CLOVER: A TEST CASE GENERATION BENCHMARK WITH COVERAGE, LONG-CONTEXT, AND VERIFICATION

Jiacheng Xu, Bo Pang, Jin Qu, Hiroaki Hayashi, Caiming Xiong & Yingbo Zhou *
Salesforce AI Research
jiacheng.xu@salesforce.com

ABSTRACT

Software testing is a critical aspect of software development, yet generating test cases remains a routine task for engineers. This paper presents a benchmark, CLOVER, to evaluate models' capabilities in generating and completing test cases under specific conditions. Spanning from simple assertion completions to writing test cases that cover specific code blocks across multiple files, these tasks are based on 12 python repositories, analyzing 845 problems with context lengths ranging from 4k to 128k tokens. Utilizing code testing frameworks, we propose a method to construct retrieval contexts using coverage information. While models exhibit comparable performance with short contexts, notable differences emerge with 16k contexts. Notably, models like GPT-4o and Claude 3.5 can effectively leverage relevant snippets; however, all models score below 35% on the complex Task III, even with the oracle context provided, underscoring the benchmark's significance and the potential for model improvement. The benchmark is containerized for code execution across tasks, and we will release the code, data, and construction methodologies.

1 INTRODUCTION

Software testing is integral to the software development lifecycle Yoo & Harman (2012); Wang et al. (2024a); Alshahwan et al. (2024). From test-driven development (Mathews & Nagappan, 2024) to program repair Yasunaga & Liang (2021); Jimenez et al. (2024), crafting efficient and high-quality test cases is a routine task. Recently, large language models (LLMs) have gained attention for their potential in code and software testing enhancements. These models utilize context, user prompts, history, and code prefixes for code suggestions Nijkamp et al. (2023); 01.AI (2024); Roziere et al. (2023); Yang et al. (2024). To evaluate models' capability in writing code, many benchmarks have been proposed in the past few years. These benchmarks vary in focus, tackling areas such as basic coding problems Austin et al. (2021b); Chen et al. (2021), data science tasks Lai et al. (2022), reasoning challenges Gu et al. (2024), and issue resolution Jimenez et al. (2024). We summarize the most relevant work in Table 1.

Can LLMs write executable test cases with specific requirements in a realistic setup? To address this question, we create a benchmark, CLoVer, focusing on automatic evaluation of unit test case generated by LLMs. We create an automatic pipeline with little human intervention to scrape permissive repositories from GitHub and configure the execution environment. We identify potential problems by extracting and verifying test cases from the existing codebase. After this step, we take these problems and structure three challenging tasks: (1) *Task I* simulates a code completion tool by focusing on cloze-filling style questions; (2) *Task II* addresses scenarios requiring coverage and testing of specific methods or classes within the source code; (3) *Task III* involves improving code coverage, where models are challenged to cover certain code blocks within the source. The selection of example is driven by AST parser and code coverage results. We evaluate model performance by executing the generated code and capturing line coverage, offering a tangible measure of their effectiveness.

*Author contributions are listed in Section 8.

	Use Case	Data Source	PL	Size	Exec Repo	Constructed Context	Coverage
SWE-bench Jimenez et al. (2024)	issue resolution	SWE-Bench	Python	2294	√	Up to 50k	×
TestEval Wang et al. (2024b)	test case gen	Leetcode	Python	210	×	×	√
TestBench Zhang et al. (2024a)	test case gen	Github	Java	108	√	×	√
SWT-Bench Mündler et al. (2024)	tcg for issue reproduction	SWE-Bench	Python	1900+	√	×	√
TestGenEval Jain et al. (2024a)	test case gen	SWE-Bench	Python	1210	√	×	√
CLOVER	test case gen (3 tasks covering completion & generation)	Github (new)	Python	845 from 16,234	√	Up to 128k	√

Table 1: Comparison of CLOVER and other benchmarks pertaining to test case generation. CLOVER encompasses three unique tasks for generating test cases. It includes 845 problems, leading to a total of 5312 instances when accounting for different context settings. Numerous benchmarks for test case generation are based on the work by Jimenez et al. (2024).

In practical software testing, leveraging a comprehensive context window is crucial, encompassing dependencies and their antecedents. To evaluate models in a realistic context-aware fashion, we construct *oracle context* via test coverage for each example. We assess model performance across three tasks with context lengths spanning 4k to 128k tokens and introduce *context utilization* as a metric to assess how effectively models leverage extended contexts, independent of their absolute performance.

Our evaluation includes 10 open-source and 4 proprietary models. In Task I, many open-source models, such as MISTRAL-7B and QWEN 2.5CI-14B, underperform with longer contexts, indicating a decline in response quality despite their technical capacity to handle such lengths. In Tasks II and III, all models encounter difficulties in generating executable code, even when provided with oracle context. A notable trend is the sharp performance drop among open-source models starting at a 16k context window. The highest performance across all tasks is demonstrated by CLAUDE 3.5-S and GPT-4O, with GPT-4O achieving a 32.7% success rate on the most demanding task, Task III. *We identified a significant disparity in context utilization and long-context instruction-following capabilities between leading proprietary models and others.* Our data pipeline and evaluation sandbox are designed for scalability. We plan to release the code, benchmark, Dockerized environment, and recipes to enable the community to use these resources for further development and training. The benchmark also supports code agent by providing APIs and task instructions.

2 DATA & SANDBOX CONSTRUCTION

In Figure 1, we describe the overall pipeline from data collection to final evaluation. In this section, we will focus on data collection and environment setup.

	#ex	avail	# templ	PO	4k	8k	16k	32k	64k	128k
I	513	14952	1	0.9	2.7	5.5	11.0	21.9	42.8	-
II	151	184	2	1.0	-	3.9	11.8	27.7	58.7	-
III	181	1098	2	-	-	-	-	-	57.7	93.5

Table 2: Benchmark statistics. Number of unique ex(amples), number of templates and average number of tokens for different settings. Problem Only setting includes only the task instruction without supplementary context. One unique example will be populated into n examples for various settings. For instance, in Task II, one unique example will be expanded into $2 \times 5 = 10$ examples.

2.1 DATA COLLECTION

Source identification Following Jimenez et al. (2024), we began by scraping 42 new python repositories and ultimately narrowed it down to 12 repositories for actual use. Details and reasons for exclusions are provided in Appendix A. In our methodology, we identified folders containing source and test code by matching filenames with the pattern `test_*.py`. For eleven repositories, we could not extract test suites. This process resulted in the identification of test modules, each comprising at least one test function.

Problem extraction from file Test cases are extracted from modules by parsing Python files using the AST tool to identify test functions. Using heuristics, we isolate setup code s to remove unrelated test functions. In Figure 1, test functions `test_simple` and `test_iter` are preserved with the necessary setup code, resulting in self-contained problems named `tmp_test_lexnparse_[*].py`. We maintain the original structure and path of test modules.

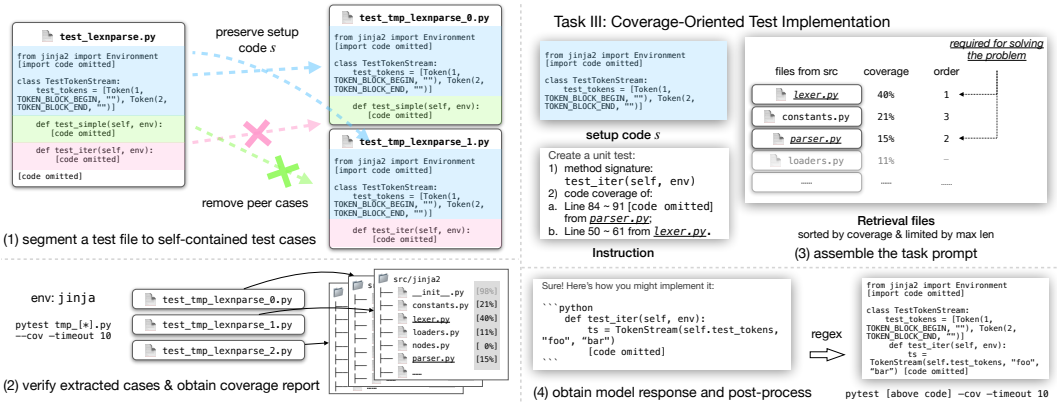

Figure 1: Pipeline overview. In this example, we focus on a test function `test_iter`, which covers the use of `Token` and `TokenStream` classes from the source code. There are four major steps: 1) we extract the problem from a test file `test_lexnparse.py`, 2) verify of the extracted case(s) by running pytest, 3) assemble task prompts with pre-constructed oracle dependent files, and 4) obtain model response and verify the execution status. In Task I, we mask part of the assertion statements. In Task II and III, we ask model to complete the test code almost from scratch with constraints imposed.

Verification API `verify` Executing new unit tests requires careful design. During the design and testing of our `verify` API, we considered several points: (1) Consistency check. Evaluate model-generated implementations against ground-truth code to identify issues from extraction, heuristics, or system conditions such as caching; (2) Batchify operations. Enable batch evaluation of test cases to decrease overhead from test framework executions and setups; (3) Timeout management. Prevent infinite loops in model-generated code; (4) Error handling and logging; (5) Repository restoration. Ensure repository state is reset before and after each use. We wrap this verification process to an API $\texttt{verify}(case) \rightarrow \{\text{true}, \text{false}\}$ where the output indicates whether the $case$ can execute successfully.

Coverage API `cov` The coverage API provides line coverage metrics for a test case across the entire repository. Utilizing pytest-cov, it reports hit and missed lines, and computes a file-level coverage rate, even if execution fails. Unlike `verify`, `cov` cannot be parallelized due to shared cache dependencies but can still deliver coverage reports on failed tests.

2.2 Sandbox Construction

To run test programs across different repositories, we create sandboxes and package them in a Docker image, maintaining minimal intervention to ensure the process is scalable to a larger number of repositories.

Procedure First, we create a conda virtual environment with Python version set to 3.10. Then we install packages including `poetry`[1] and `tox`[2]. We exhaustively search for `txt` files, and try to `pip install` those files. Then, `git submodule` related operations will handle submodules under the project if any. After this step, we try to install the package from the current project directory with `pip`, `poetry` and `tox`. After all the steps, we run `pytest` to check if we can find a significant number of passed test cases. In practice, the procedure above can automatically configure the environment of 25 out of 42 (59.5%) repositories. We describe more detail about construction failure in Section A.

Efficiency Tasks are evaluated sequentially, while evaluations within each task run concurrently across different repositories. The longest-running repository determines the evaluation's time bottleneck. To limit evaluation to 2 hours per model on a CPU Linux machine, we capped the maximum number of examples per repository: 50 for Task I, and 25 each for Task II and III.

[1] `https://python-poetry.org/`
[2] `https://tox.wiki/en/stable/`

2.3 EVALUATED MODELS

We utilized vLLM Kwon et al. (2023) for model inference with temperature and top_p set to 0.2 and 1.0. Maximum output lengths were 200, 4,000, and 4,000 for Tasks I, II, and III, respectively. To accommodate output tokens without exceeding model length limits, we adjusted the maximum sequence length by the output length during data preparation. The tokenizer from MISTRAL-7B was used in this process. We evaluated open-source models including CODEGEMMA-7B (Team et al., 2024), MAGICODER 6.7B Wei et al. (2024), QWEN 2.5CI-14B (Coder-Instruct) Yang et al. (2024), YI-CODER-9B 01.AI (2024), STARCODER2-15B Lozhkov et al. (2024), CODELLAMA-13B (Roziere et al., 2023), LLAMA 3.1-8B, LLAMA 3.1-70B (Dubey et al., 2024), CODESTRAL-22B, and MISTRAL-7B (Jiang et al., 2023). For proprietary models, we evaluated CLAUDE 3.5-S(onnet), GEMINI 1.5-F(lash), GPT-4O (2024-08-06), and GPT-4O-MINI (2024-07-18).

3 CONSTRUCTION OF ORACLE RETRIEVAL

To write or complete test cases, models need access to the related source code. To offer a simplified but *realistic* evaluation setting without using agents or retrievers, we provide oracle retrieval code in this benchmark. This leverages our executable environment and the `coverage` API for detailed coverage information. This setup aims to: (1) explore models' near-upper bound performance, (2) and test models in long-context scenarios. Our approach constructs long-contexts naturally and demands a multi-hop understanding of code and effective information use. Our setup is also perfect for software agent development.

Motivation Files such as `__init__.py` are often highly covered by most test cases, but they contribute little value in terms of addressing specific problems, as per information theory. These files can quickly deplete the context budget due to their high coverage rates. Hence, we need to calibrate the coverage information to reflect the importance of certain informative files.

Objective The objective of constructing the oracle retrieval is to provide the most relevant or informative content within a constrained context budget. For the rest of this section, we will describe how we prioritize salient information with coverage information.

Task I

Right after the above code block, you need to predict and fill in the blank _____ in the following assertion statement:
```python { cloze_question_with_blank } ```
Use the context and the provided prefix in the unit test method to infer the missing part of the assertion.

Task II

Here is the checklist:
The generated test code should contain ALL of the following calls:
- {target_name}, a Python {type} defined in file {file_name}.

Task III

Coverage Requirement
Please ensure that the following lines of code from the specified files are included and tested. Complete code snippets have already been provided earlier. When referring to coverage, it pertains to pytest coverage. If your code calls a function or method that indirectly utilizes the specified lines, this will also be considered as covering those lines.
{block.file_path}
```python { block.code_snippet } ```
This includes {block.line_count} lines from line numbers {block.start_line_number} to { block.end_line_number }.
[more code blocks if available]
Summary
In summary, there are {num_files} files and {num_code} code segments that need to be covered. Please make an effort to enhance the coverage of these lines, either directly or indirectly, to ensure thorough testing with pytest.

Figure 2: Task specific prompt template for Task I, II and III. For the complete prompts, check Sec C.

3.1 CALIBRATION OF COVERAGE

Within a file, we represent the test cases as $Y = \{y_1, y_2, \ldots, y_T\}$. Typically, these cases share some setup code and are organized under the same testing topic. The collection of all source files is denoted as $X = \{x_1, x_2, \ldots, x_F\}$. When using the `verify` API on T, we get a coverage tensor $\mathbb{C} \in \{1, 0\}^{T \times F \times L}$, where $C_{t,f,l} = 1$ indicates test case y_t covers the l-th line of file x_f, and $C_{t,f,l} = 0$ otherwise. $T = |Y|$ represents the total number of test cases in this file, $F = |X|$ is the total number of source files, and L is the max number of lines in src. We run pytest-cov two times for each test case y_t:

- a regular run C_t^{base}. This will return the regular coverage report of y_t over X.

- empty run C_t^{empty}. In this setting, we replace the code with an empty test statement: `def test(): assert True` and it will be deployed to the same location of y_t. For instance, if `test_iter` was implemented in `tests/util`, we will deploy the empty test to that directory as well.

Repository Baseline We propose a repository baseline is established by comparing C_t^{base} and C_t^{empty}:

$$Q_t^{\text{repo}} = \left\{ x_f \mid arg_f[\mathbb{1}(C_{t,f}^{base} = C_{t,f}^{\text{empty}})] \right\}$$

where $\mathbb{1}(\cdot)$ is the indicator function. The set Q_t^{repo} comprises the files x_f for which, for any test case index t, the coverage remains unchanged after executing the actual test case. This implies that the files in Q_t^{repo} offer minimal information gain in terms of entropy for generating test case y_t.

Peer Baseline To uniquely identify each test case, we set a Peer Baseline. The aim is to identify the most distinctive information across test cases. For a particular test case y_t, the Peer Baseline is defined as follows:

$$Q_t^{\text{peer}} = \left\{ x_f \mid arg_f[\mathbb{1}(\sum_{t'=1}^{T} C_{t',f,l}^{base} = 1)] \right\}$$

where $\mathbb{1}(\cdot)$ is the indicator function. $\sum_{t'=1}^{T} C_{t',f,l}^{base} = 1$ ensures that the line l is covered by exactly one test case (test case y_t), meaning it's only covered by the test case y_t. Next, we define Q', which is the set not meeting the criteria for either Q^{peer} or Q^{repo}: $Q'_t = F \setminus \left(Q_t^{\text{repo}} \cup Q_t^{\text{peer}}\right)$. We regard the value of files in Q' as lower than those in Q^{peer} but higher than the repository baseline Q^{repo}.

Calibration of Coverage Source files are classified into three categories: Q^{repo}, Q^{peer}, and Q'. The approach for assembling context for test case t gives precedence to files in the order of Q_t^{peer}, followed by Q'_t, and ultimately Q_t^{repo}. Within each category, we randomly select files if the context budget does not permit using them all.

3.2 Task Setup

Before diving into these three tasks, we define some terminologies which share across these tasks. For one task instance, we provide three categories of contents:

- Task instruction. We show examples in Fig 1 and 2.
- In-file code, including setup code s and function declaration f. Setup s prepares necessary components, such as imports, fixtures, and any initial configurations, required for the test. Function declaration f specifies the function's name, arguments, and any return types, if applicable. In Task I, we also provide code prefix, which will be discussed later.
- Source files per task requirement and from oracle retrieval. Files required by task are guaranteed to be provided unless in the Problem Only setting. It also has higher priority compared to oracle retrieval when we try to fill the context budget.

Setting We introduced two settings across three tasks, Problem Only (PO) and Contextual. In Problem Only setting, we only provide the Task instruction and in-file code. In contextual setting, we provide code snippets capped by context budget.

4 Task I: Mask Prediction in Assertion Statements

This task challenges the model to predict the missing element in an assertion statement within a test case, addressing the code completion feature offered by coding companions.

Problem Formulation For each problem x, it has following elements in the prompt of the task $[s, f, p, q, ref]$ in the Problem-Only setting:

- Prefix (p) refers to the existing code within the test function, serving as the context for solving assertion statements.

- Assertion statement with a `MASK` (q) represents the task for models to complete. Based on the surrounding code and any referenced materials, the model is expected to fill the gap and complete the assertion statements. q contains exactly one `MASK`.
- Reference answer (ref) is the original answer from the code. Any valid answer passing RER, a metric defined next, is acceptable as correct.

The model relies solely on the problem details, without extensive information about the method under test.

Cloze construction AST tools identify all possible assertion statements, including unary (e.g., variables, functions) and binary operations (comparisons). For binary, either operand can be masked. The suffix of q is removed to avoid hints, ensuring q is the last code line.

Example selection Preliminary study find that within each repository, there exists certain high frequent assertion statements, which provides unwanted hint to models. For instance, "200" (string literal) and 200 (number) are the most frequent candidates for the `MASK`. So we filter out problems with common ref. The chosen probability of a problem x_i is defined as:

$$p(x_i) = \begin{cases} 0, & \text{if } \frac{\text{count}(ref_i)}{N} > 0.01 \\ \frac{\text{len}(ref_i)}{\sum_{j=0}^{N} \text{len}(ref_j)}, & \text{otherwise} \end{cases}$$

where N is the total number of problems in one repository. We downsample to 50 problems per repository to maintain a diverse set of problems.

Prompt Template We explore two elicitation methods: (1) answer only ($pred$): the model yields the answer directly in a code block; (2) assertion statement with answer filled $q.\texttt{replace}(\texttt{MASK}, pred)$: the model returns the line with blank filled. Our studies with CODELLAMA-13B, MISTRAL-7B, CODEGEMMA-7B, and CODESTRAL-22B show the filled assertion method improves execution rate by at least 6.0%, thus we use it for Task I experiments.

	EM	ER	RER
CODEGEMMA-7B	30.3%	46.5%	43.7%
MAGICODER 6.7B	23.3%	40.6%	34.3%
CODELLAMA-13B	31.3%	52.6%	42.3%
STARCODER2-15B	32.7%	52.4%	41.1%
MISTRAL-7B	21.5%	39.4%	37.2%
QWEN 2.5CI-14B	52.6%	64.7%	59.8%
CODESTRAL-22B	45.5%	68.2%	63.3%
YI-CODER-9B	41.4%	61.7%	56.5%
LLAMA 3.1-8B	35.7%	57.4%	53.5%
LLAMA 3.1-70B	49.9%	74.2%	69.0%
GPT-4O-MINI	43.1%	70.8%	66.3%
GEMINI 1.5-F	49.7%	71.9%	67.0%
CLAUDE 3.5-S	56.3%	78.2%	72.4%
GPT-4O	55.2%	72.0%	68.0%

Table 3: Model performance on Task I in the Problem-Only setting. Refined execution rate (RER) is the recommended metric which reflects the models' ability in completing compilable and non-cheating assertion statements. Open source models are organized in ascending order based on their maximum supported length, followed by their model size.

Metrics & Verification Let the model prediction for `MASK` be $pred$. We implement three evaluation metrics for this task:: (1) Exact match is defined as $\text{EM} = \mathbf{1}(ref = pred)$; (2) Execution Rate (ER) indicates the execution result of the assertion statements filled with pred; (3) Refined execution rate (RER) is based on ER but we applied post-processing steps to remove trivial assertions and prohibit the copy of an existing assertion from the context.

Post-processing discards the following invalid scenarios: (1) $pred$ is a constant in a unary operation; (2) $pred$ is a constant in a binary operation where the other operand is also a constant; (3) in an equation comparison, $pred$ matches the operand on the opposite side. We follow the definition of constant in AST. Since the problems are selected given its surface length, as a proxy to its difficulty, the false negative ratio is considered low.

4.1 RESULTS IN THE PROBLEM ONLY SETTING

In this setting, we only provide the problem itself, excluding external context like the code of MUT. We present the result in Table 3. The best open-source model in this setting is QWEN 2.5CI-14B, achieving comparable performance compared to proprietary models. CLAUDE 3.5-S performs the best in all metrics.

Gap between EM and (R)ER We analyzed instances where predictions were successful in terms of RER but failed under the exact match (EM) criteria. The model's predictions averaged 25.8 characters compared to 30.7 characters in ground truth answers, suggesting a tendency toward brevity

Max Len	Model	PO	4k	8k	16k	32k	64k	Best	$\Delta_{max}(\uparrow)$	$\Delta_{min}(\uparrow)$
8k	CODEGEMMA-7B	43.7%	40.6%	41.5%	-	-	-	43.7%	-2.2%	-3.1%
16k	MAGICODER 6.7B	34.3%	33.0%	34.2%	31.0%	-	-	34.3%	-0.1%	-3.3%
16k	CODELLAMA-13B	42.3%	41.7%	42.3%	12.2%	-	-	42.3%	0.0%	-30.1%
16k	STARCODER2-15B	41.1%	37.3%	25.0%	25.4%	-	-	41.1%	-3.8%	-16.1%
32k	MISTRAL-7B	37.2%	34.1%	36.2%	12.2%	12.6%	-	37.2%	-1.0%	-25.0%
32k	QWEN 2.5CI-14B	59.8%	59.2%	60.4%	31.0%	30.7%	-	60.4%	0.6%	-29.1%
32k	CODESTRAL-22B	63.3%	65.0%	64.2%	22.3%	22.1%	-	65.0%	1.7%	-41.2%
131k+	YI-CODER-9B	56.5%	55.6%	53.4%	43.9%	40.2%	40.6%	56.5%	-0.9%	-16.3%
131k	LLAMA 3.1-8B	53.5%	52.1%	51.3%	49.1%	49.0%	47.3%	53.5%	-1.4%	-6.2%
131k	LLAMA 3.1-70B	69.0%	71.3%	71.1%	72.7%	71.7%	70.5%	72.7%	3.7%	1.5%
131k	GPT-4O-MINI	66.3%	66.0%	67.3%	65.3%	66.7%	67.7%	67.7%	1.4%	-1.0%
131k+	GEMINI 1.5-F	67.0%	67.6%	66.6%	66.0%	66.0%	66.2%	67.6%	0.6%	-1.0%
131k	CLAUDE 3.5-S	72.4%	71.5%	73.4%	73.4%	74.4%	75.4%	75.4%	3.0%	-0.9%
131k	GPT-4O	68.0%	71.0%	71.5%	70.9%	69.8%	67.2%	71.5%	3.5%	-0.8%

Table 4: Refined execution rate (RER) of models in contextual settings for Task I. The table lists models according to their maximum supported sequence length. The Problem-Only (PO) column indicates the baseline performance of models with average short input lengths (0.9k tokens). 'Best' highlights the highest performance achieved across both settings (underscored for each model). Ctx Util measures the maximum and minimum performance change when shifting from baseline to contextual inputs. We highlight Δ in green if $\Delta > 0\%$ and in red if $\Delta < -20\%$.

Model	Task II Performance at Max Seq Length					Best	Ctx Util		Task III Perf	
	PO	8k	16k	32k	64k		$\Delta_{max}\uparrow$	$\Delta_{min}\uparrow$	64k	128k
CODEGEMMA-7B	14.9%	10.5%	-	-	-	14.9%	-4.4%	-4.4%	-	-
MAGICODER 6.7B	13.7%	20.2%	0.0%	-	-	20.2%	6.5%	-13.7%	-	-
CODELLAMA-13B	8.4%	8.4%	0.0%	-	-	8.4%	0.0%	-8.4%	-	-
STARCODER2-15B	13.8%	19.1%	8.5%	-	-	19.1%	5.3%	-5.3%	-	-
MISTRAL-7B	8.4%	11.6%	0.0%	0.0%	-	11.6%	3.2%	-8.4%	-	-
QWEN 2.5CI-14B	23.2%	25.3%	0.0%	0.0%	-	25.3%	2.1%	-23.2%	-	-
CODESTRAL-22B	20.0%	28.4%	0.0%	0.0%	-	28.4%	8.4%	-20.0%	-	-
YI-CODER-9B	14.7%	20.0%	14.7%	13.7%	13.7%	20.0%	5.3%	-1.0%	4.7%	3.7%
LLAMA 3.1-8B	5.3%	5.3%	4.2%	9.5%	5.3%	9.5%	4.2%	-1.1%	1.9%	0.9%
LLAMA 3.1-70B	12.9%	14.9%	13.8%	24.7%	18.3%	24.7%	11.8%	0.9%	6.5%	3.7%
GPT-4O-MINI	17.0%	22.3%	25.5%	26.6%	25.5%	26.6%	9.6%	5.3%	19.6%	21.5%
GEMINI 1.5-F	17.9%	17.9%	23.2%	21.1%	16.8%	23.2%	5.3%	-1.1%	28.0%	27.1%
CLAUDE 3.5-S	26.9%	46.2%	39.8%	48.4%	46.2%	48.4%	21.5%	12.9%	29.0%	30.8%
GPT-4O	28.7%	37.2%	42.6%	40.9%	48.4%	48.4%	19.7%	8.5%	32.7%	31.8%

Table 5: Success rate on Task II and Task III (rightmost two columns). In Task II, we constructed the Problem-Only setting and contextual setting from 8k to 64k. In Task III, we only have the contextual setting since it heavily depends on retrieval files. For models that had a 0.0% success rate, with a manual inspection we discovered the outputs contained gibberish, such as backticks, line break symbols, and random words. We did not alter any configurations in vLLM for these models while testing with various lengths.

or shortcuts. Common scenarios where execution succeeds but predictions do not exactly match the original code include: (1) Unary operations in q with non-constant pred (e.g., `assert` pred); (2) Use of syntactic sugar, such as considering `x in dict` equivalent to `x in dict.keys()`; (3) String manipulations, including `strip` functions and interchangeable quotation marks; (4) Assertions for non-existence, like `assert x not in y`, where x is flexible Currently, the execution method lacks understanding of contextual semantics or user intent. A future goal is to develop a tool that can evaluate responses based on execution success and alignment with user intent.

4.2 RESULTS IN CONTEXTUAL SETTINGS

We present the result in Table 4. The best overall performance is achieved by CLAUDE 3.5-S in both settings, with a 3.0% gain from 72.6% to 75.6%. In the contextual setting, we found there is a sharp decrease after 8k max length in most open source models including CODELLAMA-13B, STARCODER2-15B, MISTRAL-7B, QWEN 2.5CI-14B, and CODESTRAL-22B. We examined the model response in these cases and we find that the chance of getting gibberish output increases along with the increase of input length. Note that the prompt template remains the same for context free and contextual setting where the only change applied is the additional code snippets.

Context Utilization Δ We introduce a novel metric to measure models' capability in effectively utilizing the context. On the performance gain side, we define it as $\Delta_{max} = \max(r_{4k}, r_{8k}, \ldots, r_{\text{maxLen}}) - r_0$ where r_0 is the context free baseline performance. Since we provide oracle context to the model, shorter context carrying strong hint could be sufficient and even better than longer sequence. Δ_{max} measures the best possible gain a model could get. Vice versa, we define $\Delta_{min} = \min(r_{4k}, r_{8k}, \ldots, r_{\text{maxLen}}) - r_0$. This set of metrics focuses on the relative performance change given longer context. The ideal value, if context provides good source of information, for this metric follows this equation $\Delta_{max} > \Delta_{min} > 0$.

5 TASK II: TARGETED TEST IMPLEMENTATION

In Task II and Task III, we will shift from code completion to open-ended code generation, which is more challenging and requires longer context. In Task II, given a python class or function from the source code, the model needs to complete the test code by using the target.

Problem Formulation For each problem, we provide setup s, function declaration f and a specification to use the target. We show the specification template and an example in Figure 2. The "target_name" here is the name of the class or function. The "type" is either "class" or "function". "file_name" is where the target object was implemented.

Data Construction We use AST tools to parse the code and identify all `Attribute` type nodes through a recursive walk to find suitable targets. These identified classes and functions become potential targets. They are then matched with those covered by this case in C_t^{base}. A random target is selected as the requirement. In settings ranging from 8k to 64k, we ensure the inclusion of the necessary file as specified. The maximum length constraint is 8k and the output length is 4k, thus the combined length of instructions and the required file must not exceed 4k. Any cases exceeding this limit are discarded. By setting a single target, we maximize the inclusion of examples.

Answer Format In this task, the generated code is the completion of the function declaration f. We designed two prompt templates to capture the output, one with the completion part only ($prompt_{part}$), and one with the full code block (s, f) along with the completion $prompt_{full}$.

Metrics We define two metrics for Task II and Task III. *Execution Rate* measures if the generated code can be captured, and executed successfully. Any test failures, exceptions and timeout will count as execution failure. *Success Rate* Besides the execution rate, we also check whether the specification was satisfied. For Task II, we check whether the required target was in the generated code. For Task III, we check for code coverage.

Result We report the Success Rate using $prompt_{full}$ in Table 5 and $prompt_{part}$ in Table 6. In the context free setting of this task, it provides *no* context to the model, not even the "file_name" required to complete the task. For most of the models, there is a significant performance boost from context-free to 8k. With only 8k context length, CLAUDE 3.5-S achieves surprisingly high performance (46.2%), 9.0% ahead of the second best model GPT-4O. Some models, however, remain the same or even get slightly worse performance, including CODEGEMMA-7B, CODELLAMA-13B, LLAMA 3.1-8B, and GEMINI 1.5-F. The best performance is achieved by CLAUDE 3.5-S and GPT-4O at 32k and 64k respectively. Starting at 16k, we have seen a sudden performance drop on CODELLAMA-13B, MISTRAL-7B, QWEN 2.5CI-14B, and CODESTRAL-22B.

6 TASK III: COVERAGE-ORIENTED TEST IMPLEMENTATION

In this task, given some code blocks from source code, the model needs to complete the test code and cover those target blocks. This task shares a lot of similarity with Task II, so we will focus on the different part.

Problem Formulation For each problem, we provide $[s, f]$ and a specification to cover up to 10 code blocks from the source code. We provide the full code snippet in the "Retrieved Code Snippets

for Reference" section of the prompt along with other oracle retrieval files. In the specification prompt, we provide the file name, the code blocks to cover, and the starting and ending line number for these code blocks in the original file.

Data Construction We took a deterministic approach to select code blocks rather than randomly choosing code spans. We use the Q_t^{peer} to guide the selection of code blocks to cover. For a case y_t, we check if there is some code blocks only covered by it, not any other peer cases. Typically it's the case where a conditional branch or a function is hit by only one case. We also filter out code blocks with fewer than 5 lines as we do not want to include many code fragments. The max number of files to cover is set at 10. With this approach, we can guide the model with a feasible and reasonable coverage requirement which also aligns with the function name and arguments.

The answer format and metrics of Task III remains the same with Task II. The different task specific prompt is shown in Figure 2. In this setting, since we include up to 10 files, we set the total sequence length to 64k and 128k to keep as many examples as possible.

Results We present the success rate of Task III in the rightmost 2 columns in Table 5. GPT-4O achieves the best performance in 64k setting with 32.7%. None of the open-source models passed 10% in this task. CLAUDE 3.5-S and GPT-4O-MINI are the models performing better with longer context provided.

Can LLMs satisfy the coverage requirement? To answer this question, we compare the execution rate and the success rate in Table 3. The gap of whether the coverage requirements can be met is around 5 to 10% for different models. During evaluation, we only consider it a success if *all* of the code blocks' coverage requirements are satisfied. As the result shows, there is generally a gap between execution rate v.s. success rate, indicates the generated test cases are generally not satisfying all the coverage requirements.

	Success Rate		Execution Rate	
Model	64k	128k	64k	128k
YI-CODER-9B	4.7%	3.7%	11.2%	8.4%
LLAMA 3.1-8B	1.9%	0.9%	2.8%	4.7%
LLAMA 3.1-70B	6.5%	3.7%	8.4%	10.3%
GPT-4O-MINI	19.6%	21.5%	25.2%	25.2%
GEMINI 1.5-F	28.0%	27.1%	38.3%	36.4%
CLAUDE 3.5-S	29.0%	30.8%	34.6%	35.5%
GPT-4O	32.7%	31.8%	42.1%	39.3%

Figure 3: Success rate and execution rate of models on Task III.

7 RELATED WORKS

Code & Test Case Generation Benchmarks The development of benchmarks for evaluating code generation models has been active. Earlier benchmarks like HumanEval Chen et al. (2021) and MBPP Austin et al. (2021a) focused on basic or LeetCode-style programming problems. BigCodeBench Zhuo et al. (2024) extends this with complex function calls, while DevBench Li et al. (2024) evaluates model performance in entire software development lifecycle. We summarized several recent benchmarks related to test case generation in Table 1. The source data and dev environment of Jimenez et al. (2024) has been widely adopted to develop new benchmarks. Jain et al. (2024b) demonstrated a scalable framework to turn any GitHub repo into an interactive environment for agents.

Test Case Generation LLMs are widely used for automating test case generation. Chen et al. (2024) employs LLMs for efficient test creation. Liu et al. (2024a) utilize LLMs for bug detection, while Tang et al. (2024); Yuan et al. (2024) enhance ChatGPT's unit test generation capabilities. Alshahwan et al. (2024) explore LLM use in industry. Neural models for test case generation were proposed by Tufano et al. (2020); Nie et al. (2023). Ryan et al. (2024) investigated coverage-guided test case generation with LLMs.

Code LLMs and Agents Recent studies on code-specific LLMs Roziere et al. (2023); Lozhkov et al. (2024); Hui et al. (2024) showcase the potential of specialized models for code generation. The StarCoder 2 Lozhkov et al. (2024) and open code models based on GEMMA Team et al. (2024) show the evolution of LLMs tailored for programming tasks.

Long Context for Code Existing long context benchmarks either exclude coding tasks or have restricted access to code LLMs. RULER Hsieh et al. (2024) and ∞Bench Zhang et al. (2024b) fail to replicate real-world software development scenarios. Meanwhile, code benchmarks are insufficient

for long context evaluation. RepoBench Liu et al. (2024b) sets the maximum token threshold of 12k for Python and 24k for Java. Most test cases in TestBench Zhang et al. (2024a) are within 32k tokens and TestGenEval Jain et al. (2024a) evaluates context length up to 32k. SWE-Bench Jimenez et al. (2024) focuses on context length less than 50k, which is far behind many recent models often claiming to be able to consume 128k+ context length.

8 Limitation & Conclusion

Our study is confined to Python and specific pytest tools. We didn't explore using code agents to tackle problems in this benchmark. Preliminary findings indicate that code agents, such as those by Wang et al. (2024c), are costly to run due to the need to explore entire repositories, primarily because of the overhead from reading large files.

In this study, we introduce a benchmark designed for multiple real-world software testing scenarios. We identified significant gaps in long-context handling, context utilization, and instruction-following capabilities between open-source and closed-source models, pointing to substantial opportunities for improvement. The coverage-driven oracle context could advance research on long-context evaluation in (code) LLMs. Researchers can use the API for real-time feedback to enhance models' coding and reasoning skills. Additionally, the data pipeline can be adapted to create high-quality training datasets.

Author Contributions

All authors contributed to discussions throughout the project. JX led the project, developed most of the code, executed the experiments, and wrote the majority of the manuscript. BP supported the model inference tasks for open-source models. JQ provided insights into long-context models and related literature. HH applied code agents to the benchmark and conducted various analyses. CX and YZ supervised the project, overseeing its initiation, scope definition, infrastructure, and publication.

Acknowledgments

The authors wish to thank the anonymous reviewers for their constructive feedback.

References

01.AI. Meet yi-coder: A small but mighty llm for code, September 2024. URL https://01-ai.github.io/blog.html?post=en/2024-09-05-A-Small-but-Mighty-LLM-for-Code.md.

Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 185–196, 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021a.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021b.

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

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 572–576, 2024.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. In *Forty-first International Conference on Machine Learning*, 2024.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark, 2024a. URL https://arxiv.org/abs/2410.00752.

Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2e: Turning any github repository into a programming agent environment. In *Forty-first International Conference on Machine Learning*, 2024b.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501, 2022.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. Devbench: A comprehensive benchmark for software development, 2024. URL https://arxiv.org/abs/2403.08604.

Kaibo Liu, Yiyang Liu, Zhenpeng Chen, Jie M Zhang, Yudong Han, Yun Ma, Ge Li, and Gang Huang. Llm-powered test case generation for detecting tricky bugs. *arXiv preprint arXiv:2404.10304*, 2024a.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024b.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa

Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024. URL https://arxiv.org/abs/2402.19173.

Noble Saji Mathews and Meiyappan Nagappan. Test-driven development for code generation, 2024. URL https://arxiv.org/abs/2402.13521.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents, 2024. URL https://arxiv.org/abs/2406.12952.

Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. Learning deep semantics for test completion. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 2111–2123. IEEE, 2023.

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

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

Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971, 2024.

Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering*, 2024.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, et al. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*, 2024.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*, 2020.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 2024a.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. Testeval: Benchmarking large language models for test case generation, 2024b. URL https://arxiv.org/abs/2406.04531.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An Open Platform for AI Software Developers as Generalist Agents, 2024c. URL https://arxiv.org/abs/2407.16741.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with OSS-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 52632–52657. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/wei24h.html.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 11941–11952. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/yasunaga21a.html.

Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1703–1726, 2024.

Quanjun Zhang, Ye Shang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. Testbench: Evaluating class-level test case generation capability of large language models, 2024a. URL https://arxiv.org/abs/2409.17561.

Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Hao, Xu Han, Zhen Thai, Shuo Wang, Zhiyuan Liu, and Maosong Sun. ∞Bench: Extending long context evaluation beyond 100K tokens. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15262–15277, Bangkok, Thailand, August 2024b. Association for Computational Linguistics. URL https://aclanthology.org/2024.acl-long.814.

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

	Performance at Max Seq Length						Ctx Util	
	PO	8k	16k	32k	64k	Best	$\Delta_{max} \uparrow$	$\Delta_{min} \uparrow$
CODEGEMMA-7B	10.6%	11.7%	-	-	-	11.7%	1.1%	1.1%
MAGICODER 6.7B	13.8%	19.1%	0.0%	0.0%	3.2%	19.1%	5.3%	-13.8%
CODELLAMA-13B	9.6%	8.4%	0.0%	-	-	9.6%	-1.2%	-9.6%
STARCODER2-15B	12.6%	22.1%	9.5%	-	-	22.1%	9.5%	-3.1%
MISTRAL-7B	9.6%	9.6%	0.0%	0.0%	-	9.6%	0.0%	-9.6%
QWEN 2.5CI-14B	27.4%	20.0%	0.0%	0.0%	-	27.4%	-7.4%	-27.4%
CODESTRAL-22B	20.2%	27.4%	0.0%	0.0%	-	27.4%	7.2%	-20.2%
YI-CODER-9B	12.6%	23.2%	15.8%	6.3%	7.4%	23.2%	10.6%	-6.3%
LLAMA 3.1-8B	7.4%	11.6%	6.4%	9.5%	9.6%	11.6%	4.2%	-1.0%
LLAMA 3.1-70B	14.0%	12.9%	14.9%	13.7%	12.8%	14.9%	0.9%	-1.2%
GPT-4O-MINI	14.7%	27.7%	22.1%	24.5%	20.2%	27.7%	13.0%	5.5%
GEMINI 1.5-F	20.0%	21.1%	26.3%	24.2%	23.2%	26.3%	6.3%	1.1%
CLAUDE 3.5-S	28.4%	45.3%	44.2%	46.3%	48.4%	48.4%	20.0%	15.8%
GPT-4O	25.8%	33.3%	39.8%	39.4%	44.7%	44.7%	18.9%	7.5%

Table 6: Success Rate of Task II with $prompt_{part}$.

A REPOSITORIES SCRAPPED & USED

The benchmark incorporates the following repositories from GitHub: Pillow, elasticsearch-py, flask, httpx, jinja, kombu, paramiko, pip, requests, sqlalchemy, starlette, and pylint. Conversely, the repositories not utilized include: salt, celery, aiohttp, pytest, sphinx, docker-py, channels, mongoengine, boto3, scrapy, requests-html, black, dd-trace-py, ansible, pyzmq, python-prompt-toolkit, blessed, fastText, google-api-python-client, h2, scikit-learn, httpbin, ipython, libcloud, matplotlib, numpy, pandas, twisted, and voluptuous.

The cutoff date for this benchmark is set for August 30, 2024. Various reasons account for not using all repositories, such as:

- some repositories do not support `pytest` and/or `pytest-cov`,
- challenges in automatically configuring the environment or extracting test cases,
- non-standard naming of tests that disrupts our heuristics,
- failure of rule-based folder localization approach (a.k.a. finding `tests` and `src` folder) due to non-standard naming or project structure. For instance, `elasticsearch-py` has source code folder and test code folder named as `elasticsearch` and `test_elasticsearch`. There are 11 repositories where we manually specified its folder name,
- some repositories being very slow or causing issues during evaluation, and
- requirements for external setup, non-Python setup, or specific system configurations for some repositories.

B RESULTS WITH $prompt_{part}$

In Table 6 we present the results with $prompt_{part}$ on Task II. In Table 7 we demonstrate the results on Task III. We found for most of the models, the $prompt_{full}$ which asks for the whole code block works better than $prompt_{part}$ in practice.

C PROMPT TEMPLATES AND EXAMPLES

We list the prompts for Task I in Fig 4 and Fig 5, Task II in Fig 6 and Task III in Fig 7.

Task I (Problem Only)

Task Overview
You need to complete the assertion in the provided unit test method by considering the context from the corresponding Python file.
Context Information
Below is the file context, which includes necessary imports and setups for the unittest:

```
{{ context }}
```

Target Unit Test Method
This is the unit test method for which you need to complete the assertion statement:

```
{{ prefix }}
```

Your Task
Right after the above code block, you need to predict and fill in the blank (`cloze_key`) in the following assertion statement:

```
{{ cloze_question }}
```

Use the context and the provided prefix in the unit test method to infer the missing part of the assertion.
Output Format
Your output should be a finished assertion statement. Do **not** generate the entire unit test method; only produce the assertion statement.
Example
For instance, if the cloze question is `assert ______ == (25, 25, 75, 75)`, and your prediction for the blank is `im.getbbox(alpha_only=False)`, your output should be:

```
assert im.getbbox(alpha_only=False) == (25, 25, 75, 75)
```

Please provide your output in the desired format without additional explanations or step-by-step guidance.
Notes

- The completed assertion should not be trivial. For instance, `assert True == True` and `assert str("a") == "a"` are considered trivial assertions.
- There will be precisely one blank in the assertion statement to be filled in.

Figure 4: PO prompt of Task I.

	Success Rate		Execution Rate	
Model	64k	128k	64k	128k
YI-CODER-9B	10.3%	10.3%	22.4%	17.8%
LLAMA 3.1-8B	0.9%	0.9%	4.7%	1.9%
LLAMA 3.1-70B	11.2%	4.7%	14.0%	7.5%
GPT-4O-MINI	21.5%	20.6%	29.9%	28.0%
GEMINI 1.5-F	26.2%	25.2%	36.4%	34.6%
CLAUDE 3.5-S	32.7%	29.9%	42.1%	37.4%
GPT-4O	28.0%	29.9%	35.5%	41.1%

Table 7: Success rate and execution rate of Task III with $prompt_{part}$.

Task I (Contextual)

Task Overview
You need to complete the assertion in the provided unit test method by considering the context from the corresponding Python file.
Retrieved Code Snippets for Reference
Here are a few code snippets retrieved for your reference while making your prediction:

```
{{ retrieved_snippet }}
```

Context Information
Below is the file context, which includes necessary imports and setups for the unittest:

```
{{ context }}
```

Target Unit Test Method
This is the unit test method for which you need to complete the assertion statement:

```
{{ prefix }}
```

Your Task
Right after the above code block, you need to predict and fill in the blank (`cloze_key`) in the following assertion statement:

```
{{ cloze_question }}
```

Use the context and the provided prefix in the unit test method to infer the missing part of the assertion.
Output Format
Your output should be a finished assertion statement. Do **not** generate the entire unit test method; only produce the assertion statement.
Example
For instance, if the cloze question is `assert _____ == (25, 25, 75, 75)`, and your prediction for the blank is `im.getbbox(alpha_only=False)`, your output should be:

```
assert im.getbbox(alpha_only=False) == (25, 25, 75, 75)
```

Please provide your output in the desired format without additional explanations or step-by-step guidance.
Notes

1. The completed assertion should not be trivial. For instance, `assert True == True` and `assert str("a") == "a"` are considered trivial assertions.
2. There will be precisely one blank in the assertion statement to be filled in.

Figure 5: Contextual prompt of Task I.

Task Overview
Create a unit test based on the provided method signature and given context. All necessary imports for the test have been included, so focus solely on writing the unit test method. No additional libraries can be imported.
Retrieved Code Snippets for Reference
Several code snippets have been provided for your reference:

```
{{ retrieved_snippet }}
```

Hint: Depending on the target unit test method's signature, you might want to utilize or test these code snippets.
Requirement: Each retrieved code snippet must be used at least once in the final unit test method.
Context Information
Below is the test code context, including necessary imports and setups for unittest:

```
{{ context }}
```

Signature of Target Unit Test Method
This is the unit test method signature you need to complete:

```
{{ prefix }}
```

Your Task
Continue writing the unit test method immediately following the provided method signature, ensuring to include at least one substantial assertion. Your task is to replace "<YOUR CODE HERE>" with correctly indented code.
Output Format
Please return the **WHOLE FILE** content (including the context and the completed unit test method, but not the Retrieved Code Snippets). You should not modify the context part of the code.
Example 1:

```
class A:
    def __init__(self):
        self.a = 1
        self.b = 2
    def test_comparing_a_b():
<YOUR CODE HERE>
    def test_assert_b():
        assert self.b == 2
```

The output should follow the correct indentation principles. For example:

```
class A:
    def __init__(self):
        self.a = 1
        self.b = 2
    def test_comparing_a_b():
        assert self.a == 1       # Correct indentation with eight spaces before `assert`
        assert self.a != self.b  # Correct indentation with eight spaces before `assert`
    def test_assert_b():
        assert self.b == 2
```

Here is one WRONG example:

```
# wrong example
class A:
    def __init__(self):
        self.a = 1
        self.b = 2
    def test_comparing_a_b():
    assert self.a == 1       # Incorrect indentation
assert self.a != self.b  # Incorrect indentation
    def test_assert_b():
        return           # YOU SHOULD NEVER modify the context code
```

Example 2:

```
def test_value_c():
<YOUR CODE HERE>
```

The output should follow the correct indentation principles. For example:

```
def test_value_c():
    assert c == 3
```

- Keep in mind that the "<YOUR CODE HERE>" section lacks any leading spaces. - Ensure you include the appropriate amount of indentation before the 'assert' statements. - If the method signature necessitates returning an object, implement that as well.

Figure 6: Contextual $prompt_{full}$ of Task II.

Task Overview
Create a unit test based on the provided method signature and given context. All necessary imports for the test have been included, so focus solely on writing the unit test method. No additional libraries can be imported.
Retrieved Code Snippets for Reference
Several code snippets have been provided for your reference:

```
{{ retrieved_snippet }}
```

Output Format
When completing a code snippet, the output should include the entire code context provided, including any imports, class definitions, or function signatures. Replace placeholders with meaningful code that fits the context. Example:
Given:

```
import pytest
from aiohttp import web
from aiohttp.web_urldispatcher import UrlDispatcher

@pytest.fixture
def router() -> UrlDispatcher:
    return UrlDispatcher()

def test_get(router: UrlDispatcher) -> None:
{{ SYMBOL_YOUR_CODE }}
```

The goal is to complete the method `test_get(router: UrlDispatcher) -> None:` with correct indentation and necessary logic while retaining the full context.
Example output:

```
import pytest
from aiohttp import web
from aiohttp.web_urldispatcher import UrlDispatcher

@pytest.fixture
def router() -> UrlDispatcher:
    return UrlDispatcher()

def test_get(router: UrlDispatcher) -> None:
    async def handler(request: web.Request) -> NoReturn:
        assert False

    router.add_routes([web.get("/", handler)])
    route = list(router.routes())[1]
    assert route.handler is handler
    assert route.method == "GET"
```

Task with Context Provided
Below is the test code context, including necessary imports and setups for `unittest`. Following the method signature " `method_signature` ", you need to complete the unit test method. Ensure to include at least one substantial assertion. Replace `SYMBOL_YOUR_CODE` with the correctly indented test code.

```
{{ context }}
```

Hint: Depending on the target unit test method's signature, you might want to utilize or test these code snippets.
Requirement:

- Ensure you return the unit test method including the given method signature. Do not modify the method signature.
- You are not allowed to import anything else as all necessary imports for the case have been provided.
- Properly indent each line before including it.
- Avoid using ANY trivial assertions such as `assert True == True` or `assert str("a") == "a"`, as they will be deemed incorrect.

```
{{ coverage_requirement }}
```

Figure 7: Contextual $prompt_{full}$ for Task III.