MMSciCode: Real-world Evaluation of Multilingual Multi-Discipline Scientific Research Coding

Xue Xia[1]* **Zheyuan Yang**[2]†* **Arman Cohan**[3] **Yilun Zhao**[3]

[1]HKUST [2]Tongji University [3]Yale University

Abstract

We introduce MMSciCode, a comprehensive expert-level, multilingual multi-discipline benchmark for evaluating foundation models in scientific code generation. It includes 624 expert-annotated research coding problems spanning six core scientific disciplines. Compared to prior benchmarks, MMSciCode features three key advancements. First, it challenges models to integrate domain-specific knowledge with algorithmic reasoning to implement core functions from research papers. Second, each problem is meticulously annotated by domain experts through a rigorous paper-grounded process, with strict quality controls implemented to ensure dataset integrity and authenticity. Finally, each problem is equipped with comprehensive unit test suites and containerized environments, enabling reproducible and diagnostic evaluation of both functional correctness and domain validity. We conduct an extensive evaluation of 23 state-of-the-art foundation models and 2 coding agents on MMSciCode. We identify substantial performance gaps between models and human experts, providing actionable insights for advancing expert-level scientific code generation.

Data MMSciCode
Code MMSciCode

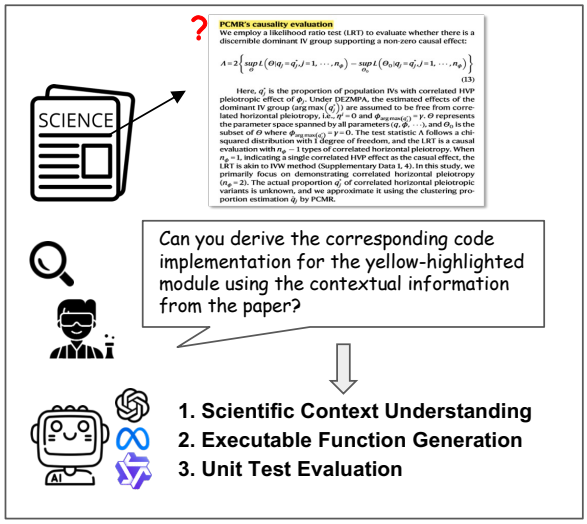

Figure 1: An overview of the MMSciCode benchmark.

1 Introduction

In recent years, researchers have developed a variety of benchmarks to evaluate foundation models' capabilities in code generation, with an emphasis on single-language (Yan et al., 2025), single-domain (Liu et al., 2025), and function-isolated programming tasks (Zhuo et al., 2025; Jain et al., 2025). While existing benchmarks effectively evaluate general programming proficiency, they fail to capture the distinctive challenges of scientific research coding, where successful implementation demands a deep integration of domain expertise and algorithmic reasoning (Tian et al., 2024). This gap is particularly critical because scientific programming inherently spans multiple disciplines, each requiring specialized knowledge, complex computational methods, and fluency across diverse programming languages and paradigms. Moreover, scientific coding often necessitates translating mathematical formulations and algorithmic descriptions from research papers into executable code while ensuring both domain-specific validity and computational correctness, which requires capabilities far beyond those assessed by existing code generation benchmarks.

To bridge this gap, we introduce MMSciCode, a comprehensive benchmark for measuring foundation models in expert-level, **M**ultilingual **M**ulti-discipline **Sci**entific **Code** generation. MMSciCode consists of 624 expert-annotated core functions extracted from 285 specialized-domain papers and their corresponding codebases,

*Equal contributions. †Project Lead. Correspondence to: Yilun Zhao (yilun.zhao@yale.edu)

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 33981–33999

Dataset	# Examples	Task Setting	Multi-lingual?	Multi-Discipline?	Objective Evaluation?
BLADE (Gu et al., 2024)	12	End-to-End Research	✗	✓	✓
MLRBench (Chen et al., 2025a)	201	End-to-End Research	✗	✗	✗
Paper2CodeBench (Seo et al., 2025)	90	End-to-End Research	✗	✗	✗
MLEBench (Chan et al., 2025)	75	End-to-End Research	✗	✗	✓
PaperBench (Starace et al., 2025)	20	Code Repository Generation	✗	✗	✗
ScienceAgentBench (Chen et al., 2025c)	44	File-Level Code Generation	✗	✓	✗
ResearchCodeBench (Hua et al., 2025)	20	Fill-in-the-blank Code Completion	✗	✗	✓
SciCode (Tian et al., 2024)	338	Function-Level Code Generation	✗	✓	✓
LMRBench (Yan et al., 2025)	23	Function-Level Code Generation	✗	✗	✓
SciReplicate-Bench (Xiang et al., 2025)	100	Function-Level Code Generation	✗	✗	✓
RECODE-H (Miao et al., 2025)	102	Function-Level Code Generation (Interactive)	✗	✓	✓
MMSciCode	**624**	**Function-Level Code Generation**	✓	✓	✓

Table 1: Comparison between MMSciCode and existing scientific research coding benchmarks.

spanning 38 subjects across six key disciplines: Computer Science, Physical Sciences, Earth and Environmental Sciences, Biological Sciences, Health Sciences, and Scientific Community and Society. To ensure both the breadth of domain knowledge and the depth of reasoning required for MMSciCode, we implement a paper-grounded data annotation process, where problems are curated from specialized research papers and validated by domain experts. Specifically, expert annotators first identify key algorithmic concepts from scientific papers in their respective fields, then locate the corresponding code implementations from associated codebases, and finally develop comprehensive unit test suites that require both domain expertise and expert-level reasoning to pass. Thorough data quality controls are implemented to ensure high quality of MMSciCode.

We conduct an extensive evaluation on MMSciCode, covering 23 frontier foundation models, spanning proprietary and open-source models. We also evaluate 2 coding agents, Claude Code (Sonnet 4.6) and Claude Code (Haiku 4.6). Across all evaluated non-agentic models, even the top-performing GPT-5.4 (proprietary) and Qwen3.5-397B-A17B (open-source) achieve only around 59.9% and 50.8% function-level accuracy, respectively, well below human open-book (86.7%) and oracle (93.3%) performance. Coding agents perform substantially better, with Claude Code (Sonnet 4.6) reaching 88.8%, yet they still fall behind human-level performance. These findings provide valuable guidance for advancing the development of scientific research coding.

2 Related Work

Code Generation. Code generation represents a long-standing and fundamental challenge in software engineering. Recent progress in large language models has spurred the development of numerous benchmarks and evaluation frameworks for assessing code synthesis capabilities. Early benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and DS-1000 (Lai et al., 2023) established standardized datasets for functional code synthesis and execution-based evaluation. Building upon these datasets, Yu et al. (2024) introduced HumanEval-Pro and MBPP-Pro, which emphasize compositional reasoning and multi-step synthesis beyond single-function completion. To further enhance realism and reduce contamination, Jain et al. (2025) proposed LiveCodeBench, a continuously updated benchmark that collects fresh programming tasks from competitive platforms. In addition, Zhuo et al. (2025) introduced BigCodeBench, a large-scale function-level benchmark spanning multiple domains and libraries, designed to evaluate models on diverse and compositionally complex code generation tasks.

Scientific Research Coding. Foundation models have emerged as powerful tools for automating research code generation across scientific domains (Chen et al., 2025b; Zhang et al., 2025). This capability has catalyzed a surge of benchmarks targeting research code generation—tasks that demand domain expertise (Peng et al., 2023; Liu et al., 2025), end-to-end automated research (Chan et al., 2025; Chen et al., 2025a; Seo et al., 2025), and executable implementations for scientific workflows (Yan et al., 2025; Chen et al., 2025c). Benchmarks such as SciCode (Tian et al., 2024) and ScienceAgentBench (Chen et al., 2025c) evaluate foundation models on scientific research coding tasks, ranging from function-level to agent-based file-level program generation and execution.

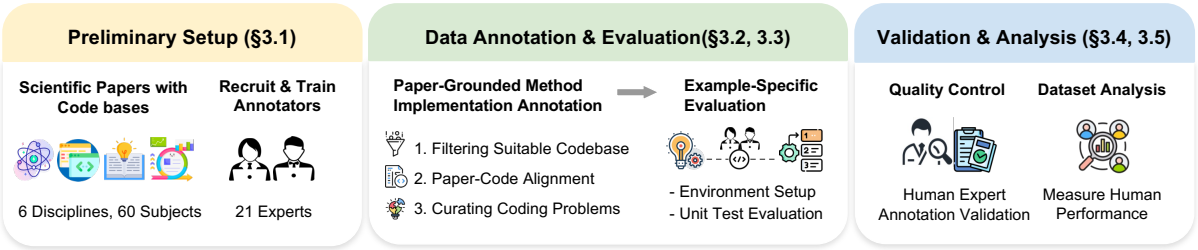

Figure 2: An overview of the MMSciCode benchmark construction pipeline.

Despite this progress, most scientific research coding benchmarks remain *single-language* (predominantly Python) and *single-domain*, lacking a unified evaluation that jointly spans multiple scientific disciplines and programming languages. Table 1 further distinguishes the difference between MMSciCode and existing benchmarks.

3 MMSciCode Benchmark Construction

We present MMSciCode, a comprehensive evaluation benchmark designed to measure progress in multilingual, multi-discipline scientific research coding within real-world research contexts. MMSciCode has the following key features: (1) **Breadth across scientific fields and programming languages**: MMSciCode spans six major scientific disciplines encompassing 38 diverse subjects and evaluates code generation across Python, C/C++, and R (§3.1). (2) **Paper-grounded scientific method implementation**: Each problem is derived from peer-reviewed scientific papers and their associated codebases, ensuring authenticity to real-world research (§3.2). (3) **Reproducible and diagnostic evaluation**: MMSciCode provides automated assessment through comprehensive unit tests designed by expert annotators, which measure functional correctness and require both domain expertise and sophisticated reasoning to pass (§3.3). Figure 2 provides an overview of the three stages involved in constructing MMSciCode, which is detailed in the following subsections.

3.1 Preliminary Setup

We first discuss the preliminary setup for the data construction process.

Source Paper Collection. We collect research articles from two sources: (1) Papers published at **ICLR 2025** that were selected as oral or spotlight presentations. These cover core AI fields such as machine learning, natural language processing, and computer vision. (2) Papers published in ***Nature Communications*** from 2024 onwards, covering 38 subjects across five core scientific disciplines: Physical Sciences, Earth and Environmental Sciences, Biological Sciences, Health Sciences, and Scientific Community and Society. This selection ensures broad coverage, extending beyond computer science into diverse scientific domains. We apply heuristic methods to exclude papers without publicly available code bases.

Expert Annotator Recruitment and Training. To ensure reliable data collection, we engaged 21 expert annotators, including 18 PhD students and three of the authors, from different scientific disciplines. All the annotators have substantial experience in research coding within their respective fields, and the papers and code they annotated are chosen to match their areas of expertise. Collectively, they dedicated approximately 1,400 hours to constructing MMSciCode. Each annotator is required to finish a training session to learn the annotation process before official annotation.

3.2 Paper-Grounded Method Implementation Annotation

To design problems that reflect real-world challenges of scientific research coding, we develop a paper-grounded method implementation pipeline, described in detail as follows:

Filtering Suitable Codebase. Annotators are first instructed to review the available code bases to determine whether they contain code implementations that correspond to the core algorithmic descriptions presented in the papers. To ensure annotation quality, we establish clear guidelines for identifying high-quality codebases that meet the standards. The codebase selection criteria include: (1) adequate code documentation and structure that facilitate understanding of the implementation logic, (2) robust code quality features, including meaningful variable naming, modular architecture, and well-defined functions, which support precise alignment

between theoretical concepts and their code realizations, and (3) adherence to a single programming language, with mixed-language projects excluded as unsuitable.

Paper–Code Alignment. In this phase, annotators identify core algorithms within the paper, such as key formula definitions, models, or innovative techniques. They then locate corresponding code snippets in the codebase that implement these algorithms by leveraging the codebase's Abstract Syntax Tree (AST). For each alignment, the original content is extracted from the paper to ensure clarity and fidelity to the source material, and the specific information of the function is recorded.

Curating Research Coding Problems. After identifying core functions and their alignments in the papers, annotators transform the extracted code implementations into structured coding problems. Where code snippets lack proper encapsulation, annotators refactor them into well-defined functions equipped with suitable parameters and return values. This problem curation process encompasses several key tasks, including cleaning the original code structure to remove the implementation code, formulating function descriptions that elucidate the algorithm concept and anticipated outcomes, and extracting core functions into standalone target files to address complex dependencies.

3.3 Example-Specific Evaluation Development

For each annotated paper-code alignment, we establish containerized testing environments using Docker to guarantee consistent dependencies and runtime configurations. This setup ensures that the selected core functions can reliably execute or perform inference within these isolated containers. The approach supports codebases in diverse programming languages via tailored, independent container configurations.

Developing Comprehensive Unit Tests. Finally, annotators develop a unit test suite for each curated coding problem to enable rigorous, automated evaluation of generated code completions. These tests are designed to cover a spectrum of input scenarios, including standard cases derived from the paper's examples, boundary conditions, and edge cases anticipated from the algorithmic descriptions. To validate the quality of these unit tests, we ensure that the target core functions extracted from the original codebases can pass all tests. By specifying precise expected outputs and integrating assertions for functional equivalence, the test suite facilitates objective assessment of correctness, efficiency, and adherence to the algorithm concept.

Building a Reliable Evaluation Pipeline. To enable robust and reproducible assessment of model-generated code, we construct an automated evaluation pipeline that standardizes the end-to-end process from function completion to performance measurement. For each coding problem, the pipeline first supplies the masked target function and its scientific context to the model to solicit a code completion. The generated function is then programmatically integrated into the original project by replacing only the masked region, ensuring that the overall codebase structure and dependencies are preserved. This replacement is fully automated and does not require manual intervention or further code modification. The modified project is executed within an isolated containerized environment, where a comprehensive suite of pre-defined unit tests is run to evaluate functional correctness and efficiency. The pipeline records outputs, execution time, and resource usage for each test, and systematically compares these results with those from the reference implementation. This approach yields quantitative metrics for correctness and performance, providing a scalable and objective framework for benchmarking code generation models across diverse scientific domains.

3.4 Data Quality Control

To ensure that MMSciCode maintains high scientific rigor and reproducibility, we implemented a multi-stage quality control pipeline. Each example underwent dual expert review to verify annotation accuracy, algorithmic soundness, and scientific validity. Problems that could be solved without genuine domain knowledge, lacked sufficient algorithmic complexity, or contained excessive dependencies that hindered independent testing were either refined or excluded. All revisions were tracked through iterative validation until the dataset satisfied both executability and domain-relevance criteria. After this process, we retained 624 functions and excluded 34 candidates from an initial pool of 658 examples. Among the initial examples, 54 required expert intervention, including both revised and excluded cases.

3.5 Dataset Analysis

Statistics	Value
Total Problems	624
Number of Subjects	38
Number of Source Papers	285
Scientific Domains (papers/problems)	6
Physical Sciences	44 / 101
Earth & Environmental Sciences	15 / 29
Biological Sciences	145 / 290
Health Sciences	17 / 30
Community & Society	4 / 7
Computer Science	60 / 167
Programming Languages (papers/problems)	3
Python	203 / 488
C/C++	22 / 44
R	60 / 92
Code Complexity	
Function Length (`avg/max`, lines)	54.1 / 817
Problem Description Length (`avg/max`, words)	77.7 / 213
Paper Context Length (`avg/max`, words)	3034.2 / 5963
Unit Tests per Problem (`avg/max`)	7.2 / 19
Test Suite Coverage (`avg`, %)	100
Algorithmic Complexity Level	
Basic	275
Intermediate	237
Advanced	112

Table 2: Key statistics of the MMSciCode benchmark.

Data Statistics. Table 2 presents the key statistics of MMSciCode. The dataset comprises 624 problems derived from 285 scientific papers spanning six disciplines and 38 subjects. It features realistic scientific code of moderate complexity and broad domain diversity, serving as a solid basis for cross-disciplinary code generation and reasoning.

Measuring Human Expert Performance. To contextualize model performance on MMSciCode, we conducted a human evaluation across multiple scientific disciplines. We randomly sampled 5 problems per discipline from the test set (30 in total) and recruited six participants, each specializing in one of the corresponding scientific domains, to independently solve the problems within their field of expertise. (1) In the **Closed-book** setting, participants solved their assigned problems within 3.5 hours without external resources, achieving an average accuracy of 60.0%. (2) In the **Open-book** setting, they were allowed to consult textbooks and online materials for four hours, increasing accuracy to 86.7%. (3) In the **Oracle** setting, participants revised all remaining errors with access to ground-truth references, reaching 93.3% accuracy, which approximates the upper bound of human performance on MMSciCode.

4 Experiments

This section describes our experimental setup and provides an analysis of the experimental results.

4.1 Experiment Setup

Evaluated Models. We evaluate the performance of state-of-the-art foundation models on our benchmark, including coding agents, proprietary models, and open-source models spanning a wide range of scales and reasoning capabilities. The coding agents evaluated are Claude Code (Sonnet 4.6) and Claude Code (Haiku 4.6) (Anthropic, 2025). Proprietary models evaluated are GPT-5.4 (OpenAI, 2026a), GPT-5.4-Mini (OpenAI, 2026b), Claude-Sonnet-4.6 (Anthropic, 2026) and Gemini-3-Flash-Preview (Google DeepMind, 2025). Open-source models include Llama-3.1-8B-Instruct (Dubey et al., 2024), Llama-4-Maverick-17B-128E-Instruct (Meta, 2025), Qwen2.5-Coder-7B-Instruct (Hui et al., 2024), Qwen3-30B-A3B-Instruct-2507, Qwen3-Coder-30B-A3B-Instruct, Qwen3-Coder-480B-A35B-Instruct (Qwen, 2025), Qwen3.5-9B, Qwen3.5-27B, Qwen3.5-35B-A3B, Qwen3.5-122B-A10B, Qwen3.5-397B-A17B (Qwen, 2026), Gemma-4-E4B-it, Gemma-4-26B-A4B-it, Gemma-4-31B-it (Google DeepMind, 2026), gpt-oss-20b, gpt-oss-120b (Agarwal et al., 2025), MiniMax-M2.7 (Minimax, 2026), Codestral-22B-v0.1 (Mistral AI, 2024), and Seed-Coder-8B-Instruct (ByteDance Seed et al., 2025). This selection covers both general-purpose instruction/coding models and specialized reasoning variants, accessed via standardized serving backends (API or vLLM). Detailed model specifications and versions are provided in Table 4.

Prompt Design. For each function completion task, the model is provided with a prompt containing the masked target function, its surrounding code context, and supplementary materials derived from the original paper, such as descriptive paragraphs, relevant formulas, a summary of the function's intended purpose, and key implementation points. This prompt formulation ensures that the model has sufficient scientific and contextual information to generate code that faithfully implements the algorithmic specifications described in paper.

Implementation Details. Upon receiving model-generated function completions, we programmatically replace the masked functions in the origi-

Model	Scientific Discipline							Programming Language				Overall
	Phys.	Earth	Bio.	Health	SocSci	CS	Avg.	Python	C/C++	R	Avg.	
Human Performance												
Human Oracle	100.0	100.0	80.0	100.0	100.0	80.0	93.3	90.0	100.0	100.0	96.7	93.3
Human Open-book	100.0	100.0	80.0	80.0	100.0	60.0	86.7	90.0	75.0	83.3	82.8	86.7
Human Closed-book	40.0	80.0	60.0	60.0	80.0	40.0	60.0	60.0	50.0	66.7	58.9	60.0
Coding Agents												
Claude Code (Sonnet 4.6)	92.1	93.1	88.3	90.0	100.0	86.2	91.6	89.8	79.5	88.0	85.8	88.8
Claude Code (Haiku 4.6)	83.2	89.7	82.4	86.7	100.0	80.2	87.0	83.4	72.7	83.7	79.9	82.7
Proprietary Models												
GPT-5.4	51.5	58.6	53.8	80.0	85.7	71.3	66.8	63.3	11.4	65.2	46.6	59.9
Claude-Sonnet-4.6	52.5	48.3	53.1	63.3	57.1	61.7	56.0	59.2	11.4	57.6	42.7	55.6
Gemini-3-Flash-Preview	48.5	55.2	47.9	60.0	42.9	61.7	52.7	53.9	11.4	65.2	43.5	52.6
GPT-5.4-Mini	45.5	48.3	45.5	60.0	71.4	65.9	56.1	53.9	13.6	60.9	42.8	52.1
Open-Source Models												
Qwen3.5-397B-A17B	54.5	48.3	43.4	66.7	71.4	58.1	57.1	53.3	11.4	56.5	40.4	50.8
Qwen3-Coder-480B-A35B-Instruct	41.6	41.4	48.3	60.0	57.1	56.3	50.8	51.2	11.4	59.8	40.8	49.7
gpt-oss-120b	47.5	41.4	45.5	60.0	42.9	56.9	49.0	51.4	11.4	56.5	39.8	49.4
Llama-4-Maverick-17B-128E-Instruct-FP8	42.6	48.3	43.1	56.7	28.6	58.1	46.2	48.4	13.6	60.9	41.0	47.8
MiniMax-M2.7	40.6	51.7	45.2	63.3	57.1	52.7	51.8	49.0	11.4	58.7	39.7	47.8
Gemma-4-31B-it	40.6	41.4	44.1	56.7	42.9	55.7	46.9	48.6	11.4	56.5	38.8	47.1
Qwen3.5-122B-A10B	39.6	44.8	43.8	50.0	42.9	56.3	46.2	48.2	11.4	56.5	38.7	46.8
Qwen3.5-27B	43.6	48.3	42.4	70.0	42.9	51.5	49.8	47.1	11.4	60.9	39.8	46.6
Qwen3-Coder-30B-A3B-Instruct	40.6	44.8	41.0	60.0	71.4	55.1	52.2	46.9	13.6	57.6	39.4	46.2
gpt-oss-20b	39.6	41.4	42.4	63.3	28.6	52.7	44.7	46.9	9.1	55.4	37.1	45.5
Qwen3.5-9B	42.6	44.8	41.4	56.7	57.1	51.5	49.0	45.9	13.6	57.6	39.0	45.4
Gemma-4-26B-A4B-it	36.6	44.8	43.8	50.0	42.9	51.5	44.9	45.1	11.4	60.9	39.1	45.0
Qwen3.5-35B-A3B	42.6	41.4	42.8	53.3	28.6	47.3	42.7	45.1	9.1	56.5	36.9	44.2
Qwen3-30B-A3B-Instruct-2507	39.6	44.8	40.0	53.3	28.6	48.5	42.5	42.6	13.6	58.7	38.3	42.9
Codestral-22B-v0.1	31.7	44.8	38.6	40.0	28.6	41.9	37.6	38.3	13.6	52.2	34.7	38.6
Gemma-4-E4B-it	28.7	37.9	34.1	36.7	28.6	40.1	34.3	34.8	11.4	47.8	31.3	35.1
Qwen2.5-Coder-7B-Instruct	25.7	34.5	31.4	30.0	28.6	35.9	31.0	31.4	9.1	44.6	28.4	31.7
Seed-Coder-8B-Instruct	21.8	31.0	27.9	26.7	14.3	32.9	25.8	27.9	6.8	40.2	25.0	28.2
Llama-3.1-8B-Instruct	18.8	27.6	25.2	23.3	14.3	29.9	23.2	24.6	4.5	39.1	22.7	25.3

Table 3: Accuracy of models on the MMSciCode test sets using CoT prompts. Model performance is ranked based on overall results. **Phys.** = Physical Sciences; **Earth** = Earth & Environmental Sciences; **Bio.** = Biological Sciences; **Health** = Health Sciences; **SocSci** = Scientific Community & Society; **CS** = Computer Science.

nal project files, strictly preserving the structure and dependencies. Each modified file is executed within a pre-configured Docker container to ensure reproducibility and consistent runtime environments. Comprehensive unit test suites are automatically run on the updated files, with outputs, execution time, and resource utilization systematically recorded and compared to golden reference implementations, yielding quantitative metrics for correctness and performance. For all experiments, we set the temperature to 1.0 and the maximum output length to 8,192 tokens. For reasoning models, the maximum output length is set to 16,384 tokens. For coding agents (Claude Code), we invoke the host-installed CLI with default model configurations; temperature and token limits are managed internally by the CLI and are not explicitly set.

4.2 Main findings

Table 3 presents the evaluated models' performance on the MMSciCode benchmark using CoT prompts, while Figure 3 illustrates a comparison between CoT reasoning and direct output prompt. Our key findings are as follows:

Our benchmark remains highly challenging for current foundation models, especially without agentic execution. Despite rapid progress in large language models, non-agentic foundation models still fall far short of human expert performance on MMSciCode. The best proprietary model, GPT-5.4, achieves 59.9% overall function-level accuracy, while the strongest open-source model, Qwen3.5-397B-A17B, reaches 50.8%. These results remain substantially below human open-book performance (86.7%) and the human oracle upper bound (93.3%), leaving gaps of 26.8 and 33.4 percentage points, respectively, for the best non-agentic model. Coding agents close much of this gap: Claude Code (Sonnet 4.6) achieves 88.8% overall accuracy, surpassing the human open-book setting but still trailing the oracle upper bound by 4.5 percentage points. These re-

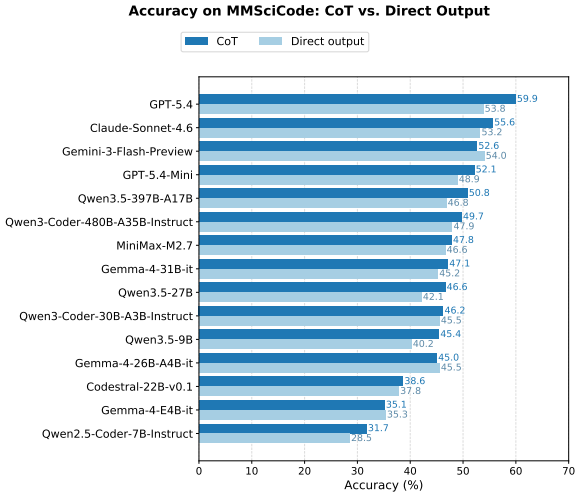

Figure 3: Comparison of model performance between CoT and direct output prompt.

sults suggest that MMSciCode requires more than fluent code synthesis; strong performance depends on sustained paper-grounded reasoning, faithful implementation of scientific algorithms, and effective interaction with executable environments.

Proprietary models retain the lead, but strong open-source models are increasingly competitive. Across all evaluated non-agentic models, proprietary models achieve higher average overall accuracy than open-source models. The proprietary models average 55.1% overall accuracy, compared with 42.8% for open-source models, yielding a 12.3 percentage-point gap. At the model level, GPT-5.4 leads all non-agentic systems with 59.9% overall accuracy, followed by Claude-Sonnet-4.6 at 55.6%, Gemini-3-Flash-Preview at 52.6%, and GPT-5.4-Mini at 52.1%. The strongest open-source model, Qwen3.5-397B-A17B, reaches 50.8%, approaching the lower end of the proprietary frontier. Other open-source models, including Qwen3-Coder-480B-A35B-Instruct (49.7%), gpt-oss-120b (49.4%), Llama-4-Maverick-17B-128E-Instruct-FP8 (47.8%), and MiniMax-M2.7 (47.8%), also deliver competitive performance. Nevertheless, coding agents form a distinct performance tier, with Claude Code with Sonnet 4.6 and Haiku 4.6 reaching 88.8% and 82.7%, respectively.

Performance varies substantially across scientific disciplines and programming languages. Model accuracy differs markedly across both scientific disciplines and programming languages. Among proprietary models, GPT-5.4 achieves the best overall performance and is especially strong on Health Sciences (80.0%), Scientific Community & Society (85.7%), and Computer Science (71.3%), while Claude-Sonnet-4.6 slightly leads on Physical Sciences (52.5%). Among open-source models, Qwen3.5-397B-A17B performs best overall and attains particularly strong results on Physical Sciences (54.5%), Health Sciences (66.7%), Scientific Community & Society (71.4%), and Computer Science (58.1%). Across programming languages, C/C++ is the most severe bottleneck for non-agentic models: even the best non-agentic C/C++ score is only 13.6%, far below the corresponding Python and R scores. By contrast, coding agents remain comparatively robust across languages, with Claude Code (Sonnet 4.6) achieving 89.8% on Python, 79.5% on C/C++, and 88.0% on R. This uneven performance highlights the importance of evaluating scientific code generation across both disciplinary contexts and programming languages, rather than relying on Python-centric or single-domain benchmarks.

4.3 Error analysis

To better understand the strengths and limitations of frontier foundation models, we conduct a human-in-the-loop error analysis and case study, which reveals the following primary error categories:

Environment and Execution Errors (33%). Models generate code that is incompatible with the target execution environment. Common failures include importing unavailable packages, calling deprecated functions, using mismatched API signatures, or assuming unsupported library versions. These errors show that LLMs still struggle to follow runtime constraints and may hallucinate plausible but non-functional library interactions.

Algorithmic Reasoning Errors (19%). These errors reflect misunderstandings of the algorithms described in the source papers. Models may implement incorrect algorithmic logic, misinterpret key steps, or fail to translate mathematical formulations into executable procedures. Typical failures include incorrect loop conditions, wrong control flow, and missing components such as convergence criteria or optimization updates. This category highlights limitations in multi-step reasoning over complex scientific algorithms.

Implementation Errors (18%). This category covers local coding mistakes such as syntax errors, type mismatches, incorrect variable usage,

and logic bugs. Although often more superficial than algorithmic reasoning failures, these errors directly affect functional correctness. Common examples include off-by-one errors, wrong operators, scope issues, and improper exception handling.

Data Structure and Schema Errors (14%). Models frequently choose inappropriate data structures or misinterpret the schema, dimensionality, or layout of scientific data. Since scientific computation often involves structured high-dimensional inputs, such as time series, spatial grids, or hierarchical records, these errors can invalidate downstream computation.

Domain Knowledge Errors (9%). These errors arise from insufficient understanding of scientific concepts and discipline-specific constraints. Models may violate physical conservation laws, ignore biological plausibility constraints, misapply statistical assumptions, or misunderstand domain-specific notation. This underscores the difficulty of integrating scientific expertise into code LLMs.

Others (7%). This category includes incomplete solutions, misread problem requirements, numerical precision issues, hardcoded values that should be parameterized, and failures on edge cases.

4.4 Why Coding Agents Perform Better

The large gap between coding agents and one-shot prompting shows that MMSciCode evaluates not only model reasoning, but also the ability to ground implementations through executable feedback. On MMSciCode, Claude Code (Sonnet 4.6) achieves 88.8% function-level accuracy, while the strongest one-shot setting, GPT-5.4 with CoT prompting, reaches 59.9%. The same trend appears within the Claude family: Claude-Sonnet-4.6 improves only slightly from direct prompting to CoT prompting, from 53.2% to 55.6%, whereas Claude Code (Sonnet 4.6) reaches 88.8% as a coding agent. This suggests that the main gain comes from the coding-agent harness rather than from model scale or a stronger single-pass reasoning trace.

Coding agents can verify and repair implementations through self-generated feedback. Coding agents do not access the held-out unit tests used for final evaluation, which are reserved only for scoring. Their advantage instead comes from creating auxiliary verification signals during problem solving. They can write sanity checks, construct toy inputs from paper formulas, test edge cases implied by the function contract, run import and type checks, and execute candidate implementations locally. When these checks expose stack traces, shape mismatches, numerical inconsistencies, or invalid return structures, the agent can localize the issue and revise the code.

Coding agents obtain stronger repository grounding than prompt-only models. Although our CoT prompt includes paper excerpts, the masked source file, sibling functions, symbol mappings, and return contracts, it remains a compressed view of the repository. Coding agents can inspect additional context when needed, including adjacent files, helper functions, import definitions, type declarations, and configuration files. This helps resolve details that are often underspecified in papers or prompts, such as expected array shapes, object fields, numerical conventions, and library-specific behavior. In contrast, one-shot models must infer these details from the prompt alone.

The main bottleneck is executable grounding rather than longer reasoning alone. CoT prompting can improve a single forward pass by making reasoning more explicit, but it does not provide external evidence that the implementation satisfies the target environment. Coding agents combine reasoning with tool use, execution feedback, and iterative repair. The gap between Claude-Sonnet-4.6 with CoT prompting and Claude Code (Sonnet 4.6) indicates that the dominant advantage comes from this verification-and-revision loop.

5 Conclusion

We present MMSciCode, a high-quality, multilingual, and multi-disciplinary benchmark designed to evaluate the scientific research coding capabilities of foundation models. Each example in MMSciCode is meticulously annotated from scratch by human experts. Our evaluation of 23 frontier foundation models and 2 coding agents reveals that even the best-performing non-agentic model, GPT-5.4, and the best coding agent, Claude Code (Sonnet 4.6), still lag behind human experts. Through comprehensive error analyses and case studies, we uncover persistent challenges highlighted by MMSciCode, providing valuable insights for advancing the research-driven scientific coding abilities of future foundation models.

Limitations

This work focuses on evaluating frontier models for scientific research coding rather than proposing new training methods to enhance model performance on our tasks. Therefore, MMSciCode provides diagnostic evidence about current model capabilities, but does not directly study how different training strategies, data recipes, or optimization methods affect scientific programming performance. Moreover, our benchmark currently covers three programming languages: Python, C/C++, and R. Although these languages are widely used across scientific domains, other languages such as MATLAB, Julia, and Fortran are also common in specific research communities. As a result, our findings may not fully generalize to all scientific programming settings.

References

OpenAI Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Hai-Biao Bao, Boaz Barak, Ally Bennett, Tyler Bertao, N. Archer Brett, Eugene Brevdo, Greg Brockman, Sébastien Bubeck, Cheng Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, C. Dvorak, K Fives, Vlad Fomenko, T. Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, A. B. Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominika Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran N. Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub W. Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, K. Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, Daniel Sculley, Harshit S. Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Phil Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy E. Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Hu Zhang, Eddie Zhang, and Shengjia Zhao. 2025. gpt-oss-120b&gpt-oss-20b model card.

Anthropic. 2025. Claude code. https://docs.anthropic.com/en/docs/claude-code/overview.

Anthropic. 2026. Claude sonnet 4.6 system card.

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

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, Yifan Huang, Guanghan Ning, Xierui Song, Jiaze Chen, Siyao Liu, Kai Shen, Liang Xiang, and Yonghui Wu. 2025. Seed-Coder: Let the code model curate data for itself.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. 2025. Mle-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*.

Hui Chen, Miao Xiong, Yujie Lu, Wei Han, Ailin Deng, Yufei He, Jiaying Wu, Yibo Li, Yue Liu, and Bryan Hooi. 2025a. Mlr-bench: Evaluating AI agents on open-ended machine learning research. *CoRR*, abs/2505.19955.

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

Q. Chen, M. Yang, L. Qin, et al. 2025b. Ai4research: A survey of artificial intelligence for scientific research. *arXiv preprint arXiv:2507.01903*.

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. 2025c. 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.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony S. Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aur'elien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie

Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab A. AlBadawy, E I Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriele Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guanglong Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Ju-Qing Jia, Kalyan Vasuden Alwala, K. Upasani, Kate Plawiak, Keqian Li, Kenneth Heafield, Kevin R. Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuen ley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Ma hesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melissa Hall Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Niko lay Bashlykov, Nikolay Bogoychev, Niladri S. Chatterji, Olivier Duchenne, Onur cCelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasić, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ron nie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sa hana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Chandra Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stéphane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Vir ginie Do, Vish Vogeti, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whit ney Meers, Xavier Martinet, Xiaodong Wang, Xiaoqing Ellen Tan, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yiqian Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zhengxu Yan, Zhengxing Chen, Zoe Papakipos, Aaditya K. Singh, Aaron Grattafiori, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adi Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alex Vaughan, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Franco, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Po-Yao (Bernie) Huang, Beth Loyd, Beto de Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Damon Civin, Dana Beaty, Daniel Kreymer, Shang-Wen Li, Danny Wyatt, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Firat Ozgenel, Francesco Caggioni, Francisco (Paco) Guzmán, Frank J. Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Govind Thattai, Grant Herman, Grigory Sizov, Guangyi Zhang, Guna Lakshminarayanan, Hamid Shojanazeri, Han Zou, Hannah Wang, Han Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Igor Molybog, Igor Tufanov, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kaixing(Kai) Wu, U KamHou, Karan Saxena, Karthik Prasad, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kun Huang, Kunal Chawla, Kushal Lakhotia, Kyle Huang, Lailin Chen, Lakshya Garg, A Lavender, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Maria Tsimpoukelli, Martynas Mankus, Matan Hasson, Matthias Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Mun ish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navy ata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikolay Pavlovich Laptev, Ning Dong, Ning Zhang, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pe dro Rittner, Philip Bontrager, Pierre

Roux, Piotr Dollár, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Rohan Maheswari, Russ Howes, Ruty Rinott, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, S. Yu. Sidorov, Satadru Pan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Zha, Shiva Shankar, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Kumar Gupta, Sung-Bae Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Kohler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Andrei Poenaru, Vlad T. Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xia Tang, Xiaofang Wang, Xiaojian Wu, Xiaolan Wang, Xide Xia, Xilun Wu, Xinbo Gao, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu Wang, Yuchen Hao, Yundi Qian, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, and Zhiwei Zhao. 2024. The llama 3 herd of models.

Google DeepMind. 2025. Gemini 3 flash - model card.

Google DeepMind. 2026. Gemma 4.

Ken Gu, Ruoxi Shang, Ruien Jiang, Keying Kuang, Richard-John Lin, Donghe Lyu, Yue Mao, Youran Pan, Teng Wu, Jiaqian Yu, Yikun Zhang, Tianmai M. Zhang, Lanyi Zhu, Mike A Merrill, Jeffrey Heer, and Tim Althoff. 2024. BLADE: Benchmarking language model agents for data-driven science. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 13936–13971, Miami, Florida, USA. Association for Computational Linguistics.

Tianyu Hua, Harper Hua, Violet Xiang, Benjamin Klieger, Sang T. Truong, Weixin Liang, Fan-Yun Sun, and Nick Haber. 2025. Researchcodebench: Benchmarking llms on implementing novel machine learning research code. *CoRR*, abs/2506.02314.

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, Shanghaoran Quan, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *ArXiv*, abs/2409.12186.

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*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Siyu Liu, Jiamin Xu, Beilin Ye, Bo Hu, David J Srolovitz, and Tongqi Wen. 2025. Mattools: Benchmarking large language models for materials science tools. *arXiv preprint arXiv:2505.10852*.

Meta. 2025. Llama 4.

Chunyu Miao, Henry Peng Zou, Yangning Li, Yankai Chen, Yibo Wang, Fangxin Wang, Yifan Li, Wooseong Yang, Bowei He, Xinni Zhang, Dianzhi Yu, Hanchen Yang, Hoang H. Nguyen, Yue Zhou, Jie Yang, Jizhou Guo, Wenzhe Fan, Chin-Yuan Yeh, Panpan Meng, Liancheng Fang, Jinhu Qi, Wei-Chieh Huang, Zhengyao Gu, Yuwei Han, Langzhou He, Yuyao Yang, Yinghui Li, Hai-Tao Zheng, Xue Liu, Irwin King, and Philip S. Yu. 2025. RECODE-H: A benchmark for research code development with interactive human feedback. *CoRR*, abs/2510.06186.

Minimax. 2026. Minimax m2.7: Early echoes of self-evolution.

Mistral AI. 2024. Codestral.

OpenAI. 2026a. Gpt-5.4.

OpenAI. 2026b. Gpt-5.4-mini.

Cheng Peng, Xi Yang, Aokun Chen, Kaleb E Smith, Nima PourNejatian, Anthony B Costa, Cheryl Martin, Mona G Flores, Ying Zhang, Tanja Magoc, et al. 2023. A study of generative large language model for medical research and healthcare. *NPJ digital medicine*, 6(1):210.

Qwen. 2025. Qwen3-coder: Agentic coding in the world.

Qwen. 2026. Qwen3.5: Towards native multimodal agents.

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

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. In *Forty-second International Conference on Machine Learning*.

Minyang Tian, Luyu Gao, Shizhuo Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, et al. 2024. Scicode: A research coding benchmark curated by scientists. *Advances in Neural Information Processing Systems*, 37:30624–30650.

Yanzheng Xiang, Hanqi Yan, Shuyin Ouyang, Lin Gui, and Yulan He. 2025. Scireplicate-bench: Benchmarking llms in agent-driven algorithmic reproduction from research papers. *CoRR*, abs/2504.00255.

Shuo Yan, Ruochen Li, Ziming Luo, Zimu Wang, Daoyang Li, Liqiang Jing, Kaiyu He, Peilin Wu, George Michalopoulos, Yue Zhang, et al. 2025. Lmr-bench: Evaluating llm agent's ability on reproducing language modeling research. *arXiv preprint arXiv:2506.17335*.

Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. 2024. Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation. *CoRR*.

Haoxuan Zhang, Ruochi Li, Yang Zhang, Ting Xiao, Jiangping Chen, Junhua Ding, and Haihua Chen. 2025. The evolving role of large language models in scientific innovation: Evaluator, collaborator, and scientist. *arXiv preprint arXiv:2507.11810*.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*.

A Experiment Setup

A.1 Configuration of Evaluated Models

Table 4 details the configuration of each evaluated model. We use the default configurations provided by the official implementations of all models. For consistency, the decoding temperature is fixed at 1.0, with a maximum output length of 8,192 tokens for standard models and 16,384 tokens for reasoning-oriented models across all experiments. All inferences are conducted using either the official API endpoints or local `vLLM` deployment for most open-source models. Our experiments are reproducible on a workstation equipped with multiple NVIDIA A100 (80 GB) GPUs.

A.2 Chain-of-Thought and Direct Output Prompts

A.2.1 Chain-of-Thought Prompt

Figure 4 illustrates the Chain-of-Thought prompt used in the non-agentic one-shot evaluations, where the model is guided to produce intermediate reasoning steps before generating the final solution code.

A.2.2 Direct Output Prompt

Figure 5 illustrates the Direct Output prompt used in the non-agentic one-shot evaluations, where the model is asked to directly generate the final solution code without intermediate reasoning traces.

Organization	Model	Release	Version	Reasoning?	Serving Backend
			Proprietary Models		
OpenAI	GPT-5.4	2026-03	`gpt-5.4`	✓	API
	GPT-5.4-Mini	2026-03	`gpt-5.4-mini`	✓	API
Anthropic	Claude-Sonnet-4.6	2026-02	`claude-sonnet-4-6`	✓	API
Google	Gemini-3-Flash-Preview	2025-12	`gemini-3-flash-preview`	✓	API
			Open-Source Models		
Meta	Llama-3.1-8B-Instruct	2024-07	`meta-llama/Llama-3.1-8B-Instruct`	✗	vLLM
	Llama-4-Maverick-17B-128E-Instruct-FP8	2025-04	`meta-llama/Llama-4-Maverick-17B-128E-Instruct-FP8`	✗	API
Qwen	Qwen2.5-Coder-7B-Instruct	2024-09	`Qwen/Qwen2.5-Coder-7B-Instruct`	✗	vLLM
	Qwen3-30B-A3B-Instruct-2507	2025-07	`Qwen/Qwen3-30B-A3B-Instruct-2507`	✗	vLLM
	Qwen3-Coder-30B-A3B-Instruct	2025-07	`Qwen/Qwen3-Coder-30B-A3B-Instruct`	✗	vLLM
	Qwen3-Coder-480B-A35B-Instruct	2025-07	`Qwen/Qwen3-Coder-480B-A35B-Instruct`	✗	API
	Qwen3.5-9B	2026-02	`Qwen/Qwen3.5-9B`	✓	vLLM
	Qwen3.5-27B	2026-02	`Qwen/Qwen3.5-27B`	✓	vLLM
	Qwen3.5-35B-A3B	2026-02	`Qwen/Qwen3.5-35B-A3B`	✓	vLLM
	Qwen3.5-122B-A10B	2026-02	`Qwen/Qwen3.5-122B-A10B`	✓	vLLM
	Qwen3.5-397B-A17B	2026-02	`Qwen/Qwen3.5-397B-A17B`	✓	API
Google	Gemma-4-E4B-it	2026-04	`google/gemma-4-E4B-it`	✓	vLLM
	Gemma-4-26B-A4B-it	2026-04	`google/gemma-4-26B-A4B-it`	✓	vLLM
	Gemma-4-31B-it	2026-04	`google/gemma-4-31B-it`	✓	vLLM
OpenAI	gpt-oss-20b	2025-08	`openai/gpt-oss-20b`	✓	vLLM
	gpt-oss-120b	2025-08	`openai/gpt-oss-120b`	✓	vLLM
MiniMax	MiniMax-M2.7	2026-04	`MiniMaxAI/MiniMax-M2.7`	✓	API
Mistral AI	Codestral-22B-v0.1	2024-05	`mistralai/Codestral-22B-v0.1`	✗	vLLM
ByteDance	Seed-Coder-8B-Instruct	2025-05	`ByteDance-Seed/Seed-Coder-8B-Instruct`	✗	vLLM

Table 4: Details of the foundation models evaluated in MMSciCode.

Chain-of-Thought Prompt

SYSTEM

You are an expert research software engineer and computational scientist. Implement functions from peer-reviewed scientific papers rigorously: map the paper's notation to code, respect the exact signature and return contract, use only available imports or the standard runtime, and do not invent behavior unsupported by the paper or codebase.

USER

 # Paper-grounded function implementation ({LANGUAGE})

You are given a function-implementation task from a peer-reviewed paper, relevant paper excerpts, and the original source file with the target body masked. Fill in the masked function body so that it realizes the paper algorithm.

Paper metadata

- **Title**: `{paper_title}`
- **Domain**: `{paper_subject}`
- **Language**: `{LANGUAGE}`

Paper excerpts

`{paper_context}`

Target function: `{function_name}`

- **Paper section**: `{paper_section}`
- **Description**: `{description}`
- **Paper quote**: "`{article_reference}`"
- **Formula**: `{formula}`
- **Symbol map**: `{key_terms_bullets}`
- **Signature**: `{signature_block}`
- **Return contract**: `{return_signatures_bullets}`
- **In-scope sibling functions**: `{siblings_bullets}`

Masked source

`{masked_file_content}`

Your task

First reason through the implementation, then output the complete function code.

Phase 1 — Reasoning. Write inside `<reasoning>` tags. Cover: algorithm summary, paper-symbol-to-code-variable mapping, input/output contract, edge cases, numerical or runtime constraints, and a concise execution plan.

Phase 2 — Implementation. After the reasoning block, return one fenced code block per target function, tagged as: `{LANGUAGE}:{function_name}`.

Rules. Match the declared signature exactly; preserve return arity, ordering, and types; use only existing imports or the standard runtime; honor the paper formula and symbol mapping; preserve required side effects; do not print, log, or add tests; after `</reasoning>`, output only the code blocks.

Figure 4: Chain-of-thought prompt template for paper-grounded function completion. The template provides paper metadata, excerpts, target-function context, return constraints, sibling functions, and masked source code, then requires a structured reasoning phase before the final drop-in implementation.

Direct Output Prompt

SYSTEM

You are an expert research software engineer and computational scientist. Implement functions from peer-reviewed scientific papers rigorously: map the paper's notation to code, respect the exact signature and return contract, use only available imports or the standard runtime, and do not invent behavior unsupported by the paper or codebase.

USER

 # Paper-grounded function implementation ({LANGUAGE})

You are given a function-implementation task from a peer-reviewed paper, relevant paper excerpts, and the original source file with the target body masked. Fill in the masked function body so that it realizes the paper algorithm.

Paper metadata

- **Title**: `{paper_title}`
- **Domain**: `{paper_subject}`
- **Language**: `{LANGUAGE}`

Paper excerpts
`{paper_context}`

Target function: `{function_name}`

- **Paper section**: `{paper_section}`
- **Description**: `{description}`
- **Paper quote**: "`{article_reference}`"
- **Formula**: `{formula}`
- **Symbol map**: `{key_terms_bullets}`
- **Signature**: `{signature_block}`
- **Return contract**: `{return_signatures_bullets}`
- **In-scope sibling functions**: `{siblings_bullets}`

Masked source
`{masked_file_content}`

Your task

Produce the complete implementation of every function listed above.

Rules. Match the declared signature exactly; preserve return arity, ordering, and types; use only existing imports or the standard runtime; honor the paper formula and symbol mapping; preserve required side effects; do not print, log, or add tests.

Output format. Return one fenced code block per target function, tagged as: `{LANGUAGE}:{function_name}`. Only output the code blocks — no prose, explanation, or extra text.

Figure 5: Direct output prompt template for paper-grounded function completion. The template provides paper metadata, excerpts, target-function context, return constraints, sibling functions, and masked source code, then requires the model to output only the final drop-in implementation.

B Error Analysis

Target Function: get_categories

Error Category: Environment and Execution Errors

Model / Mode: GPT-5.4 / CoT

Paper Title: "The Geometry of Categorical and Hierarchical Concepts in Large Language Models"

Function Description: "Constructs a WordNet-based taxonomic hierarchy for categorical and hierarchical concept analysis; collects descendant word sets, augments word forms, and filters categories by model vocabulary coverage."

Formula / Algorithm: "WordNet method: define each synset as attribute w, define Y(w) as descendant words of w, augment word forms, and keep synsets with at least 50 Gemma-vocabulary words."

Paper Reference: "WordNet synsets are used as binary concepts; Y(w) contains descendant words, with plural/tense/case augmentation and vocabulary coverage filtering."

Inserted Code: focused excerpt around the defect

```python
import os
import json
import networkx as nx
from code.hierarchical.from_models import get_vocab  # required for vocab filtering

import inflect
p = inflect.engine()

# ... lines 9-33 omitted ...
def get_categories(noun_or_verb='noun', model_name='gemma'):
    # Load WordNet data from JSON file in the same directory
    base_dir = os.path.dirname(__file__)
    json_path = os.path.join(base_dir, f"wordnet_{noun_or_verb}.json")
    with open(json_path, 'r') as f:
        wordnet_data = json.load(f)

    # Build graph G (hyponym hierarchy)
    G = nx.DiGraph()
# ... lines 43-62 omitted ...
    # Get model vocabulary
    vocab_dict, _ = get_vocab(model_name)
    vocab_set = set(vocab_dict.keys())
# ... lines 66-85 omitted ...
```

X Wrong package qualifier on import

The model writes from code.hierarchical.from_models import get_vocab, mirroring the on-disk path code/hierarchical/from_models.py. But the repository has no code/__init__.py, so code is not a Python package; Python falls back to the stdlib single-file code module, the hierarchical submodule lookup fails, and every test errors out at import time. The correct import is from hierarchical.from_models import ... since the test harness adds code/ itself to sys.path.

Execution Results: focused detailed error output

```
### Test entry 1: python:unit_test.test_estimate_single_dir_from_embeddings,unit_test.test_get_categories,unit_test.test_get_g
status=env_broken passed=0/37 failed=0 errors=37 duration_s=9.908
...
test_estimate_single_dir_from_embeddings (unittest.loader._FailedTest) ... ERROR
test_get_categories (unittest.loader._FailedTest) ... ERROR
test_get_g (unittest.loader._FailedTest) ... ERROR
...
ERROR: test_estimate_single_dir_from_embeddings (unittest.loader._FailedTest)
...
    import hierarchical.category_golden as test_module
  File "code/hierarchical/__init__.py", line 6, in <module>
    from hierarchical.category import *
  File "code/hierarchical/category.py", line 11, in <module>
    from code.hierarchical.from_models import get_vocab  # required for vocab filtering
ModuleNotFoundError: No module named 'code.hierarchical'; 'code' is not a package

...
ERROR: test_get_categories (unittest.loader._FailedTest)
...
ERROR: test_get_g (unittest.loader._FailedTest)
...
Ran 3 tests in 0.001s
...
FAILED (errors=3)
```

Figure 6: An error case of Environment and Execution Error.

Target Function: entropy

Error Category: Algorithmic Reasoning Errors

Model / Mode: GPT-5.4 / CoT

Paper Title: "Model-free estimation of completeness, uncertainties, and outliers in atomistic machine learning using information theory"

Function Description: "Computes information entropy of atomistic datasets using Gaussian-kernel density estimation and batched distance calculations."

Formula / Algorithm: "Entropy via KDE: H = -sum_i p(x_i) log p(x_i). The reference implementation normalizes kernel sums by N before log: log(p_x / N)."

Paper Reference: "The paper estimates information entropy from feature-vector distributions using a kernel K_h with bandwidth h over n atomic environments."

Inserted Code: focused excerpt around the defect

```python
def entropy(
    x: np.ndarray,
    h: Union[float, List[float]] = DEFAULT_BANDWIDTH,
    batch_size: int = DEFAULT_BATCH,
):
# ... lines 47-65 omitted ...
    if type(h) is np.ndarray:
        p_x = kernel_sum_multi_bandwidth(x, x, h=h, batch_size=batch_size)
        return -np.mean(np.log(p_x), axis=1)
    else:
        p_x = kernel_sum(x, x, h=h, batch_size=batch_size)
        return -np.mean(np.log(p_x))
# ... lines 72-583 omitted ...
```

X Wrong mathematical formula implementation

The inserted code applies -mean(log(p_x)) instead of -mean(log(p_x / N)). Missing normalization causes negative entropy values in custom bandwidth and identical-input edge cases.

Execution Results: focused detailed error output

```
### Test entry 1: python:unit_test.unit_test_1,unit_test.unit_test_2
status=fail passed=14/16 failed=2 errors=0 duration_s=14.036
...
Test entropy calculation with custom bandwidth parameter ... FAIL
...
Test edge cases for entropy function ... FAIL
...
FAIL: test_custom_bandwidth (unit_test.unit_test_1.TestEntropy)
...
During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "quests/unit_test/unit_test_1.py", line 86, in test_custom_bandwidth
    self.fail(f"Custom bandwidth entropy test failed: {e}")
AssertionError: Custom bandwidth entropy test failed: -0.1948274083688962 not greater than or equal to 0.0

...
FAIL: test_edge_cases (unit_test.unit_test_1.TestEntropy)
...
During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "quests/unit_test/unit_test_1.py", line 154, in test_edge_cases
    self.fail(f"Edge cases test failed: {e}")
AssertionError: Edge cases test failed: -1.3862943611198906 not greater than or equal to 0.0

...
Ran 16 tests in 9.895s
...
FAILED (failures=2)
```

Figure 7: An error case of Algorithmic Reasoning Error.

Target Function: instant_simpShearXz

Error Category: Implementation Errors

Model / Mode: GPT-5.4 / CoT

Paper Title: "Jamming is a first-order transition with quenched disorder in amorphous materials sheared by cyclic quasistatic deformations"

Function Description: "Applies instantaneous simple shear deformation in the xz-plane as part of the cyclic athermal quasistatic shear protocol."

Formula / Algorithm: "Affine shear: x' = x + gamma z. The full implementation must also update the simulation box transform and invalidate stale update/neighbor-list state."

Paper Reference: "The method uses cyclic athermal quasistatic shear; local metadata references gamma_max = 0.7 unless otherwise specified."

Inserted Code: focused excerpt around the defect

```
void instant_simpShearXz(Box *box, Particle *particle, Update *update, double
deltaGamma) {
    // 1. Validate input parameters
    if (!box || !particle || deltaGamma < 0.0 || deltaGamma > 0.7) {
        return;
    }

    // 2. Apply shear transformation to each particle
    for (int i = 0; i < particle->nAtom; i++) {
        // Get current position
        double x = particle->pos[i][0];
        double z = particle->pos[i][2];

        // Apply affine transformation: x' = x + γz
        particle->pos[i][0] = x + deltaGamma * z;
    }

    // 3. Normalize box geometry if necessary
    // Note: For simple shear in 2D, no transformation of box edges is needed
    // as the affine transformation preserves the volume and shape of the simulation
box

    // 4. Update box structure
    // Not applicable for this simple shear transformation
}
```

X Missing required side effects

The code shifts particle x positions but skips boxH update and update/force/neighbor-list invalidation, so the behavioral tests detect stale simulation state.

Execution Results: focused detailed error output

```
### Test entry 1: c:gcc-fsyntax-only
status=compile_ok passed=1/1 failed=0 errors=0 duration_s=0.162
...
### Test entry 2: c:realtest realtest_FIRE
status=fail passed=0/1 failed=1 errors=0 duration_s=0.397
...
### Test entry 3: c:realtest realtest_simpShearXz
status=fail passed=2/4 failed=2 errors=0 duration_s=0.011
notes:
- behavioral test against real Library-... sources

FAIL test_boxH_offdiagonal: H[0,z] 0 != expected 1
FAIL test_flag_side_effects: Edone/Pdone/Tdone not cleared (1 1 1)
  [PASS] position_shift
  [FAIL] boxH_offdiagonal_update
  [FAIL] flag_side_effects
  [PASS] zero_shear

Results: 2/4 passed
```

Figure 8: An error case of Implementation Error.

Target Function: forward

Error Category: Data Structure and Schema Errors

Model / Mode: GPT-5.4 / CoT

Paper Title: "Self-supervised learning for characterising histomorphological diversity and spatial RNA expression prediction across 23 human tissue types"

Function Description: "Implements RNAPath multiple-instance learning: histology tile representations are processed by parallel gene regressors and aggregated with mean pooling."

Formula / Algorithm: "MIL aggregation: patch-level gene-expression predictions are computed by parallel regressors; whole-tissue expression is produced by mean pooling."

Paper Reference: "RNAPath takes histology tile representations and outputs spatial expression maps plus whole-tissue expression predictions."

Inserted Code: focused excerpt around the defect

```python
def forward(self, h, genes_list=None, return_patch_expression=False):
    if h.numel() == 0:
        if genes_list is None:
            return torch.zeros(self.n_classes, device=h.device)
        else:
            return torch.zeros(len(genes_list), device=h.device)

    h = self.dropout(h)

    if genes_list is None:
        classifiers = self.classifiers
    else:
        classifiers = [self.classifiers[i] for i in genes_list]

    patch_logits = torch.cat([classifier(h) for classifier in classifiers], dim=1)
    logits = patch_logits.mean(dim=0)

    if return_patch_expression:
        patch_logits = patch_logits.reshape(-1, len(classifiers))

    if genes_list is not None:
        logits = logits[genes_list]
        if return_patch_expression:
            patch_logits = patch_logits[:, genes_list]

    if return_patch_expression:
        return logits, patch_logits.reshape(-1, len(genes_list))
    else:
        return logits,
```

X Wrong return schema

The trailing comma in return logits, creates a singleton tuple. Tests expect a torch.Tensor, so tensor operations, shape access, and type checks fail.

Execution Results: focused detailed error output

```
ERROR: test_deterministic_output (unit_test.unit_test_2.TestRNAPathForward)
Test deterministic output for known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "RNAPath/unit_test/unit_test_2.py", line 78, in test_deterministic_output
    self.assertTrue(torch.allclose(result1, result2, rtol=1e-5, atol=1e-6))
TypeError: allclose(): argument 'input' (position 1) must be Tensor, not tuple
...
ERROR: test_edge_cases (unit_test.unit_test_2.TestRNAPathForward)
Test edge cases and boundary conditions
----------------------------------------------------------------------
Traceback (most recent call last):
  File "RNAPath/unit_test/unit_test_2.py", line 128, in test_edge_cases
    self.assertEqual(result_single.shape, (self.n_genes,))
AttributeError: 'tuple' object has no attribute 'shape'
...
Ran 13 tests in 0.176s
...
FAILED (failures=2, errors=4)
```

Figure 9: An error case of Data Structure and Schema Error.