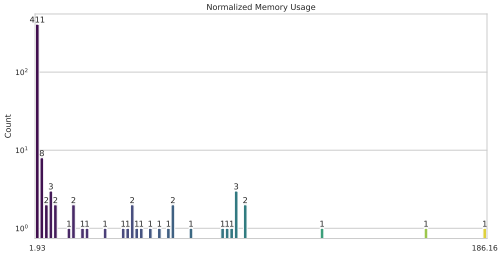
(c) Maximum memory usage distribution.



(d) Normalized maximum memory usage distribution.



(e) Completion memory usage distribution.



(f) Normalized memory usage distribution.

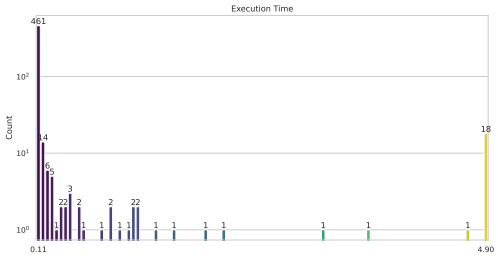
Figure 6. Various distributions of computational resources used by GPT-3.5 Turbo 0613 version.

Table 15. Evaluation result of GPT-4-turbo for different algorithm subsets.

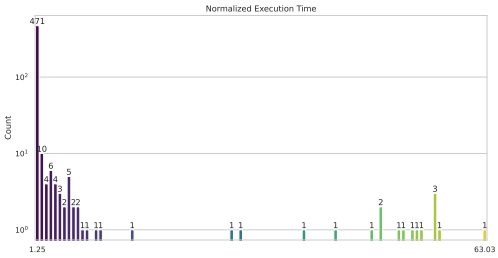
Model	**ET (s)**	**NET**	max NET	NET>5	**MU (Mb)**	**NMU**	max NMU	NMU>5	**TMU (Mb*s)**	**NTMU**	max NTMU	NTMU>5	Pass@1
greedy	0.18	1.45	40.20	1.9	36.16	1.22	7.79	1.9	14.30	2.29	60.06	2.6	58.3
dynamic_programming	0.23	1.97	42.45	3.4	28.25	1.11	4.99	0.0	11.82	3.22	139.16	5.7	50.7
backtracking	0.30	2.23	9.88	12.9	82.53	1.69	12.68	6.5	60.56	8.15	142.89	25.8	43.7
divide_and_conquer	0.12	1.35	2.87	0.0	28.64	1.20	2.41	0.0	2.42	1.68	3.94	0.0	63.6
dfs	0.13	1.32	3.80	0.0	37.15	1.35	4.73	0.0	4.83	2.33	16.39	8.1	29.6
bfs	0.36	3.20	45.49	7.9	35.50	1.33	4.73	0.0	12.11	6.32	79.15	13.2	39.2
binary_search	0.16	1.40	6.73	4.8	53.29	1.42	8.04	6.0	19.14	3.68	63.49	6.0	51.5
two_pointers	0.13	1.25	6.73	3.0	40.57	1.24	8.04	3.0	13.39	2.62	63.49	3.0	56.8
sliding_window	0.13	1.30	6.73	2.3	39.14	1.23	8.04	2.3	12.95	2.65	63.49	2.3	55.1
bit_manipulation	0.18	1.50	9.88	3.3	52.90	1.33	12.68	3.3	31.10	4.29	142.89	6.7	44.8
sorting	0.15	1.36	10.55	3.3	41.71	1.30	8.04	2.6	15.26	2.77	63.49	4.6	59.1

Table 16. Evaluation result of GPT-4 for different algorithm subsets.

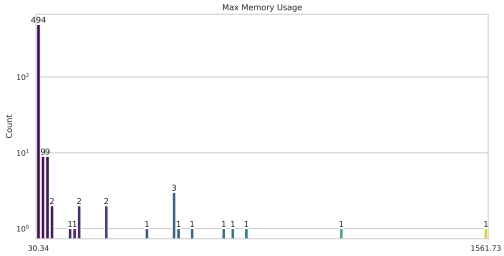
Model	**ET (s)**	**NET**	max NET	NET>5	**MU (Mb)**	**NMU**	max NMU	NMU>5	**TMU (Mb*s)**	**NTMU**	max NTMU	NTMU>5	Pass@1
greedy	0.26	2.23	56.99	3.4	132.35	5.25	278.63	4.3	332.71	235.74	14818.60	4.3	44.0
dynamic_programming	0.32	3.31	58.08	7.1	62.47	2.52	187.85	1.4	131.77	93.76	12497.02	10.0	40.6
backtracking	0.28	2.37	9.99	15.4	69.42	1.66	12.68	3.8	49.48	8.67	144.82	23.1	36.6
divide_and_conquer	0.13	1.40	2.98	0.0	28.32	1.19	2.41	0.0	2.45	1.73	4.13	0.0	40.9
dfs	0.12	1.32	3.84	0.0	34.99	1.33	4.73	0.0	4.12	2.17	16.52	6.5	24.8
bfs	0.26	2.26	27.44	6.2	32.82	1.30	4.73	0.0	8.70	4.66	86.04	9.4	33.0
binary_search	0.20	1.65	8.12	7.1	62.61	1.57	8.04	8.6	27.53	5.02	61.32	10.0	42.9
two_pointers	0.14	1.30	6.55	3.4	42.60	1.25	8.04	3.4	15.05	2.86	61.32	3.4	50.0
sliding_window	0.14	1.34	6.55	2.7	41.61	1.25	8.04	2.7	14.39	2.86	61.32	2.7	47.4
bit_manipulation	0.23	2.36	54.46	3.5	42.64	1.26	12.68	1.8	23.98	5.34	144.82	5.3	42.5
sorting	0.19	1.74	50.71	4.0	41.00	1.28	8.04	2.4	16.15	3.22	79.45	5.6	48.6



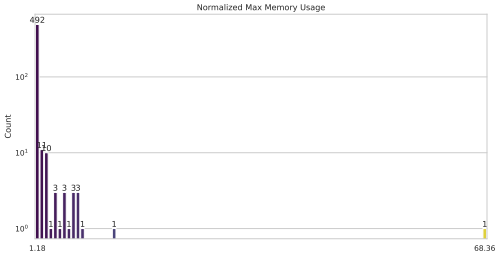
(a) Execution time distribution for GPT-3.5 Turbo.



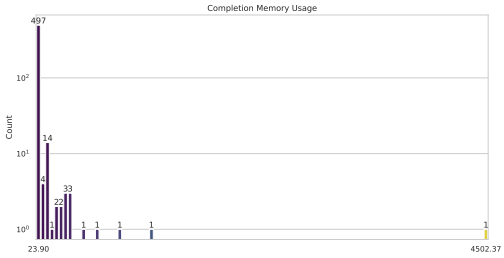
(b) Normalized execution time distribution.



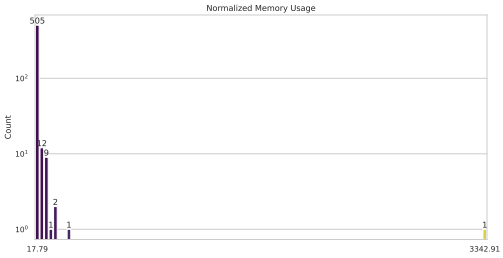
(c) Maximum memory usage distribution.



(d) Normalized maximum memory usage distribution.



(e) Completion memory usage distribution.



(f) Normalized memory usage distribution.

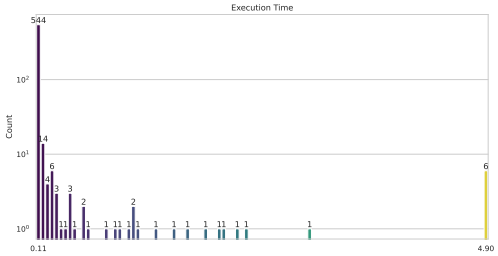
Figure 7. Various distributions of computational resources used by GPT-3.5 Turbo 1106 version.

Table 17. Evaluation result of PaLM-2-chat-bison for different algorithm subsets.

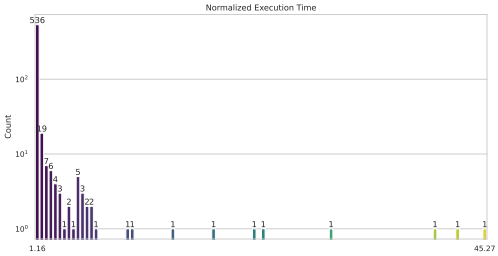
Model	**ET (s)**	**NET**	max NET	NET>5	**MU (Mb)**	**NMU**	max NMU	NMU>5	**TMU (Mb*s)**	**NTMU**	max NTMU	NTMU>5	Pass@1
greedy	0.46	4.92	57.74	7.2	118.14	4.95	187.85	3.6	353.88	262.40	12555.34	8.4	31.2
dynamic_programming	0.58	5.31	57.74	10.7	203.54	7.92	399.96	4.9	544.80	281.16	12555.34	12.6	29.9
backtracking	0.49	3.98	41.23	11.1	278.47	7.23	97.51	11.1	513.44	200.69	3440.77	16.7	25.4
divide_and_conquer	0.17	1.80	5.28	16.7	23.88	1.04	1.12	0.0	3.11	2.19	7.71	16.7	27.3
dfs	0.12	1.26	3.65	0.0	34.28	1.31	4.72	0.0	3.92	2.07	16.12	6.2	25.6
bfs	0.20	1.53	6.34	3.8	33.94	1.32	4.72	0.0	5.40	2.30	16.12	7.7	26.8
binary_search	0.32	2.70	51.92	7.7	57.26	1.55	8.04	7.7	35.86	7.53	121.64	9.6	31.9
two_pointers	0.12	1.20	6.58	2.1	36.89	1.18	8.04	2.1	11.34	2.34	61.98	2.1	40.7
sliding_window	0.15	1.43	6.58	6.9	45.56	1.28	8.04	3.4	18.19	3.41	61.98	6.9	37.2
bit_manipulation	0.67	6.11	54.42	15.2	354.82	12.07	399.96	6.5	778.70	254.40	7824.38	15.2	34.3
sorting	0.18	1.81	51.62	3.1	45.10	1.59	31.37	3.1	28.72	15.68	1283.21	5.1	38.1

Table 18. Evaluation result of Claude-instant-1 for different algorithm subsets.

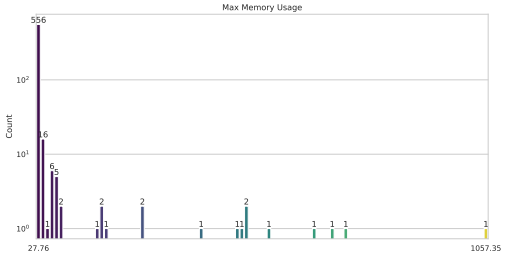
Model	**ET (s)**	**NET**	max NET	NET>5	**MU (Mb)**	**NMU**	max NMU	NMU>5	**TMU (Mb*s)**	**NTMU**	max NTMU	NTMU>5	Pass@1
greedy	0.44	5.03	57.78	6.9	34.26	1.46	21.21	1.7	26.04	19.52	804.21	6.9	21.8
dynamic_programming	0.45	4.38	57.71	9.8	29.50	1.16	4.99	0.0	15.67	6.67	86.92	11.5	17.7
backtracking	0.35	2.55	9.92	16.7	121.21	2.31	12.68	8.3	99.54	14.74	143.06	25.0	16.9
divide_and_conquer	0.09	1.08	1.13	0.0	24.14	1.04	1.13	0.0	1.45	1.08	1.15	0.0	22.7
dfs	0.15	1.55	3.74	0.0	47.92	1.63	4.73	0.0	7.56	3.53	16.30	15.4	10.4
bfs	0.98	8.20	46.74	21.4	41.30	1.50	4.73	0.0	41.81	21.46	184.43	28.6	14.4
binary_search	0.37	3.16	50.46	13.9	91.34	1.89	8.04	13.9	45.78	9.20	78.36	16.7	22.1
two_pointers	0.13	1.29	6.65	2.8	41.83	1.25	8.04	2.8	14.81	2.83	62.42	2.8	30.5
sliding_window	0.16	1.37	6.65	5.0	55.05	1.37	8.04	5.0	25.39	4.15	62.42	5.0	25.6
bit_manipulation	0.45	4.50	57.79	10.0	105.22	2.85	23.64	10.0	115.64	52.21	875.31	10.0	14.9
sorting	0.20	2.09	57.04	4.1	47.42	1.58	21.21	4.1	27.07	13.74	804.21	5.4	28.8



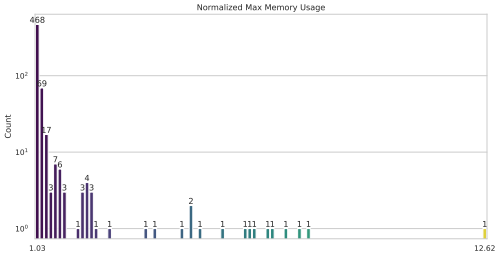
(a) Execution time distribution for GPT-4 Turbo.



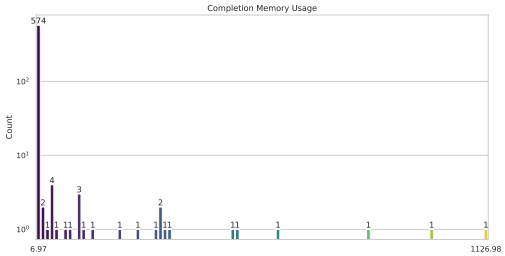
(b) Normalized execution time distribution.



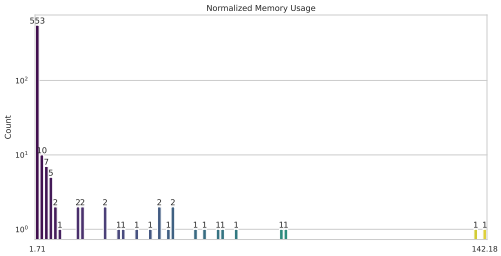
(c) Maximum memory usage distribution.



(d) Normalized maximum memory usage distribution.



(e) Completion memory usage distribution.

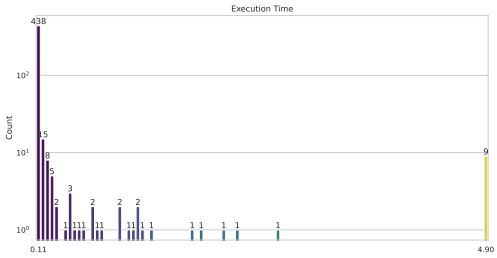


(f) Normalized memory usage distribution.

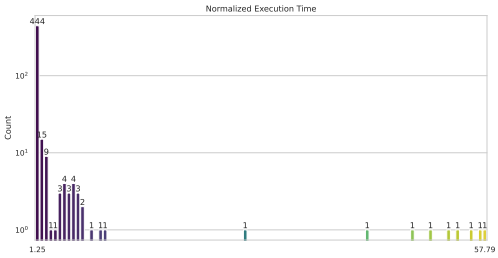
Figure 8. Various distributions of computational resources used by GPT-4 Turbo.

Table 19. Evaluation result of Gemini-pro for different algorithm subsets.

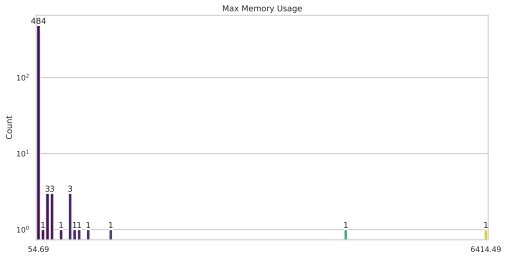
Model	**ET (s)**	**NET**	max NET	NET>5	**MU (Mb)**	**NMU**	max NMU	NMU>5	**TMU (Mb*s)**	**NTMU**	max NTMU	NTMU>5	Pass@1
greedy	0.33	3.61	56.94	5.6	68.38	2.55	94.41	2.8	148.15	104.55	7099.48	7.0	26.7
dynamic_programming	0.51	4.64	57.31	8.0	69.13	2.80	94.41	2.7	166.74	117.26	7099.48	10.7	21.7
backtracking	0.67	6.26	53.26	23.1	106.48	2.03	12.68	7.7	103.43	21.28	146.98	23.1	18.3
divide_and_conquer	0.12	1.28	1.89	0.0	28.90	1.22	1.73	0.0	2.55	1.69	3.51	0.0	18.2
dfs	0.13	1.38	3.80	0.0	39.18	1.42	4.73	0.0	5.44	2.68	16.37	10.0	16.0
bfs	1.10	10.32	53.26	22.7	79.03	3.15	32.77	9.1	149.83	96.39	1333.37	27.3	22.7
binary_search	0.29	2.88	57.85	7.3	58.35	1.50	8.04	7.3	26.37	6.41	86.81	9.8	25.2
two_pointers	0.26	2.69	57.51	7.0	49.50	1.33	8.04	4.7	23.72	5.67	88.99	7.0	36.4
sliding_window	0.64	6.81	56.78	13.8	46.13	1.30	8.04	3.4	29.98	11.74	83.93	13.8	37.2
bit_manipulation	0.58	5.37	57.36	11.8	55.49	1.42	12.68	2.9	47.19	12.18	146.98	11.8	25.4
sorting	0.25	2.52	57.31	5.4	125.58	4.94	335.74	4.3	272.20	189.74	17268.88	7.5	36.2



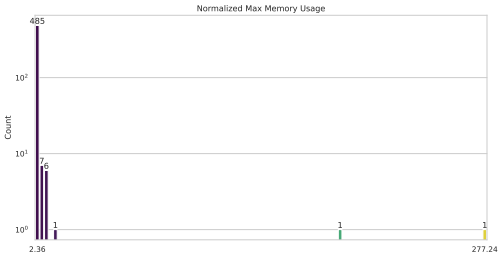
(a) Execution time distribution for GPT-4.



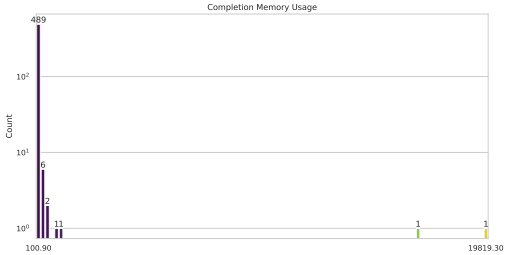
(b) Normalized execution time distribution.



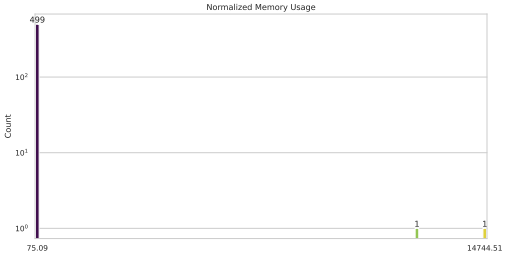
(c) Maximum memory usage distribution.



(d) Normalized maximum memory usage distribution.

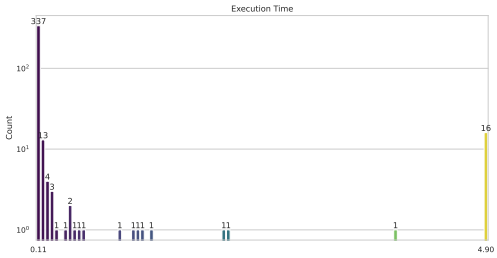


(e) Completion memory usage distribution.

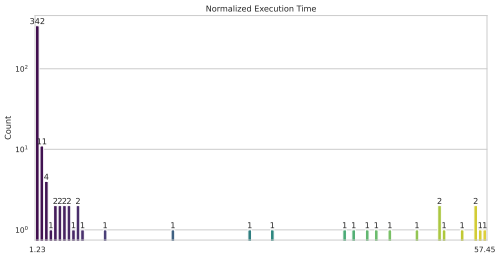


(f) Normalized memory usage distribution.

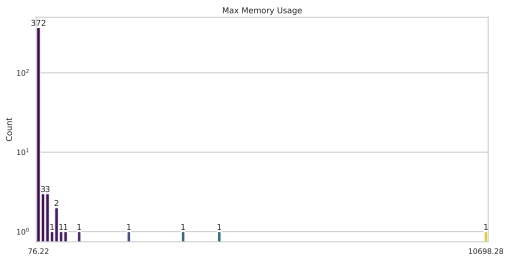
Figure 9. Various distributions of computational resources used by GPT-4.



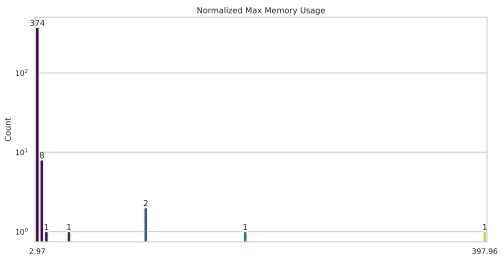
(a) Execution time distribution for palm-2-chat-bison.



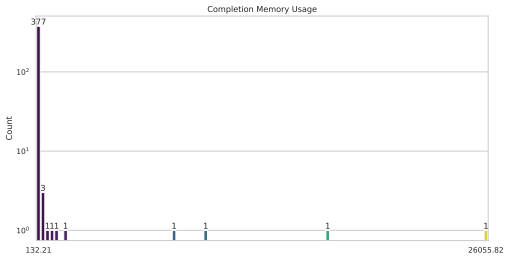
(b) Normalized execution time distribution.



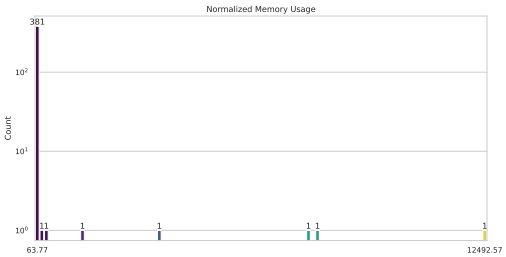
(c) Maximum memory usage distribution.



(d) Normalized maximum memory usage distribution.

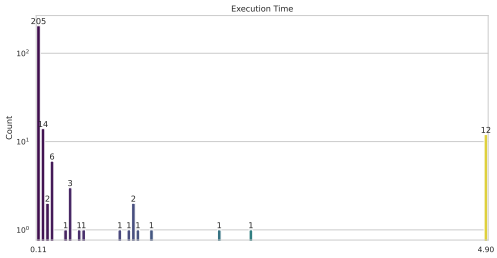


(e) Completion memory usage distribution.

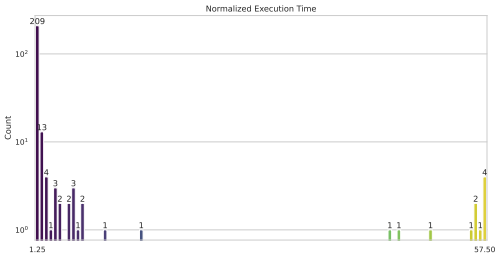


(f) Normalized memory usage distribution.

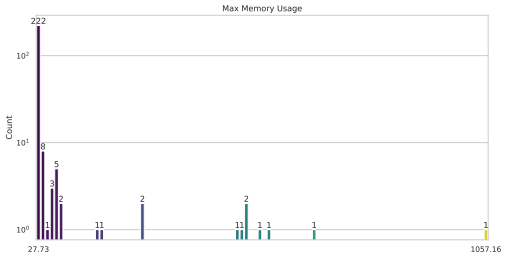
Figure 10. Various distributions of computational resources used by PaLM-2-chat-bison.



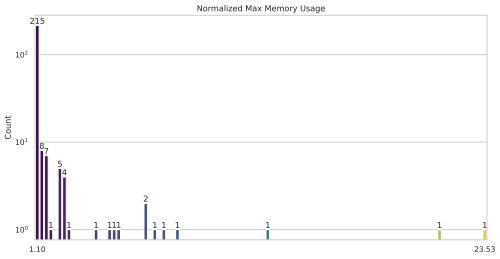
(a) Execution time distribution for claude-instant-1.



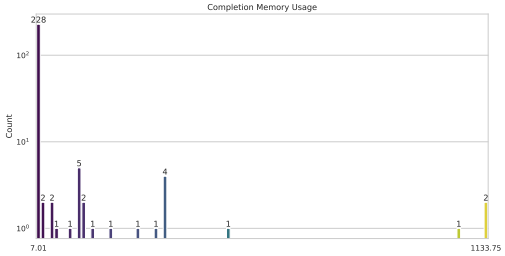
(b) Normalized execution time distribution.



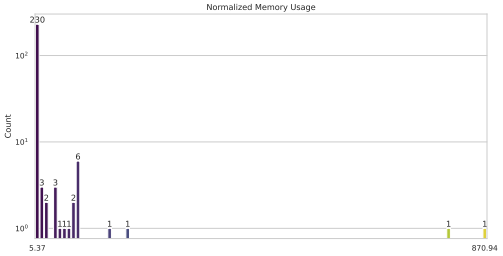
(c) Maximum memory usage distribution.



(d) Normalized maximum memory usage distribution.

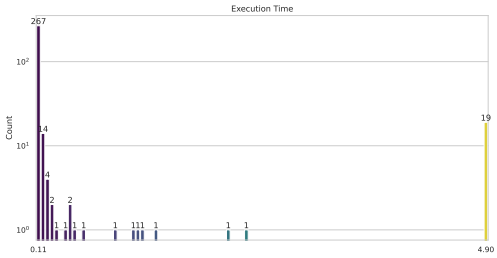


(e) Completion memory usage distribution.

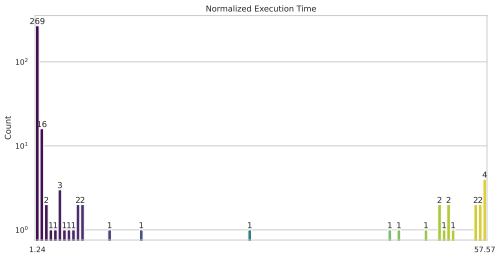


(f) Normalized memory usage distribution.

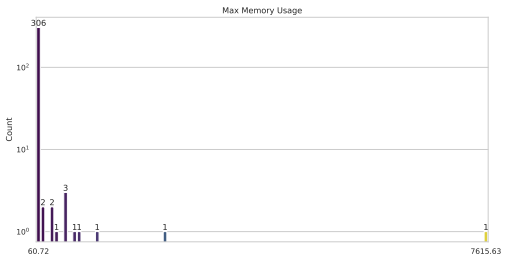
Figure 11. Various distributions of computational resources used by Claude-instant-1.



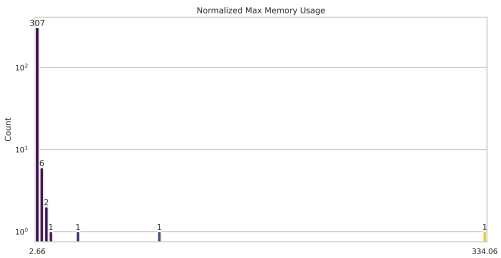
(a) Execution time distribution for gemini-pro.



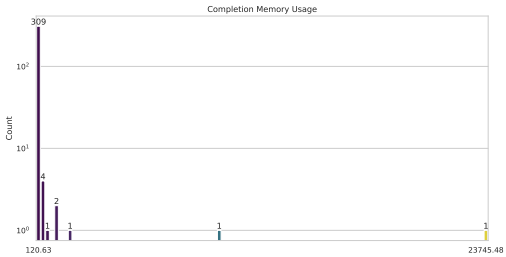
(b) Normalized execution time distribution.



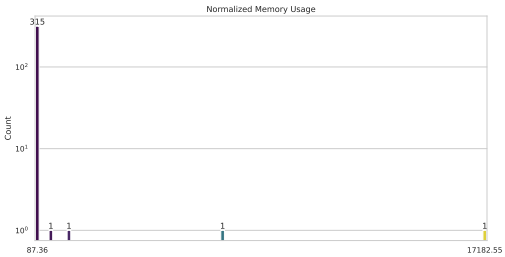
(c) Maximum memory usage distribution.



(d) Normalized maximum memory usage distribution.



(e) Completion memory usage distribution.



(f) Normalized memory usage distribution.

Figure 12. Various distributions of computational resources used by Gemini-pro.

Table 20. Standard deviation for five times execution results. Values in the () mean the standard deviation results.

Model	ET (s)	NET	MU (Mb)	NMU	TMU (Mb*s)	NTMU	Pass@1
gpt-3.5-turbo-0301	0.27 (0.0)	2.62 (0.01)	52.57 (0.28)	1.92 (0.01)	50.97 (0.49)	30.61 (0.35)	43.9
gpt-3.5-turbo-0613	0.30 (0.0)	2.77 (0.01)	37.15 (0.0)	1.25 (0.0)	19.17 (0.05)	5.26 (0.01)	45.5
gpt-3.5-turbo-1106	0.32 (0.0)	2.91 (0.03)	41.41 (0.0)	1.38 (0.0)	25.41 (0.08)	11.61 (0.06)	53.1
gpt-4-turbo	0.20 (0.0)	1.69 (0.02)	37.06 (0.01)	1.24 (0.0)	14.36 (0.11)	3.18 (0.03)	60.2
gpt-4	0.24 (0.0)	2.27 (0.01)	59.09 (0.01)	2.19 (0.0)	87.54 (0.22)	58.21 (0.13)	50.1
palm-2-chat-bison	0.34 (0.0)	3.25 (0.01)	91.90 (1.48)	3.35 (0.05)	179.21 (1.48)	89.71 (0.45)	38.8
claude-instant-1	0.38 (0.0)	3.80 (0.01)	47.61 (0.05)	1.52 (0.0)	29.66 (0.05)	13.33 (0.03)	25.3
gemini-pro	0.43 (0.0)	4.27 (0.01)	71.95 (0.08)	2.72 (0.0)	131.65 (0.5)	88.71 (0.37)	31.9

solely on these for several reasons. Firstly, identical time and space complexity annotations do not guarantee equivalent performance across different implementations. For instance, two algorithms with time complexities expressed as $T(2n)$ and $T(n)$ might both be classified under the same complexity order $O(n)$. However, their practical execution times and resource utilization can vary significantly, underscoring the limitations of using complexity classes as the sole measure of efficiency. Secondly, accurately determining the time and space complexity of a given piece of code typically requires manual analysis and labeling. This process is inherently subjective and prone to human error, making it less suitable for automated, large-scale evaluation of code generation models. The necessity for manual intervention contradicts our goal of automating the efficiency evaluation process as much as possible. Thirdly, although there are models designed to predict the time and space complexity of code, these predictions are often sub-optimal and can be inaccurate[2]. Relying on such models for critical evaluations might introduce significant errors, leading to misleading conclusions about a code generation model's efficiency. Given these considerations, we chose to focus on direct measurements of execution time and memory usage through our specified metrics. These measurements provide a more accurate, objective, and practical assessment of the generated code's efficiency, reflecting real-world performance more closely than theoretical complexity classes. This approach allows for a nuanced analysis of the models' output, enabling a comprehensive evaluation of their practical utility in software development scenarios.

GPT-4-turbo

```python
from typing import List

class Solution:
    def minGroups(self, intervals: List[List[int]]) ->
    ↪  int:
        points = []
        for start, end in intervals:
            points.append((start, 1))  # Record as the
            ↪  start of an interval
            points.append((end + 1, -1))  # Record as the
            ↪  end of an interval
        points.sort()

        # Sweep line algorithm to find the maximum number
        ↪  of overlapping intervals
        max_overlap = 0
        current_overlap = 0
        for _, diff in points:
            current_overlap += diff
            max_overlap = max(max_overlap,
            ↪  current_overlap)

        return max_overlap
```

Canonica Solution

```python
from typing import *

class Solution:
    def minGroups(self, intervals:
    ↪  List[List[int]]) -> int:
        h = []
        for a, b in
        ↪  sorted(intervals):
            if h and h[0] < a:
                heappop(h)
            heappush(h, b)
        return len(h)
```

Figure 13. A case illustration of GPT-4-turbo and Canonical solution. GPT-4-turbo obtains 0.93x NET compared to the canonical solution, which is the Min NET in Table 9. The problem ID in LeetCode for this problem is 2604.

[2] https://community.ibm.com/community/user/ai-datascience/blogs/sepideh-seifzadeh1/2021/10/05/ai-for-code-predict-code-complexity-using-ibms-cod