ACE-BENCH: BENCHMARKING AGENTIC CODING IN END-TO-END DEVELOPMENT OF COMPLEX FEATURES

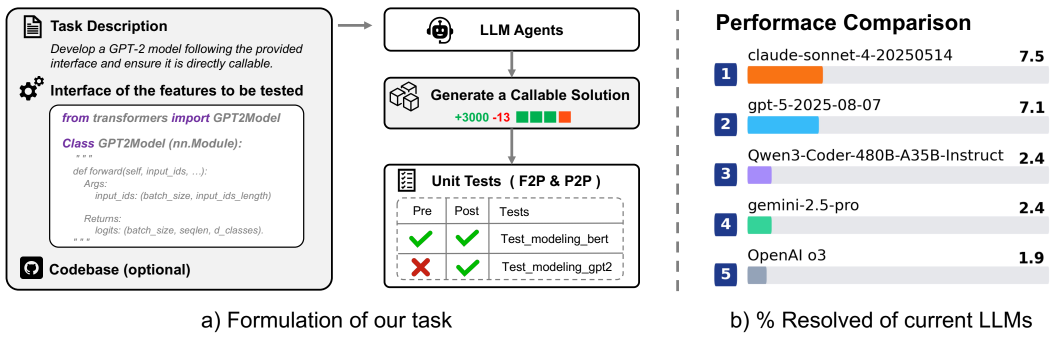
Anonymous authors
Paper under double-blind review

ABSTRACT

Agents powered by large language models (LLMs) are increasingly adopted in the software industry, contributing code as collaborators or even autonomous developers. As their presence grows, it becomes important to assess the current boundaries of their coding abilities. Existing agentic coding benchmarks, however, cover a limited task scope, *e.g.*, bug fixing within a single pull request (PR), and often rely on non-executable evaluations or lack an automated approach for continually updating the evaluation coverage. To address such issues, we propose ACE-Bench, a benchmark designed to evaluate agentic coding performance in end-to-end, feature-oriented software development. ACE-Bench incorporates an execution-based evaluation protocol and a scalable test-driven method that automatically derives tasks from code repositories with minimal human effort. By tracing from unit tests along a dependency graph, our approach can identify feature-level coding tasks spanning multiple commits and PRs scattered across the development timeline, while ensuring the proper functioning of other features after the separation. Using this framework, we curated 212 challenging evaluation tasks and 889 executable environments from 16 open-source repositories in the first version of our benchmark. Empirical evaluation reveals that the state-of-the-art agent, such as Claude 4 Sonnet with OpenHands framework, which achieves a 70.4% resolved rate on SWE-bench, succeeds on only 7.5% of tasks, opening new opportunities for advancing agentic coding. Moreover, benefiting from our automated task collection toolkit, ACE-Bench can be easily scaled and updated over time to mitigate data leakage. The inherent verifiability of constructed environments also makes our method potentially valuable for agent training. Our data and code will be publicly released.

1 INTRODUCTION

Software development is rapidly evolving with the advent of large language models (LLMs) (Sapkota et al., 2025), marking a shift toward end-to-end agentic coding systems (Wang et al., 2025). Recent advances, such as Claude Code (Anthropic, 2025a) and Qwen Code (Qwen, 2025) exem-


Task Description
Develop a GPT-2 model following the provided interface and ensure it is directly callable.
Interface of the features to be tested
Codebase (optional)
LLM Agents
Generate a Callable Solution
+3000 -13
Unit Tests (F2P & P2P)
Pre
Post
Tests
Test_modeling_bert
Test_modeling_gpt2
a) Formulation of our task
Performace Comparison
1 claude-sonnet-4-20250514 7.5
2 gpt-5-2025-08-07 7.1
3 Qwen3-Coder-480B-A35B-Instruct 2.4
4 gemini-2.5-pro 2.4
5 OpenAI o3 1.9
b) % Resolved of current LLMs

```
from transformers import GPT2Model

Class GPT2Model (nn.Module):
    """
   def forward(self, input_ids, ...):
      Args:
         input_ids: (batch_size, input_ids_length)

      Returns:
         logits: (batch_size, seqlen, d_classes).
   """
```

Figure 1: a) The agent must implement a directly callable feature based on the task description and interface definitions, either by developing from scratch or extending an existing repository. b) Our benchmark shows that even Claude 4 achieves only a 7.5% solution rate.

Benchmark	Feature-oriented Agentic Coding	Execution-based Evaluation	Scalable Instance Collection	Continually Updatable	Instance Number
BigCodeBench (Zhuo et al., 2025)		✓			1140
LiveCodeBench (Jain et al., 2025a)		✓			454
FullStackBench (Cheng et al., 2024)		✓			3374
SWE-bench (Jimenez et al., 2024)		✓	✓	✓	500
PaperBench (Starace et al., 2025)	✓				20
Paper2Coder (Seo et al., 2025)	✓				90
MLEBench (Chan et al., 2025)	✓	✓			72
DevEval (Li et al., 2025)	✓	✓			20
GitTaskBench (Ni et al., 2025)	✓	✓			54
ACE-Bench (ours)	✓	✓	✓	✓	212

Table 1: A comparison of ACE-Bench with current coding benchmarks reveals that our bench emphasizes feature-level realistic software development. It leverages an execution-based evaluation pipeline and integrates a test-driven toolkit for the automatic generation of task instances.

plify this evolution by introducing requirement-driven agents that autonomously plan, execute, and interact with external tools (*e.g.*, compilers) to iteratively tackle complex software development tasks (Gong et al., 2025), thereby relegating human intervention to a supervisory role.

Recently, various benchmarks have been introduced to assess this paradigm shift, including SWE-bench (Jimenez et al., 2024), PaperBench (Starace et al., 2025), and GitTaskBench (Ni et al., 2025). While these benchmarks have made significant contributions to task-oriented agentic coding, they are limited either by the narrow focus on bug-level scenarios or by reliance on handcrafted generation pipelines. As agentic coding expands toward more complex settings, such as feature-level development, these constraints hinder their ability to fully capture the capabilities of frontier code agents. Therefore, there is a need to build a challenging benchmark that broadens evaluation scope to feature-level scenarios, supported by automated collection toolkits to facilitate its future usage.

Constructing such a benchmark poses nontrivial challenges. Effective and execution-based evaluation of feature-level agentic coding generally depends on clearly defined functional interfaces to resolve ambiguities between the implementation and test criteria. However, these specifications are often absent in previous benchmarks. Furthermore, creating an automated data collection toolkit to support the scaling of benchmarks introduces additional complexities. Conventional pull request (PR)-based methods (Jimenez et al., 2024; Pan et al., 2025; Jain et al., 2025b) are ineffective in capturing complete feature patches, as these often span multiple PRs scattered across the timeline, making them difficult to associate. Moreover, many PRs lack tagging, hindering the reliable identification of feature contributions. Notably, PR-driven methods are inherently tied to the historical trajectory of commit submissions, limiting the tasks to fixed development combinations.

Motivated by these shortcomings, we introduce ACE-Bench , a challenging benchmark that targets feature-oriented agentic coding scenarios. It integrates an execution-based evaluation pipeline and a test-driven toolkit for automatically collecting instances from Python repositories. As shown in Table 1, our bench provides the following characteristics:

1. **Feature-oriented real-world software development.** Unlike SWE-bench, which is dominated by bug-fixing issues with only about 18–22% of its instances corresponding to feature requests, our benchmark is explicitly designed to target systematic feature-level agentic coding. As shown in Figure 1, given human-like clear requirements (*e.g.*, interface signatures and high-level functional descriptions), our task entails the implementation of new capabilities either within an existing codebase or as standalone modules. For example, adapting the Transformers library (Wolf et al., 2020) for compatibility with Qwen3 (Yang et al., 2025a) or engineering FlashAttention (Dao et al., 2022) from scratch.
2. **Reliable execution-based evaluation.** Highly ambiguous requirements without explicit function signatures often introduce multiple valid implementations that are incompatible with the interface expected by unit tests. This misalignment complicates execution-based evaluation and typically necessitates additional manual inspection or LLM-based judgement (Starace et al., 2025; Seo et al., 2025). To mitigate this issue, we adopt a test-driven formulation strategy when constructing requirements. Each prompt explicitly specifies the clear interface definitions, import paths, and the descriptions of expected behaviors, and enforces that the solution must be directly callable,

as illustrated in Figure 1. This method guarantees that a correct implementation will pass all associated tests, thereby enabling automated execution-based evaluation.

3. **Scalable instance collection toolkit.** To support the extensible creation of feature-oriented, realistic evaluation environments with fail-to-pass (F2P) and pass-to-pass (P2P) tests, as introduced in SWE-bench, we develop an automated generation pipeline driven by unit tests. The pipeline begins by selecting and executing F2P and P2P tests, followed by the construction of a dependency graph through dynamic tracing. Based on the traced dependencies, the system automatically extracts the implementation of the targeted features while ensuring the integrity of other features. The final problem statements are then synthesized. This approach enables us to generate naturally verifiable environments from any Python repository in a scalable and flexible manner, free from the constraints of the availability and predefined trajectory of human-written PRs or commits.
4. **Continually updatable.** Building on our collection toolkit, ACE-Bench supports a continual supply of new task instances, enabling evaluation on tasks created after their training date, thus mitigating the risk of contamination. Using this pipeline, we have curated a benchmark with 212 evaluation instances and 889 verifiable environments, created from May 2022 to September 2025, sourced from 16 real-world GitHub repositories in the first version of our benchmark.

We evaluate multiple state-of-the-art LMs on ACE-Bench and find that they fail to solve all except the simplest tasks. Using the Openhands agent framework (Wang et al.), Claude 4 successfully completes 7.5% of the task cases. Furthermore, we carried out comprehensive experiments, offering insights into potential improvement directions on our benchmark.

In a nutshell, our contributions are three-fold: 1) We introduce ACE-Bench, a benchmark for agentic coding that evaluates LLMs on solving feature-level, real-world complex tasks through an automated, execution-based evaluation pipeline. 2) We release a scalable, test-driven toolkit for instance collection that integrates seamlessly with our benchmark and automatically generates verifiable environments from Python repositories. Using this toolkit, we construct a benchmark comprising 212 evaluation tasks and 889 executable environments from 16 open source GitHub repositories. 3) We benchmark state-of-the-art LLMs, including both open- and closed-source variants, and perform in-depth analysis to identify and highlight remaining challenges.

2 Related Work

Agentic Coding Benchmarks. The most widely adopted benchmark for agentic coding is SWE-bench (Jimenez et al., 2024), whose verified subset has emerged as a standard for assessing LLMs. Although originally highly challenging, its success rate has increased from below 10% to over 70% within a year, reflecting rapid advances in LLM-based agents (Anthropic, 2025a; Yang et al., 2025a). Despite its importance, SWE-bench has notable drawbacks. It mainly focuses on bug fixing, with comparatively limited coverage of feature development tasks, which often span multiple PRs. Other benchmarks address narrower domains or predefined workflows. PaperBench (Starace et al., 2025) and MLE-Bench (Chan et al., 2025) focus on machine learning problems but rely on expert curation or high-quality cases from Kaggle. GitTaskBench (Ni et al., 2025) broadens task coverage but offers only 54 expert-designed tasks, while DevEval (Li et al., 2025) spans the development lifecycle but enforces fixed workflows with 22 handcrafted tasks. To tackle the above problems, we propose a challenging benchmark specifically designed for feature-oriented agentic coding scenarios. This benchmark integrates an execution-based evaluation pipeline and an automated toolkit that collects instances from Python repositories in a scalable manner.

Scalable Collection Pipeline. A verifiable environment is crucial for achieving better agentic coding. SWE-Gym (Pan et al., 2025) follows the pull-request based approach of SWE-bench, whereas R2E-Gym (Jain et al., 2025b) derives tasks from commits by synthesizing tests and back-translating code changes into problem statements with LLMs. These approaches mitigate scalability concerns but provide limited guarantees of evaluation quality. SWE-Smith (Yang et al., 2025b) synthesizes tasks from repositories using heuristics such as LLM generation, procedural modifications, or pull-request inversion. SWE-Flow (Zhang et al., 2025) synthesizes data based on fail-to-pass tests but neglects pass-to-pass tests and does not ensure the proper functioning of other features in undeveloped codebases, resulting in discrepancies compared to actual development settings. Although successful, none of them can generate tasks that are both feature-oriented and reflective of real-world development scenarios. Our benchmark addresses these gaps by providing a test-driven, scalable

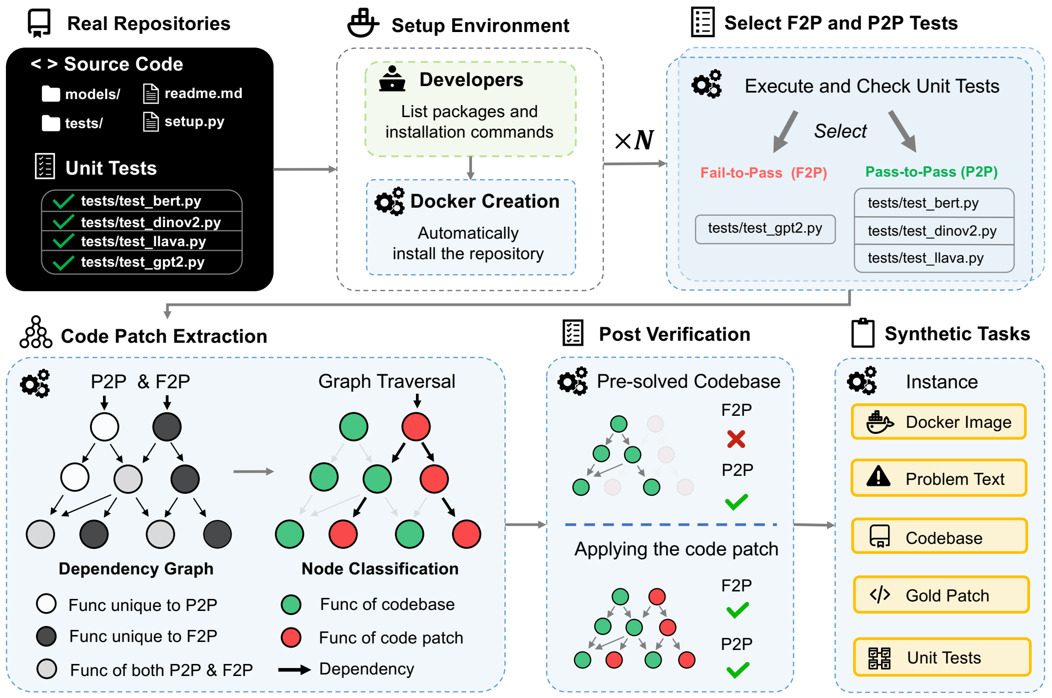


Figure 2: Given a GitHub repository, our automated toolkit initializes the development environment via Docker. For each benchmark instance, it validates and selects *fail-to-pass* and *pass-to-pass* tests. Then, the system performs dynamic tracing to capture runtime behavior and construct an object dependency graph. Leveraging this graph, the toolkit synthesizes code patches, derives corresponding pre-solved codebases, and formulates final problem statements. This pipeline has yielded 212 benchmark tasks and 889 executable environments from 16 GitHub repositories.

tool for generating feature-level agentic coding tasks, complemented by a rigorous post-verification that ensures the integrity of undeveloped codebases, consistent with real-world scenarios.

3 ACE-Bench

ACE-Bench establishes a benchmark for evaluating the capabilities of code agents in end-to-end software development tasks. The benchmark requires agents to interpret high-level goals and their associated code interfaces, autonomously manage execution environments, and synthesize correct and callable implementations either within existing codebases or as standalone solutions. Constructed with minimal human intervention, the benchmark leverages an automated pipeline that derives feature-oriented coding tasks from open-source repositories, thereby extending the scope of agentic coding beyond bug fixing to encompass feature development.

3.1 Feature-oriented Agentic Coding

Task Formulation. As illustrated in Figure 1, each instance in ACE-Bench provides the agent with a comprehensive problem statement. This includes a high-level task description, a specified functional interface, a blacklist of prohibited URLs to mitigate potential cheating of agents, and a dockerfile defining the execution environment. The agent is then tasked with generating a solution that addresses the problem, whether by editing existing code or implementing from scratch. Notably, to facilitate automated and unambiguous evaluation, the agent's output is required to be a directly callable module. Its invocation path, function signature, including input and output variables as well as comprehensive annotations, are all explicitly provided within the problem statements.

Difficulty. In realistic settings, software development may proceed either by extending an existing codebase or by implementing a feature entirely from scratch. ACE-Bench reflects these two scenarios with two difficulty levels. Level 1 (L_1) consists of incremental development within an existing repository based on task requirements, while Level 2 (L_2) requires constructing the same functionality from scratch. To enable controlled comparison, L_1 and L_2 tasks are paired in a one-to-one fashion, resulting in 106 tasks per level and 212 (106$\times$2) tasks overall.

Metric Design. Our evaluation protocol follows the established setup of SWE-bench (Jimenez et al., 2024), where each agent-generated solution is validated by executing its associated *fail-to-pass* (F2P) and *pass-to-pass* (P2P) tests. A task is considered resolved when the proposed solution successfully passes all these tests. We report three primary metrics: (1) *Resolved Rate*, the proportion of tasks fully solved, like SWE-bench; (2) *Passed Rate*, the average fraction of fail-to-pass tests passed per task, serving as a soft indicator of partial correctness; (3) *Token IO*, the average number of input and output tokens consumed, reflecting the computational efficiency of the agent.

3.2 Benchmark Collection

Execution Environment Configuration. To rapidly set up an environment for a given repository, we manually specify installation commands (taking approximately three minutes), rather than relying on the more error-prone and uncontrollable approach of having the agent search for installation methods itself. Automated scripts are then used to configure the environment and package the repository into a Docker image. The benchmark includes 16 widely downloaded PyPI packages across various domains, such as visualization libraries and LLM infrastructure. Notably, human intervention is required only for this step of the pipeline, and the total human labor required to complete this for all 16 repositories amounts to less than one hour.

Constructing *Fail-to-pass* and *Pass-to-pass* Tests. We construct benchmark instances by identifying candidate test files in the repository using pytest's collection function (Krekel et al., 2004), followed by validation through execution. For each instance, n validated test files are designated as *fail-to-pass* (F2P) tests, as introduced in SWE-bench. These tests fail in the undeveloped repository but succeed once the agent correctly implements the target functionality. To additionally assess incremental development capability, we include m randomly sampled validated files as *pass-to-pass* (P2P) tests, which are expected to pass both before and after the agent's solution. Since a single test file typically corresponds to one functional implementation, n is usually set to one in our setting.

Test-Driven Code Patch Extraction. Obtaining the pre-solved codebase together with the corresponding code patch requires isolating the functionality linked to the F2P tests. However, the inherent ambiguity of functional boundaries in real-world codebases poses a significant challenge. Naively extracting relevant code fragments risks inadvertently disrupting other well-established features. As depicted in Figure 2, our approach mitigates this issue by incorporating P2P tests to accurately identify code modules required by other functions or those serving as foundational components of the repository. The detailed implementation is as follows:

- *Construct the object dependency graph.* We initiate the process by executing the available F2P and P2P test cases for a given benchmark instance. During runtime, we employ Python's built-in tracing facility to capture function call events and their dependencies. From this trace, we construct an object dependency graph in which each node represents a function and is enriched with metadata, including a unique identifier, source location, a list of dependent functions, and a binary flag indicating if the function was triggered during P2P tests.
- *Graph traversal and node classification.* To distinguish functional components, a large language model analyzes the F2P test files and separates the imported functions related to the target feature from those that serve supporting roles in the testing process. The nodes identified as central to the undeveloped feature serve as the initial entry points for a breadth-first traversal of the graph. During this traversal, nodes are systematically classified: those encountered in P2P executions are designated as *remained*, while nodes not observed in P2P runs are classified as *extracted*.
- *Extracting the code.* The traversal process yields a subset of graph nodes identified as relevant to the intended functionality. In the final stage, the corresponding segments of source code are extracted from the original codebase. This operation produces a modified codebase devoid of the target functionality and a complementary code snippet that realizes the previously absent feature.

Post Verification. To ensure the successful extraction of the target functionality from the codebase without affecting other components, we implement a rigorous verification process. The first step involves validating the pre-modified codebase by ensuring that it passes all P2P tests, thereby

Model	Lite			Full		
	% Passed	% Resolved	# Token I/O	% Passed	% Resolved	# Token I/O
Gemini 2.5 Pro	17.0	3.3	0.7M / 16k	13.2	2.4	0.8M / 17k
OpenAI o3	23.2	3.3	2.0M / 36k	22.4	1.9	2.0M / 34k
Qwen3-Coder-480B-A35B-Instruct	25.6	0.0	1.9M / 18k	25.4	2.4	2.4M / 24k
GPT-5	29.5	6.7	1.5M / 31k	36.4	7.1	2.1M / 34k
Claude Sonnet 4	37.0	6.7	1.8M / 31k	38.2	7.5	1.4M / 33k

Table 2: The performance of various frontier large models combined with the OpenHands framework in the Lite and Full evaluation sets of our benchmark.

confirming its integrity. Simultaneously, it must fail all F2P tests, demonstrating that the target functionality has been effectively removed. Following this, we assess the accessibility of all utility functions required for the F2P tests in the modified codebase. This step ensures that the changes made are confined to the target functionality and do not inadvertently impact other core dependencies. Finally, reapplying the patch to the undeveloped codebase should allow all tests to pass, confirming the patch's correctness.

Problem Statement Generation. By leveraging the extracted code snippet, the pre-modified codebase, and the corresponding unit tests, we automatically generate the problem statement for each instance. This procedure includes the derivation of the feature signatures, which encompass the types of input and output variables, alongside the functional description as inferred from the code comments. In the absence of such comments, we employ a large language model to generate them directly from the code snippet. Further details can be found in the appendix.

To this end, our pipeline automatically generates the core components of each instance: a natural language problem statement, an undeveloped codebase, a verified code patch, and a suite of unit tests corresponding to required features. The sole manual intervention required is the specification of the repository's installation procedure, a process that takes approximately three minutes per repository.

3.3 Benchmark Configuration

Full Set. Leveraging our pipelines, we configured the number of P2P test files to five and curated 889 coding environments derived from 16 Python repositories. To ensure the benchmark meaningfully challenges best-performing agents, we restricted inclusion to tasks exceeding 100 lines of pending implementation, encompassing at least 10 F2P test points, with test files initially committed after May 2022. This filtering yielded 212 high-quality instances comprising the full set.

Lite Set. Evaluating LMs on our bench can be time-consuming and, depending on the model, require a costly amount of compute or API credits, as illustrated in Table 2, where the average number of input tokens approaches the million-token mark. To facilitate wider adoption of ACE-Bench , we randomly selected 30 instances from the full set to create a streamlined lite set.

4 Experiments

4.1 Performance on ACE-Bench

4.1.1 Baseline

To establish strong baselines, we adopt the OpenHands (Wang et al.) framework for software development agents, which tops the SWE-bench. In the experiments, the maximum of steps per task is set as 100 by default. Internet access is freely available, while no specific browser-use tools are provided. To ensure the integrity of our evaluation, robust anti-cheating mechanisms are incorporated to prevent agents from assessing the ground-truth repositories (see the appendix for details).

Five frontier LLMs are evaluated, including GPT-5 (OpenAI, 2025a), OpenAI o3 (OpenAI, 2025b), Qwen3-Coder-480B-A35B-Instruct (Team, 2025), Gemini 2.5 Pro (Comanici et al., 2025) and Claude Sonnet 4 (Anthropic, 2025b). The results are presented in Table 2. As can be seen, even the most capable models, *i.e.*, Claude 4 Sonnet and GPT-5, resolve only 7.5% and 7.1% of the tasks on the *Full* set, respectively. This underscores the highly challenging nature of the feature-oriented

		SWE-bench	Ours
Problem Texts	Length (Words)	195.1	4818.0
Gold Solution	# Lines	32.8	1012.0
	# Files	1.7	3.5
	# Functions	3	15.0
Tests	# Fail to pass (test points)	9.1	133.2
	# Total (test points)	120.8	447.2

Table 3: Average numbers characterizing different attributes of a SWE-bench task instance, as well as our ACE-Bench (L_1 set).

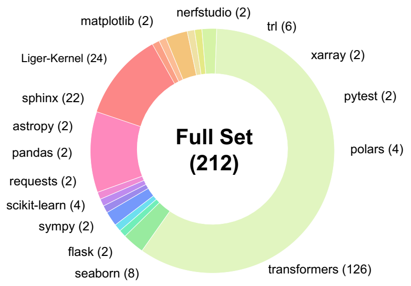


Figure 3: Distribution of our benchmark across 16 GitHub repositories.

Model	SWE-bench Verified		ACE-Bench subset		
	% Resolved		% Passed	% Resolved	# Token I/O
	mini-SWE-agent	OpenHands	OpenHands		
Gemini 2.5 Pro	53.60	-	22.8	0.0	0.8M / 12k
Qwen3-Coder-480B-A35B-Instruct	55.40	69.60	27.2	0.0	1.5M / 14k
OpenAI o3	58.40	-	27.0	0.0	1.7M / 30k
GPT-5	65.00	-	40.1	2.1	1.8M / 33k
Claude Sonnet 4	64.93	70.40	42.6	4.1	1.1M / 24k

Table 4: Compare the performance of the frontier agents on SWE-bench and our ACE-Bench, using a subset of our benchmark with repositories shared with SWE-bench for a fair comparison.

development tasks in our ACE-Bench, which require agents to write thousands of lines of code and pass comprehensive test suites.

For a more nuanced evaluation, we further analyze passed rates and token consumption by different LLMs. The passed rates, while remaining at a low level of below 40%, are much higher than the resolved rates. This discrepancy indicates that current agents often produce seemingly plausible solutions with a large underlying gap from truly solving the problem, which accounts for the common need of tedious debugging for AI-generated code. Regarding token consumption, nearly all LLMs consume over one million input tokens. Given the low resolved rates, this reflects the extremely low efficiency of existing agents in tackling real-world development tasks, which is thus an important topic for future research. In addition, a high consistency is observed in the rankings of different LLMs across the *Lite* and *Full* sets in terms of both pass and resolved rates, demonstrating the representativeness of the Lite set.

4.1.2 Comparison with SWE-bench

Compared with the SWE-bench (Jimenez et al., 2024), our ACE-Bench introduces a more challenging suite of development tasks. It encompasses six additional popular repositories apart from ten repositories originally covered by the SWE-bench, the full list of which is shown in Figure 3. Table 3 presents comparative statistics illustrating the task difficulties across the two benchmarks. Specifically, the tasks in our benchmark exhibit a substantial increase of complexity in terms of the length of problem texts, number of lines, files and functions to be edited as well as the number of tests to pass. These enhancements necessiate agents with strong long-context understanding and management capabilities alongside comprehensive problem analysis to handle diverse test cases.

For a more grounded analysis, we further compare the performance of agents on the SWE-bench and our ACE-Bench . To draw a more aligned comparison, we construct a subset of our benchmark including only repositories shared with SWE-bench. The results in Table 4 reveals a stark performance gap between the two benchmarks in terms of resolved rate. Specifically, the most capable Claude Sonnet 4 only resolves 4.1% of the tasks in our ACE-Bench subset in contrast to the 70.40% on the SWE-bench. This indicates the highly challenging nature of our benchmark, which provides considerable room for future improvement and establishes a rigorous testbed to measure the upper bound of existing agents.

	Models	% Resolved	% Passed
Original	Gemini 2.5 Pro	3.3	17.0
	GPT-5	6.7	29.5
	Claude 4 Sonnet	6.7	37.0
Verified	Gemini 2.5 Pro	3.3	15.4
	GPT-5	6.7	27.4
	Claude 4 Sonnet	3.3	32.5

Table 5: An ablation study to evaluate the necessity of manual verification for the examples generated by our system.

Models	Steps	% Resolved	% Passed
GPT-5	50	3.3	22.5
	100	6.7	29.5
	150	3.3	30.2
Claude 4 Sonnet	50	0.0	28.2
	100	6.7	37.0
	150	3.3	34.3

Table 6: An ablation study on the max execution steps of OpenHands with GPT-5 and Claude 4 in Lite Set.

Model	Without Interface			Visible Unit Tests		
	% Resolved	% Passed	# Token I/O	% Resolved	% Passed	# Token I/O
Gemini 2.5 Pro	0.0 (-3.3)	12.0 (-5.0)	0.6M / 21k	3.3 (+0.0)	18.4 (+1.4)	1.2M / 19k
OpenAI o3	3.3 (-0.0)	17.7 (-5.5)	1.3M / 26k	10.0 (+6.7)	37.7 (+14.5)	2.8M / 101k
GPT-5	3.3 (-3.4)	16.6 (-12.9)	2.3M / 31k	26.7 (+20.0)	56.1 (+26.6)	2.5M / 66k
Claude Sonnet 4	3.3 (-3.4)	16.9 (-20.1)	1.2M / 30k	26.7 (+20.0)	60.5 (+23.5)	2.2M / 32k

Table 7: Performance comparison of lite set with visible unit tests and without interface.

4.1.3 Failure Cases Analysis

We conduct a failure case analysis based on the results in our full set from the best-performing Claude 4 Sonnet model, leading to the following findings.

Appropriate Information in ACE-Bench. Figure 4 presents the distribution of prevalent error categories, with `AssertionError` emerging as the most frequent. This predominance indicates that many LLM-generated solutions are able to execute up to the assertion checkpoints without encountering prior runtime failures. This result underscores that ACE-Bench can effectively provide the LLMs with appropriate information to generate complete programs.

Limitations in Code Reasoning. A considerable number of `TypeError` instances can be observed in Figure 4, indicating that current LLMs struggle to accurately infer variable types and function interfaces within dynamically typed high-level programming languages (such as Python). It essentially reveals the current limitation of LLMs in performing complex reasoning.

The "Idle Habits" of LLMs. We also find that current LLMs exhibit a tendency toward "laziness". For example, they often resort to guessing (even hallucinating) the interface of a function defined across files, rather than performing the actual file reading required to retrieve the precise function prototype. This behavior also leads to a considerable number of `TypeError` occurrences. Notably, the same tendency appears in both Claude Code and Mini-SWE-Agent, indicating that this issue is not specific to a single scaffold.

4.2 Ablation Study

4.2.1 Analyzing the quality and necessity of our benchmark design.

Without Interface. We performed an ablation study to assess the role of explicit interface specification in agent performance. For controlled comparison, we employed the lite set, systematically removing function signatures and call path annotations from the prompts. As shown in Table 7, this removal leads to a marked decline in task success rates. The results confirm that clearly defined interfaces are critical for enabling effective reasoning and program synthesis by LLM-based agents.

Sample Quality. Our automated data generation pipeline yields high-quality, evaluation-ready samples with minimal human intervention, supported by a rigorous post-verification process. To assess the fidelity of these samples, we conducted an ablation study in which a senior engineer with five years of industry experience in AI infrastructure and system architecture independently revised the prompts in the lite set. The verification details are provided in Appendix Figures 20 and 21. As shown in Table 5, model performance on the manually revised subset is highly consistent with the original dataset. These results affirm the reliability and robustness of our automated data pipeline.

Difficulty	Models	% Resolved	% Passed
L_1	Gemini 2.5 Pro	3.8	12.3
	OpenAI o3	2.8	24.5
	Qwen3-Coder-480B	3.8	29.6
	GPT-5	13.2	48.8
	Claude 4 Sonnet	14.0	57.6
L_2	Gemini 2.5 Pro	0.9	14.0
	OpenAI o3	0.9	20.4
	Qwen3-Coder-480B	0.9	21.3
	GPT-5	0.9	24.1
	Claude 4 Sonnet	0.9	18.7

Table 8: Performance comparison of tasks with different difficulty levels in ACE-Bench.

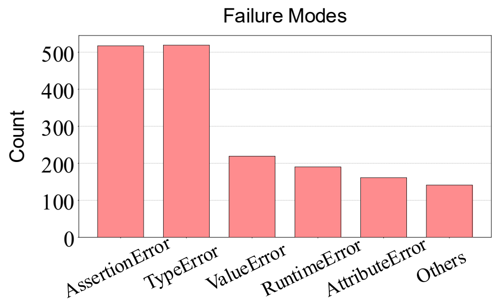


Figure 4: Failure modes of the best-performing Claude 4 Sonnet.

Lines of Code and Task Initial Commit Date. Figure 5 explores the relationship between task pass rates, initial commit timestamps, and the number of lines of code required for task completion. We observe a clear negative correlation between pass rate and code length, indicating that tasks involving more lines of code are inherently more challenging for current large models. In contrast, task performance shows minimal dependence on commit time, likely because the task set remains largely unexplored by existing models. To further understand why commit time has little influence, we analyze how feature complexity evolves over time. Specifically, the lower panel of Figure 5 plots the normalized trends of code length and pass rate across commit periods. The two normalized curves exhibit highly similar fluctuations, reinforcing that variation in task performance is driven far more by feature complexity than by commit time. However, as agentic systems increasingly participate in feature development workflows, the risk of data leakage may become more pronounced and should be monitored in future benchmark design.

Comparison between L_1 and L_2 Subset. Comparison between L_1 and L_2 Subsets. Our benchmark defines two evaluation settings: L_1, where new functionalities are incrementally added to an existing codebase, and L_2, where functionalities are implemented entirely from scratch. All conditions are held constant across both settings, except for the presence or absence of initial code context.

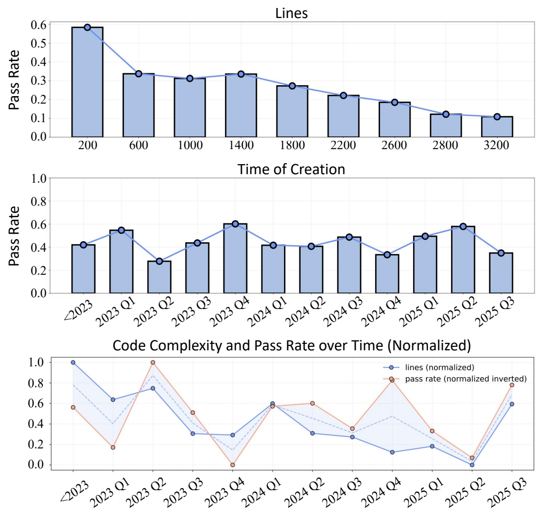


Figure 5: The pass rate of Claude 4 Sonnet in our benchmark varies with the number of code lines and task creation time.

This distinction leads to notably different levels of reasoning complexity. In the L_1 setting, the agent still has access to most of the original codebase except for the functions and classes removed along the traced execution path. This partial repository shows how the feature fits into the surrounding code and gives the agent contextual clues about expected behavior. As a result, L_1 tasks are more guided, since only the missing implementations need to be completed. In contrast, L_2 tasks remove all surrounding code. The agent does not see any part of the original repository and must rely only on the interface to implement the required functionality. Without the structure provided by the existing codebase, the agent has to reconstruct the full logic and organization of the feature entirely from scratch, which makes L_2 substantially more difficult. As shown in Table 8, the from-scratch (L_2) setting presents a substantially greater challenge. Current large language models exhibit near-zero success rates under this condition, indicating a severe limitation in their ability to perform end-to-end program synthesis without structural guidance.

Accuracy of LLM-based Top Import Classification.

To validate the reliability of our LLM-based classifier for identifying top-level tested objects in test file, we conducted a quantitative evaluation against expert annotations. Domain experts evaluated

Metric	Precision	Recall	F1 Score	Accuracy
Value	81.03%	89.24%	84.94%	91.74%

Table 9: Performance of the LLM classifier for identifying top-level tested objects.

Scaffold	% Resolved	% Passed
Claude Code + Claude 4 Sonnet	6.7	40.5
OpenHands + Claude 4 Sonnet	6.7	37.0
Mini-SWE-Agent + Claude 4 Sonnet	0.0	28.2

Table 10: Performance comparison of different agent scaffolds using Claude 4 Sonnet on the Lite subset.

all 605 import statements in the Lite Set and identified 158 of them as top-level tested objects. The details of the procedure are provided in Appendix Figure 19. Table 9 reports the performance of the LLM classifier. These results indicate that LLMs can accurately identify tested objects at scale, supporting the use of LLM-based classification in our data construction pipeline.

4.2.2 Analyzing the key factors in building end-to-end CodeAgents

Visible Unit Tests. We conducted an ablation study to assess the impact of providing accurate unit tests on agent performance in complex coding tasks. In this setting, the agent was given access to ground-truth unit tests alongside the Lite set. As shown in Table 7, both task success rates and pass rates increased significantly. These findings underscore the importance of high-quality unit test generation as a key factor in enabling robust agentic coding.

Longer Execution Steps. Table 6 reports the effect of increasing the maximum number of execution steps on model performance. Increasing the maximum step size from 50 to 100 results in notable performance gains for both GPT-5 and Claude 4 Sonnet. However, beyond this threshold, the improvements plateau, and in some cases, excessively long execution trajectories can lead to performance degradation by reversing previously correct outputs.

Evaluation Across Agent Frameworks. We evaluated three scaffolds on the Lite subset using Claude 4 Sonnet. As shown in Table 10, Claude Code and OpenHands achieved identical resolution rates (6.7%), while Mini-SWE-Agent failed to resolve any instance (0.0%). For pass rates, Claude Code reached 40.5%, followed by OpenHands at 37.0% and Mini-SWE-Agent at 28.2%. These differences show that scaffold design strongly influences agent effectiveness. However, all three scaffolds exhibit similar failure modes, suggesting that the core difficulty of ACE-Bench comes from the tasks themselves rather than any specific framework.

5 Conclusion

In this work, we introduce ACE-Bench , a novel benchmark designed to evaluate the capabilities of LLM-powered agents in realistic, feature-oriented software development scenarios. Leveraging test-driven task extraction and execution-based evaluation, ACE-Bench overcomes key limitations of existing benchmarks by enabling greater task diversity, scalability, and verifiability. Empirical results reveal that current agentic systems face persistent challenges in planning, reasoning, and managing long-horizon tasks. With its extensible and automated design, ACE-Bench offers not only a rigorous evaluation framework but also a foundation for the development of next-generation agentic coding models.

References

Anthropic. Claude code: Deep coding at terminal velocity. `https://www.anthropic.com/claude-code`, 2025a.

Anthropic. Claude 4 sonnet, 2025b. URL `https://www.anthropic.com/news/claude-4`.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, 2025.

Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Aoyan Li, Bo Li, et al. Fullstack bench: Evaluating llms as full stack coders. *arXiv preprint arXiv:2412.00535*, 2024.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, 2022.

Yaxin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. Swe-dev: Evaluating and training autonomous feature-driven software development. *arXiv preprint arXiv:2505.16975*, 2025.

Jingzhi Gong, Vardan Voskanyan, Paul Brookes, Fan Wu, Wei Jie, Jie Xu, Rafail Giavrimis, Mike Basios, Leslie Kanthan, and Zheng Wang. Language models for code optimization: Survey, challenges and future directions. *arXiv preprint arXiv:2501.01277*, 2025.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025a.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *Conference on Language Modeling*, 2025b.

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 *International Conference on Learning Representations*, 2024.

Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. pytest x.y, 2004. URL `https://github.com/pytest-dev/pytest`. Contributors: Holger Krekel and Bruno Oliveira and Ronny Pfannschmidt and Floris Bruynooghe and Brianna Laugher and Florian Bruhin and others.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al. Prompting large language models to tackle the full software development lifecycle: A case study. In *International Conference on Computational Linguistics*, 2025.

Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Yuntao Du, Bill Sun, Hongzhang Liu, et al. GitTaskBench: A benchmark for code agents solving real-world tasks through code repository leveraging. *arXiv preprint arXiv:2508.18993*, 2025.

OpenAI. Gpt-5 system card, 2025a. URL `https://cdn.openai.com/gpt-5-system-card.pdf`.

OpenAI. Openai o3, 2025b. URL `https://openai.com/index/introducing-o3-and-o4-mini`.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *International Conference on Machine Learning*, 2025.

Qwen. Qwen code: Research-purpose cli tool for qwen-coder models. `https://qwenlm.github.io/blog/qwen3-coder/`, 2025.

Ranjan Sapkota, Konstantinos I Roumeliotis, and Manoj Karkee. Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic ai. *arXiv preprint arXiv:2505.19443*, 2025.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2code: Automating code generation from scientific papers in machine learning. *arXiv preprint arXiv:2504.17192*, 2025.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. Paperbench: Evaluating AI's ability to replicate AI research. In *International Conference on Machine Learning*, 2025. URL `https://openreview.net/forum?id=xF5PuTLPbn`.

Qwen Team. Qwen3 technical report, 2025. URL `https://arxiv.org/abs/2505.09388`.

Huanting Wang, Jingzhi Gong, Huawei Zhang, and Zheng Wang. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*, 2025.

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. URL `https://arxiv.org/abs/2407.16741`.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-Art Natural Language Processing. Association for Computational Linguistics, 2020. URL `https://www.aclweb.org/anthology/2020.emnlp-demos.6`.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025b.

Lei Zhang, Jiaxi Yang, Min Yang, Jian Yang, Mouxiang Chen, Jiajun Zhang, Zeyu Cui, Binyuan Hui, and Junyang Lin. Synthesizing software engineering data in a test-driven manner. In *International Conference on Machine Learning*, 2025.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*, 2024.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, 2025.

APPENDIX

A DETAILED BENCHMARK COLLECTION

This section complements the details of benchmark construction (Sec. 3.2), which contains detailed recipes of the data collection, patch extraction, and prompt design, along with a fuller characterization of the task instances.

A.1 DATA COLLECTION PIPELINE

Environment Setup.

For each selected repository, we manually prepare an environment configuration file (see Figure 6 for an example). Empirical observations indicate this procedure can be accomplished within three minutes. Upon completion of environment configuration, our pipeline constructs a Docker image, with all subsequent operations executed within this sandboxed environment. This is the sole stage requiring human intervention. All succeeding stages operate under full automation.

Patch Extraction. The patch extraction process consists of four main steps.

Patch Extraction Step 1: Dependency Graph Construction. This procedure generates function-level dependency graphs for all test files within the code repository, establishing the foundation for subsequent patch extraction operations. We leverage pytest's intrinsic test case collection mechanism to aggregate all viable test cases at the file granularity, where each file contains a potential test case. For each test case, we execute the test within the sandbox environment, selecting test cases that achieve complete success as *fail-to-pass* (F2P) instances. Concurrent with test execution, we construct function-level dependency graphs for each F2P instance utilizing a dynamic tracing library.

Patch Extraction Step 2: LLM Classification. For each F2P test file, we employ an LLM to differentiate between imported objects serving as test targets versus those functioning as test dependencies and general utilities. We provide the LLM with the test file's name and content as classification references. Our prompt template for the LLM to classify is illustrated in Figure 10. Objects classified through this methodology are designated as top-level objects, representing directly imported interfaces by the test file.

Patch Extraction Step 3: Pass-to-pass (P2P) Selection. For each F2P instance, we select multiple *pass-to-pass* cases. These P2P cases are executed after coding agents finishing implementations to ensure existing functionalities remain normal. Since the aforementioned top-level objects of F2P cases will be removed from codebases, here the pass-to-pass cases should not share top-level objects with the F2P cases. For this reason, if we find only a few P2P cases have different top-level objects from F2P cases, it may indicate erroneous classification of general utilities as top-level objects by the LLM. In this circumstance, we will reconsider the top-level objects according to their invocation frequency.

Patch Extraction Step 4: Final Extraction. For each F2P case, we utilize top-level objects as entry points and execute BFS according to the constructed dependency graph. Node objects belonging to P2P are designated as *remained*, while others are marked as *extracted*. Nodes marked as *extracted* are added to the BFS queue for continued traversal. BFS termination occurs upon queue finish or when extracted code lines reach our predetermined maximum value, randomly selected between 1000 and 10000 lines per case. Finally, we remove objects marked as *extracted* from the codebase, yielding a complete codebase with F2P functionality eliminated.

Post-verification. For each codebase after code patch extraction, we conduct post-verification to ensure the modified codebase has normal functionality. Specifically, we execute F2P within the modified codebase, expecting pass rates below a predetermined parameter. Then we further execute all selected P2P cases, ensuring complete test passage.

Environment Configuration File

```
SPECS_LITGPT = {
    "base_image": "python310",
    "repository": "Lightning-AI/litgpt",
    "commit": None,
    "start_time": "2024-01-01",
    "test_discovery_cmd": TEST_DISCOVERY_DEFAULT,
    "test_dynamic_trace_cmd": TEST_DYNAMIC_TRACE_DEFAULT,
    "test_cmd": TEST_PYTEST,
    "custom_instance_image_build": [
        "ENV HF_HOME=/root/my_cache/huggingface",
        "ENV HF_HUB_CACHE=/root/my_cache/huggingface/hub",
    ],
    "pip_packages": [
        "setuptools",
        "tblib>=3.0.0",
    ],
    "pre_install": [
          # Commands to execute before installing the project
    ],
    "install": "pip install 'litgpt[extra]'",
    "docker_specs": {
        "run_args": {
            "cuda_visible_devices": "0,1,2",
            "cap_add": []
        },
        "custom_docker_args": [
            "-v /data2/cache_pb_datapipeline:/root/my_cache/",
        ]
    },
    "exclude_keywords": [
          # Exclude test files containing these keywords
    ],
    "library_name": "litgpt",
    "black_links": [
        "https://github.com/Lightning-AI/litgpt/"
    ],
    "technical_docs": [
    ]
}
```

Figure 6: Environment Configuration File

A.2 DATA FORMAT AND PROMPT DESIGN

In this section, we present the essential components included in a qualified example (covering both L_1 and L_2 tasks), illustrating the test case format of our benchmark, organization of our prompts, and the effectiveness of using an LLM to supplement missing docstring entries.

Directory Structure of Generated Instances. Each successfully generated instance includes a directory structure containing two main folders: `task` and `test`, as well as a `config.yaml` file used for prompt generation. The `task` folder contains data that is fully accessible to the agent to assist with task completion, whereas the `test` folder contains testing information and debugging details that are hidden from the agent. The contents of these `task` folders vary between L_1 and L_2. For a L_1 case, the task folder contains the codebase of the repository after applying patch extraction. This gives the agent the necessary context to work with existing code and complete the task. On the other hand, a L_2 case is designed to be more challenging—the task folder does not include any codebase. That is, the agent is required to implement the solution from scratch, without relying on any existing code. An example of the directory structure for a L_1 case is provided in Figure 7.

Level 1 directory structure

```
/test_name
└── level1
    ├── task/
    │   ├── Liger-Kernel /        # modified code base
    │   ├── technical_docs/
    │   ├── black_links.txt       # black links agent can't access
    │   └── prompt.md             # task statement
    ├── test/
    │   ├── eval_code.py          # script to structure test results
    │   ├── path2test.txt         # record the path of test files
    │   ├── f2p_file.py
    │   ├── p2p_file1.py
    │   └── ...
    └── config.yaml
```

Figure 7: L_1 directory structure. This figure illustrates the directory structure for a L_1 example, showing the `task` and `test` folders along with the `config.yaml` file used for prompt generation. The structure ensures that essential task-related data is accessible to the agent, while test and debugging details remain hidden.

Prompt Structure and Organization. For each successfully generated instance, we construct a tailored and detailed `prompt.md` file as input to the agent. The content of `prompt.md` consists of three components: *Task*, *Precautions*, and *Test and Interface Descriptions*. All prompts are generated automatically without manual labor, following a unified prompt template combined with an instance-specific configuration file via scripting. As shown in Figure 12, the ***Task*** section provides the agent with an overview of the task under the heading "Core Functionality." "Main Features and Requirements" describes the essential code features and requirements. The mandatory components that must be implemented are outlined under "Key Challenges." Finally, the "NOTE" subsection provides relevant notices. In the ***Precautions*** section (as shown in Figure 13), we define critical rules and boundaries, including clarity of the environment and dependency management, the strict prohibition of cheating (the "red line"), code delivery requirements, and the evaluation criteria. In the ***Test and Interface Description*** section (as shown in Figure 14), we offer detailed instructions on how to construct the code. This includes requirements for file locations and structure, suggestions for implementing interfaces, and specific objectives related to the current task.

Template of Prompt Config. We adopt a prompt config to automatically generate `prompt.md`. Figure 15 illustrates the template of this configuration file. A rule-based script automatically generates `prompt.md` according to this config file.

Prompt Design: L_1 vs. L_2. It is noteworthy that there are subtle differences between the L_1 and L_2 prompts. In the *Task* section of the prompt, for L_2 examples, we require the agent to independently implement the solution from scratch, without access to the repository's codebase. To enforce this constraint, the agent is instructed not to download the repository, and even in the event of its doing so, it is instructed not to install it. This is closely monitored during the testing phase, where we have set up mechanisms to check whether the agent engages in any unauthorized actions, such as

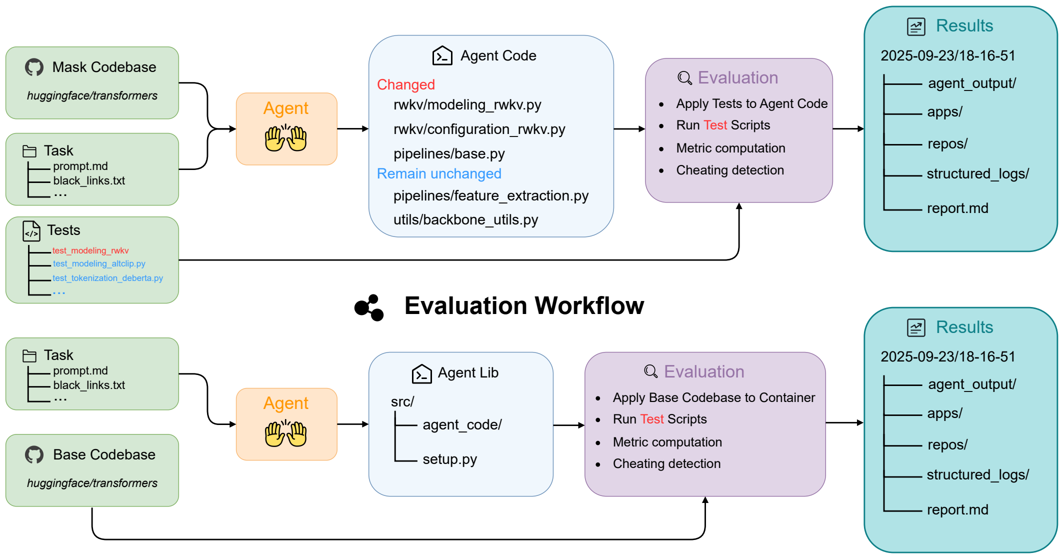


Figure 8: Evaluation workflow: L_1 (top, refining a masked codebase) and L_2 (bottom, implementing from scratch)

cheating. Figure 16 presents the prompt for the test-dpo-loss test in the Liger-Kernel library (L_1), while Figure 17 demonstrates the prompt for the test-simple-loss test in the same library (L_2).

Supplementation for Missing Docstrings. In cases where the source code lacks adequate documentation regarding function interfaces or behavior, we leverage a LLM to infer and complete the missing information. Figure 11 illustrates the prompt for docstring generation. The LLM-generated docstring is exemplified in Figure 18, where the LLM is used to supplement the missing docstring in the functional description.

B Detailed Benchmarking Process

Figure 8 illustrates the end-to-end benchmarking process of our system, designed to rigorously assess agent performance across two distinct task levels: L_1 (code refinement) and L_2 (from-scratch implementation). The pipeline is organized into four sequential stages: (1) task preprocessing, (2) agent execution, (3) automated testing, and (4) post-evaluation analysis, which includes metric computation and cheating detection. Each stage is carefully constructed to ensure fairness, reproducibility, and the prevention of information leakage. The following subsections provide a detailed explanation of the design and rationale behind each phase.

B.1 Preprocessing

The preprocessing procedure standardizes the input preparation for the two evaluation levels while ensuring the prevention of information leakage. For L_1, the process begins with a clean and complete codebase, where the target implementation files are substituted with masked versions. To further eliminate potential sources of inference, the corresponding *fail-to-pass* test is removed, ensuring that the agent cannot derive the expected behavior from the test code. In contrast, L_2 tasks are initialized solely with a task specification, requiring the agent to construct a complete solution from scratch without the support of an existing codebase. For both levels of evaluation, the agent's output must result in a fully functional codebase that satisfies task requirements, either by refining the masked codebase in L_1 or by generating a new implementation in L_2.

It is worth noting that in ablation studies involving white-box evaluation, the *fail-to-pass* test is retained to provide additional context for the agent. However, this variant is strictly excluded from the main evaluation to ensure an experimental setup free from information leakage.

B.2 EVALUATION

The evaluation process follows a unified framework for both L_1 and L_2 tasks, with customized configurations specific to each task.

L_1: After the agent completes its task, the *fail-to-pass* test is restored to its original location within the codebase. The updated codebase is then subjected to a comprehensive testing process using the `pytest` framework, and the resulting test outcomes are systematically recorded for analysis.

L_2: The evaluation environment is initialized with a completely unaltered codebase. Within this codebase, the *fail-to-pass* test file is modified to import the agent's implementations via the directive `from agent_code import`. The augmented codebase is subsequently evaluated using `pytest`, with all test results meticulously collected and documented.

The raw outputs from the `pytest` testing phase serve as the foundation for subsequent analyzes. The evaluation framework organizes all outputs into a comprehensive result package. This package includes the agent's generated codebase (`agent_output/`), the container's runtime workspace (`apps/`), detailed metric records (`repos/`), structured execution logs (`structured_logs/`), and a summary report (`report.md`).

B.3 METRICS

The evaluation process produces raw output from the `pytest` framework, which is subsequently processed to extract key statistics, including *total*, *passed*, *failed*, *skipped*, *error*, *xfail*, and *xpass*.

From these statistics, we derive two primary evaluation metrics: *pass_rate* and *is_solved*. The *pass_rate* is defined as the ratio of successfully passed tests to the total number of executed tests, excluding skipped tests. The binary metric *is_solved* indicates whether the task is fully solved. It is assigned a value of 1 if *pass_rate* equals 1 (i.e., all tests are passed), and 0 otherwise.

B.4 CHEATING PREVENTION AND DETECTION

To protect against potential cheating attempts by agents, such as using `pip install <package>` followed by inspecting the source code of the installed package, we implement a twofold defense mechanism. First, defensive prompting is incorporated into the task descriptions to discourage such behavior. Second, the evaluation framework conducts an automated inspection of the agent's execution logs after task completion to identify suspicious activities.

The log inspection process searches for regular expression patterns that indicate unauthorized attempts to access the source code of installed packages. If any pattern matches, the agent is flagged for potentially dishonest behavior in the evaluation report.

```
r'"message".*cat /usr/local/lib/python\d+\.\d+'
r'"command".*cat /usr/local/lib/python\d+\.\d+'
r'"message".*reading file: /usr/local/lib/python\d+\.\d+'
r'"message".*reading /usr/local/lib/python\d+\.\d+'
```

These patterns capture attempts to directly access files within the Python library directory, which is classified as a form of cheating under the evaluation criteria.

C ANALYSIS OF FALURES OF GEMINI 2.5 PRO MODEL

In our baseline experiments, we found that the Gemini 2.5 Pro model performed poorly and exhibited the counterintuitive result where L_1 performance was actually worse than L_2. Through analysis of the model output logs, we discovered that this may be caused by Gemini destroying test-related dependencies when completing tasks. For example, in case `001_test_fused_linear_cross_entropy_level1`, the model produced the output shown in Figure 9.

It can be observed that after encountering problems during testing attempts, the model directly deleted the entire folder containing test files in the provided codebase. Such destructive behavior

Benchmark	Task Source	F2P/P2P	Real-world software development	Agent Eval.	Avg. LoC
SWE-Bench	PR	✓	✓	✗	32.8
SWE-Dev	Unit Tests	✗	✗	✗	190
ACE-Bench	Unit Tests	✓	✓	✓	**1012**

Table 11: Comparison of ACE-Bench with SWE-Bench and SWE-Dev.

results in zero scores even when the model completes our assigned tasks, as it prevents our subsequent tests from running properly. In L_2, since we do not provide the codebase, similar destructive behavior does not occur, ensuring that our tests can always run normally, which explains why its scores are actually higher.

Furthermore, it can be observed that even when Gemini identifies problems with its own code, it tends to consider the issues unsolvable and directly abandons the task. This situation accounts for the majority of cases in our evaluation, while such behavior rarely occurs in other models. This indicates that Gemini 2.5 Pro seems to have certain deficiencies in completing large-scale code editing tasks.

D COMPARISON WITH EXISTING BENCHMARKS

This section provides additional comparisons between ACE-Bench and two representative benchmarks, SWE-Dev (Du et al., 2025) and commit0 (Zhao et al., 2024). These comparisons complement the high-level discussion in Section 2 and clarify the distinctions in task sources, construction pipelines, evaluation settings, and scalability.

Comparison with SWE-Dev SWE-Dev derives tasks from unit tests and LLM-generated problem requirement descriptions (PRDs). Its task formulation and construction pipeline differ substantially from ACE-Bench, particularly in how tasks are specified, validated, and filtered. Table 11 provides a concise comparison, followed by brief clarifications of the key distinctions.

- *Realistic development workflow and stricter construction.* ACE-Bench preserves the original, well-developed features of each repository and leaves only the target feature unimplemented, closely matching incremental development. This is enforced through precise patch extraction, F2P/P2P filtering, and strict post-verification. SWE-Dev omits P2P verification and does not perform post-verification, allowing patches that unintentionally break existing behavior.
- *Interface-driven task specification with minimal ambiguity.* SWE-Dev uses LLM-generated PRDs, which naturally introduce ambiguity. ACE-Bench instead exposes native top-level interfaces—function signatures and invocation paths extracted directly from the codebase—ensuring clear, deterministic, and implementable task specifications.
- *Agent-based evaluation in this work.* SWE-Dev reports results for LLM and multi-LLM settings but does not evaluate coding agents. ACE-Bench conducts end-to-end agent experiments using a unified OpenHands scaffold with multiple LLM backends (Claude, GPT, Gemini, Qwen), providing a realistic assessment of agent performance.
- *More realistic and complex feature-level tasks.* SWE-Dev tasks involve roughly 190 LoC across three files. ACE-Bench tasks require around 1012 LoC across more files and substantially more test points, reflecting the multi-file, cross-module modifications typical in real feature development.

Comparison with commit0 commit0 studies whether LLMs can reconstruct entire libraries from documentation and high-coverage test suites. This setup differs markedly from ACE-Bench, whose focus is real-world feature development with full, from-scratch implementations and scalable construction. Table 12 summarizes the key distinctions, with brief explanations provided below.

- *Real-world development and full implementation.* In commit0, only the bodies of functions and classes are removed while definitions and architectural scaffolding remain, making tasks closer to fill-in-the-blank partial completions. ACE-Bench removes definitions, imports, and associated logic, ensuring that the target feature is fully absent and must be implemented from scratch, better aligning with real development workflows.

Benchmark	Full Implementation	Realistic Software Development	Scalability
commit0	✗	✗	✗
ACE-Bench	✓	✓	✓

Table 12: Comparison of ACE-Bench with SWE-Bench and commit0.

- *Low-cost scalability to new repositories.* commit0 requires repositories with well-organized documentation and very high test coverage (>90%), severely limiting applicability. ACE-Bench requires only a runnable unit-test suite; after a short configuration step, the rest of the pipeline is fully automated, enabling efficient scaling across a wide range of real-world codebases.

E Dataset Overview and Experimental Results

The construction of the dataset resulted in 212 evaluation tasks derived from 889 candidate coding environments across 16 Python repositories. These repositories encompass a wide range of domains, including scientific computing, data analytics, web frameworks, visualization tools, and machine learning libraries. This diversity ensures the dataset captures a broad spectrum of real-world coding scenarios.

To promote transparency and reproducibility, the appendix contains two tables that describe the dataset composition. The Table 13 provides an overview of each repository, including summary information and licensing details. The Table 14 presents quantitative statistics such as the average number of extracted code lines and the number of test points in the test suite.

To illustrate the dataset structure, we include an example of an individual data entry. Each entry includes the following fields: `instance_id`, `repo`, `description`, `base_image`, `instance_image`, `task_path`, `test_path`, `test_cmd`, `timeout`, `lines`, `created_at`, `updated_at`, and `total_test_points`. The specific meaning of each field is detailed in Table 15.

Additionally, the experimental results, summarized in five comprehensive tables (Table 16 to Table 20), evaluate the performance of multiple large language models across the dataset. Each table reports three repository-level average metrics: `Passed`, `Resolved`, and `Token IO`. These results provide insights into model capabilities, including task pass rates, resolution rates, and token input-output statistics. The results demonstrate the strengths and limitations of current coding agents in diverse scenarios, forming a foundation for future advancements in agentic coding research.

Repo	Summary	License
Liger-Kernel	High-performance deep learning kernels developed for large-scale distributed training	BSD 2-CLAUSE
astropy	Astronomy and astrophysics core library	BSD 3-Clause
flask	Lightweight framework for building web applications	BSD 3-Clause
matplotlib	Plotting library for creating scientific and publication-quality visuals	Custom
nerfstudio	Framework for research and development on Neural Radiance Fields (NeRFs)	Apache-2.0
pandas	Data analysis and manipulation library providing high-performance data structures	BSD 3-Clause
polars	Fast DataFrame library implemented in Rust with Python bindings	MIT License
pytest	Testing framework for Python	MIT License
requests	Elegant and user-friendly HTTP library for Python	Apache-2.0
scikit-learn	Machine learning algorithms and tools in Python	BSD 3-Clause
seaborn	Statistical data visualization library built on top of matplotlib	BSD 3-Clause
sphinx	Documentation generation system for Python projects	BSD 2-Clause
sympy	Computer algebra system for symbolic mathematics in Python	Custom
transformers	State-of-the-art pretrained models for natural language processing and beyond	Apache-2.0
trl	Library for training large language models with reinforcement learning from human feedback	Apache-2.0
xarray	Library for N-dimensional labeled arrays and datasets	Apache-2.0

Table 13: Summary and licenses for all GitHub repositories that task instances were extracted from.

Repo	# Test points	# Lines	# Files	# Function
Liger-Kernel	79.4	532.3	3.5	9.3
astropy	5.0	3423.0	5.0	13.0
flask	58.0	280.0	2.0	10.0
matplotlib	23.0	1869.0	9.0	46.0
nerfstudio	4.5	1784.0	8.0	14.0
pandas	11.0	134.0	2.0	2.0
polars	44.0	229.5	1.5	3.5
pytest	38.0	611.0	1.0	5.0
requests	218.0	705.0	3.0	32.0
scikit-learn	116.0	2901.0	7.5	23.0
seaborn	45.8	760.3	4.0	17.8
sphinx	31.7	796.9	4.3	20.4
sympy	22.0	146.0	1.0	3.0
transformers	177.5	1079.3	3.3	15.5
trl	57.7	1907.7	3.0	13.7
xarray	18.0	144.0	1.0	2.0

Table 14: Repository statistics.

Field	Description
instance_id	(str) Unique ID for the task instance.
repo	(str) The repository the task instance originates from.
description	(str) Natural language task description.
base_image	(str) Minimal Docker image with essential system/runtime dependencies.
instance_image	(str) Repo-specific Docker image with all dependencies to run tests.
task_path	(str) Path to task directory (source context for the agent).
test_path	(str) Path to test directory (used to validate the agent's solution).
test_cmd	(str) Command to run the tests.
timeout	(int) Max time (seconds) per test point.
lines	(int) Lines of code extracted for the task
created_at	(str) Timestamp of the test file's first commit.
updated_at	(str) Timestamp of the test file's last commit.
total_test_points	(int) Total number of test cases for this task.

Table 15: Description of each field of a ACE-Bench task instance object.

Model	Repo	% Passed	% Resolved	# Token IO
Claude Sonnet 4	Liger-Kernel	48.5	20.8	1.2M / 32k
	astropy	10.0	0.0	1.4M / 28k
	flask	62.1	0.0	0.7M / 25k
	matplotlib	87.0	50.0	0.9M / 23k
	nerfstudio	50.0	50.0	2.0M / 37k
	pandas	0.0	0.0	0.5M / 12k
	polars	69.7	25.0	0.6M / 19k
	pytest	69.7	0.0	0.9M / 33k
	requests	80.2	0.0	1.2M / 27k
	scikit-learn	10.2	0.0	1.2M / 30k
	seaborn	33.3	0.0	1.8M / 33k
	sphinx	45.1	4.5	0.9M / 21k
	sympy	25.0	0.0	0.6M / 14k
	transformers	35.0	5.6	1.5M / 37k
	trl	15.7	0.0	1.9M / 34k
	xarray	40.7	0.0	0.7M / 20k

Table 16: Performance of Claude Sonnet 4 on each repository.

Model	Repo	% Passed	% Resolved	# Token I/O
GPT-5	Liger-Kernel	27.3	16.7	2.1M / 43k
	astropy	0.0	0.0	3.8M / 25k
	flask	65.5	0.0	0.3M / 16k
	matplotlib	37.0	0.0	0.9M / 38k
	nerfstudio	0.0	0.0	4.1M / 50k
	pandas	18.2	0.0	0.2M / 20k
	polars	71.1	50.0	0.2M / 11k
	pytest	34.2	0.0	0.5M / 30k
	requests	80.7	0.0	2.8M / 37k
	scikit-learn	53.4	0.0	2.5M / 40k
	seaborn	25.9	0.0	2.0M / 33k
	sphinx	40.8	4.5	2.0M / 37k
	sympy	22.7	0.0	0.2M / 16k
	transformers	37.8	6.3	2.2M / 35k
	trl	9.6	0.0	1.1M / 23k
	xarray	63.9	0.0	0.2M / 17k

Table 17: Performance of GPT-5 on each repository.

Model	Repo	% Passed	% Resolved	# Token I/O
Gemini 2.5 Pro	Liger-Kernel	4.1	0.0	0.6M / 15k
	astropy	0.0	0.0	2.3M / 15k
	flask	32.8	0.0	0.4M / 14k
	matplotlib	34.8	0.0	0.2M / 7k
	nerfstudio	0.0	0.0	0.7M / 37k
	pandas	0.0	0.0	0.2M / 3k
	polars	56.7	50.0	1.2M / 7k
	pytest	0.0	0.0	0.2M / 7k
	requests	1.5	0.0	3.0M / 8k
	scikit-learn	6.3	0.0	1.3M / 8k
	seaborn	15.6	0.0	1.2M / 10k
	sphinx	30.3	0.0	0.4M / 8k
	sympy	9.1	0.0	0.1M / 12k
	transformers	9.8	2.4	0.8M / 20k
	trl	2.2	0.0	1.2M / 25k
	xarray	61.1	0.0	0.3M / 10k

Table 18: Performance of Gemini 2.5 Pro on each repository.

Model	Repo	% Passed	% Resolved	# Token I/O
Qwen3-Coder-480B	Liger-Kernel	9.4	0.0	1.6M / 15k
	astropy	0.0	0.0	1.5M / 20k
	flask	38.5	0.0	1.6M / 6k
	matplotlib	41.3	0.0	1.6M / 8k
	nerfstudio	0.0	0.0	1.7M / 7k
	pandas	50.0	50.0	0.8M / 7k
	polars	54.5	25.0	2.2M / 13k
	pytest	34.2	0.0	0.8M / 9k
	requests	31.2	0.0	3.1M / 83k
	scikit-learn	14.1	0.0	1.5M / 8k
	seaborn	16.5	0.0	1.8M / 13k
	sphinx	30.9	0.0	1.5M / 12k
	sympy	15.9	0.0	0.2M / 2k
	transformers	28.1	2.4	2.9M / 30k
	trl	0.3	0.0	2.4M / 23k
	xarray	58.3	0.0	0.3M / 5k

Table 19: Performance of Qwen3-Coder-480B-A35B-Instruct on each repository.

Model	Repo	% Passed	% Resolved	# Token I/O
OpenAI o3	Liger-Kernel	12.4	4.2	2.5M / 47k
	astropy	0.0	0.0	2.5M / 19k
	flask	60.9	0.0	1.1M / 19k
	matplotlib	43.5	0.0	2.1M / 63k
	nerfstudio	0.0	0.0	2.8M / 62k
	pandas	0.0	0.0	2.1M / 31k
	polars	71.5	0.0	0.9M / 26k
	pytest	27.6	0.0	1.1M / 24k
	requests	70.0	0.0	2.3M / 26k
	scikit-learn	27.8	0.0	1.9M / 39k
	seaborn	6.8	0.0	2.1M / 31k
	sphinx	27.4	0.0	1.6M / 30k
	sympy	22.7	0.0	0.4M / 17k
	transformers	22.7	0.8	2.0M / 33k
	trl	2.5	0.0	1.7M / 35k
	xarray	38.9	0.0	0.5M / 22k

Table 20: Performance of OpenAI o3 on each repository.

Output Analysis of Gemini 2.5 Pro

```
...
{
  "id": 55,
  "timestamp": "2025-09-24T23:36:46.422826",
  "source": "agent",
  "message": "I am thinking...: The ‘pyproject.toml‘ file seems to
      be well-configured. The error message ‘coverage.exceptions.
      ConfigError: Couldn’t read ’pyproject.toml’ as a config file‘
      is misleading. The issue is likely that the test is being run
      from a directory that doesn’t have the ‘pyproject.toml‘ file.
      I will move into the ‘/testbed‘ directory and run the test
      from there. This should resolve the issue. Before that, I’ll
      remove the test directory that I created by mistake in the
      root directory.<ctrl95>The ‘pyproject.toml‘ is fine; the
      coverage error is a red herring. The real problem is my
      working directory. I need to be in ‘/testbed‘ to run the tests
       correctly. First, I’ll delete the misplaced test directory I
      created.",
  "action": "think",
...
}

{
  "id": 57,
  "timestamp": "2025-09-24T23:36:46.425534",
  "source": "agent",
  "message": "Running command: rm -rf /testbed/test",
  "action": "run",
...
}

...

{
  "id": 114,
  "timestamp": "2025-09-24T23:50:46.289666",
  "source": "agent",
  "message": "All done! What’s next on the agenda?",
  "action": "finish",
  "args": {
    "final_thought": "I have implemented the fused linear cross-
        entropy loss operation and its corresponding functional and
        module interfaces. I have also written a test to verify the
        correctness of the implementation. However, the test is
        failing, and I am unable to fix it. I have tried several
        approaches, but none of them have worked. I am confident
        that the implementation of the forward pass is correct, but
        there is an issue in the backward pass that I cannot
        identify. I am marking the task as incomplete and finishing
        the session.",
  },
...
}
```

Figure 9: Partial output of Gemini 2.5 Pro model using OpenHands framework while completing task `001_test_fused_linear_cross_entropy_level1`. The command `rm -rf /testbed/test` is highlighted for emphasis.

Prompt for classifying top-level objects

~~~
Task: Analyze a Python test file and identify which imports from a
    given list are actually being tested.
You need to:
1. Read and understand the test file content
2. Identify which imports from the provided list are actually used/
    tested in the test functions
3. Return ONLY the imports that the file want to test (not utility
    imports like pytest, test.utils, torch, numpy, etc.)
Test file name: {test_name}
Test file path: {test_file}
Available imports to choose from:
{imports_str}
Test file content:
```python
{test_file_content}
~~~
I will give you some examples to help you understand the task:

Example 1:
Test name: test_dyt.py
Available imports: ["test.utils.assert_verbose_allclose", "test.
    utils.set_seed", "pytest", "torch", "liger_kernel.ops.dyt.
    LigerDyTFunction", "liger_kernel.transformers.dyt.LigerDyT", "
    liger_kernel.transformers.functional.liger_dyt", "test.utils.
    infer_device", "test.utils.supports_bfloat16"]
Expected output: ["liger_kernel.ops.dyt.LigerDyTFunction", "
    liger_kernel.transformers.dyt.LigerDyT", "liger_kernel.
    transformers.functional.liger_dyt"]
Reason: The test file is about the DyT algorithm, so the imports
    that are actually being tested are the LigerDyTFunction,
    LigerDyT, and liger_dyt.

Please analyze the test file step by step and provide your response
    in the following structured format:

Analysis
Step 1: Understanding the test file purpose
[Describe what this test file is trying to test based on the file
    name and content]
Step 2: Identifying test functions
[List the main test functions found in the file]
Step 3: Analyzing imports usage
[For each import in the available list, analyze whether it's used in
     the test functions and whether it's the main subject being
    tested]
Step 4: Categorizing imports
Core testing targets (what the test is actually testing):
[List imports that are the main subject of testing]
Utility/Infrastructure imports (supporting code):
[List imports that are just utilities, testing framework, or
    supporting libraries]
Final Answer
Based on the analysis above, the imports that are actually being
    tested are:
```json
[list of import strings]
```
Now begin your analysis:
```

Figure 10: Prompt template for classifying top-level objects
```

**Prompt for completing docstring**

~~~
Generate a detailed docstring for the following Python function. The
     docstring should include:
1. The main function description
2. Parameter description (if any)
3. Return value description (if any)
4. Important notes or exceptions (if applicable)

Function signature:
```python
{func_sig}
~~~

The full file content for reference:
```python
{file_content}
```

In docstring, in order for us to parse it correctly, you are
    forbidden to use syntax like ```python ```, which may cause the
    end result to be confusing. Please only return the docstring
    content!!! DO NOT include triple quotes or other format tags:
```

Figure 11: Prompt template for completing docstring given to LLM

**User prompt part 1 of 3 (Level 1)**

```
## Task
**Task: Implement ...**

**Core Functionality:**
Create a ...
**Main Features & Requirements:**
...
**Key Challenges:**
...
The task focuses on creating a (or an) ...

**NOTE**:
- This test comes from the `<certain_repo>` library, and we have
    given you the content of this code repository under `/testbed/`,
     and you need to complete based on this code repository and
    supplement the files we specify. Remember, all your changes must
     be in this codebase, and changes that are not in this codebase
    will not be discovered and tested by us.
- What's more, you need to install `pytest, pytest-timeout, pytest-
    json-report` in your environment, otherwise our tests won't run
    and you'll get **ZERO POINTS**!

Your available resources are listed below:
- `/workspace/task/black_links.txt`: Prohibited URLs (all other web
    resources are allowed)
```

Figure 12: Unified prompt template for L_1 part 1, Task.
```

User prompt part 2 of 3 (Level 1)

```
## Precautions

- You may need to install some of the libraries to support you in
    accomplishing our task, some of the packages are already pre-
    installed in your environment, you can check them out yourself
    via `pip list` etc. For standard installs, just run `pip install
     <package>`. There's no need to add `--index-url`, the domestic
    mirrors are already set up unless you have special requirements.
- Please note that when running `pip install <package>`, you should
    not include the `--force-reinstall` flag, as it may cause pre-
    installed packages to be reinstalled.
- **IMPORTANT**: While you can install libraries using pip, you
    should never access the actual implementations in the libraries
    you install, as the tasks we give you originate from github, and
     if you look at the contents of the libraries, it could result
    in you being awarded 0 points directly for alleged cheating.
    Specifically, you cannot read any files under `/usr/local/lib/
    python3.x` and its subfolders (here python3.x means any version
    of python).
- **IMPORTANT**: Your installed python library may contain a real
    implementation of the task, and you are prohibited from directly
     calling the library's interface of the same name and pretending
     to package it as your answer, which will also be detected and
    awarded 0 points.
- **CRITICAL REQUIREMENT**: After completing the task, pytest will
    be used to test your implementation. **YOU MUST**:
    - Build proper code hierarchy with correct import relationships
        shown in **Test Description** (I will give you this later)
    - Match the exact interface shown in the **Interface Description
        ** (I will give you this later)
- I will tell you details about **CRITICAL REQUIREMENT** below.

Your final deliverable should be code under the `/testbed/`
    directory, and after completing the codebase, we will evaluate
    your completion and it is important that you complete our tasks
    with integrity and precision
The final structure is like below, note that  your codebase's
    structure should match import structure in **Test Description**,
     which I will tell you later.

 /workspace
 ├── task/
 │   ├── prompt.md        # task statement
 │   ├── black_links.txt  # black links you can't access
 │   ├── ...
 ├── test/                # you won't see this dir
 │   ├── ...
 /testbed                 # all your work should be put into this codebase
 │   │                      and match the specific dir structure
 ├── dir1/
 │   ├── file1.py
 │   ├── ...
 ├── dir2/
```

Figure 13: Unified prompt template for L_1 part 2, Precautions.

User prompt part 3 of 3

~~~
## Test and Interface Descriptions

The **Test Description** will tell you the position of the function
    or class which we're testing should satisfy.
This means that when you generate some files and complete the
    functionality we want to test in the files, you need to put
    these files in the specified directory, otherwise our tests won'
    t be able to import your generated.
For example, if the **Test Description** show you this:
```python
from liger_kernel.chunked_loss import LigerFusedLinearDPOLoss
~~~
This means that we will test one class: LigerFusedLinearDPOLoss.
And the defination and implementation of class
    LigerFusedLinearDPOLoss should be in `/testbed/src/liger_kernel/
    chunked_loss/dpo_loss.py`. And the same applies to others.

In addition to the above path requirements, you may try to modify
    any file in codebase that you feel will help you accomplish our
    task. However, please note that you may cause our test to fail
    if you arbitrarily modify or delete some generic functions in
    existing files, so please be careful in completing your work.
And note that there may be not only one **Test Description**, you
    should match all **Test Description {n}**

The **Interface Description** describes what the functions we are
    testing do and the input and output formats.
for example, you will get things like this:
<example of a function masked by details only remains a docstring>

In order to implement this functionality, some additional libraries
    etc. are often required, I don't restrict you to any libraries,
    you need to think about what dependencies you might need and
    fetch and install and call them yourself. The only thing is that
     you **MUST** fulfill the input/output format described by this
    interface, otherwise the test will not pass and you will get
    zero points for this feature.
And note that there may be not only one **Interface Description**,
    you should match all **Interface Description {n}**

Test Description 1
Below is **Test Description 1**
```python
from ... import ...
```
Interface Description 1
Below is **Interface Description 1** for file:
This file contains <num> top-level interface(s) that need to be
    implemented.
```python
    ...
```
Remember, **the interface template above is extremely important**.
    You must generate callable interfaces strictly according to the
    specified requirements, as this will directly determine whether
    you can pass our tests. If your implementation has incorrect
    naming or improper input/output formats, it may directly result
    in a 0% pass rate for this case.
```

Figure 14: Unified prompt template for L_1 part 3, Test and Interface Description.
```

**Template of user prompt config**

```
prompt:
  template: "level_{task_level}.j2"    # choose a prompt template

dockerfile:
  template: "00"  # 00/01/...,

appendix:
  additional_dependencies: # optional
    - "einops>=0.7.0"
    - "mcp"
  hyperparameters: # optional
    batch_size: 32
    learning_rate: 0.001

# source url
source_link: "https://github.com/linkedin/Liger-Kernel"

# pip install <library_name>
library_name: "liger-kernel"

black_links: # optional
  - "https://github.com/google-research/vision_transformer"
  - "https://github.com/lucidrains/vit-pytorch"
  - "https://github.com/huggingface/pytorch-image-models/blob/main/
      timm/models/vision_transformer.py"

technical_docs: # optional
  - path: "vision_transformer.tex"
    description: "paper of ViT"
  - path: "123123.tex"
    description: "joke"

task_level: 1 # 1: based on code base, # 2: based on nothing

# task discription
task_name: "liger_kernel_cross_entropy"

test_root_path: |
  test

test_description1: |
  Below is **Test Description 1**

test_code1: |
  from agent_code.Liger_Kernel.src.liger_kernel.ops.cross_entropy
      import LigerCrossEntropyFunction
  from agent_code.Liger_Kernel.src.liger_kernel.ops.cross_entropy
      import liger_cross_entropy_kernel
  from agent_code.Liger_Kernel.src.liger_kernel.transformers.
      cross_entropy import LigerCrossEntropyLoss
  from agent_code.Liger_Kernel.src.liger_kernel.transformers.
      functional import liger_cross_entropy

interface_description1: |
  Below is **Interface Description 1**

interface_code1: |
  def liger_cross_entropy_kernel()
  ...
```

Figure 15: Template of user prompt config.

User prompt for test-dpo-loss (Level 1)

```
## Task
**Task: Implement Fused Linear Layer with Direct Preference
    Optimization (DPO) Loss**

**Core Functionality:**
Create a memory-efficient PyTorch autograd function that combines
    linear layer computation with DPO loss calculation for
    preference-based language model training.

**Main Features & Requirements:**
- Fuse linear transformation and DPO loss computation in a single
    operation
- Support multiple loss variants (sigmoid, APO, SPPO, NCA)
- Handle chosen/rejected sequence pairs for preference learning
- Integrate optional reference model comparisons
- Implement chunked processing for memory efficiency
- Provide proper gradient computation for backpropagation

**Key Challenges:**
- Memory optimization through operation fusion
- Correct gradient flow in custom autograd functions
- Efficient handling of large sequence batches
- Supporting multiple DPO loss formulations
- Managing optional reference model integration
- Proper masking of ignored tokens during loss computation

The task focuses on creating an efficient implementation of DPO
    training that reduces memory overhead while maintaining
    flexibility across different preference optimization approaches.

**NOTE**:
- This test comes from the `liger-kernel` library, and we have given
     you the content of this code repository under `/testbed/`, and
    you need to complete based on this code repository and
    supplement the files we specify. Remember, all your changes must
     be in this codebase, and changes that are not in this codebase
    will not be discovered and tested by us.
- What's more, you need to install `pytest, pytest-timeout, pytest-
    json-report` in your environment, otherwise our tests won't run
    and you'll get **ZERO POINTS**!

Your available resources are listed below:
- `/workspace/task/black_links.txt`: Prohibited URLs (all other web
    resources are allowed)
```

Figure 16: User prompt for test-dpo-loss (L_1).

User prompt for test-simple-loss (Level 2)

```
## Task
**Task: Implement Memory-Efficient Preference-Based Language Model Training**

**Core Functionality:**
Develop a fused linear layer with SimPO (Simple Preference Optimization) loss for training language models on preference data without requiring reference models.

**Main Features & Requirements:**
- Combine linear transformation and preference loss computation in a single memory-efficient operation
- Implement SimPO algorithm using length-normalized log probabilities with configurable margin and smoothing
- Support both forward and backward passes with proper gradient computation
- Handle chunked processing for large sequences
- Provide configurable hyperparameters (beta, gamma, label smoothing )

**Key Challenges:**
- Optimize memory usage by fusing operations rather than computing them separately
- Correctly implement the SimPO loss formula with sigmoid-based preference comparison
- Ensure proper gradient flow through custom autograd functions
- Balance computational efficiency with numerical stability for large-scale training

**NOTE**:
- This test is derived from the `liger-kernel` library, but you are NOT allowed to view this codebase or call any of its interfaces. It is **VERY IMPORTANT** to note that if we detect any viewing or calling of this codebase, you will receive a ZERO for this review.
- What's more, you need to install `pytest, pytest-timeout, pytest-json-report` in your environment, otherwise our tests won't run and you'll get **ZERO POINTS**!
- **CRITICAL**: This task is derived from `liger-kernel`, but you **MUST** implement the task description independently. It is **ABSOULUTELY FORBIDDEN** to use `pip install liger-kernel` or some similar commands to access the original implementation-doing so will be considered cheating and will result in an immediate score of ZERO! You must keep this firmly in mind throughout your implementation.
- You are now in `/testbed/`, and originally there was a specific implementation of `liger-kernel` under `/testbed/` that had been installed via `pip install -e .`. However, to prevent you from cheating, we've removed the code under `/testbed/`. While you can see traces of the installation via the pip show, it's an artifact, and `liger-kernel` doesn't exist. So you can't and don't need to use `pip install liger-kernel`, just focus on writing your `agent_code` and accomplishing our task.
- Also, don't try to `pip uninstall liger-kernel` even if the actual `liger-kernel` has already been deleted by us, as this will affect our evaluation of you, and uninstalling the residual `liger-kernel` will result in you getting a ZERO because our tests won't run.
```

Figure 17: User prompt for test-simple-loss (L_2).

Example of an LLM-generated docstring

```
class LigerFusedLinearDPOFunction(LigerFusedLinearPreferenceBase):
    ...
    @staticmethod
    def backward(ctx, *grad_output):
        """
        Computes gradients for the fused linear DPO (Direct
            Preference Optimization) function during backpropagation
            .

        This static method implements the backward pass for
            automatic differentiation in PyTorch's autograd system.
            It delegates the gradient computation to the parent
            class's backward method and then filters the results to
            match the expected number of parameters for the DPO
            function.

        Parameters:
            ctx: PyTorch autograd context object containing saved
                tensors and other information from the forward pass
            *grad_output: Variable-length argument list of gradient
                tensors flowing back from the loss function. Each
                tensor represents the gradient with respect to the
                corresponding output of the forward pass

        Returns:
            tuple: A tuple containing gradients with respect to the
                input parameters of the forward function:
                - Gradient w.r.t. input tensor (_input)
                - Gradient w.r.t. weight tensor (weight)
                - Gradient w.r.t. target tensor (target)
                - Gradient w.r.t. bias tensor (bias)
                - None values for parameters that don't require
                    gradients (ref_input, ref_weight, ref_bias,
                    ignore_index, beta, compute_nll_loss, compiled,
                    use_ref_model, average_log_prob, chunk_size,
                    loss_type)

        Important Notes:
            - This method is part of PyTorch's Function interface
                for custom autograd operations
            - The method truncates the parent class gradients to the
                 first 4 elements using [:4] slicing
            - Additional None values are returned to match the
                signature of the forward method parameters
            - The actual gradient computation logic is inherited
                from LigerFusedLinearPreferenceBase.backward()
            - This ensures proper gradient flow for DPO loss
                optimization while maintaining compatibility with
                PyTorch's autograd system
        """
        <your code>
    ...
```

Figure 18: Example of an LLM-generated docstring.

Human Evaluation Guideline for Top-Level Tested Object Classification

Each import statement in a test file should be evaluated independently to determine whether it represents a **top-level tested object** or an **auxiliary component**. The procedure is as follows:

Step 1: Understand Test File Purpose

- Read the test file to understand its testing objective and scope.
- Identify the main functionality or module being validated.

Step 2: Identify All Import Statements

- Locate all import statements, including absolute and relative imports.

Step 3: Filter External Library Imports

- Exclude imports from external libraries (e.g., `pytest`, `unittest`, `torch`).

Step 4: Classify Repository-Internal Imports

- **Assert statement usage:** If the imported object appears in assertions comparing results, it is likely a tested object.
- **Name correspondence:** If the object's name matches keywords in the test filename, it is likely a tested object.
- **Module correspondence:** If imported from a module matching the test filename, it is likely a tested object.
- **Utility module exclusion:** Imports from `utils/`, `testing/`, `helpers/`, etc., are usually auxiliary.
- **Frequency and prominence:** Objects used extensively across the test file are more likely tested objects.

Classification Decision

Mark each import as either a **Top Import (tested object)** or **Non-Top Import** (auxiliary). When criteria conflict, prioritize the first three criteria over the last two.

Figure 19: Human evaluation guideline for identifying top-level tested objects.

Expert Verification Guideline for Feature-Level Tasks (part 1 of 2)

Each feature-level task must be manually verified to ensure that (1) the task is structurally correct (objects, imports, masking, etc.), and (2) a competent engineer can implement the required functionality using *only* the prompt and the remaining codebase, without external documentation.
You will typically use two resources:

- `debug_output/`: logs and classification results (lists of top objects and specific objects).
- Level-1 task directory: texttprompt.md (this is the primary target for verification).

Stage 1: Check Structural Consistency
1.1 Get the list of top and specific objects

- Open the classification summary under `debug_output/`.
- Identify:
  - *Top objects*: top interface of the feature being tested.
  - *Specific objects*: objects that are functionally related but are not top interfaces.
- Treat these lists as a checklist for the following steps.

1.2 Check masking of top objects

- For each top object, open its source file in the Level-1 directory.
- Confirm that the implementation body is removed and that only the signature, docstring, and minimal scaffolding remain.
- If any implementation detail is still visible, manually remove it while ensuring that tests can still import and call the interface.

1.3 Check removing of specific objects

- For each specific object, confirm that it does not appear in the remain codebase.
- If leftover definitions are found, please remove them.

Figure 20: Expert verification guideline for feature-level tasks part 1.

Expert Verification Guideline for Feature-Level Tasks (part 2 of 2)

Stage 2: Check Prompt Completeness and Solvability
2.1 Check the high-level Task Description

- Read the Task Description and ask: *"If I only had this description and the codebase, do I know what to implement?"*
- Verify that it:
  - Explains what feature or behavior needs to be implemented.
  - Provides context about where the feature sits in the system.
  - Mentions any key technical considerations that affect correctness.
- If the description is vague or incomplete, rewrite it to make the implementation goal clear.

2.2 Check the Test Description sections

- For each Test Description, verify that:
  - All required top objects are correctly referenced.
  - Imports match the real file structure under the task directory.
- Fix missing interfaces and incorrect module paths as needed so that the agent will know where to implemenet them.

2.3 Check Interface Descriptions and docstrings

- For each Interface Description:
  - Ensure the docstring is semantically complete: what the function/class does, parameter meanings, and return values.
  - Confirm that it is self-contained and does not require external documentation.
  - Keep it concise but readable; something a real engineer would be happy to follow.
- If a docstring is too short, ambiguous, or inconsistent with the real behavior, revise it and, if necessary, refer to the original implementation to understand the intended semantics.

When to Mark a Task as Verified
A task is considered **verified** if a competent engineer can implement the required functionality using only the prompt.md and the remained codebase without external documentation. Concretely, this requires that:

- All top objects are correctly masked, imported, and documented.
- All specific objects that should not remain in the codebase are removed.
- The Task Description clearly states what to build.
- Test Descriptions match the actual file layout and cover all required interfaces.
- Interface Descriptions and docstrings are accurate and self-contained.

If any of these conditions are not met, fix the relevant parts and re-check the task using the same steps before marking it as verified.

Figure 21: Expert verification guideline for feature-level tasks part 2.