LMR-BENCH: Evaluating LLM Agents' Ability on Reproducing Language Modeling Research

Shuo Yan*, Ruochen Li*, Ziming Luo*, Zimu Wang*, Daoyang Li*, Liqiang Jing, Kaiyu He, Peilin Wu, George Michalopoulos, Yue Zhang, Ziyang Zhang, Mian Zhang, Zhiyu Chen, Xinya Du
University of Texas at Dallas
`{shuo.yan, ruochen.li, ziming.luo, zimu.wang}@utdallas.edu`

Abstract

Large language model (LLM) agents have demonstrated remarkable potential in advancing scientific discovery. However, their capability in the fundamental yet crucial task of reproducing code from research papers, especially in the NLP domain, remains underexplored. This task includes unique complex reasoning challenges in the intellectual synthesis of abstract concepts and the comprehension of code repositories with interdependent files. Motivated by this gap, we present LMR-BENCH, a benchmark designed to systematically evaluate the capability of LLM agents on code reproduction from **L**anguage **M**odeling **R**esearch. It consists of 28 code reproduction tasks derived from 23 research papers published in top-tier NLP venues over the past five years, spanning nine fundamental categories. Models are provided with a research paper, a code repository containing one or more masked functions, and instructions for implementing these functions. We conduct extensive experiments in standard prompting and LLM agent settings with state-of-the-art LLMs, evaluating the accuracy of unit tests and performing LLM-based evaluation of code correctness. Experimental results reveal that even the most advanced models still exhibit persistent limitations in scientific reasoning and code synthesis, highlighting critical gaps in LLM agents' ability to autonomously reproduce scientific research[1].

1 Introduction

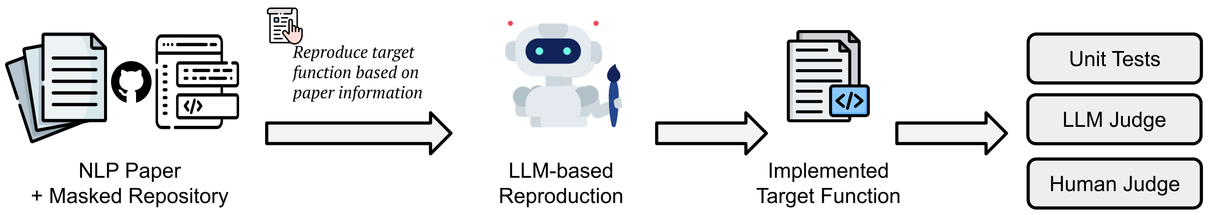
The advent of large language model (LLM) agents has revolutionized beyond language generation, being recognized as a transformative force in advancing scientific discovery (Peng et al., 2023; Ma et al., 2024; Yang et al., 2024; Luo et al., 2025). These agents have shown to be capable of executing the entire scientific discovery pipeline (Li et al., 2024c), from generating research ideas, designing experiments (Li et al., 2024a,b; Baek et al., 2025) to implementing code (Jiang et al., 2024a,b; Zhang et al., 2024), drafting academic papers (Lee et al., 2022; Wang et al., 2024; Ifargan et al., 2025), and even producing complete papers that potentially pass peer review (Yamada et al., 2025). They have also been integrated with tools such as Scholar Inbox (Flicke et al., 2025) to accelerate humans' research, highlighting the extraordinary capability of LLM agents to understand, synthesize, and generate complex knowledge in scientific discovery.

Despite the agents' remarkable advancement in research acceleration, there remains a notable gap regarding their ability in a foundational yet crucial aspect of scientific validation, i.e., code reproduction from academic papers in real-world environments. This task poses unique challenges in complex reasoning, especially in the following two aspects: (1) **Logic understanding**, such as the intellectual synthesis of concise and abstract mathematical equations, algorithm outlines, and generalized flowcharts; (2) **Code implementation**, particularly at the repository level that spreads across multiple interdependent files. Reproducing algorithms necessitates a thorough analysis of these complex dependencies, ensuring the consistency of both the internal codebase and the external environment, thereby amplifying the challenges associated with reproduction.

However, while reproducing code from research papers is a critical capability for LLMs, there is a lack of a dedicated benchmark that systematically evaluates the capability of LLMs to reproduce research papers in real-world scientific contexts. Existing efforts fall into several categories: ML engineering (e.g., MLAgentBench (Huang et al., 2024), MLE-Bench (Chan et al., 2025)), data-driven scientific discovery (e.g., DSBench (Jing et al., 2025), ScienceAgentBench (Chen et al., 2025)), and code debugging and issues resolving (e.g., SWE-bench

*Equal contribution.

[1]Data, code, and leaderboard of LMR-BENCH are available at `https://github.com/du-nlp-lab/LMR-Bench`.



Reproduce target function based on paper information
NLP Paper + Masked Repository
LLM-based Reproduction
Implemented Target Function
Unit Tests
LLM Judge
Human Judge

Figure 1: Overview of LMR-BENCH. Given an NLP research paper and a corresponding codebase with masked functions, the LLM agent is tasked with reproducing the function, requiring its ability of scientific method understanding, abstract reasoning, and cross-file understanding.

(Jimenez et al., 2024), DebugBench (Tian et al., 2024)). While these benchmarks are valuable, they typically evaluate isolated technical capabilities using task-specific inputs and metrics, rather than evaluate end-to-end paper reproduction. A concurrent effort, PaperBench (Starace et al., 2025), evaluates LLM agents on the reproduction of 20 ICML 2024 papers. However, PaperBench requires reproducing the entire project from scratch—an unrealistic expectation for current LLM agents, making it difficult to offer valuable insights in guiding model improvements. More importantly, its evaluation protocol relies solely on LLM-as-a-judge, lacking curated unit tests or automated checks to ensure objective and reproducible assessments.

Motivated by this gap, we present LMR-BENCH, a benchmark designed to systematically evaluate the LLM agents' ability on reproducing **L**anguage **M**odeling **R**esearch. It consists of 28 code reproduction tasks derived from 23 research papers published in top-tier NLP venues over the past five years, spanning nine fundamental categories, such as generative models and reinforcement learning, which are central to current LLM research. In each task, LLM agents are provided with a research paper, a code repository containing one or more masked functions, and instructions for implementing these functions. Successful completion requires the model to comprehend the algorithmic details accurately and generate functionally correct, syntactically consistent code (see Figure 1). To ensure an objective evaluation, we design two distinct metrics: the accuracy of unit tests curated by human expert annotators and the distribution of LLM-as-a-judge classifications of generated implementations into three categories.

We conduct extensive experiments in standard prompting and LLM agent settings with OpenHands (Wang et al., 2025) on state-of-the-art LLMs, such as GPT-4o (Hurst et al., 2024), GPT-4.1, and o4-mini. Experimental results reveal that even the most advanced models and LLM agents exhibit persistent limitations in scientific reasoning and code synthesis, such as unsuccessful paper parsing and failure in reasoning across steps and files, highlighting critical gaps in agents' ability to autonomously reproduce scientific research.

The key contributions of our work can be summarized as follows: **(1)** We present LMR-BENCH, a benchmark designed to systematically evaluate the ability of LLM agents to reproduce scientific research projects. It consists of 28 code reproduction tasks across nine core categories in LLM research; **(2)** We introduce two complementary evaluation metrics: the accuracy of unit tests and the distribution of LLM-as-a-judge classifications of generated implementations, offering an objective evaluation of the LLM agents' capabilities. The unit tests are curated by human expert annotators and executed within separate Docker containers to faithfully reproduce the original environment; **(3)** We conduct extensive experiments in standard prompting and LLM agent settings, highlighting critical gaps and producing valuable insights in the current LLM agent's ability in reproducing scientific research.

2 Related Work

LLM Agents for Scientific Discovery. Recent research has increasingly focused on leveraging LLM agents to advance scientific discovery, spanning the entire research pipeline, from research idea generation and experimental design (Li et al., 2024a,b; Baek et al., 2025) to implementing code (Jiang et al., 2024a,b; Zhang et al., 2024) and drafting academic papers (Lee et al., 2022; Wang et al., 2024; Ifargan et al., 2025). Some studies have introduced agent-based systems that can automate an end-to-end research flow. MLR-Copilot (Li et al., 2024c) utilizes LLM agents to autonomously generate and implement research ideas. The AI Scientist (Lu et al., 2024) is an iterative framework designed to generate research concepts, conduct experiments, write papers, and perform peer reviews.

The AI Scientist-v2 (Yamada et al., 2025) extends this pipeline by generalizing idea generation, incorporating coarse-grained experiment management, and employing an agentic tree search-based exploration. This approach has been shown to produce manuscripts that successfully pass peer review at well-recognized machine-learning workshops.

However, the ideas and experiments in these frameworks are typically synthesized by agent themselves, which limits their ability to capture the complexity of real-world scenarios. In contrast, our research focuses on the capability of LLM agents to faithfully reproduce peer-reviewed research papers, bridging the gap between agent-synthesized information and real-world publications.

LLM Agents for Code Generation. Code generation serves as a recognized benchmark for evaluating models' problem-solving abilities and their practicality in software development. Models such as Codex (Chen et al., 2021) and Qwen-Coder (Hui et al., 2024), accompanied by agent-based frameworks like MapCoder (Islam et al., 2024), AgentCoder (Huang et al., 2023), and OpenHands (Wang et al., 2025) have been introduced to enhance the scalability of code intelligence. At the same time, some benchmarks dedicated to code generation have been proposed. MBPP, MathQA-Python (Austin et al., 2021), FC2Code (Liu et al., 2022), and LiveCodeBench (Jain et al., 2025) evaluate models' capability to generate code based on natural language instructions. MLAgentBench (Huang et al., 2024) and MLE-bench (Chan et al., 2025) are based on Kaggle competitions to evaluate LLMs' machine-learning engineering capabilities. SUPER (Zhuo, 2024) evaluates agents on end-to-end repository setup and execution for real research repos, stressing environment provisioning, dependency resolution, and task orchestration. RepoBench (Liu et al., 2024) and ML-Bench (Liu et al., 2023) focus on code generation at the repository level. DSBench (Jing et al., 2025) and SciAgentBench (Chen et al., 2025) emphasize data-driven scientific discovery. SWE-bench (Jimenez et al., 2024) and DebugBench (Tian et al., 2024) center on resolving bugs and issues within the codebase. However, there is little work on *research code implementation and execution*, which is a fundamental capability of LLM research agents (Luo et al., 2025). PaperBench (Starace et al., 2025) assesses LLM agents to replicate 20 research papers from ICML 2024. However, this dataset requires reproduction entirely from scratch, which is far beyond existing agents' ability and may lead to discrepancies compared to human-curated codebases. In this paper, we build upon the concept of code generation but focus specifically on the exciting and fundamental skill of code reproduction from NLP research papers with human instructions.

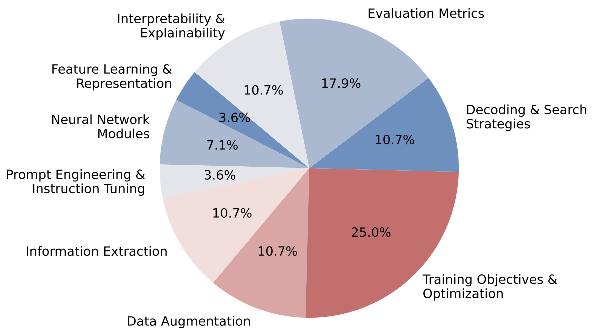


Figure 2: Question distribution in LMR-BENCH.

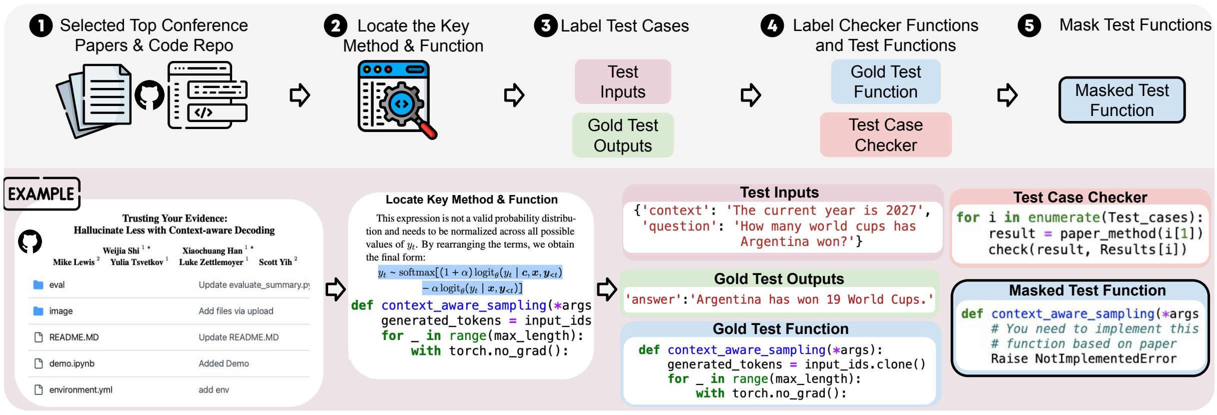
3 LMR-BENCH

LMR-BENCH is a benchmark designed to evaluate the LLM agents' ability on reproducing language modeling research papers on related functions in the repository. It consists of 23 research papers and 28 distinct questions, each corresponding to a key task within the LLM/NLP research field. It covers nine essential task categories (see Table 4), with the distributions illustrated in Figure 2. Training Objectives & Optimization and Evaluation Metrics are the most prevalent. This distribution aligns with the significance and practical challenges of real-world reproducibility, as these areas often involve high levels of abstraction and require rigorous methodological precision.

3.1 Task Formulation

Given a research paper and its corresponding open-source code repository, the aim of the task is to reproduce the implementation of the missing function using the information provided in the paper, focusing on repository-level code generation. This simulates the real-world scenario where one reproduces or verifies key components of a research method based on its textual description.

As shown in Figure 1, the components provided as inputs to LLMs include (1) the original paper obtained from the proceedings; (2) the source files and codes within the repository, with masked functions for reproduction; and (3) the definition of the target function, including the description of its definition, input, output, and any additional steps required for implementation. The reproduction process involves



1 Selected Top Conference Papers & Code Repo
2 Locate the Key Method & Function
3 Label Test Cases
Test Inputs
Gold Test Outputs
4 Label Checker Functions and Test Functions
Gold Test Function
Test Case Checker
5 Mask Test Functions
Masked Test Function
EXAMPLE
Trusting Your Evidence: Hallucinate Less with Context-aware Decoding
Weijia Shi 1 * Xiaochuang Han 1 * Mike Lewis 2 Yulia Tsvetkov 1 Luke Zettlemoyer 1 Scott Yih 2
eval Update evaluate_summary.py
image Add files via upload
README.MD Update README.MD
demo.ipynb Added Demo
environment.yml add env
Locate Key Method & Function
Test Inputs
Gold Test Outputs
Gold Test Function
Test Case Checker
Masked Test Function

This expression is not a valid probability distribution and needs to be normalized across all possible values of y_t. By rearranging the terms, we obtain the final form:

$$y_t \sim \mathrm{softmax}[(1+\alpha)\,\mathrm{logit}_\theta(y_t \mid \boldsymbol{c}, \boldsymbol{x}, \boldsymbol{y}_{<t}) - \alpha\,\mathrm{logit}_\theta(y_t \mid \boldsymbol{x}, \boldsymbol{y}_{<t})]$$

```
def context_aware_sampling(*args
    generated_tokens = input_ids
    for _ in range(max_length):
        with torch.no_grad():
```

```
{'context': 'The current year is 2027',
 'question': 'How many world cups has
              Argentina won?'}
```

```
'answer':'Argentina has won 19 World Cups.'
```

```
def context_aware_sampling(*args):
    generated_tokens = input_ids.clone()
    for _ in range(max_length):
        with torch.no_grad():
```

```
for i in enumerate(Test_cases):
    result = paper_method(i[1])
    check(result, Results[i])
```

```
def context_aware_sampling(*args
    # You need to implement this
    # function based on paper
    Raise NotImplementedError
```

Figure 3: Dataset annotation pipeline of LMR-BENCH.

two different setups: standard prompting and LLM agent settings. The output function is evaluated via a combination of unit tests and the LLM-as-a-judge method, offering a multi-faceted evaluation of correctness and fidelity.

3.2 Data Collection

Figure 3 illustrates the collection pipeline of LMR-BENCH. In this section, we introduce each process in detail.

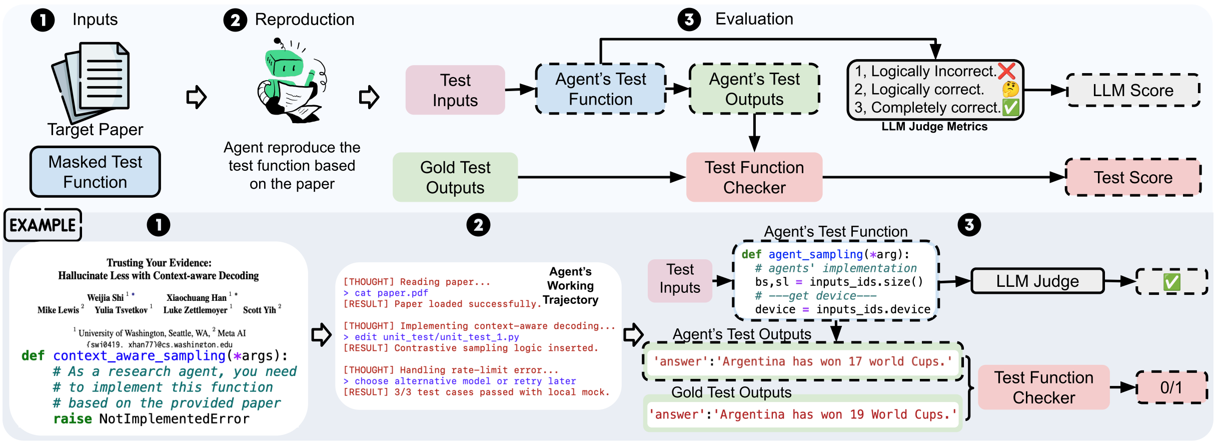
Paper Selection. We form a group of 12 experienced researchers in the LLM/NLP community as our co-authors and annotators (see Appendix B). Each annotator has been provided with a detailed annotation guideline, accompanied by examples.

We instruct annotators to collect research papers published within the past five years from top-tier NLP conferences, including ACL, EMNLP, NAACL, EACL, and COLING, and select appropriate papers for annotation. Each candidate paper must satisfy the following criteria: (1) **Methodological Focus:** The paper should retain method-driven research rather than survey papers or benchmarks; (2) **Reproducibility:** The proposed method should have an official, up-to-date repository with most issues resolved, ensuring that results are reproducible; (3) **Clarity and Complexity:** The method should be well-documented, with clear instructions and sufficient implementation details provided in the paper, and should involve a level of complexity beyond basic examples (e.g., simple prompts like "*Let's think step by step*"). Before the annotation process, we perform a manual review of the candidate papers selected by annotators to ensure their adherence to these criteria.

Reproducibility Check. For each selected paper, annotators are required to reproduce its official codebase to guarantee its reproducibility. Environmental setup remains a persistent challenge, particularly due to the dependency conflicts across different repositories, especially those with proprietary components. To mitigate this issue, we ask annotators to create a Dockerfile for each paper, following the repository's README. This ensures a consistent and functional execution environment. More specifically, this process involves pulling an official PyTorch Docker image[2], specifying the repository's dependencies, and incorporating any additional setup specific to the repository. Annotators are required to resolve issues encountered in the environmental setup. Repositories with unresolved errors are excluded from the process.

Data Annotation. During the data annotation process, annotators begin by selecting an algorithm presented in the paper and map it to the corresponding code block within the repository, ensuring the alignment between its theoretical description and code implementation. Next, for code blocks that are not organized into functions, they refactor the code into self-contained functions, ensuring them have appropriate inputs and outputs to encapsulate the core functionality. Then, for each aligned function, annotators meticulously document essential implementation details in a structured format as comments, including its primary objective, inputs and outputs, intra- and inter-file dependencies, and additional steps required for implementation (e.g., usage of external APIs). Following this documentation process, annotators craft detailed task instructions describing the algorithm's intended behavior. Subsequently, they generate a masked version of the function and save a golden file that will be utilized for evaluation.

[2]https://hub.docker.com/r/pytorch/pytorch



```
def context_aware_sampling(*args):
    # As a research agent, you need
    # to implement this function
    # based on the provided paper
    raise NotImplementedError
```

```
[THOUGHT] Reading paper...
> cat paper.pdf
[RESULT] Paper loaded successfully.

[THOUGHT] Implementing context-aware decoding...
> edit unit_test/unit_test_1.py
[RESULT] Contrastive sampling logic inserted.

[THOUGHT] Handling rate-limit error...
> choose alternative model or retry later
[RESULT] 3/3 test cases passed with local mock.
```

```
def agent_sampling(*arg):
  # agents' implementation
  bs,sl = inputs_ids.size()
  # ---get device---
  device = inputs_ids.device
```

```
'answer':'Argentina has won 17 world Cups.'
```

```
'answer':'Argentina has won 19 World Cups.'
```

Figure 4: Dataset evaluation pipeline of LMR-BENCH. The agent is presented with a target paper and a masked test function. After reproduction, the test function is evaluated in two stages. First, an LLM judge assesses the code for correctness and alignment with the paper's logic (in this example, the judge deems the implementation correct). Second, the test function is executed on labeled inputs and its outputs are automatically compared against the golden outputs. In this figure, since there is only one test case, the final score is $0/1$.

Unit Test Evaluation Preparation. Finally, annotators construct a unit test suite of around 3 test cases derived from the original datasets, where both the inputs and corresponding expected outputs are faithfully recorded during the reproduction phase. For scalable and consistent evaluation, annotators create task-specific evaluation scripts based on these pre-defined metrics that measure output accuracy. LLMs are leveraged to support the creation of unit tests to enhance efficiency. To account for the inherent variability that may be present in NLP implementations, we design checker functions specific to each task. For instance, we evaluate the value differences between predictions and ground truths for optimization tasks, while setting a threshold for the BERTScore (Zhang et al., 2020) in prompt engineering tasks. Following the annotation process, all annotations undergo rigorous human review and refinement to guarantee correctness and reproducibility.

3.3 Evaluation

The overall evaluation pipeline is illustrated in Figure 4. Since evaluating code quality involves complementary aspects of correctness and robustness, we employ two automated methods: (1) unit test evaluation and (2) LLM-as-a-judge evaluation.

Unit Test Evaluation. We first measure functional correctness using an automated unit test framework, following a protocol similar to LeetCode's (Liu et al., 2023; Zhao et al., 2025). As described in Section 3.2, we generate a suite of test cases and create a container as the testing environment. During evaluation, each candidate implementation is executed inside its own container, and we run the predefined test suite. We then compute the unit test accuracy as the fraction of problems for which all associated test cases pass.

LLM-as-a-Judge Evaluation. Unit tests ensure basic functionality but cannot capture code readability, style conformance, or subtle semantic discrepancies that do not trigger test failures (Tong and Zhang, 2024; Starace et al., 2025). To obtain a more granular assessment, we introduce LLM-as-a-judge as the second evaluation method. We present both the model-generated function and the reference solution to the LLM judge and evaluate the generated code from two distinct but complementary perspectives: algorithmic logic correctness and implementation correctness. The *algorithmic logic correctness* evaluation verifies that the algorithm's underlying mathematical design and theoretical logic are conceptually sound and consistent with the intended methodology, ensuring that for each valid input, the algorithm would produce an output meeting its formal specification. In parallel, the *implementation correctness* evaluation scrutinizes the code to ensure it faithfully realizes the intended algorithmic logic. This involves checking that the code's procedures and data handling strictly follow the algorithm's design and that it robustly handles edge cases (e.g., empty inputs or unexpected input formats), uses appropriate data types, and behaves reliably at runtime.

Based on the combined outcomes of these two evaluations, we classify the generated code's cor-

Benchmark	Pub.	Repo.	Unit	Docker	Source	Task
MLE-Bench (Chan et al., 2025)	✗	✗	✗	✗	Kaggle	Machine Learning Engineering
MLAgentBench (Huang et al., 2024)	✗	✓	✓	✗	Kaggle	Machine Learning Engineering
RepoBench (Liu et al., 2024)	✗	✓	✗	✗	GitHub	Code Auto-Completion
ML-Bench (Liu et al., 2023)	✗	✓	✓	✓	GitHub	Code Auto-Completion
SWE-bench (Jimenez et al., 2024)	✗	✓	✗	✗	GitHub	Resolve GitHub Issues
DebugBench (Tian et al., 2024)	✗	✗	✓	✗	LeetCode	Resolve Code Bugs
DSBench (Jing et al., 2025)	✗	✗	✓	✗	Kaggle	Data-Driven Discovery
ScienceAgentBench (Chen et al., 2025)	✓	✗	✓	✗	Research	Data-Driven Discovery
PaperBench (Starace et al., 2025)	✓	✓	✗	✓	Research	Reproduce ICML Papers
LMR-BENCH	✓	✓	✓	✓	Research	Reproduce LLM/NLP Papers

Table 1: Comparison between LMR-BENCH and existing benchmarks.

rectness into three categories: (1) *Logically Incorrect*: the code's foundational logic is flawed, rendering it incapable of producing correct results even with a perfect implementation; (2) *Logically Correct*: the design of the logic is sound in principle, but the implementation fails to realize that design accurately (e.g., bugs or improper handling of edge cases); (3) *Completely Correct*: the logic is conceptually sound and its implementation in code is faithful and error-free, satisfying all specified requirements. We report the percentage of implementations falling into each category to provide a more fine-grained performance analysis of LLM agents' implementation accuracy. We also include the prompt in Appendix C.

3.4 Comparison with Existing Benchmarks

Table 1 presents a systematic comparison between LMR-BENCH and nine existing benchmarks across four essential dimensions: derived from published research papers (**Pub.**), repository-level operation (**Repo.**), standard unit tests (**Unit**), and task-specific Docker environments (**Docker**). From the table, it is evident that LMR-BENCH is the only benchmark combining all four features, distinguishing it as a uniquely robust benchmark in contemporary LLM/NLP research. An example of LMR-BENCH is depicted in Appendix E.

4 Experiments

4.1 Experimental Setup

Our experiments are conducted under two settings: *standard prompting* and *LLM agent* settings, where backbone LLMs used include GPT-4o (Hurst et al., 2024), GPT-4.1[3], and o4-mini[4], serving as representative models for general-purpose, complex, and reasoning-oriented tasks, respectively.

Standard Prompting. In the *standard prompting* setting, since LLMs cannot directly process an entire repository as input, we design a straightforward pipeline for data pre-processing, extracting relevant information by parsing the paper and presenting the associated code, which is then formatted into a prompt and passed to the LLM (see Appendix D). Specifically, the prompt includes the JSON format of the paper parsed by PyMuPDF[5], the code of the goal file, the instruction, and the related code snippet in other files in the repository.

LLM Agent Setting. In contrast, the *LLM agent* setting aims for an end-to-end problem-solving approach. Here, the agent is provided with a folder consisting of the paper's PDF file and the code repository with masked functions as input, tasked with addressing the problem using any available method. OpenHands (Wang et al., 2025), a well-known coding agent, meets these requirements. Under this setup, the objective is to allow the agent to handle as much of the task as possible, with minimal intervention, i.e., only providing an instruction specifying which function needs to be implemented.

4.2 Results and Analysis

Table 2 presents the overall performance comparison between standard prompting and LLM agents. GPT-4.1 and o4-mini achieve the highest accuracy on unit tests. However, when it comes to LLM-as-a-judge, o4-mini significantly outperforms GPT-4.1 in the number of samples deemed correct. On the other hand, GPT-4o exhibits the weakest performance, underscoring its limits in code reproduction.

[3]https://openai.com/index/gpt-4-1/

[4]https://openai.com/index/introducing-o3-and-o4-mini/

[5]https://github.com/pymupdf/PyMuPDF

Model	Unit Test	LLM-as-a-Judge Evaluation			Human Evaluation		
	Accuracy	CC	LC	LI	CC	LC	LI
GPT-4o	39.3	17.9	10.7	71.4	25	14.3	60.7
GPT-4.1	42.9	7.1	28.6	64.3	32.1	28.6	39.3
o4-mini	42.9	25.0	21.4	53.6	35.7	32.1	32.1
OpenHands (GPT-4o)	25.0	7.1	7.1	85.7	21.4	17.9	60.7
OpenHands (GPT-4.1)	32.1	17.9	14.3	67.9	32.1	25	42.9
OpenHands (o4-mini)	35.7	35.7	14.3	50.0	39.3	21.4	39.3

Table 2: Evaluation results for standard prompting and LLM agent settings (CC: Completely Correct, LC: Logically Correct, LI: Logically Incorrect). All results are reported as percentages.

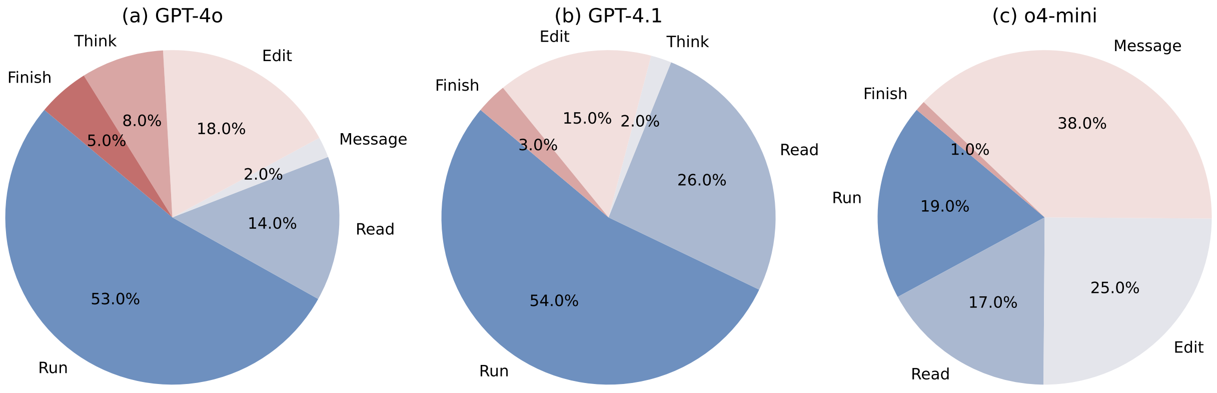


Figure 5: Action distribution in OpenHands agents with different backbone models.

Although the absolute unit test accuracy remains relatively low across all models, the category *logically correct* also reflects that the algorithms match the specification but contain implementation mistakes or omissions. This emphasizes the necessity of future models with enhanced abstract reasoning and better cross-file integration capabilities.

Compared to standard prompting, LLM agents show a slight decrease in accuracy across all models, with reductions of 14.3%, 10.8%, and 7.2% for GPT-4o, GPT-4.1, and o4-mini, respectively. However, these agents tend to produce a higher number of functions identified as correct. This observation underscores the enhanced ability of LLM agents to understand paper details and generate accurate functions, while also revealing their limitations in repository-level paper reproduction that passes unit tests. Challenges such as repository-level code understanding and dependency handling emerge as key areas for improvement, offering valuable directions for future research.

5 Further Analysis

5.1 Action Analysis for OpenHands Agents

To examine whether the distribution of actions of the agent is related to implementation success, we analyze the logs of OpenHands (Figure 5). The explanations of the actions are listed in Appendix F. Foundation models (GPT-4o, GPT-4.1) and reasoning models (o4-mini) exhibit markedly different interaction profiles. Although GPT-4o and GPT-4.1 share a similar overall action distribution, GPT-4.1 performs more concrete operations (e.g., code execution, file queries) and fewer conversational "think" steps. By contrast, the reasoning-oriented o4-mini devotes a larger fraction of its workflow to in-depth analysis before invoking execution steps.

Despite these behavioral differences, GPT-4.1 and o4-mini achieve comparable success rates under human evaluation. Moreover, when we split logs by outcome (passed vs. failed), the relative balance between analysis and execution remains consistent across all model types (see Figure 6), suggesting that it is the ratio of "think" to "run" actions, rather than their absolute counts, that best predicts successful code implementation. The data analysis shows that the action distribution is not related to the success or failure of the unit test since o4-mini and GPT-4.1 have different behavior patterns but show similar performance.

To quantify the impact of repository structure on implementation success, we fit a logistic regression model (Table 5 in Appendix G) with binary success as the dependent variable, as predictors we included:

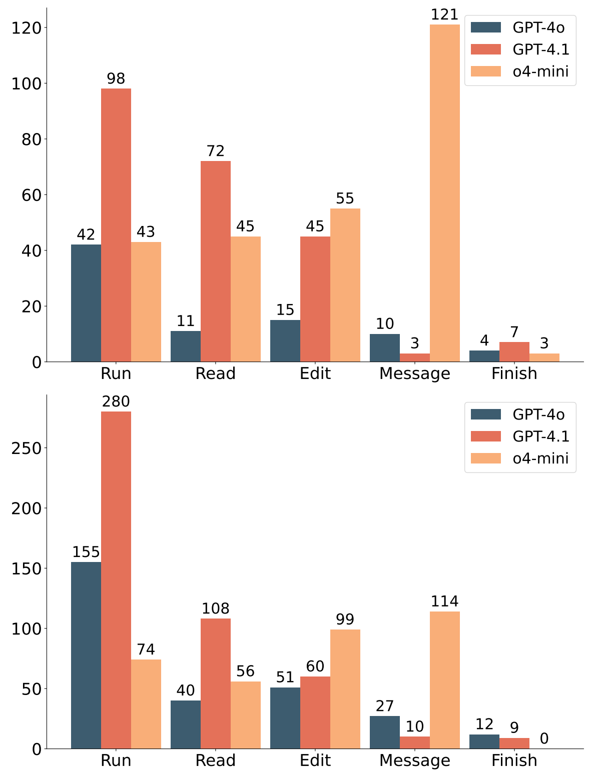


Figure 6: Action counts comparison for pass papers (*upper figure*) and fail papers (*bottom figure*).

- *Average Directory Depth*: Sum the depth of each file or folder (where depth is the number of edges from the repository root to that node) and divide by the total number of nodes;
- *Average Branch Factor*: Sum the number of immediate subdirectories for node and divide by the number of directories with at least one child;
- *Goal Function Length*: The number of source lines of code in the goal function file;
- *Dummy Indicators for Model Type*: The current model is GPT-4.1 and two dummy variables to represent GPT-4o and o4-mini.

The regression reveals that deeper directory hierarchies are strongly associated with success ($\beta = 0.8049$, $z = 3.232$, $p = 0.001$), whereas more highly branched structures significantly reduce success likelihood ($\beta = -0.6750$, $z = -2.995$, $p = 0.003$). Neither directory imbalance nor goal-function length reached statistical significance ($p > 0.10$), nor did the model-type indicators ($p > 0.30$). These results suggest that repositories organized into deeper but less divergent folder structures facilitate correct implementation, while shallow, highly forked hierarchies impede it.

5.2 Ablation Study

We conduct ablation with two different input settings to identify the contribution of the paper and code context: (1) "*Paper-Only*" setting, where the repository is entirely removed, so agents rely solely on paper context; (2) "*Paper + Goal File*" setting, where all repository files except the goal file are removed, exposing only the minimal context needed for reproduction. Results are reported in Table 3. Under "Paper-Only," GPT-4o and GPT-4.1 show lower performance, while o4-mini exhibits larger variance, consistent with its higher decoding temperature. Under "Paper + Goal File," isolating repository context has only a minor effect on aggregate performance. This might come from that most target functions are self-contained and require limited cross-file information, so agents are able to implement and test them successfully without additional repository context.

Input	Model	Unit Test	CC	LC	LI
Paper	GPT-4o	21.4	10.7	7.1	82.1
	GPT-4.1	32.1	28.6	14.3	57.1
	o4-mini	39.3	46.4	3.6	50.0
+ Goal File	GPT-4o	25.0	3.6	7.1	89.3
	GPT-4.1	28.6	7.1	14.3	78.6
	o4-mini	39.3	42.9	17.9	39.3

Table 3: Performance (%) of OpenHands agents under two input settings across different backbone models.

5.3 Error Analysis

We observe recurrent failure modes when OpenHands agents translate papers into code, each revealing a distinct challenge.

Unsuccessful Paper Parsing. Complex layouts and formula-heavy descriptions often yield incomplete or garbled inputs. For example, mathematical formulas and pseudo-code are often mis-extracted (missing symbols, mis-ordering), stripping critical information of the algorithm computation.

Incomplete Comprehension of Problem Context. The agent often demonstrates a shallow understanding of the paper context. Specifically, it can capture the general idea (e.g., "*apply an attention mechanism*") but struggles to expand this into concrete code behaviors. Key implementation details implied in texts (e.g., stopping criteria and parameter initializations) are sometimes omitted in the generated code. This indicates the models are not decomposing the task sufficiently and instead produce an incomplete or overly generic solution.

Lack of Robustness in Code Generation. Many errors originate from intrinsic weaknesses in the LLM's generated code, including syntax errors, log-

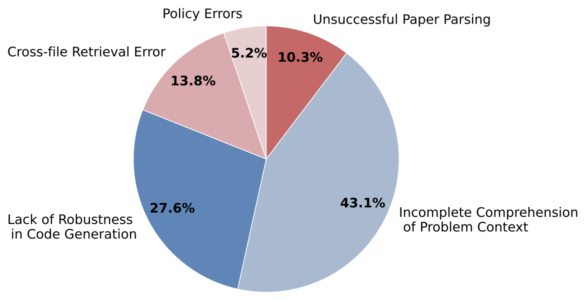


Figure 7: Error distribution of OpenHands on LMR-BENCH.

ical mistakes, and incorrect handling of edge cases. Employing code-verification tools and integrating iterative code refinement loops may significantly reduce such errors, improving overall robustness.

Cross-file Retrieval Error. Often, functions to be implemented rely on constants, helper functions, or class definitions located in other parts of the project. The code agent, with a limited context window, sometimes fails to recall or look up these dependencies but omits or redefines them using placeholders. These mistakes underscore the difficulty of repository-level code generation when not all relevant context fits in the prompt.

Policy Errors. In some trials, attempts are made to revise the prompt or inject additional context (such as appending external code snippets or altering the task description), which trigger LLM's safety policies or confuse its understanding of the task. In such cases, the model's performance degrades: it might refuse to continue, produce irrelevant output, or reset its earlier reasoning.

To facilitate better understanding, we include a concrete example for each error category in Appendix H. Additionally, we provide a statistical analysis of OpenHands' error distribution on our benchmark, as shown in Figure 7. The figure reveals a clear hierarchy of failure sources: nearly half (43.1%) arise from incomplete comprehension of the problem context, while 27.6% stem from brittle code that breaks under minor input or environment changes. Cross-file retrieval issues account for 13.8%, underscoring limitations in tracking multi-file dependencies.

5.4 Discussions on Evaluation Methods

We evaluate agent-generated code using two automated methods, supplemented by detailed human annotation (Appendix B). Table 2 shows that unit tests achieve the highest accuracy, followed by human evaluation with LLM-as-a-judge showing the worst performance, contrary to our expectations. As LLMs improve, they can generate executable and well-structured code, but there can be multiple valid implementations. Agents may generate correct solutions that diverge from the golden answer, which LLM-as-a-judge (biased toward surface similarity) can miss, whereas humans are more tolerant of variation but often overly permissive on "logically correct" category.

We also observe mismatches between LLM and human judgments. The LLM-human agreement is 62.5% with 9.5% direct conflicts, and the two annotators disagree more often with the LLM (an average of 16%), concentrated on specific papers. Most disagreements are human-correct but LLM-incorrect, while the reverse mainly occurs when execution fails. Overall, LLMs weigh local similarity while humans prioritize global reasoning, increasing divergence on complex cases (yet total agreement still exceeds 70%). We further collapse the evaluation into two categories—*Completely Correct* vs. *Not Completely Correct*. The aggregated results are reported in Table 6.

These findings support using both unit tests and LLM-as-a-judge. Unit tests capture *correctness* regardless of implementation strategy, while LLM-as-a-judge measures *alignment* with the reference and instruction *fidelity*.

6 Conclusion

We present LMR-BENCH, a benchmark designed to systematically evaluate the LLM agents' ability on reproducing language modeling research. To ensure an objective evaluation of the code reproduction results, we employ two distinct metrics: the accuracy of unit tests and the distribution of LLM-as-a-judge classifications of generated implementations. Experimental results on both standard generation and LLM agent settings reveal the persistent limitations in scientific reasoning and code synthesis of existing models, highlighting critical gaps in agent's ability to autonomously reproduce scientific research. In the future, we will focus on automatic or semi-automatic data collection and design more capable agents to improve reproduction outcomes.

Limitations

To ensure the high quality of our benchmark, the annotation cost is high, and scalability is difficult since it requires PhD-level expertise. How to enable automatic or semi-automatic data points collection is an open problem.

Ethical Considerations

We developed LMR-BENCH based on research papers from top-tier NLP conference proceedings and their publicly available code repositories. This project has been classified as exempt by our Institutional Review Board (IRB). All human annotations and evaluations were conducted by our co-authors who are researchers with substantial NLP research experience. The systems trained on our dataset are intended to augment—not replace—human decision-making in scientific research.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This research is supported in part by the National Science Foundation CAREER Grant IIS-2340435 and an Amazon Research Award. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily represent the views, either expressed or implied, of the U.S. Government.

References

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

Jinheon Baek, Sujay Kumar Jauhar, Silviu Cucerzan, and Sung Ju Hwang. 2025. Researchagent: Iterative research idea generation over scientific literature with large language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2025 - Volume 1: Long Papers, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, pages 6709–6738. Association for Computational Linguistics.

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. 2025. Mle-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. 2025. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Markus Flicke, Glenn Angrabeit, Madhav Iyengar, Vitalii Protsenko, Illia Shakun, Jovan Cicvaric, Bora Kargi, Haoyu He, Lukas Schuler, Lewin Scholz, Kavyanjali Agnihotri, Yong Cao, and Andreas Geiger. 2025. Scholar inbox: Personalized paper recommendations for scientists. *CoRR*, abs/2504.08385.

Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *CoRR*, abs/2312.13010.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2024. Mlagentbench: Evaluating language agents on machine learning experimentation. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Madry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alex Kirillov, Alex Nichol, Alex Paino, and 79 others. 2024. Gpt-4o system card. *CoRR*, abs/2410.21276.

Tal Ifargan, Lukas Hafner, Maor Kern, Ori Alcalay, and Roy Kishony. 2025. Autonomous llm-driven research — from data to human-verifiable research papers. *NEJM AI*, 2(1):AIoa2400555.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md. Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 4912–4944. Association for Computational Linguistics.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024a. A survey on large language models for code generation. *CoRR*, abs/2406.00515.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024b. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7):182:1–182:30.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. 2025. Dsbench: How far are data science agents from becoming data science experts? In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Mina Lee, Percy Liang, and Qian Yang. 2022. Coauthor: Designing a human-ai collaborative writing dataset for exploring language model capabilities. In *CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022 - 5 May 2022*, pages 388:1–388:19. ACM.

Long Li, Weiwen Xu, Jiayan Guo, Ruochen Zhao, Xingxuan Li, Yuqian Yuan, Boqiang Zhang, Yuming Jiang, Yifei Xin, Ronghao Dang, Deli Zhao, Yu Rong, Tian Feng, and Lidong Bing. 2024a. Chain of ideas: Revolutionizing research via novel idea development with LLM agents. *CoRR*, abs/2410.13185.

Ruochen Li, Liqiang Jing, Chi Han, Jiawei Zhou, and Xinya Du. 2024b. Learning to generate research idea with dynamic control. *CoRR*, abs/2412.14626.

Ruochen Li, Teerth Patel, Qingyun Wang, and Xinya Du. 2024c. Mlr-copilot: Autonomous machine learning research based on large language models agents. *CoRR*, abs/2408.14033.

Tianyang Liu, Canwen Xu, and Julian J. McAuley. 2024. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Yuliang Liu, Xiangru Tang, Zefan Cai, Junjie Lu, Yichi Zhang, Yanjun Shao, Zexuan Deng, Helan Hu, Zengxian Yang, Kaikai An, Ruijun Huang, Shuzheng Si, Sheng Chen, Haozhe Zhao, Zhengliang Li, Liang Chen, Yiming Zong, Yan Wang, Tianyu Liu, and 7 others. 2023. Ml-bench: Large language models leverage open-source libraries for machine learning tasks. *CoRR*, abs/2311.09835.

Zejie Liu, Xiaoyu Hu, Deyu Zhou, Lin Li, Xu Zhang, and Yanzheng Xiang. 2022. Code generation from flowcharts with texts: A benchmark dataset and an approach. In *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 6069–6077. Association for Computational Linguistics.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob N. Foerster, Jeff Clune, and David Ha. 2024. The AI scientist: Towards fully automated open-ended scientific discovery. *CoRR*, abs/2408.06292.

Ziming Luo, Zonglin Yang, Zexin Xu, Wei Yang, and Xinya Du. 2025. LLM4SR: A survey on large language models for scientific research. *CoRR*, abs/2501.04306.

Yubo Ma, Zhibin Gou, Junheng Hao, Ruochen Xu, Shuohang Wang, Liangming Pan, Yujiu Yang, Yixin Cao, and Aixin Sun. 2024. Sciagent: Tool-augmented language models for scientific reasoning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 15701–15736. Association for Computational Linguistics.

Hao Peng, Xiaozhi Wang, Jianhui Chen, Weikai Li, Yunjia Qi, Zimu Wang, Zhili Wu, Kaisheng Zeng, Bin Xu, Lei Hou, and Juanzi Li. 2023. When does in-context learning fall short and why? A study on specification-heavy tasks. *CoRR*, abs/2311.08993.

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. 2025. Paperbench: Evaluating ai's ability to replicate AI research. *CoRR*, abs/2504.01848.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and Maosong Sun. 2024. Debugbench: Evaluating debugging capability of large language models. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 4173–4198. Association for Computational Linguistics.

Weixi Tong and Tianyi Zhang. 2024. Codejudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP*

2024, Miami, FL, USA, November 12-16, 2024, pages 20032–20051. Association for Computational Linguistics.

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, and 2 others. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Yidong Wang, Qi Guo, Wenjin Yao, Hongbo Zhang, Xin Zhang, Zhen Wu, Meishan Zhang, Xinyu Dai, Min Zhang, Qingsong Wen, Wei Ye, Shikun Zhang, and Yue Zhang. 2024. Autosurvey: Large language models can automatically write surveys. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob N. Foerster, Jeff Clune, and David Ha. 2025. The AI scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *CoRR*, abs/2504.08066.

Zonglin Yang, Xinya Du, Junxian Li, Jie Zheng, Soujanya Poria, and Erik Cambria. 2024. Large language models for automated open-domain scientific hypotheses discovery. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 13545–13565. Association for Computational Linguistics.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. 2025. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *CoRR*, abs/2502.11089.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 13643–13658. Association for Computational Linguistics.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. Bertscore: Evaluating text generation with BERT. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Yuwei Zhao, Ziyang Luo, Yuchen Tian, Hongzhan Lin, Weixiang Yan, Annan Li, and Jing Ma. 2025. Codejudge-eval: Can large language models be good judges in code understanding? In *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu Dhabi, UAE, January 19-24, 2025*, pages 73–95. Association for Computational Linguistics.

Terry Yue Zhuo. 2024. Ice-score: Instructing large language models to evaluate code. In *Findings of the Association for Computational Linguistics: EACL 2024, St. Julian's, Malta, March 17-22, 2024*, pages 2232–2242. Association for Computational Linguistics.

A Paper Categories in LMR-Bench

Category	Definition
Feature Learning & Representation	Creating and refining vector representation of texts.
Neural Network Architectures	Designing building blocks within neural networks.
Prompt Engineering & Instruction Tuning	Crafting prompts or fine-tuning models to control and optimize model behaviors.
Information Extraction	Extracting structured knowledge from unstructured texts.
Data Augmentation	Augmenting training samples with curated strategies.
Training Objectives & Optimization	Designing loss functions or optimization algorithms to govern model training.
Decoding & Search Strategies	Employing inference-time algorithms for decoding and search in text generation.
Evaluation Metrics	Calculating quantitative measures of text generation results.
Interpretability & Explainability	Focusing on techniques that illuminate model internals and decision rationales.

Table 4: Paper categories within the LMR-Bench benchmark.

B Human Annotator Profile

To conduct the manual annotation evaluation, we first draw a random sample of 40 papers. We then recruit a panel of 30 subject-matter specialists drawn from a variety of institutions and including several faculty members to annotate their papers. Each annotated paper is examined independently by three different experts, ensuring the correctness of unit tests. On average, completing one paper annotation requires roughly five hours.

Qualification Standards. All annotators are selected to satisfy the reviewer expectations of premier NLP and machine learning venues. Each expert met at least *two* of the following conditions:

- holds a Ph.D. or has authored multiple peer-reviewed publications in a relevant discipline;
- has published a minimum of two *first-authored* papers in top-tier conferences or journals (AAAI, NeurIPS, ICML, ICLR, ACL, EMNLP, NAACL, etc.) within the past five years;
- has served as a reviewer for one of these venues, or has comparable research standing as demonstrated by citation metrics and scholarly record.

Evaluation Protocol. During the manual evaluation stage, each annotator are provided with the uniform evaluate criteria, and are be required to write a brief justification of their scoring. On average, it takes around ten minutes to complete a code evaluation.

C Prompt for LLM-as-a-Judge Evaluation

Prompt for LLM-as-a-Judge Evaluation

```
Instruction: {instruction}

You are an expert NLP software engineer tasked with evaluating the correctness of a function
implementation by comparing two code artifacts:

- Golden Reference ({golden file}):
  {golden content}

- Agent Implementation ({goal file}):
  {goal content}

Instructions:
1. Examine both implementations in detail, focusing on:
   - Logical correctness relative to the specification provided above
   - Handling of edge cases and error conditions
   - Subtle deviations such as off-by-one errors or missing checks

2. Classify your judgment into exactly one of the following categories:
   1. Incorrect Logic: the core algorithm deviates from the specification and produces wrong results
   2. Logic Correct but Subtle Errors: the main algorithm matches the specification, but there are other
    implementation mistakes or omissions
   3. Completely Correct: the implementation is fully faithful to the specification with no errors

3. For the chosen category, provide a concise rationale with two to four bullet points illustrating the
key discrepancies or confirmations

Output Format (JSON):
{
  category: <1 | 2 | 3>,
  rationale: [
    First key point ...,
    Second key point ...
  ]
}
```

D Prompt for Standard Prompting

Prompt for Standard Prompting

```
You are a code assistant.
Below is the entire Python source file.
Please implement only the function/method named <method_name>.
Return only its def line and indented body--no fences or explanations.

=== FILE BEGIN ===
<full_source_code>
=== FILE END ===

Paper (JSON):
<paper_json>

Instruction:
<instruction>

Related code for reference:
# Path: <relative_path_1>
<retrieval_content_1>
# Path: <relative_path_2>
<retrieval_content_2>
```

E Example of LMR-Bench

Example of the DPO Paper in LMR-Bench

```
Structure of data folder:
- 1-DPO
    - direct-preference-optimization (main code repository)
    - Golden_files (reference implementation files)
    - Dockerfile (defines Docker environment for unit-test evaluation)
    - info.json (metadata and implementation details)

Content of info.json:
{
    "instance_id": 1,
    "paper_name": "Direct Preference Optimization: Your Language Model is Secretly a Reward Model",
    "folder_name": "1-DPO",
    "paper_url": "https://arxiv.org/pdf/2305.18290",
    "year": 2023,
    "repo_url": "https://github.com/eric-mitchell/direct-preference-optimization",
    "repo_folder_name": "direct-preference-optimization",
    "implementations": [
        {
            "instruction": "Implement the preference_loss function in trainers.py based on the DPO loss
            mentioned in the paper and the code repository. You may ignore the following parameters: ipo
            , reference_free and label_smoothing.",
            "index": 1,
            "category": "Training Objectives & Optimization Techniques",
            "goal_file": "trainers.py",
            "goal_function": "preference_loss",
            "class_name": "",
            "golden_file": "golden_files/trainers_golden.py",
            "retrieval_context": [],
            "unit_test_file": "unit_test/unit_test_1.py"
        }
    ]
}
```

F Actions of the OpenHands Agent

- **Edit:** Edits a file using various sub-commands (e.g., view, create, str_replace, insert, undo_edit).
- **Finish:** Signals that the agent has completed the task.
- **Message:** We combine "message" and "think" here, treating "think" as an internal message step ("getting information and reasoning" without executing code).
- **Read:** Reads a file from a given path into the agent's context.
- **Run:** Combines all execution steps—whether via command-line or IPython—into a single "run" category, since both execute code in the sandbox.

G Logistic Regression Analysis

Variable	coef	std err	z	P>|z|	[0.025	0.975]
const	-0.4313	0.330	-1.306	0.192	-1.079	0.216
avg_dir_depth	0.8049	0.249	3.232	0.001	0.317	1.293
avg_branch_factor	-0.6750	0.225	-2.995	0.003	-1.117	-0.233
directory_imbalance	0.2426	0.226	1.071	0.284	-0.201	0.686
goal_function_length	-0.3478	0.225	-1.547	0.122	-0.788	0.093
model_name_gpt4o	-0.4458	0.475	-0.939	0.348	-1.376	0.484
model_name_o4mini	0.2807	0.476	0.590	0.555	-0.653	1.214

Table 5: Logistic regression analysis of repository structure and model type on implementation success.

H Example of Incorrect Implementations by OpenHands

Unsuccessful Paper Parsing. Blow is the agent implementation of Native Sparse Attention (Yuan et al., 2025). A comparison with the original algorithmic design reveals that the agent's implementation fails to capture the core algorithmic flow described in the paper, indicating a fundamental failure in parsing and understanding the intended methodology.

OpenHands' Implementation

```
def nsa(q: torch.Tensor,
    k: torch.Tensor,
    v: torch.Tensor,
    g_slc: torch.Tensor,
    g_swa: torch.Tensor,
    block_indices: torch.LongTensor,
    block_counts: Optional[Union[torch.LongTensor, int]] = None,
    block_size: int = 64,
    window_size: int = 0,
    scale: Optional[float] = None,
    cu_seqlens: Optional[torch.LongTensor] = None,
    head_first: bool = False) -> torch.Tensor:
    r"""
    Args:
        q (torch.Tensor): Queries of shape `[B, T, HQ, K]` if `head_first=False` else `[B, HQ, T, K]`.
        k (torch.Tensor): Keys of shape `[B, T, H, K]` if `head_first=False` else `[B, H, T, K]`.
            GQA is enforced here. The ratio of query heads (HQ) to key/value heads (H) must be a power
            of 2 and >=16.
        v (torch.Tensor): Values of shape `[B, T, H, V]` if `head_first=False` else `[B, H, T, V]`.
        g_slc (torch.Tensor): Gate score for selected attention of shape `[B, T, HQ]` if  `head_first=
        False` else `[B, HQ, T]`.
        g_swa (torch.Tensor): Gate score for sliding attentionof shape `[B, T, HQ]` if  `head_first=
        False` else `[B, HQ, T]`.
        block_indices (torch.LongTensor): Block indices of shape `[B, T, H, S]` if `head_first=False`
        else `[B, H, T, S]`.
            `S` is the maximum number of selected blocks for each query token, which is set to 16 in the
             paper.
        block_counts (Union[torch.LongTensor, int]): Number of selected blocks for each token.  If a
        tensor is provided, with shape `[B, T, H]` if `head_first=True` else `[B, T, H]`, each token can
         select the same number of blocks.If not provided, it will default to `S`, Default: `None`.
        block_size (int): Selected block size. Default: 64.
        window_size (int): Sliding window size. Default: 0.
        scale (Optional[int]): Scale factor for attention scores.
            If not provided, it will default to `1 / sqrt(K)`. Default: `None`.
        cu_seqlens (torch.LongTensor): Cumulative sequence lengths of shape `[N+1]` used for variable-
        length training,
            consistent with the FlashAttention API.
        head_first (Optional[bool]):
            Whether the inputs are in the head-first format. Default: `False`.
    Returns:
        o (torch.Tensor):
            Outputs of shape `[B, T, HQ, V]` if `head_first=False` else `[B, HQ, T, V]`.
    """

    # Determine scale if not provided
    if scale is None:
        scale = 1 / math.sqrt(q.size(-1))

    # Compute scaled dot-product attention
    q = q * scale

    # Hierarchical Sparse Strategy: Coarse-grained token compression
    # Assuming g_slc and g_swa are gate scores for selected and sliding attention
    # Apply gate scores to queries
    q_slc = q * g_slc.unsqueeze(-1)
    q_swa = q * g_swa.unsqueeze(-1)

    # Fine-grained token selection
    # Compute attention scores for selected blocks
    attn_scores_slc = torch.einsum('bthd,bshd->bths', q_slc, k)
    attn_scores_swa = torch.einsum('bthd,bshd->bths', q_swa, k)
    # Apply softmax to get attention probabilities
    attn_probs_slc = F.softmax(attn_scores_slc, dim=-1)
    attn_probs_swa = F.softmax(attn_scores_swa, dim=-1)

    # Compute attention outputs
    o_slc = torch.einsum('bths,bshd->bthd', attn_probs_slc, v)
    o_swa = torch.einsum('bths,bshd->bthd', attn_probs_swa, v)

    # Combine outputs from selected and sliding attention
    o = o_slc + o_swa

    return o
```

Incomplete Comprehension of Problem Context. OpenHands' implementation compared with the reference implementation, and the backbone of OpenHands is o4-mini. Parameter `beta` is not used correctly in OpenHands' implementation.

OpenHands' Implementation

```
def preference_loss(policy_chosen_logps: torch.FloatTensor,
                    policy_rejected_logps: torch.FloatTensor,
                    reference_chosen_logps: torch.FloatTensor,
                    reference_rejected_logps: torch.FloatTensor,
                    beta: float,
                    label_smoothing: float = 0.0,
                    ipo: bool = False,
                    reference_free: bool = False) -> Tuple[torch.FloatTensor, torch.FloatTensor, torch.
                    FloatTensor]:
    """Compute the DPO loss for a batch of policy and reference model log probabilities.

    Args:
        policy_chosen_logps: Log probabilities of the policy model for the chosen responses. Shape: (
        batch_size,)
        policy_rejected_logps: Log probabilities of the policy model for the rejected responses. Shape:
        (batch_size,)
        reference_chosen_logps: Log probabilities of the reference model for the chosen responses. Shape
        : (batch_size,)
        reference_rejected_logps: Log probabilities of the reference model for the rejected responses.
        Shape: (batch_size,)
        beta: Temperature parameter for the DPO loss, typically something in the range of 0.1 to 0.5. We
         ignore the reference model as beta -> 0.
        label_smoothing: conservativeness for DPO loss, which assumes that preferences are noisy (
        flipped with probability label_smoothing)
        ipo: If True, use the IPO loss instead of the DPO loss.
        reference_free: If True, we ignore the _provided_ reference model and implicitly use a reference
         model that assigns equal probability to all responses.

    Returns:
        A tuple of three tensors: (losses, chosen_rewards, rejected_rewards).
        The losses tensor contains the DPO loss for each example in the batch.
        The chosen_rewards and rejected_rewards tensors contain the rewards for the chosen and rejected
        responses, respectively.
    """
if reference_free:
        reference_chosen_logps = torch.zeros_like(reference_chosen_logps)
        reference_rejected_logps = torch.zeros_like(reference_rejected_logps)
    chosen_rewards = (policy_chosen_logps - reference_chosen_logps) / beta
    rejected_rewards = (policy_rejected_logps - reference_rejected_logps) / beta
    differences = chosen_rewards - rejected_rewards
    losses = -F.logsigmoid(differences)
    return losses, chosen_rewards, rejected_rewards
```

Reference Implementation

```
def preference_loss(policy_chosen_logps: torch.FloatTensor,
                    policy_rejected_logps: torch.FloatTensor,
                    reference_chosen_logps: torch.FloatTensor,
                    reference_rejected_logps: torch.FloatTensor,
                    beta: float,
                    label_smoothing: float = 0.0,
                    ipo: bool = False,
                    reference_free: bool = False) -> Tuple[torch.FloatTensor, torch.FloatTensor, torch.
                    FloatTensor]:
    """Compute the DPO loss for a batch of policy and reference model log probabilities.

    Args:
        policy_chosen_logps: Log probabilities of the policy model for the chosen responses. Shape: (
        batch_size,)
        policy_rejected_logps: Log probabilities of the policy model for the rejected responses. Shape:
        (batch_size,)
        reference_chosen_logps: Log probabilities of the reference model for the chosen responses. Shape
        : (batch_size,)
        reference_rejected_logps: Log probabilities of the reference model for the rejected responses.
        Shape: (batch_size,)
        beta: Temperature parameter for the DPO loss, typically something in the range of 0.1 to 0.5. We
         ignore the reference model as beta -> 0.
        label_smoothing: conservativeness for DPO loss, which assumes that preferences are noisy (
        flipped with probability label_smoothing)
        ipo: If True, use the IPO loss instead of the DPO loss.
        reference_free: If True, we ignore the _provided_ reference model and implicitly use a reference
         model that assigns equal probability to all responses.

    Returns:
        A tuple of three tensors: (losses, chosen_rewards, rejected_rewards).
        The losses tensor contains the DPO loss for each example in the batch.
        The chosen_rewards and rejected_rewards tensors contain the rewards for the chosen and rejected
        responses, respectively.
    """
    pi_logratios = policy_chosen_logps - policy_rejected_logps
    ref_logratios = reference_chosen_logps - reference_rejected_logps

    if reference_free:
        ref_logratios = 0

    logits = pi_logratios - ref_logratios  # also known as h_{\pi_\theta}^{y_w,y_l}

    if ipo:
        losses = (logits - 1/(2 * beta)) ** 2  # Eq. 17 of https://arxiv.org/pdf/2310.12036v2.pdf
    else:
        # Eq. 3 https://ericmitchell.ai/cdpo.pdf; label_smoothing=0 gives original DPO (Eq. 7 of https
        ://arxiv.org/pdf/2305.18290.pdf)
        losses = -F.logsigmoid(beta * logits) * (1 - label_smoothing) - F.logsigmoid(-beta * logits) *
        label_smoothing

    chosen_rewards = beta * (policy_chosen_logps - reference_chosen_logps).detach()
    rejected_rewards = beta * (policy_rejected_logps - reference_rejected_logps).detach()

    return losses, chosen_rewards, rejected_rewards
```

Lack of Robustness in Code Generation. Below is an example of OpenHands' implementation compared with the reference implementation, and the backbone of OpenHands is GPT-4o. There is two case in the reference implementation determined by parameter `if_tdpo2` whether to use method TDPO2. However, the implementation by OpenHands neglects to consider these cases.

OpenHands' Implementation

```
def tdpo_loss(chosen_logps_margin: torch.FloatTensor,
              rejected_logps_margin: torch.FloatTensor,
              chosen_position_kl: torch.FloatTensor,
              rejected_position_kl: torch.FloatTensor,
              beta: float, alpha: float = 0.5, if_tdpo2: bool = True) -> Tuple[torch.FloatTensor, torch.
              FloatTensor, torch.FloatTensor]:
    """Compute the TDPO loss for a batch of policy and reference model log probabilities.

    Args:
        chosen_logps_margin: The difference of log probabilities between the policy model and the
        reference model for the chosen responses. Shape: (batch_size,)
        rejected_logps_margin: The difference of log probabilities between the policy model and the
        reference model for the rejected responses. Shape: (batch_size,)
        chosen_position_kl: The difference of sequential kl divergence between the policy model and the
        reference model for the chosen responses. Shape: (batch_size,)
        rejected_position_kl: The difference of sequential kl divergence between the policy model and
        the reference model for the rejected responses. Shape: (batch_size,)
        beta: Temperature parameter for the TDPO loss, typically something in the range of 0.1 to 0.5.
        We ignore the reference model as beta -> 0.
        alpha: Temperature parameter for the TDPO loss, used to adjust the impact of sequential kl
        divergence.
        if_tdpo2: Determine whether to use method TDPO2, default is True; if False, then use method
        TDPO1.

    Returns:
        A tuple of two tensors: (losses, rewards).
        The losses tensor contains the TDPO loss for each example in the batch.
        The rewards tensors contain the rewards for response pair.
    """
    # Calculate the rewards using the Bradley-Terry model
    chosen_rewards = torch.sigmoid(chosen_logps_margin / beta - alpha * chosen_position_kl)
    rejected_rewards = torch.sigmoid(rejected_logps_margin / beta - alpha * rejected_position_kl)

    # Calculate the losses as the negative log likelihood of the rewards
    losses = -torch.log(chosen_rewards) - torch.log(1 - rejected_rewards)

    return losses, chosen_rewards, rejected_rewards
```

Reference Implementation

```
def tdpo_loss(chosen_logps_margin: torch.FloatTensor,
              rejected_logps_margin: torch.FloatTensor,
              chosen_position_kl: torch.FloatTensor,
              rejected_position_kl: torch.FloatTensor,
              beta: float, alpha: float = 0.5, if_tdpo2: bool = True) -> Tuple[torch.FloatTensor, torch.
              FloatTensor, torch.FloatTensor]:
    """Compute the TDPO loss for a batch of policy and reference model log probabilities.

    Args:
        chosen_logps_margin: The difference of log probabilities between the policy model and the
        reference model for the chosen responses. Shape: (batch_size,)
        rejected_logps_margin: The difference of log probabilities between the policy model and the
        reference model for the rejected responses. Shape: (batch_size,)
        chosen_position_kl: The difference of sequential kl divergence between the policy model and the
        reference model for the chosen responses. Shape: (batch_size,)
        rejected_position_kl: The difference of sequential kl divergence between the policy model and
        the reference model for the rejected responses. Shape: (batch_size,)
        beta: Temperature parameter for the TDPO loss, typically something in the range of 0.1 to 0.5.
        We ignore the reference model as beta -> 0.
        alpha: Temperature parameter for the TDPO loss, used to adjust the impact of sequential kl
        divergence.
        if_tdpo2: Determine whether to use method TDPO2, default is True; if False, then use method
        TDPO1.

    Returns:
        A tuple of two tensors: (losses, rewards).
        The losses tensor contains the TDPO loss for each example in the batch.
        The rewards tensors contain the rewards for response pair.
    """

    chosen_values = chosen_logps_margin + chosen_position_kl
    rejected_values = rejected_logps_margin + rejected_position_kl

    chosen_rejected_logps_margin = chosen_logps_margin - rejected_logps_margin


    if not if_tdpo2:
        logits = chosen_rejected_logps_margin - (rejected_position_kl - chosen_position_kl)    # tdpo1
    else:
        logits = chosen_rejected_logps_margin - alpha * (rejected_position_kl - chosen_position_kl.
        detach())  # tdpo2
    losses = -F.logsigmoid(beta * logits)

    chosen_rewards = beta * chosen_values.detach()
    rejected_rewards = beta * rejected_values.detach()

    return losses, chosen_rewards, rejected_rewards
```

Cross-file Retrieval Error. In the following example, `simpo_loss` depends on class-level attributes (e.g., `self.gamma_beta_ratio` and `self.beta`) defined outside the function body. As a result, the agent implementation fails to access these key parameters from `self`, leading to an incorrect implementation of the `simpo_loss` function.

OpenHands' Implementation

```
def simpo_loss(
    self,
    policy_chosen_logps: torch.FloatTensor,
    policy_rejected_logps: torch.FloatTensor,
) -> Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]:
    """Compute the SimPO loss for a batch of policy model log probabilities.

    Args:
        policy_chosen_logps: Log probabilities of the policy model for the chosen responses. Shape:
        (batch_size,)
        policy_rejected_logps: Log probabilities of the policy model for the rejected responses.
        Shape: (batch_size,)

    Returns:
        A tuple of three tensors: (losses, chosen_rewards, rejected_rewards).
        The losses tensor contains the SimPO loss for each example in the batch.
        The chosen_rewards and rejected_rewards tensors contain the rewards for the chosen and
        rejected responses, respectively.
    """

    # Calculate rewards using average log probabilities
    chosen_rewards = policy_chosen_logps.mean(dim=-1)
    rejected_rewards = policy_rejected_logps.mean(dim=-1)

    # Implement the Bradley-Terry objective with a target reward margin
    target_margin = 1.0  # This can be a hyperparameter
    margin = chosen_rewards - rejected_rewards - target_margin
    losses = -F.logsigmoid(margin)

    return losses, chosen_rewards, rejected_rewards
```

Reference Implementation

```
def simpo_loss(
    self,
    policy_chosen_logps: torch.FloatTensor,
    policy_rejected_logps: torch.FloatTensor,
) -> Tuple[torch.FloatTensor, torch.FloatTensor, torch.FloatTensor]:
    """Compute the SimPO loss for a batch of policy model log probabilities.

    Args:
        policy_chosen_logps: Log probabilities of the policy model for the chosen responses. Shape:
        (batch_size,)
        policy_rejected_logps: Log probabilities of the policy model for the rejected responses.
        Shape: (batch_size,)

    Returns:
        A tuple of three tensors: (losses, chosen_rewards, rejected_rewards).
        The losses tensor contains the SimPO loss for each example in the batch.
        The chosen_rewards and rejected_rewards tensors contain the rewards for the chosen and
        rejected responses, respectively.
    """
    pi_logratios = policy_chosen_logps - policy_rejected_logps
    logits = pi_logratios - self.gamma_beta_ratio

    if self.loss_type == "sigmoid":
        losses = (
            -F.logsigmoid(self.beta * logits) * (1 - self.label_smoothing)
            - F.logsigmoid(-self.beta * logits) * self.label_smoothing
        )
    elif self.loss_type == "hinge":
        losses = torch.relu(1 - self.beta * logits)
    else:
        raise ValueError(
            f"Unknown loss type: {self.loss_type}. Should be one of ['sigmoid', 'hinge']"
        )

    chosen_rewards = self.beta * policy_chosen_logps
    rejected_rewards = self.beta * policy_rejected_logps

    return losses, chosen_rewards, rejected_rewards
```

Policy Errors. Below is an OpenHands implementation using GPT-4o as its backbone. Notably, the target function remains unimplemented.

Agent implementation

```
def info_nce_loss(self, features):
    """
    Compute the InfoNCE loss for a batch of features.

    Args:
        features (torch.Tensor): Normalized feature representations from the encoder.
            Shape: (batch_size * n_views, feature_dim).
            It is assumed that features from different augmented views of the same image
            are stacked along the batch dimension.

    Returns:
        A tuple containing:
            - logits (torch.Tensor): Similarity scores for positive and negative pairs.
              Shape: (batch_size * n_views, 1 + num_negatives).
              Each row corresponds to one positive pair and multiple negative pairs.
            - labels (torch.Tensor): Ground truth labels where the first entry is the positive.
              Shape: (batch_size * n_views,). All entries are 0 since positive is first.
    """



    return logits, labels
```

The corresponding OpenHands logs below indicate that this run resulted in a `BadRequestError`, likely because prompt revisions made by OpenHands triggered the underlying LLM's safety mechanisms.

Log Messages

```
{"id": 84, "timestamp": "2025-05-16T18:44:29.250531", "source": "environment", "message": "", "
observation": "agent_state_changed", "content": "", "extras": {"agent_state": "error", "reason": "
BadRequestError: litellm.BadRequestError: OpenAIException - Invalid prompt: your prompt was flagged as
potentially violating our usage policy. Please try again with a different prompt: https://platform.
openai.com/docs/guides/reasoning#advice-on-prompting"}}
```

I Evaluation with Binary Labels

We further collapse the evaluation into two categories—*Completely Correct* vs. *Not Completely Correct*. The aggregated results are reported in Table 6. Under this binary labeling, GPT-4o and o4-mini show minimal change, whereas GPT-4.1 shifts notably—likely reflecting bias introduced by using GPT-4.1 as the judge LLM.

Agent	Completely Correct	Not Completely Correct
No Agent (GPT-4o)	14.29%	85.71%
No Agent (GPT-4.1)	21.43%	78.57%
No Agent (o4-mini)	25.00%	75.00%
OpenHands (GPT-4o)	7.14%	92.86%
OpenHands (GPT-4.1)	28.57%	71.43%
OpenHands (o4-mini)	39.29%	60.71%

Table 6: Binary-label evaluation on LMR-Bench (*Completely Correct* vs. *Not Completely Correct*).