EternalMath: A Living Benchmark of Frontier Mathematics that Evolves with Human Discovery

Anonymous ACL submission

Abstract

Current evaluations of mathematical reasoning in large language models (LLMs) are dominated by static benchmarks, either adapted from competition problems or curated through costly expert effort. These resources are valuable, but they are difficult to refresh and provide limited coverage of recent research-level mathematics. We present **EternalMath**, a theorem-grounded pipeline that turns constructive or quantitative results from vetted mathematical work into executable and verifiable reasoning tasks. The pipeline uses arXiv-hosted or journal-hosted copies for access while admitting sources only when their authority is supported by published or accepted venues, expert-curated whitelists. It extracts theorem kernels, instantiates them into parameterized problem templates, and generates Python/SymPy verifiers for deterministic answer checking. The released snapshot contains 891 templates and 782 verified instances, focusing on executable tasks rather than open-ended proof generation. Experiments with frontier LLMs reveal substantial remaining headroom, and the failure analysis shows that many errors arise during theorem interpretation and mathematical modeling rather than final arithmetic.

1 Introduction

Large Language Models (LLMs) have recently demonstrated substantial progress in mathematical reasoning, driven by stronger architectures (McLeish et al., 2024), reinforcement learning methods (Wang et al., 2024; Shao et al., 2025), and prompting strategies (Wei et al., 2022). This progress has reduced the discriminative value of foundational benchmarks such as GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021). It also makes static public datasets harder to interpret, because benchmark performance can reflect both reasoning ability and exposure to similar public problems (Balunović et al., 2025).

Recent expert-curated evaluations such as FrontierMath (Glazer et al., 2024) and Humanity's Last Exam (HLE) (Phan et al., 2025) address part of this problem by introducing high-difficulty tasks. These resources provide useful evaluation signals, but they remain expensive to create, slow to refresh, and fixed after release. A complementary evaluation methodology should therefore be able to follow newly available mathematical work while preserving verifiable ground truth.

This paper targets a specific setting: *frontier constructive and quantitative mathematics*. We do not ask models to produce formal proofs or open-ended symbolic derivations. Instead, source theorem, proposition, and construction kernels with explicit input-output instantiations are turned into executable problem instances. The scope keeps the benchmark close to research mathematics while preserving deterministic answer checking.

We introduce a multi-agent pipeline that transforms vetted or published mathematical work, accessed through arXiv-hosted or journal-hosted versions, into parameterized and verifiable problem instances. Unlike prior automated methods that rely on unstructured LLM synthesis (Yu et al., 2024; Tang et al., 2024), the generation process is anchored in constructive source kernels from the literature. Each retained template specifies parameter domains, constraints, solution logic, and executable verification code.

This methodology supports versioned refreshes, deterministic ground truth, and targeted evaluation by mathematical subfield.

Using this pipeline, we construct **EternalMath**, a benchmark of research-level constructive and quantitative mathematical reasoning tasks. Frontier LLMs still struggle with these tasks, especially when a solution requires translating a specialized theorem into a concrete multi-step computation.

In summary, our contributions are threefold:

- We propose an executable, theorem-grounded benchmark construction pipeline that transforms recent vetted or published mathematical results into parameterized and verifiable problem instances, supporting refreshable evaluation without large-scale expert authoring.

- We introduce **EternalMath**, an instantiation of this pipeline that yields a dynamically extensible mathematical reasoning benchmark grounded in real-world research problems. The released snapshot mitigates exact leakage and static-benchmark saturation through versioned refreshes, while not ruling out broader theorem-family overlap.[1]

- We evaluate frontier LLMs on **EternalMath** and show that the released snapshot remains unsaturated, especially on tasks requiring theorem interpretation and mathematical modeling.

2 Related Work

Mathematical reasoning evaluation has moved from school-level arithmetic toward competition, expert-curated, dynamic, and formally verifiable benchmarks. EternalMath builds on these lines while targeting a narrower but under-served setting: executable tasks derived from recent constructive or quantitative research results.

Static and Expert-Curated Benchmarks Foundational datasets such as GSM8K (Cobbe et al., 2021), GAOKAO (Zhang et al., 2023), MATH (Hendrycks et al., 2021), Omni-MATH (Gao et al., 2025), and PutnamBench (Tsoukalas et al., 2024) remain useful for standardized comparison, but their fixed public problem sets make saturation and leakage concerns difficult to avoid. Expert-curated resources such as FrontierMath (Glazer et al., 2024) and HLE (Phan et al., 2025) increase task difficulty through human authoring, but the resulting snapshots are costly to expand and do not by themselves provide a low-cost refresh mechanism.

Automated, Dynamic, and Verifiable Evaluation Automated problem generation methods such as MetaMath (Yu et al., 2024), DART-Math (Tong et al., 2024), MathScale (Tang et al., 2024), SAND-Math (Manem et al., 2025), and STORM-BORN (Liu et al., 2025b) provide scalability, while dynamic benchmarks such as DynaMath (Zou et al., 2025), LiveMathBench (Liu et al., 2025a), and LiveMathematicianBench (He et al., 2026) refresh evaluation content. Formal proof benchmarks such as MiniF2F (Zheng et al., 2021), ProofNet (Azerbayev et al., 2023), and Lean-Workbook (Ying et al., 2024) provide strong verification, but they require formalized corpora and evaluate proof construction rather than natural-language mathematical problem solving.

Positioning. The closest recent efforts motivate the design space but leave different gaps. RealMath (Zhang et al., 2025) extracts verifiable QA pairs from live arXiv papers, but its released tasks are not dynamically parameterized into many executable variants. LiveMathematicianBench (He et al., 2026) also builds a live research-level benchmark from recent arXiv theorems, but it focuses on multiple-choice theorem-comprehension questions and proof-sketch-guided distractors rather than executable code templates. LemmaBench (Peyronnet et al., 2026) evaluates live research proofs, but relies on LLM-as-a-judge evaluation. VAR-MATH (Yao et al., 2025) uses programmatic multi-instance generation, but draws from static historical corpora. EternalMath combines recent vetted mathematical sources, source-kernel-grounded extraction, programmatic parameterization, and deterministic executable verification.

3 Overall Pipeline

Figure 1 illustrates the **EternalMath** construction pipeline, which grounds executable tasks in recent vetted or published mathematical work and couples source kernels, concrete instances, and executable verification.

3.1 Running Example

To make the pipeline self-contained, we first trace one concrete transformation.

[1]The anonymous artifact is available at https://anonymous.4open.science/r/anonymous-403B/README.md.

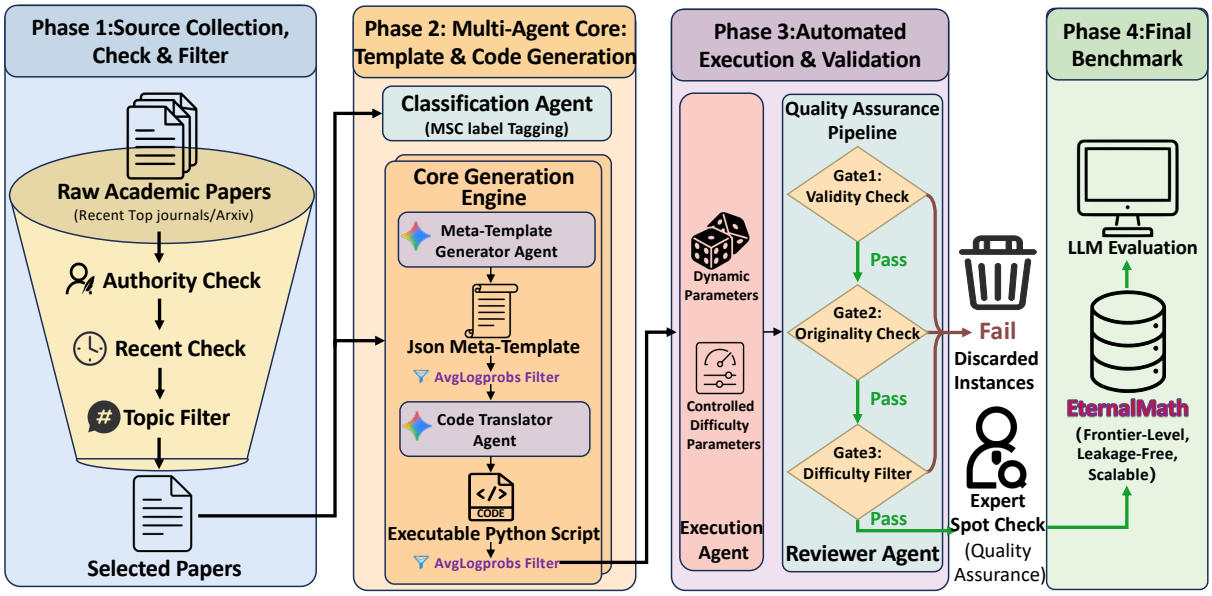

Figure 1: Overview of the **EternalMath** construction pipeline: source filtering, multi-agent theorem-to-template conversion, executable verification, and quality control.

Example 1: Theorem-to-Instance Walk-through

Source kernel. Ebrahimi (2026)'s Theorem 1.6(b) supplies the computable expression $\mathcal{E}(\Delta) = 2^{p-1}(p-1)!$ for the relevant Cayley graph when the prime parameter satisfies $p = n$.
Meta-template fields. The generator records the parameter domain ("n is prime"), the problem text template, the formal answer expression, and validation constraints in JSON.
Executable verifier. The translator emits Python code that samples a valid prime n, instantiates the graph-energy question, computes $2^{n-1}(n-1)!$, and checks that the sampled parameters satisfy the theorem preconditions.
Generated instance. With $n = 181$, the final task asks for the energy of the corresponding graph and has deterministic answer $2^{180} \times 180!$; the full template, code, and generated JSON appear in Appendices A.5–A.7.

3.2 Paper Filtering

The source corpus is assembled using three filtering criteria:

Recency We prioritize papers published within the past two years. This targets recent mathematical work and supports versioned benchmark refreshes. Recency is treated as a leakage-mitigation signal, not as a claim that all task-family overlap with model training data is impossible.

Authority For accessibility, the pipeline often uses arXiv-hosted versions of papers, but arXiv is treated as an access channel rather than an authority signal. A source is eligible only when the underlying work is published or accepted in a peer-reviewed mathematical venue, appears in a curated expert-vetted source list, or has an equivalent verifiable authority record in the source metadata. Sources without such validation are excluded.

Computability We prioritize theorems with constructive or quantitative formulations that admit explicit input-output relationships (e.g., computing an object C given conditions B). Excluding purely qualitative existence proofs keeps the retained problems programmatically instantiable and verifiable within a closed-loop pipeline. Such constructive or computable results appear across diverse mathematical landscapes, far beyond traditionally computational subfields.

3.3 Multi-agent collaboration

The construction pipeline uses four specialized stages for theorem extraction, instantiation, code generation, and validation. This separation makes intermediate artifacts inspectable and gives each stage a narrow input-output contract.

This decomposition addresses two practical needs. Extracting research-level source kernels, instantiating them into concrete problems, and ver-

ifying their solutions involves heterogeneous objectives that are brittle under a single prompt or model invocation. Separating responsibilities lets each agent operate under narrower constraints, improving stability, controllability, and reproducibility compared with single-pass Question-Answer (QA) generation (Zhang et al., 2025). Table 1 summarizes the released pipeline configuration; full prompts are provided in Appendix A.4.

In the released pipeline, classification and meta-template generation use Gemini-3-pro-preview, while code translation uses Claude 4.5. Correctness filtering is then handled by the generated Python/SymPy code and validator checks rather than by an additional LLM judge. Table 1 gives the operational contract for each stage, while Appendix A.4 records the full prompts for reproduction.

Classification Agent The Classification Agent assigns up to three MSC2020 (Dunne and Hulek, 2020) subject codes to each selected paper, prioritizing author-provided classifications when available. This step supports structured analysis and systematic filtering. Papers in non-core or descriptive categories (e.g., 00 General and overarching topics; collections, 01 History and biography, and 97 Mathematics education) are excluded because they are poor fits for quantitative problem generation.

In particular, we curated and categorized approximately 400 recent research articles, organized by mathematical subfields. This classification allows the pipeline to selectively draw from specific areas of interest, supporting the construction of domain-targeted benchmark variants. Table 3 reports the distribution of these papers across disciplines and their corresponding proportions.

Meta-Template Generator Agent Given a classified paper, this agent identifies instantiable mathematical statements such as theorems or propositions with explicit quantitative or constructive content and converts them into parameterized meta-templates. Each meta-template specifies parameter domains, generation constraints, problem statements, natural-language and formal solutions, and validation rules in a structured JSON format. Only statements admitting numerical instantiation and automated verification are retained.

Although a single article often yields multiple suitable theorems, the generator is capped at two to three unique templates per paper to keep validation focused. During template and code generation, outputs with an $avgLogprobs$ below -1 are filtered. This process yields 891 distinct templates. A representative template is presented in Appendix A.5.

Code Translator Agent The Code Translator Agent converts meta-templates into executable scripts, typically implemented in Python using symbolic and numerical libraries. These scripts instantiate parameters, compute exact solutions, and embed consistency checks derived from the original mathematical statements, enabling deterministic generation of problem-solution pairs. Appendix A.6 provides a representative example.

Execution and Validator Agent The final agent executes the generated scripts and applies a multi-stage validation pipeline, including runtime checks, solution integrity verification, and constraint consistency testing. Only problem instances that pass all validation stages are retained in the benchmark.

This modular design supports template-based data generation, parameter-controlled variation, and automated verification without requiring expert-authored problems for every instance. A representative output problem is shown in Appendix A.7.

3.4 Validation and Quality Assurance

The validation pipeline checks reliability, correctness, and evaluative value through automated verification, model-assisted screening, and targeted human inspection.

Problem Validity Review We first evaluate whether the source kernels used for template generation can be reliably instantiated as self-contained and verifiable problems. This review assesses the semantic clarity and mathematical soundness of both the problem statement and its corresponding solution, with particular attention to the consistency of parameter constraints, derivation steps, and final conclusions with the underlying source result. Problems that admit multiple plausible interpretations, rely on implicit or unstated assumptions, are ill-posed, or lack a unique and well-defined solution are excluded at this stage.

Originality and Contamination Review To operationalize duplicate and leakage screening, we employ layered checks. First, we perform exact string matching against the public benchmark corpora used in this snapshot: GSM8K,

Stage	Backbone/executor	Input	Output	Validation role
Classification	Gemini-3-pro-preview	Paper title, abstract, and metadata	Up to three top-level MSC2020 codes	Filters non-core or non-computable areas
Meta-template generation	Gemini-3-pro-preview	Vetted paper text and MSC labels	JSON template with parameters, constraints, problem text, and solution logic	Retains constructive or quantitative kernels
Code translation	Claude 4.5	Meta-template JSON	Runnable Python generator and solver	Converts template logic into executable scripts
Execution and validation	Python/SymPy runtime and validator checks	Generated code and sampled parameters	Verified problem–answer instances	Rejects runtime failures, constraint violations, and inconsistent answers

Table 1: Main agents in the EternalMath construction pipeline. The prompts and a full worked example are provided in Appendix A.4 and Appendices A.5–A.7.

Statistic	Snapshot value
Source articles	Approximately 400 recent mathematical research articles
Retained meta-templates	891 parameterized templates
Evaluated instances	782 verified problem instances
Model-relative strata	Easy: 116; Medium: 262; Hard: 404
Languages	English and Chinese
Coverage metadata	MSC2020 top-level categories; distribution in Table 3
Duplicate screening	Exact benchmark matching, embedding similarity filtering, and theorem-family metadata review
Released artifacts	Dataset snapshot, meta-templates, executable scripts, and example outputs

Table 2: Compact statistics for the EternalMath snapshot used in the experiments.

AIME, and HLE. Second, we conduct semantic duplicate detection using dense embeddings (`text-embedding-3-small`); a candidate problem is discarded if its cosine similarity to any screened public benchmark instance exceeds $\tau = 0.85$. These two checks are useful first-pass filters for surface reuse and close paraphrases, but they do not by themselves establish robustness to structural or theorem-family-level overlap.

We therefore add a source-kernel overlap review tailored to research-derived tasks. For each retained template, the pipeline records a theorem-family fingerprint consisting of the source paper identifier, theorem/proposition label, normalized assumption signature, target quantity, parameter roles, MSC code, and a short dependency profile of the derivation. Templates are flagged for manual review when they share the same source paper or theorem label with a screened item, or when their assumption signature and target quantity match a previously recorded source kernel even if the generated problem text is different. This step is intended to catch cases where two items are not lexical duplicates but instantiate the same mathematical result or proof pattern. Broader source-paper exposure and future post-release contamination cannot be ruled out for closed models, so we track source metadata, timestamped snapshots, and versioned refreshes rather than claiming contamination-free evaluation.

Difficulty Stratification We stratify problem difficulty by jointly considering model-relative solvability and template-level structural complexity. The solvability signal is estimated using a suite of frontier systems, including OpenAI GPT-5.1-high, Gemini-3-pro, DeepSeek-v3.2, Qwen3-max, and Doubao-seed-1.6, while the structural-complexity rubric records dependency depth, number of active constraints, hidden-condition checks, and required intermediate constructions; details are provided in Appendix A.2. This combined procedure yields 116 Easy, 262 Medium, and 404 Hard instances in the released snapshot. These labels are intended as practical stratification metadata rather than absolute mathematical difficulty ratings, and should be recalibrated as model capabilities and benchmark snapshots evolve.

Final Curation and Human Verification As a final sampling-based validation step, we applied a blind audit protocol to a stratified random sample of 100 generated instances. The sample was selected to cover MSC areas, model-relative solvability strata, source authority channels, and template forms. The audit panel was recruited from university mathematics departments; each auditor held a PhD in mathematics, and the panel's research areas collectively covered the major mathematical areas represented in the audited sample. The experts reviewed each item without access to the generation-stage model identity or downstream model-evaluation outcomes. The protocol required three forms of verification: independently re-deriving the solution path from the prob-

MSC Classification	Percentage	MSC Classification	Percentage
35 Partial differential equations	11.9%	18 Category theory; homological algebra	2.9%
05 Combinatorics	9.8%	13 Commutative algebra	2.7%
20 Group theory and generalizations	7.0%	82 Statistical mechanics, structure of matter	2.2%
14 Algebraic geometry	6.3%	76 Fluid mechanics	2.2%
11 Number theory	5.1%	15 Linear and multilinear algebra; matrix theory	2.0%
46 Functional analysis	4.5%	17 Nonassociative rings and algebras	2.0%
65 Numerical analysis	4.3%	22 Topological groups, Lie groups	1.8%
81 Quantum theory	3.9%	60 Probability theory and stochastic processes	1.8%
37 Dynamical systems and ergodic theory	3.7%	03 Mathematical logic and foundations	1.6%
53 Differential geometry	3.5%	49 Calculus of variations and optimal control	1.6%
16 Associative rings and algebras	3.5%	58 Global analysis, analysis on manifolds	1.6%
34 Ordinary differential equations	2.9%	Other (< 1.5% each)	11.4%

Table 3: Distribution of Mathematics Subject Classification (MSC) 2020 codes by percentage in the dataset.

lem statement, checking that the boundary conditions and parameter constraints in the generated Python script faithfully encoded the source theorem kernel, and verifying the deterministic output against the manual derivation. The 98% alignment rate means that 98 of the 100 audited instances passed all checklist items, including source-theorem faithfulness, parameter-constraint consistency, answer uniqueness, and verifier correctness. This result provides sampling-based evidence that the released snapshot contains research-level reasoning tasks rather than only code-translation artifacts; Appendix A.3 records the checklist.

4 Evaluation

4.1 Experimental Setup

For a representative assessment of **EternalMath**, we evaluate frontier LLMs through API-hosted inference: Gemini-3-pro-preview, OpenAI GPT-5.1-high, OpenAI GPT-5.2-xhigh, DeepSeek-v3.2-exp, Doubao-seed-1.6, Qwen3-max-preview-thinking, DeepSeek-v3.2-speciale, DeepSeek-v3.2-thinking, Kimi-k2-thinking, Claude-4.5-opus-thinking, Grok-4-latest, and Zhipu-GLM-4.6. Appendix B.3 reports the exact model identifiers, API interfaces, weight availability, and pretraining-data disclosure status.

During inference, each API is run with its maximum supported context/output budget to reduce token-budget artifacts.

EternalMath focuses on constructive or quantitative tasks, so scoring is based on deterministic final answers. We accept mathematically equivalent outputs, such as $2^{180} \times 180!$ or its numeric expansion, using `SymPy`-based semantic and numerical equivalence checks.

4.2 Experimental Results and Takeaways

The main empirical takeaway is that the released snapshot remains unsaturated and that many errors occur before final arithmetic. Figure 2 shows that OpenAI GPT-5.2 solves 49.4% of the snapshot, while most other models remain below 40%. The code-interpreter and open-book probes below then test whether this gap can be explained by calculation limits or missing theorem text alone; both probes show only limited gains.

Difficulty Stratification Analysis Figure 4 shows that performance drops sharply from Easy to Hard model-relative strata. Qwen3-max and OpenAI GPT-5.1-high both reach 94.8% on Easy tasks, but Hard tasks, which account for 51.7% of the snapshot, remain below 8% for all shown models. These strata summarize current model behavior. The structural-complexity rubric in Appendix A.2 provides a separate template-level view, and the two signals are reported separately.

4.3 Diagnostic Analysis

Impact of Code Interpreter To test whether failures are primarily arithmetic, we re-evaluate OpenAI GPT-5.2 with a Python code-interpreter sandbox. Accuracy rises only from 49.4% to 51.6%, a 2.2-point gain, suggesting that many failures occur before final calculation.

4.4 Failure-Mode Analysis

The small code-interpreter gain suggests that many failures occur upstream of arithmetic execution. Manual analysis of 100 failed cases yields 246 error tags. Calculation Drift accounts for only 10.6% (26/246), while Knowledge Gaps (18.3%) and Conceptual Missteps (13.8%) more often arise during theorem interpretation and mathematical modeling. Many incorrect responses therefore fail before the correct computation has been set up.

The worked failure case in Appendix B.1 illustrates this pattern. In the Ebrahimi Cayley-graph example, a model can recognize several surface facts about cycles and group components, yet

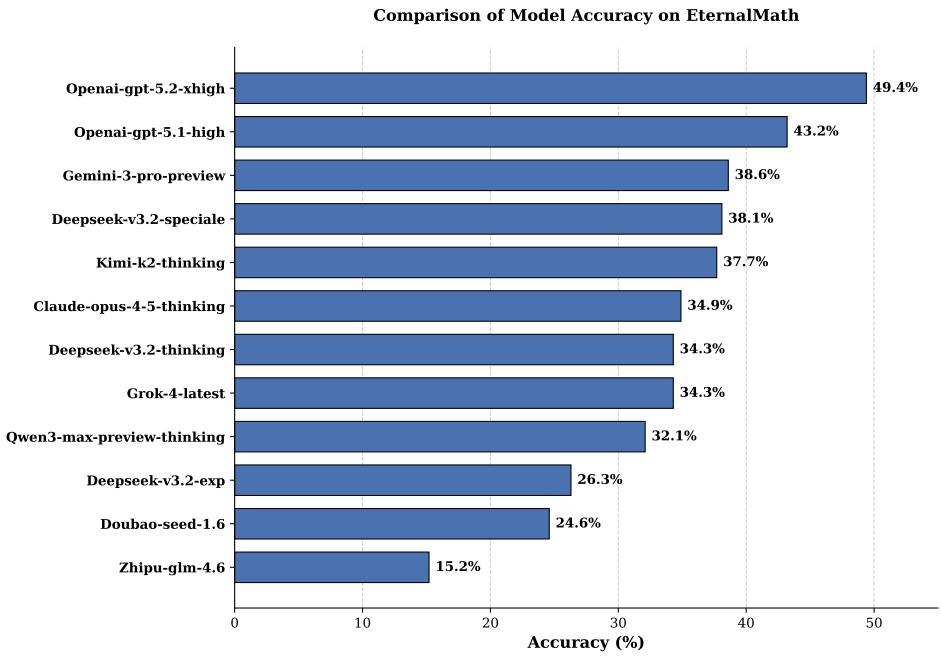

Figure 2: Accuracy of evaluated LLMs on **EternalMath**.

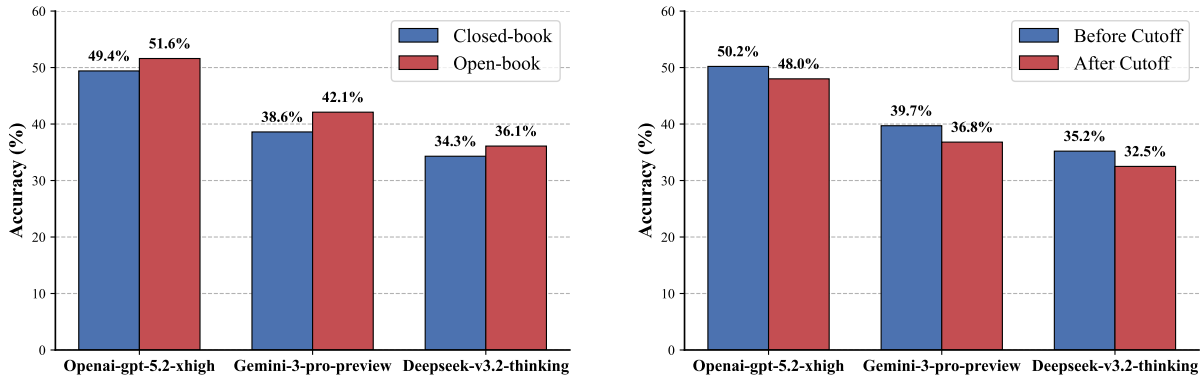

Figure 3: Auxiliary diagnostic probes: closed-book vs. open-book evaluation (left) and before/after-cutoff leakage screening (right).

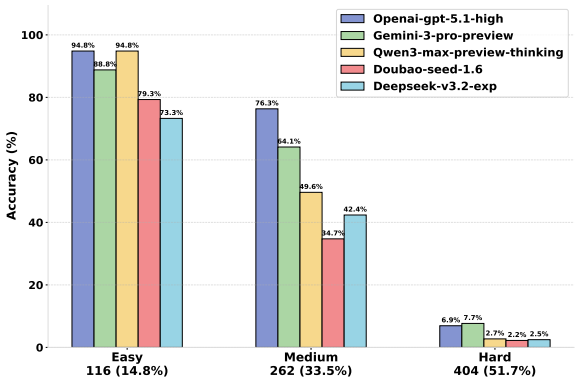

Figure 4: Performance across model-relative solvability strata.

it replaces the theorem-specific energy computation with an unsupported heuristic about the trivial eigenvalue and then extrapolates from a low-dimensional $p = 3$ case to $p = 181$. We categorize this behavior as a combination of logical hallucination and over-generalization: the response sounds structurally mathematical, but the decisive theorem application is missing.

Table 4 summarizes the main diagnostic categories used in the analysis, separating setup failures from downstream calculation failures. Appendix B.2 provides the full taxonomy and additional manifestations.

Failure mode	What the signal means
Knowledge gap 18.3% of tags	Missing specialized theorem context or prerequisite concepts; the model starts from an incomplete theorem kernel.
Conceptual misstep 13.8% of tags	Misread definitions or theorem conditions lead to an invalid setup before computation begins.
Calculation drift 10.6%, 26/246	Symbolic or arithmetic precision is lost after the main mathematical setup is already in place.
Logical hallucination Ebrahimi case	A generic shortcut or low-dimensional pattern is imported where the theorem-specific argument is required.

Table 4: Compact main-text summary of failure modes. Percentages use the 246 manual error tags from 100 failed cases; the qualitative row is illustrated by the worked case in Appendix B.1.

4.5 Auxiliary Diagnostic Probes

Impact of Context In an open-book setting with theorem statements, definitions, and source excerpts, accuracy improves only marginally (Figure 3, left), suggesting that many failures involve multi-stage mathematical modeling rather than only recall or source access.

Contamination and Leakage Probes We distinguish exact test-set leakage from broader task-family overlap. Exact matching and embedding similarity provide first-pass duplicate filters, while the source-kernel review described above targets theorem-family overlap that may survive lexical rewriting. A before/after December 2025 timestamp probe shows only a small performance difference (Figure 3, right). This pattern is consistent with the absence of widespread exact memorization in this snapshot, while broader source-paper exposure, residual theorem-family overlap, LLM-generated paraphrase overlap, and future contamination remain separate risks.

5 Conclusion

We introduced **EternalMath**, a refreshable benchmark for research-level constructive and quantitative mathematical reasoning. Its pipeline turns recent vetted mathematical work into parameterized, executable instances, supporting versioned refreshes and leakage-mitigating duplicate checks. Current frontier systems solve less than half of the released snapshot, and the failure analysis points to theorem interpretation and mathematical modeling rather than only calculation. These results suggest that research-level constructive reasoning remains an open challenge for frontier LLMs.

Limitations

EternalMath has several limitations. The methodology targets constructive or quantitative formulations and therefore underrepresents proof-oriented, qualitative, or non-executable areas of mathematics. The pipeline also relies on LLMs for source-kernel extraction and code generation, so specialized terminology or boundary conditions can still be misinterpreted even after multi-stage validation and sampling-based audits. The Easy/Medium/Hard labels are model-relative solvability strata rather than absolute mathematical difficulty levels and require recalibration as model capabilities evolve; the structural-complexity rubric is a complementary proxy, not a replacement for expert mathematical difficulty ratings. Finally, exact matching, embedding filtering, and theorem-family metadata review mitigate several forms of overlap in the released snapshot, but they cannot rule out all task-family overlap, source-paper exposure in closed pretraining data, or future contamination after public release. This motivates versioned refreshes as part of the benchmark design. Future work could explore how prompting strategies, search-augmented generation, or formal verification tools affect performance on the benchmark.

References

Art of Problem Solving. 2026. AoPS Wiki: Competition ratings. https://artofproblemsolving.com/wiki/index.php?title=AoPS_Wiki:Competition_ratings. Accessed 2026-05-26.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W Ayers, Dragomir Radev, and Jeremy Avigad. 2023. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*.

Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. 2025. MathArena: Evaluating LLMs on uncontaminated math competitions. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Edward Dunne and Klaus Hulek. 2020. Mathematics subject classification 2020. *Not. Am. Math. Soc*, 67(3):410–411.

Mahdi Ebrahimi. 2026. Singular cayley graphs and p-blocks of finite groups. *Journal of Algebra*, 687:477–491.

Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Runxin Xu, and 1 others. 2025. Omni-MATH: A universal olympiad-level mathematic benchmark for large language models. In *International Conference on Learning Representations*.

Elliot Glazer, Ege Erdil, Tamay Besiroglu, Diego Chicharro, Evan Chen, Alex Gunning, Caroline Falkman Olsson, Jean-Stanislas Denain, Anson Ho, Emily de Oliveira Santos, and 1 others. 2024. FrontierMath: A benchmark for evaluating advanced mathematical reasoning in AI. *arXiv preprint arXiv:2411.04872*.

Linyang He, Qiyao Yu, Hanze Dong, Baohao Liao, Xinxing Xu, Micah Goldblum, Jiang Bian, and Nima Mesgarani. 2026. LiveMathematicianBench: A live benchmark for mathematician-level reasoning with proof sketches. *arXiv preprint arXiv:2604.01754*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*.

Junnan Liu, Hongwei Liu, Linchen Xiao, Ziyi Wang, Kuikun Liu, Songyang Gao, Wenwei Zhang, Songyang Zhang, and Kai Chen. 2025a. Are your LLMs capable of stable reasoning? In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 17594–17632.

Wenhao Liu, Zhenyi Lu, Xinyu Hu, Jierui Zhang, Dailin Li, Jiacheng Cen, Huilin Cao, Haiteng Wang, Yuhan Li, Kun Xie, and 1 others. 2025b. STORM-BORN: A challenging mathematical derivations dataset curated via a human-in-the-loop multi-agent framework. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 23938–23958.

Chaitanya Manem, Pratik Prabhanjan Brahma, Prakamya Mishra, Zicheng Liu, and Emad Barsoum. 2025. SAND-Math: Using LLMs to generate novel, difficult and useful mathematics questions and answers. *arXiv preprint arXiv:2507.20527*.

Sean McLeish, Arpit Bansal, Alex Stein, Neel Jain, John Kirchenbauer, Brian Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, Jonas Geiping, Avi Schwarzschild, and 1 others. 2024. Transformers can do arithmetic with the right embeddings. *Advances in Neural Information Processing Systems*, 37:108012–108041.

Antoine Peyronnet, Fabian Gloeckle, and Amaury Hayat. 2026. LemmaBench: A live, research-level benchmark to evaluate LLM capabilities in mathematics. *arXiv preprint arXiv:2602.24173*.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, Michael Choi, Anish Agrawal, Arnav Chopra, Adam Khoja, Ryan Kim, Richard Ren, Jason Hausenloy, Oliver Zhang, Mantas Mazeika, and 11 others. 2025. Humanity's Last Exam. *Preprint*, arXiv:2501.14249.

Zhihong Shao, Yuxiang Luo, Chengda Lu, ZZ Ren, Jiewen Hu, Tian Ye, Zhibin Gou, Shirong Ma, and Xiaokang Zhang. 2025. DeepSeekMath-V2: Towards self-verifiable mathematical reasoning. *arXiv preprint arXiv:2511.22570*.

Zhengyang Tang, Xingxing Zhang, Benyou Wang, and Furu Wei. 2024. MathScale: Scaling instruction tuning for mathematical reasoning. In *Proceedings of the 41st International Conference on Machine Learning*.

Yuxuan Tong, Xiwen Zhang, Rui Wang, Ruidong Wu, and Junxian He. 2024. DART-Math: Difficulty-aware rejection tuning for mathematical problem-solving. *Advances in Neural Information Processing Systems*, 37:7821–7846.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. 2024. PutnamBench: Evaluating neural theorem-provers on the putnam mathematical competition. *Advances in Neural Information Processing Systems*, 37:11545–11569.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024. Math-Shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Jian Yao, Ran Cheng, and Kay Chen Tan. 2025. VAR-MATH: Probing true mathematical reasoning in LLMs via symbolic multi-instance benchmarks. *arXiv preprint arXiv:2507.12885*.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Jiayu Wang, Dahua Lin, and Kai Chen. 2024. Lean Workbook: A large-scale Lean problem set formalized from natural language math problems. *Advances in Neural Information Processing Systems*, 37:105848–105863.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2024. MetaMath: Bootstrap your own mathematical questions for large language models. In *International Conference on Learning Representations*.

Jie Zhang, Cezara Petrui, Kristina Nikolić, and Florian Tramèr. 2025. RealMath: A continuous benchmark for evaluating language models on research-level mathematics. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*.

Xiaotian Zhang, Chunyang Li, Yi Zong, Zhengyu Ying, Liang He, and Xipeng Qiu. 2023. Evaluating the performance of large language models on Gaokao benchmark. *arXiv preprint arXiv:2305.12474*.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2021. MiniF2F: A cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*.

Chengke Zou, Xingang Guo, Rui Yang, Junyu Zhang, Bin Hu, and Huan Zhang. 2025. DynaMath: A dynamic visual benchmark for evaluating mathematical reasoning robustness of vision language models. In *International Conference on Learning Representations*.

A Appendix

A.1 Radar chart indicators and data

This appendix defines the indicators used for the radar-chart comparison in Figure 5. The chart visualizes the descriptive benchmark indicators reported in Table 6.

- **Model-relative hardness** D_1

$$D_1 = 1 - \mu,$$

  where μ denotes the mean model accuracy. This indicator summarizes empirical hardness under the evaluated model set.

- **Improvement Space** D_2

$$D_2 = 1 - a_{\max},$$

  where $a_{\max}$ is the maximum accuracy achieved by the model.

- **Discriminability** D_3

$$D_3 = \frac{\sigma}{\mu},$$

  where σ denotes the standard deviation of the model accuracy.

- **Academic Level** D_4

  The academic level reflects the highest level of formal education required to solve a problem. Each problem is assigned a scalar score according to its disciplinary depth, as summarized in Table 5.

Academic Level	Score
Research-level mathematics	1.0
Graduate-level mathematics	0.8
Undergraduate mathematics	0.7
High-school mathematics olympiad	0.6
Middle / High school	0.4
Primary school	0.2

Table 5: Categorization and scoring assignment for the academic depth of mathematical problems.

- **Coverage** D_5

$$D_5 = \frac{\#\{\text{covered first-level categories}\}}{\#\{\text{first-level categories}\}}.$$

- **Originality** D_6

$$D_6 = \frac{\text{Number of original problems}}{\text{Total number of problems}}.$$

Table 6 reports the indicator values for Eternal-Math and several mainstream mathematical evaluation datasets. Figure 5 visualizes the same values as a radar chart.

Benchmark	D_1	D_2	D_3	D_4	D_5	D_6
EternalMath	0.67	0.51	0.26	1.00	0.91	1.00
EternalMath-hard	0.96	0.92	0.54	1.00	0.80	1.00
HLE-Math	0.76	0.65	0.38	1.00	0.48	1.00
AIME 2025	0.11	0.00	0.05	0.60	0.16	1.00
SuperGPQA-Math	0.13	0.07	0.03	0.80	0.48	0.20
UGMath	0.10	0.07	0.03	0.70	0.40	0.40
U-Math	0.09	0.06	0.02	0.70	0.08	0.10

Table 6: Comparison of **EternalMath** with mainstream mathematical benchmarks across six descriptive indicators: model-relative hardness (D_1), improvement space (D_2), discriminability (D_3), academic level (D_4), category coverage (D_5), and originality (D_6).

A.2 Structural Complexity Rubric

Model-relative solvability is useful for diagnosing current systems. To complement Figure 4, we define a structural-complexity rubric that can be applied to each meta-template before model evaluation. The rubric is adapted to executable research-derived tasks and follows the broader practice of recording difficulty through human-readable problem structure (Gao et al., 2025; Art of Problem Solving, 2026).

Dimension	Score	Criterion
Dependency depth	1–3	Number of nontrivial reasoning links from given parameters to the target quantity: direct use, one intermediate object, or a multi-step dependency chain.
Active constraints	0–2	Number and coupling of constraints that must be checked before applying the source kernel: none, single independent constraint, or multiple coupled constraints.
Hidden-condition checks	0–2	Whether the solver must recognize applicability conditions not stated as formula hints: none, explicit but non-formulaic check, or nontrivial theorem precondition.
Intermediate constructions	0–2	Whether solving requires constructing auxiliary objects or equivalent transformations before computation: none, one construction, or multiple transformations.

Table 7: Structural-complexity rubric for EternalMath templates.

The resulting total score ranges from 1 to 9. We use low (1–3), medium (4–6), and high (7–9) structural-complexity tags as metadata when auditing the dataset. These tags help identify whether a model-relative Hard item is structurally complex or instead reflects a possible shared blind spot, source exposure issue, or implementation artifact.

A.3 Human Audit Protocol

The final audit uses a stratified random 100-instance sample selected after automated validation. The sample covers MSC top-level areas, model-relative solvability strata, source authority channels, and template forms. The audit panel was recruited from university mathematics departments. Each auditor held a PhD in mathematics, and the panel's research areas collectively covered the major mathematical areas represented in the audited sample. Reviewers used the item-level audit sheet without access to the generation-stage model identity or downstream model-evaluation outcomes, keeping the assessment focused on the problem statement, the source result, and the executable verifier.

The audit protocol has three steps. First, reviewers independently re-derive the solution path from the problem statement. Second, they check whether the boundary conditions and parameter constraints in the generated Python script faithfully encode the source theorem kernel. Third, they verify the deterministic output against the manual derivation. The checklist in Table 8 records the pass conditions used to assign the final alignment label.

The reported 98% alignment rate means that 98 of the 100 audited instances passed all four checklist items.

A.4 System Prompts for Agents

Audit item	Pass condition
Source-theorem faithfulness	The executable task follows the cited source theorem, proposition, construction, or quantitative kernel without changing its mathematical scope.
Parameter constraints	The stated problem conditions and generated Python parameters satisfy all source preconditions needed for the answer.
Unique answer	The problem statement determines one mathematically well-defined answer under the stated constraints.
Verifier correctness	The executable verifier computes or checks the same target quantity as the manual derivation and rejects invalid parameter settings.

Table 8: Checklist for the sampling-based human audit.

Agent	Backbone/executor	Input	Output	Prompt/detail location
Classification	Gemini-3-pro-preview	Paper metadata and text snippets	Top-level MSC2020 codes	Listing A.4
Meta-template generator	Gemini-3-pro-preview	Vetted paper text and MSC labels	Parameterized JSON template	Listing A.4
Code translator	Claude 4.6	Meta-template JSON	Runnable Python generator/verifier	Listing A.4
Execution/validator	Python/SymPy runtime and validation checks	Generated script and sampled parameters	Verified problem-answer JSON	Appendices A.6–A.7

Table 9: Prompt and agent manifest for the released EternalMath construction pipeline. The table summarizes each agent's role, while the full prompt listings follow below.

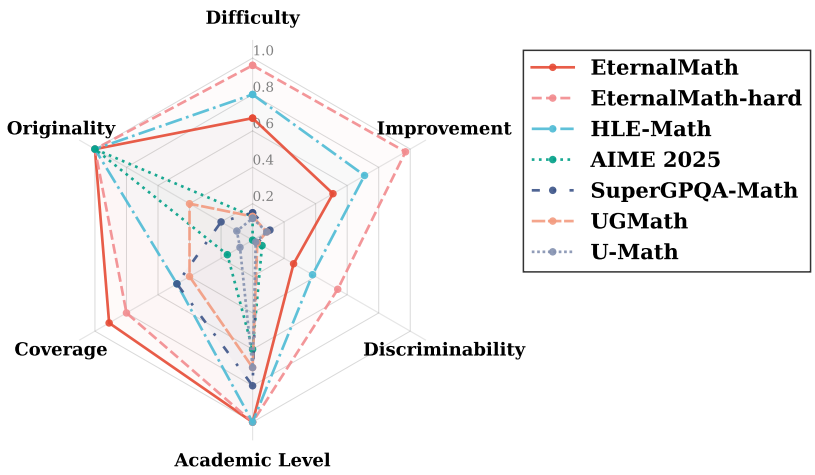

Figure 5: Radar chart of the six descriptive benchmark indicators reported in Table 6; indicator definitions are given in Appendix A.1.

Classification Agent Prompt.

```
    "**Role and Objective:**\n"
    "You are a professional Mathematics and Computer Science classification expert. Your task is to assign the most accurate Mathematics Subject Classification (MSC 2020) codes to a given academic paper.\n"
    "\n"
    "**Workflow (In Order of Priority):**\n"
    "1. **Extraction (Primary):** Carefully scan the provided abstract/content to see if the authors explicitly state any MSC codes (e.g., 'MSC 2020: 34A34, 65L04'). \n"
    "   *Crucial:* If 5-character codes are found, you MUST truncate them to their 2-digit top-level category (e.g., convert '34A34' to '34').\n"
    "2. **Inference (Secondary):** If no MSC codes are explicitly mentioned, analyze the paper's title and content to deduce and assign the most appropriate codes yourself.\n"
    "\n"
    "**Constraints & Output Format:**\n"
    "1. **Classification Level:** Strictly use **2-digit** top-level MSC codes only.\n"
    "2. **Quantity:** Return at least 1 and at most **3** of the most relevant codes.\n"
    "3. **Separator:** Separate multiple codes strictly with a comma and a space (', ').\n"
    "4. **Fallback:** If the content is completely irrelevant or cannot be classified, output '99'.\n"
    "5. **Strict Output:** Output ONLY the final numerical codes. Do not include prefixes like 'MSC:', 'Codes:', or any conversational text.\n"
    "6. **No Chain of Thought:** Do NOT output your thinking process or explanations.\n"
    "\n"
    "**Reference List of Top-Level MSC Codes:**\n"
    "00 - General and overarching topics; collections\n"
    "01 - History and biography\n"
    "03 - Mathematical logic and foundations\n"
    "05 - Combinatorics\n"
    "06 - Order, lattices, ordered algebraic structures\n"
    "08 - General algebraic systems\n"
    "11 - Number theory\n"
    "12 - Field theory and polynomials\n"
    "13 - Commutative algebra\n"
    "14 - Algebraic geometry\n"
    "15 - Linear and multilinear algebra; matrix theory\n"
    "16 - Associative rings and algebras\n"
    "17 - Nonassociative rings and algebras\n"
    "18 - Category theory; homological algebra\n"
    "19 - K-theory\n"
```

```
    "20 - Group theory and generalizations\n"
    "22 - Topological groups, Lie groups\n"
    "26 - Real functions\n"
    "28 - Measure and integration\n"
    "30 - Functions of a complex variable\n"
    "31 - Potential theory\n"
    "32 - Several complex variables and analytic spaces\n"
    "33 - Special functions\n"
    "34 - Ordinary differential equations\n"
    "35 - Partial differential equations\n"
    "37 - Dynamical systems and ergodic theory\n"
    "39 - Difference and functional equations\n"
    "40 - Sequences, series, summability\n"
    "41 - Approximations and expansions\n"
    "42 - Harmonic analysis on Euclidean spaces\n"
    "43 - Abstract harmonic analysis\n"
    "44 - Integral transforms, operational calculus\n"
    "45 - Integral equations\n"
    "46 - Functional analysis\n"
    "47 - Operator theory\n"
    "49 - Calculus of variations and optimal control; optimization\n"
    "51 - Geometry\n"
    "52 - Convex and discrete geometry\n"
    "53 - Differential geometry\n"
    "54 - General topology\n"
    "55 - Algebraic topology\n"
    "57 - Manifolds and cell complexes\n"
    "58 - Global analysis, analysis on manifolds\n"
    "60 - Probability theory and stochastic processes\n"
    "62 - Statistics\n"
    "65 - Numerical analysis\n"
    "68 - Computer science\n"
    "70 - Mechanics of particles and systems\n"
    "74 - Mechanics of deformable solids\n"
    "76 - Fluid mechanics\n"
    "78 - Optics, electromagnetic theory\n"
    "80 - Classical thermodynamics, heat transfer\n"
    "81 - Quantum theory\n"
    "82 - Statistical mechanics, structure of matter\n"
    "83 - Relativity and gravitational theory\n"
    "85 - Astronomy and astrophysics\n"
    "86 - Geophysics\n"
    "90 - Operations research, mathematical programming\n"
    "91 - Game theory, economics, finance, and other social and behavioral
        sciences\n"
    "92 - Biology and other natural sciences\n"
    "93 - Systems theory; control\n"
    "94 - Information and communication theory, circuits\n"
    "97 - Mathematics education\n"
```

Meta-Template Generator Agent Prompt.

```
You are an expert combining a background in mathematics education, cutting-edge
    mathematical research experience, and data engineering experience for large
    models.

Your task is: based on the core module of a cutting-edge mathematical paper,
    generate a "parameterized mathematical problem meta-template" used for the
    batch generation of high-difficulty, verifiable math problems that are
    instantiated from theorems and centered on reasoning ability.

 **I. Task Objectives**

Generate a structured JSON template object (strict format) to batch-generate
    mathematical problems that are highly difficult, logically complete,
    numerically unique, and emphasize reasoning skills.

**Must satisfy:**

* Only select modules from the paper that are instantiable and numerically
```

```
    calculable.
* Must involve multi-step logical reasoning, rather than being instantly
    solvable by directly substituting into a formula.
* Problems should test:
* Concept understanding
* Conditional reasoning
* Structural analysis
* Multi-variable relationship derivation


* Problems must have a unique, verifiable numerical solution.

**Cannot be:**

* Proof problems
* True/False binary choice problems
* Pure symbolic derivation

**Must ensure:**

* Conditions are complete.
* There are no logical contradictions.
* Given conditions are sufficient to deduce a unique solution.

 **Important Limitations (To prevent memory-based problems)**

**Prohibited to generate:**

* Problems where answers can be found by directly substituting into paper
    formulas.
* Single identity problems (e.g., $tr(AB)=tr(BA)$).
* Problems solvable by pure algebraic expansion.
* Questions merely testing the memorization of theorem statements.
* Problems with no intermediate judgment steps.
* Problems involving only 1 layer of variable dependency.
* Large-scale floating-point calculations.
* High-precision decimal results.
* Problems where every parameter given in the condition can be directly used in
    the final formula (this reduces reasoning difficulty).

**Allowed to generate:**

* Problems requiring the identification of hidden structures before invoking
    core concepts.
* Problems requiring multi-step transformations to trigger the paper‘s theorem
    conditions.
* Problems requiring judgment on whether applicable conditions are satisfied.
* Problems requiring intermediate constructions or equivalent deformations.
* Introducing "distracting conditions" or "redundant structures" with a
    reasonable mathematical background into the problem. The solver must use
    conceptual understanding to identify which conditions are not needed for the
     target structure calculation.

 The core objective is to test **"reasoning ability"**, not **"literature memory
      ability"**.

 **Numerical & Parameter Constraints (Precision Control)**

To avoid decimal error issues:

* **Prioritize using:**
* Integers
* Rational numbers
* Divisible structures


* **Avoid:**
* Massive floating-point operations
* High-precision approximations
* Irreducible irrational number propagation
```

```


* **If floating-point numbers must be used:**
* Limit to within four decimal places.
* Explicitly define the allowed error range (e.g., 1e-6).



 **II. Input Information (Provided by me)**

You will receive a paper. Please extract the following information from the paper:

* **Core paper domain:** e.g., "Linear Algebra (Matrix Trace Calculation)", "Calculus (Definite Integral Calculation)", "Machine Learning (Gradient Descent Iteration)", "Probability Theory (Conditional Expectation Calculation)".
* **Target paper module:** The specific example or section in the paper to be converted into a template (e.g., "Section 4.2 Example 4.3 Matrix Inversion Calculation").
* **Core mathematical logic:** The key calculation flow or formula of the module.

 **Only select modules that can be instantiated for high-difficulty, explicit numerical calculation steps; do not select parts involving proofs or purely theoretical deductions.**

**You need to:**

* Automatically identify instantiable, calculable modules.
* Filter out purely theoretical proof sections.
* Select modules with explicit logical flows.

 **III. Output Structure (Strict JSON Format)**

The output must **ONLY contain one JSON object**.

Keys and their order must strictly be as follows:

'''json
{
  "template_id": "",
  "target_paper_module": "",
  "core_math_concept": "",
  "reasoning_complexity": "",
  "param_definition": [],
  "problem_template": "",
  "output_type": "",
  "param_generation_rule": [],
  "natural_lang_solution": "",
  "formal_solution": "",
  "validation_rule": [],
  "condition_completeness_check": []
}

'''

**Field Descriptions:**

* 'template_id': Unique template ID (format like 'linear_algebra_matrix_trace_001').
* 'target_paper_module': Corresponding paper module.
* 'core_math_concept': Core mathematical concept.
* 'param_definition': Parameterizable variable definitions (see below).
* 'problem_template': Problem stem template (including variable placeholders).
* 'output_type': Problem output type (e.g., "real number", "matrix").
* 'param_generation_rule': Variable generation and constraint rules.
* 'natural_lang_solution': Natural language step-by-step solution template.
* 'formal_solution': Formal executable solution (e.g., Python/Numpy).
* 'validation_rule': Validation mechanism (see below).
```

```
* ‘condition_completeness_check‘: Completeness check for problem conditions.

 **IV. Component Definition Specifications**

 **Template Basic Information**

‘‘‘json
{
"template_id": "Domain_Module_Number",
"target_paper_module": "Full Paper Name: Corresponding Section or Theorem",
"core_math_concept": "Core mathematical principle (e.g., 'Cyclic invariance of matrix trace')"
}

‘‘‘

 **‘reasoning_complexity‘**
Explain how this template reflects "non-memory reasoning":
*Examples:*

* "Requires verifying hidden structural condition before applying theorem"
* "Requires multi-step transformation to reduce to core identity"
* "Requires constructing intermediate object satisfying theorem conditions"

 **Parameterizable Variable Definition**
Each variable should contain the following four items:

‘‘‘json
{
  "var_name": "Variable Name",
  "var_type": "Variable Type (Integer/Float/Matrix/Vector, etc.)",
  "var_constraint": "Value range or logical constraints (including relationships between variables)",
  "var_source": "Generation method (e.g., random generation / sampled from paper)"
}

‘‘‘

**Must:**

* Explicitly state the variable type.
* Explicitly state the value range (avoid large-scale decimals).
* Explicitly state logical relationships between variables.

*Example:*

‘‘‘json
{
  "var_name": "n",
  "var_type": "Integer",
  "var_constraint": "n \in [2, 80]",
  "var_source": "random_int(2,80)"
}

‘‘‘

 *The parameter range should include at least ten different sets of values.*

 **Problem Template**
**Strictly Prohibited:**

* Appearing "According to the theorem..."
* Appearing formula hints.
* Appearing problem-solving hints.
* Appearing the paper name.

**Must:**

* Only give conditions.
```

```
* Give the target.
* Not expose the method.
* Conditions must be sufficient to deduce a unique solution.

*(Bad Example ):* "Given matrix A... and B..., please calculate the difference between the trace of AB and the trace of BA according to formula xx or step yy." (This example leaks calculation steps and formula hints).
*(Good Example ):* "Given matrix $A \in \mathbb{R}^{m \times n}$ defined as $A = \{A\}$, and matrix $B \in \mathbb{R}^{n \times m}$ defined as $B = \{B\}$. Please calculate the difference between the trace of $AB$ and the trace of $BA$." (This example only states known conditions and the target without leaking any steps or hints).

And define output type: `{"output_type": "Real number"}`

 **Parameter Generation Rules**
Explain the variable generation process and constraint conditions step-by-step:

```json
"param_generation_rule": [
  "Step1: m = random_int(2,500)",
  "Step2: n = random_int(2,500)",
  "Step3: A = random_matrix(m,n,-100,100)",
  "Step4: B = random_matrix(n,m,-100,100)",
  "ensure: A.shape[1] == B.shape[0]",
  "ensure: A.shape[0] == B.shape[1]"
]

```

Must strengthen precision control and unique solution conditions.

 Solution Template
Detailed steps connecting concepts and calculations should only be included in the solution template (especially `natural_lang_solution`).

`"natural_lang_solution"`: "Solution: 1. Based on the cyclic invariance of the matrix trace ($tr(XY) = tr(YX)$), it is known that A is an m\times n matrix and B is an n\times m matrix, so both AB and BA are square matrices. 2. Theoretically, $tr(AB)$ is always equal to $tr(BA)$. 3. Therefore, the theoretical value of $tr(AB) - tr(BA)$ is 0. 4. (Verification step) By calculating... (calculation process)... the result is 0."

`"formal_solution"`:

```python
import numpy as np
A=np.array({A});
B=np.array({B});
tr_AB=np.trace(A@B);
tr_BA=np.trace(B@A);
print(tr_AB - tr_BA)

```


Must:

* Display a complete reasoning chain.
* Explicitly state key judgment steps.
* Explain why the core concept can be applied.
* Demonstrate intermediate structure construction.

Must Avoid:

* Directly quoting conclusions in a single sentence.

 Validation Mechanism
Generate validation mechanisms to check if the problem conditions are reasonable
```

```
     and if the answer type is correct.
*Example:*

```json
"validation_rule": [
  {"type": "dimension_check", "rule": "A.shape[1] == B.shape[0]"},
  {"type": "value_check", "rule": "result == 0"},
  {"type": "execution_check", "rule": "formal_solution executes without error"}
]

```

 Condition Completeness Check
`"condition_completeness_check"`: "List all implicit mathematical premises required to ensure a unique solution (e.g., the graph must be connected, the matrix must be positive semi-definite, weights must be non-negative, etc.), and confirm that these constraints are already included in the problem stem or parameter generation."

 Key Design Principles (Must Satisfy)
Generated problems must be:

* Conditionally sufficient
* Non-contradictory
* Have a unique solution
* Programmatically verifiable
* Independent of floating-point errors
* Independent of massive computing power
* Independent of memorizing the original paper

 Preferred Problem Types
Prioritize:

* Requiring judgment on whether a structure satisfies conditions before calculating.
* Requiring the construction of equivalent representations.
* Requiring the use of invariants.
* Requiring the identification of hidden symmetries.
* Requiring multi-variable linkages.

Avoid:

* Single matrix identities.
* Single norm invariance.
* Direct trace equalities.
* One-step determinant formulas.

 V. Output Format Constraints

* Output **ONE valid JSON object**.
* Do not output explanations, comments, or redundant instructions.
* Use English punctuation inside the JSON and ensure the syntax is valid.
* Every field must be present and non-empty.
* Ensure the output structure can be parsed directly.

 Advanced Logic & Engineering Constraints (Highest Priority)

* **Forced Deep Reasoning:** The problem-solving path must be "tree-like" or "deep-chained." The solver must first calculate a hidden intermediate algebraic or topological structure from the given parameters, then judge the applicability conditions of the core theorem based on that intermediate structure, and finally calculate the numerical value.
* **Precondition Closed-Loop:** The given conditions in the stem and the parameter generation rules must be perfectly tight, absolutely ensuring that the generated random parameters satisfy all non-degenerate conditions (such as non-singular, non-negative weights, strongly connected, etc.).
* **Engineering Code Output:** `param_generation_rule` and `formal_solution` must be pure, Type Hinted Python function code blocks. Any non-code explanatory text is strictly prohibited in these two fields. `solve(params)` must be able to directly receive the output dictionary of `generate_params`
```

```
    ()` and return a unique result.

 **VI. Execution Prompt**
When I input paper articles or specific content, please follow the above format to:

1. Automatically identify the core calculable logic;
2. Generate **1 complete parameterized template**;
3. Ensure each template fully complies with the JSON structure and field specifications;
4. **Do not attach any explanatory or conversational text after the output.**
```

Code Translator Agent Prompt.

```
You are an expert in Python automatic code generation. Your task is: based on the "parameterized mathematical problem meta-template" (in JSON format) provided by the user, generate complete, directly runnable Python code. This code must be able to automatically generate 10 specific, calculable, and verifiable math problems along with their standard answers (strictly adhering to the requested JSON format and field structure). Strictly follow the computational logic of the meta-template without making any simplifications.

 **[Python Code Generation Requirements]**

**Code Structure:**

*
**Core Function**: Must include a core function `generate_math_problems(template_json)`, which receives the JSON template string or dictionary input by the user as a parameter.


*
**Helper Function**: To isolate risks and improve readability, it should include a helper function (e.g., `_generate_params_and_validate`) to handle parameter generation and constraint checking.


*
**Core Logic**: The code needs to simulate and implement all core rules of the original math problem generation engine, including parameter generation, problem instantiation, and answer calculation.


*
**Formatted Output**: Ultimately return a Python list containing 10 JSON objects.


* **Dependencies**: Use standard libraries (`json`, `random`, `math`, `sympy`) and `numpy` (which is friendly for complex calculations). Only introduce other libraries if necessary.


*
**Runnability**: The generated code must be a complete script that can be directly copied, pasted, and run, including the logic to execute the main program (`if __name__ == "__main__":`).



 **Critical Enhancement Requirements: Robustness Handling for LaTeX and JSON Formats**

**A. Secure JSON Handling (Input/Output)**

*
**Input handling**: Inside the core function, first use `json.loads()` to convert the input JSON string template into a Python dictionary.
```

```


*
**Final encapsulation**: Only when returning the results, use `json.dumps(result_list, ensure_ascii=False, indent=4)` for a one-time formatted output.



**B. Precise Handling of LaTeX Strings ( Mandatory Absolute Escaping of Double Backslashes and Curly Braces)**

* **Backslash escaping (JSON/Python script safety)**:
*
**Mandatory requirement**: In the complete generated Python script (including the hardcoded `template_json_string` variable), all backslashes in LaTeX commands (like `\frac`) must be defined using double backslashes `\\\\`.


*
**Example**: `\\\\frac{{...}}{{...}}` instead of `\frac{...}{...}`.




* **Curly brace escaping (Absolute safety for Python formatting)**:
*
**Mandatory requirement**: In the `problem_template`, `natural_lang_solution`, and `formal_solution` fields:


* Only placeholders used for substituting parameters use single curly braces, and the content must be a valid parameter name, for example: `{a}`, `{beta}`.


* All other curly braces that need to appear (including `{...}` in LaTeX structures, subscripts, superscripts, sets, etc.) must be escaped using double curly braces `{{...}}`.







**Error Demonstration and Correction:**

| Error Demonstration | Error Type | Absolute Correction |
| --- | --- | --- |
| `...{m-1}...` | KeyError | <br>`...{{m-1}}...`

 |
| `...{1, O}...` | KeyError | <br>`...{{1, O}}...`

 |
| `...\\frac{a}{b}...` | KeyError | <br>`...\\frac{{{a}}}{{{b}}}...` (If a, b are parameters)

 |
| `...\\frac{A}{{{m-1}}}...` | Correct | Correct, because `{m-1}` has been escaped as a literal.

 |

**Common Negative Examples of Missing Escapes:**

| Error Demonstration (KeyError/ValueError) | Intent (LaTeX/Literal) | Absolute Correction |
| --- | --- | --- |
| `{1, O}` | Set notation | <br>`{{1, O}}`
```

```
 |
| `{v_n}` | LaTeX subscript | <br>`{{v_n}}`

 |
| `{-2\\pi^2 t}` | Complex exponential expression | <br>`{{-2\\pi^2 t}}`

 |
| `{}` | Empty placeholder | <br>`{{}}`

 |
| `{1}` | Positional placeholder | <br>**Prohibited**

 |

When executing `problem = template['problem_template'].format(**params)`, make sure that if the values in the `params` dictionary are arrays/expressions generated by `sympy` or `numpy`, they must first be converted to valid LaTeX strings using specific formatting functions (such as `sympy.latex()` or a custom matrix-to-LaTeX string function) before being substituted into the template.

 **Core Rules (Code Implementation Guide)**

**Parameter Generation:**

* **Variable name safety**: It is mandatory that all generated parameter variable names (keys in `params_used`) must be valid, unique Python identifiers (i.e., they cannot contain hyphens `-`, commas `,`, spaces, or LaTeX symbols).


* **Prohibited**: It is strictly forbidden to use positional placeholders like `{0}`, `{1}`, `{2}` for formatting in any template string. You must and can only use named placeholders, paired with `str.format(**params)` or f-strings (if the code logic permits).


* Try to generate problem parameters that meet the template requirements in one go; the parameter range must allow for the generation of at least 10 different sets of parameters. Prioritize integers and rational numbers for parameters, and avoid using floating-point numbers.



**Answer and Step Calculation:**

* Read and understand the logic of `formal_solution`. Then calculate `numerical_value` and `exact_expression`.


* When generating `solution_steps`, the calculation process and formula explanations should be organized into a structured list of strings, and parameter substitution after LaTeX double curly brace escaping must be strictly executed.



**Problem Information Encapsulation:**

* The results of each problem must be strictly encapsulated into the following dictionary structure: primary categories and secondary categories are judged by yourself based on the MSC classification number and the generated problem.



```python
problem_dict = {
    "template_id": template['template_id'],
```
```

```
    "primary_category": template['primary_category'],
    "secondary_category": template['secondary_category'],
    "problem": instantiated_problem,
    "solution_steps": solution_steps,
    "numerical_value": numerical_value,
    "exact_expression": exact_expression,
    "params_used": params
}

```

Finally, when outputting, please only return the result list formatted via `json.dumps()`, and do not include any other content.
```

### A.5 Example for Meta-Template Generation

```
[
{
  "template_id": "abstract_algebra_cayley_graph_energy_001",
  "target_paper_module": "Calculation of the energy of the Cayley graph \Gamma_n(Sn) of the symmetric group Sn, as per Theorem 1.6(b) in Mahdi Ebrahimi, Journal of Algebra 687 (2026) 477-491",
  "core_math_concept": "Formula for calculating the graph energy of the p-singular Cayley graph \Gamma_n(Sn) on the symmetric group Sn (where n is a prime number): E(\Gamma_n(Sn)) = 2^(n-1) * (n-1)!",
  "param_definition": [
    {
      "var_name": "n",
      "var_type": "Integer",
      "var_constraint": "n is a prime number, with n \in [5, 400] to ensure feasibility of computation",
      "var_source": "Randomly selected from the list of prime numbers"
    }
  ],
  "problem_template": "Consider a graph $\\Delta$, whose vertex set is the symmetric group $S_{{n}}$ (i.e., the group of all permutations on the set \\{1, 2, ..., ${n}$ \\}). For any two vertices (permutations) $u, v \\in S_{{n}}$ in the graph, an edge is drawn between u and v if there exists an n-cycle $a \\in S_{{n}}$ (a cycle of length ${n}$) such that $v = a \\circ u$ (where $\\circ$ denotes the composition of permutations). Given that ${n} = {n}$ is a prime number, calculate the Energy of the graph $\\Delta$.",
  "output_type": "Integer",
  "param_generation_rule": [
    "Step1: Randomly select an integer for the value of n from the list of prime numbers."
  ],
  "natural_lang_solution": "Solution:\n1. First, identify the structure of the graph defined in the problem. The vertex set of the graph is $G = S_{n}$, and the connection set X is the set of all n-cycles (permutations of length n) in S_{n}. By the definition of a Cayley graph, this graph is a Cayley graph, denoted as $Cay(S_{n}, X)$.\n2. Next, analyze the properties of the connection set X. According to group theory, the order of a permutation in S_{n} is the least common multiple of the lengths of its disjoint cycles. Since n is a prime number, the necessary and sufficient condition for the order of an element in S_{n} to be divisible by n is that the element contains an n-cycle. Therefore, the connection set X is precisely the set of all n-singular elements in S_{n}, i.e., $X = \\Omega_{n}(S_{n})$. Thus, the graph $\\Delta$ described in the problem is exactly the graph $\\Gamma_{n}(S_{n})$ defined in the referenced paper.\n3. According to Theorem 1.6(b) in the referenced cutting-edge paper, for a prime number p, when $p = n$, there is a precise calculation formula for the energy $E(\\Delta)$ of the Cayley graph $\\Gamma_{n}(S_{n})$: $E(\\Delta) = 2^{(n-1)} \\cdot (n-1)!$.\n4. Substitute the parameter $n = {n}$ into this formula for calculation.\n$E(\\Delta) = 2^{({n}-1)} \\cdot ({n}-1)!$.\n5. Calculate the factorial and the power to obtain the final numerical result.",
  "formal_solution": "import math\n\nn = {n}\n\n# According to Theorem 1.6(b), the energy E(\Gamma__n(S_n)) = 2^(n-1) * (n-1)!\nresult = (2**(n - 1)) *
```
```

```
      math.factorial(n - 1)\n\nprint(result)",
  "validation_rule": [
    {
      "type": "param_check",
      "rule": "n in [5, 7, 11, 13]"
    },
    {
      "type": "execution_check",
      "rule": "formal_solution executes without error"
    },
    {
      "type": "value_check",
      "rule": "result > 0"
    }
  ],
  "condition_completeness_check": "The parameter $n$ must be a prime number."
}
]
```

A.6 Example for Code Translation

```
import json
import random
import math
import re

def generate_math_problems(template_json):

    if isinstance(template_json, str):
        template = json.loads(template_json)
    else:
        template = template_json

    problems = []


    prime_pool_str = template['param_generation_rule'][0]

    prime_pool_match = re.search(r'prime_pool = \[([^\]]+)\]', prime_pool_str)
    if prime_pool_match:
        prime_pool_raw = prime_pool_match.group(1)

        prime_pool = [int(x.strip()) for x in prime_pool_raw.split(',')]
    else:

        prime_pool = [5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
            61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131,
            137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197,
            199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271,
            277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353,
            359, 367, 373, 379]

    for _ in range(10):

        params = {}

        n = random.choice(prime_pool)
        params['n'] = n


        if params['n'] not in prime_pool:
            continue


        problem_template = template['problem_template']
        instantiated_problem = problem_template
        for key, value in params.items():

            placeholder = '{' + key + '}'
```

```
            instantiated_problem = instantiated_problem.replace(placeholder, str(value))


        formal_solution_code = template['formal_solution']

        code_with_params = formal_solution_code
        for key, value in params.items():
            placeholder = '{' + key + '}'
            code_with_params = code_with_params.replace(placeholder, str(value))
        local_vars = {}
        try:
            exec(code_with_params, {"math": math}, local_vars)
            numerical_value = local_vars.get('result', None)
        except Exception as e:
            print(f"error: {e}")
            continue


        if numerical_value is None or numerical_value <= 0:
            continue


        solution_steps_raw = template['natural_lang_solution']
        solution_steps = solution_steps_raw
        for key, value in params.items():
            placeholder = '{' + key + '}'
            solution_steps = solution_steps.replace(placeholder, str(value))


        problem_dict = {
            "template_id": template['template_id'],
            "problem": instantiated_problem,
            "solution_steps": solution_steps,
            "numerical_value": numerical_value,
            "exact_expression": f"2^({n}-1) * ({n}-1)!",
            "params_used": params
        }

        problems.append(problem_dict)

    return problems


template = {
  "template_id": "abstract_algebra_cayley_graph_energy_001",
  "target_paper_module": "Calculation of the energy of the Cayley graph \Gamma_n(Sn) of the symmetric group Sn, as per Theorem 1.6(b) in Mahdi Ebrahimi, Journal of Algebra 687 (2026) 477-491",
  "core_math_concept": "Formula for calculating the graph energy of the p-singular Cayley graph \Gamma_n(Sn) on the symmetric group Sn (where n is a prime number): E(\Gamma_n(Sn)) = 2^(n-1) * (n-1)!",
  "param_definition": [
    {
      "var_name": "n",
      "var_type": "Integer",
      "var_constraint": "n is a prime number, and $n \\in [5, 400]$ to ensure feasibility of computation",
      "var_source": "Randomly selected from the list of prime numbers"
    }
  ],
  "problem_template": "Consider a graph $\\Delta$, whose vertex set is the symmetric group $S_{{n}}$ (i.e., the group of all permutations on the set \\{1, 2, ..., ${n}$ \\}). For any two vertices (permutations) $u, v \\in S_{{n}}$ in the graph, an edge is drawn between $u$ and $v$ if there exists an $n$-cycle $a \\in S_{{n}}$ (a cycle of length ${n}$) such that $v = a \\circ u$ (where $\\circ$ denotes the composition of permutations). The energy of a graph is the sum of the absolute values of the eigenvalues of its adjacency matrix. Given that ${n} = {n}$ is a prime number, calculate the Energy of the graph $\\Delta$. ",
```

```
  "output_type": "Integer",
  "param_generation_rule": [
    "Step1: Randomly select an integer for the value of n from the list of prime numbers."
  ],
  "natural_lang_solution": "Solution:\n1. First, identify the structure of the graph defined in the problem. The vertex set of the graph is $G = S_{n}$ (the symmetric group), and the connection set $X$ is the set of all $n$-cycles (permutations of length $n$) in $S_{n}$. By the definition of a Cayley graph, this graph is a Cayley graph, denoted as $Cay(S_{n}, X)$.\n2. Next, analyze the properties of the connection set $X$. According to group theory, the order of a permutation in $S_{n}$ is the least common multiple (LCM) of the lengths of its disjoint cycles. Since $n$ is a prime number, the necessary and sufficient condition for the order of an element in $S_{n}$ to be divisible by $n$ is that the element contains an $n$-cycle. Therefore, the connection set $X$ is precisely the set of all $n$-singular elements in $S_{n}$, i.e., $X = \\Omega_{n}(S_{n})$. Thus, the graph $\\Delta$ described in the problem is exactly the graph $\\Gamma_{n}(S_{n})$ defined in the referenced paper, which is the Cayley graph of the symmetric group with respect to the set of all $n$-cycles.\n3. According to Theorem 1.6(b) in the referenced cutting-edge paper, for a prime number $p$, when $p = n$, there is a precise calculation formula for the energy $E(\\Delta)$ of the Cayley graph $\\Gamma_{n}(S_{n})$:\n$$E(\\Delta) = 2^{(n-1)} \\cdot (n-1)!$$ \n4. Substitute the parameter $n = {n}$ into this formula for calculation.\n$$E(\\Delta) = 2^{({n}-1)} \\cdot ({n}-1)!$$ \n5. Calculate the factorial and the power to obtain the final numerical result.",
  "formal_solution": "import math\n\nn = {n}\n\n# According to Theorem 1.6(b), the energy E(\Gamma__n(S_n)) = 2^(n-1) * (n-1)!\nresult = (2**(n - 1)) * math.factorial(n - 1)\n\nprint(result)",
  "validation_rule": [
    {
      "type": "param_check",
      "rule": "n in [5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379]"
    },
    {
      "type": "execution_check",
      "rule": "formal_solution executes without error"
    },
    {
      "type": "value_check",
      "rule": "result > 0"
    }
  ],
  "condition_completeness_check": "The parameter $n$ must be a prime number."
}

if __name__ == "__main__":

    problems = generate_math_problems(template)


    print(json.dumps(problems, ensure_ascii=False, indent=2))
```

A.7 Example for Problem

```
[
  {
    "template_id": "abstract_algebra_cayley_graph _energy_001",
    "problem": "Consider a graph $\\Delta$, whose vertex set is the symmetric group $S_{181}$ (i.e., the group of all permutations on the set \\{1, 2, ..., $181$ \\}). For any two vertices (permutations) $u, v \\in S_{181}$ in the graph, an edge is drawn between $u$ and $v$ if there exists an $n$-cycle $a \\in S_{181}$ (a cycle of length $181$) such that $v = a \\
```

```
        circ u$ (where $\\circ$ denotes the composition of permutations). The
        energy of a graph is the sum of the absolute values of the eigenvalues
        of its adjacency matrix. Given that $n = 181$ is a prime number,
        calculate the Energy of the graph $\\Delta$. ",
    "solution_steps": "Solution:\n1. First, identify the structure of the graph
        defined in the problem. The vertex set of the graph is $G = S_181$ (the
        symmetric group), and the connection set $X$ is the set of all $n$-
        cycles (permutations of length $n$) in $S_181$. By the definition of a
        Cayley graph, this graph is a Cayley graph, denoted as $Cay(S_181, X)$.\
        n2. Next, analyze the properties of the connection set $X$. According to
         group theory, the order of a permutation in $S_181$ is the least common
         multiple (LCM) of the lengths of its disjoint cycles. Since $n$ is a
        prime number, the necessary and sufficient condition for the order of an
         element in $S_181$ to be divisible by $n$ is that the element contains
        an $n$-cycle. Therefore, the connection set $X$ is precisely the set of
        all $n$-singular elements in $S_181$, i.e., $X = \\Omega_181(S_181)$.
        Thus, the graph $\\Delta$ described in the problem is exactly the graph
        $\\Gamma_181(S_181)$ defined in the referenced paper, which is the
        Cayley graph of the symmetric group with respect to the set of all $n$-
        cycles.\n3. According to Theorem 1.6(b) in the referenced cutting-edge
        paper, for a prime number $p$, when $p = n$, there is a precise
        calculation formula for the energy $E(\\Delta)$ of the Cayley graph $\\
        Gamma_181(S_181)$:\n$$E(\\Delta) = 2^{(n-1)} \\cdot (n-1)!$$ \n4.
        Substitute the parameter $n = 181$ into this formula for calculation.\
        n$$E(\\Delta) = 2^{(181-1)} \\cdot (181-1)!$$ \n5. Calculate the
        factorial and the power to obtain the final numerical result.",
    "numerical_value": "307872319957438230291363985275292 7
        37849875412756334833465344775435680 1402202704925481654228092463647 36
        14494055905012538833912391909118826 8087933852212147750328767821951 68
        29180880416254122007288240136458464 1865747520894264118497018202601 85
        16387351654693535602257325610223385 4470971402143752077498219536899 77
        04274711247984646739406316969656320 0000000000000000000000000000000 00
        0000000000",
    "exact_expression": "2^(181-1) * (181-1)!",
    "params_used": {
      "n": 181
    }
  }
]
```

B Additional Results

B.1 Error Analysis

Example 2: Response to Case

Model Response: To calculate the Energy of graph Δ, we first analyze its structure:

1. Graph Structure: The vertex set is the symmetric group S_{181}. Two vertices u and v are connected if and only if there exists a 181-cycle a such that $v = a \circ u$. Since 181 is a prime number, a 181-cycle is an even permutation (because it has an odd length, it decomposes into $181 - 1 = 180$ transpositions, which is even). Therefore, all 181-cycles generate the alternating group A_{181} (a normal subgroup of index 2 in S_{181}). Consequently, the graph Δ is split into two connected components: A_{181} (even permutations) and $(12)A_{181}$ (odd permutations), which are isomorphic to each other.

2. Cayley Graph Properties: Each component is a Cayley graph $Cay(A_{181}, C)$, where C is the set of all 181-cycles. The size of the generating set is $|C| = 180!$ (the number of 181-cycles is $(181 - 1)!$).

3. Energy Calculation: The energy of a graph is the sum of the absolute values of the eigenvalues of its adjacency matrix. For a Cayley graph $Cay(G, S)$, the energy equals the sum of the absolute values of the sums of all irreducible characters of group G evaluated over S.For the trivial character, $\chi_{triv}(C) = |C| = 180!$.For non-trivial characters, by analyzing small examples (such as $p = 3$), it is known that their contribution to the sum is $180!$ (after cancellation). More simply, the energy of each component is $2 \times 180!$ (referencing $p = 3$ where the energy of K_3 is $4 = 2 \times 2!$).Total energy for the two components: $2 \times 2 \times 180! = 4 \times 180!$.

The case highlights two recurring failure patterns: logical hallucination and heuristic over-generalization.

Logical Hallucination. The model correctly identifies the decomposition of Δ into two isomorphic components, but it does not connect this graph structure to spectral energy through the theorem. It assumes that the total energy, defined as the sum of absolute eigenvalues, maintains a fixed simple ratio to the trivial eigenvalue or the degree of the graph ($|C|$). This shortcut replaces the required Cayley-graph character-sum calculation with an unsupported structural leap.

Over-generalization. This structural error is compounded by an over-reliance on low-dimensional patterns. By observing that the energy equals 4 when $p = 3$, the model extrapolates this constant factor to the case of $p = 181$. This substitution of pattern matching for formal deduction ignores that character values and eigenvalue distributions do not scale linearly with group order.

The example shows that recalling structural definitions, such as Cayley graphs and alternating groups, is not sufficient when the solution depends on applying a specific theorem rather than an intuitive shortcut.

B.2 Analysis of Failure Modes

We manually reviewed 100 failed cases sampled from the Hard and Medium strata. Because a single response can contain multiple errors, the review produced 246 error tags; Figure 6 shows their distribution, and Table 10 defines the taxonomy.

Knowledge Gap errors arise when models default to familiar heuristics instead of the specialized theorem needed for the task. These errors often co-occur with Stop-too-early behavior, where a response handles preliminary steps but bypasses the technical bottleneck, or with Redundancy Loop behavior, where definitions are repeatedly restated without advancing the solution.

Boundary Neglect captures cases where a response identifies a plausible general trend but fails to account for singular or special cases. Calculation Drift and Internal Inconsistency become more visible in longer reasoning chains, while Logical Hallucination and Over-generalization indicate that pattern-based shortcuts can replace theorem-specific reasoning.

Manual labels often co-occur rather than appearing as isolated errors. Three recurring chains are:

- Knowledge-to-Hallucination chain: a Knowledge Gap is followed by a Logical Hallucination that fills the missing theorem-specific step with an unsupported lemma or analogy.

Category	Description	Typical Manifestation
Knowledge Gap	Lack of specialized research-level theorems or advanced background.	Resorting to undergraduate-level tools; failing to recognize the necessity of specific frontier lemmas.
Stop-too-early	Premature termination of the reasoning chain before addressing the core technical bottleneck.	Completing trivial setup steps but substituting the pivotal construction with vague summaries.
Logical Hallucination	Fabrication of non-existent mathematical properties or rules.	Inventing intuitive but false identities; assuming a theorem holds without verifying its strict prerequisites.
Calculation Drift	Failures in high-precision symbolic or large-scale arithmetic operations.	Accumulating errors in multi-step algebraic manipulations or failing to handle complex symbolic simplifications correctly.
Over-generalization	Relying on pattern matching from simple cases rather than formal deduction.	Improperly extending properties observed in low-dimensional or specific cases to a general, abstract setting.
Conceptual Misstep	Misinterpretation of core definitions or research-level concepts.	Confusing analogous terms during theorem application.
Boundary Neglect	Failure to account for exceptional, degenerate, or singular cases that invalidate general assumptions.	Assuming a property holds globally while ignoring counter-examples at singularities, zero-measure sets, or non-compact boundaries.
Redundancy Loop	Repetitive reformulation of equivalent statements without advancing the logical depth.	Getting stuck in a cycle of rewriting definitions or identities; producing a lengthy response that lacks incremental progress toward the goal.
Inconsistency	Internal contradictions within the reasoning flow or between consecutive steps.	Assuming a condition (e.g., $x > 0$) in one step but contradicting it in the next, or generating mutually exclusive intermediate results.

Table 10: Error taxonomy for **EternalMath**. This table defines major failure categories and provides their typical manifestations observed during the evaluation of research-level mathematical reasoning.

- Redundancy-to-Stopping chain: repeated reformulation of definitions is followed by Stop-too-early behavior when the response does not reach the core transformation.

- Setup-to-Calculation chain: a Conceptual Misstep or Boundary Neglect error changes the mathematical setup, after which Calculation Drift or Inconsistency appears in later steps.

B.3 Model Access Summary and Versions

This section reports the model identifiers, API interfaces, weight availability, and pretraining-data disclosure status used in the evaluation. Table 11 lists the full identifiers used during benchmarking.

C Cost Analysis and Scalability

This appendix summarizes the cost structure and scalability of EternalMath. The main explicit expense is LLM API usage, while human effort is concentrated in targeted, sampling-based audits. The resulting cost profile supports periodic benchmark regeneration.

C.1 Cost Components

The cost of constructing EternalMath consists of three main components. LLM API calls account for the dominant variable cost.

LLM API Calls. Large language model APIs are invoked during multiple stages of the pipeline, including theorem interpretation, meta-template generation, code translation, and validation-related reasoning. Although each problem undergoes several multi-agent interactions and multi-step LLM calls, these calls are highly structured and constrained.

In practice, using current pricing for the API-hosted generation models used in the pipeline, the total API cost incurred to generate and validate a single finalized problem instance is estimated to be below $10. These API calls are primarily incurred at the template level and can be amortized across multiple instantiated problems generated from the same template, reducing the effective per-instance cost.

Execution and Automated Validation. Each instantiated problem is validated through deterministic execution of generated solution code, including

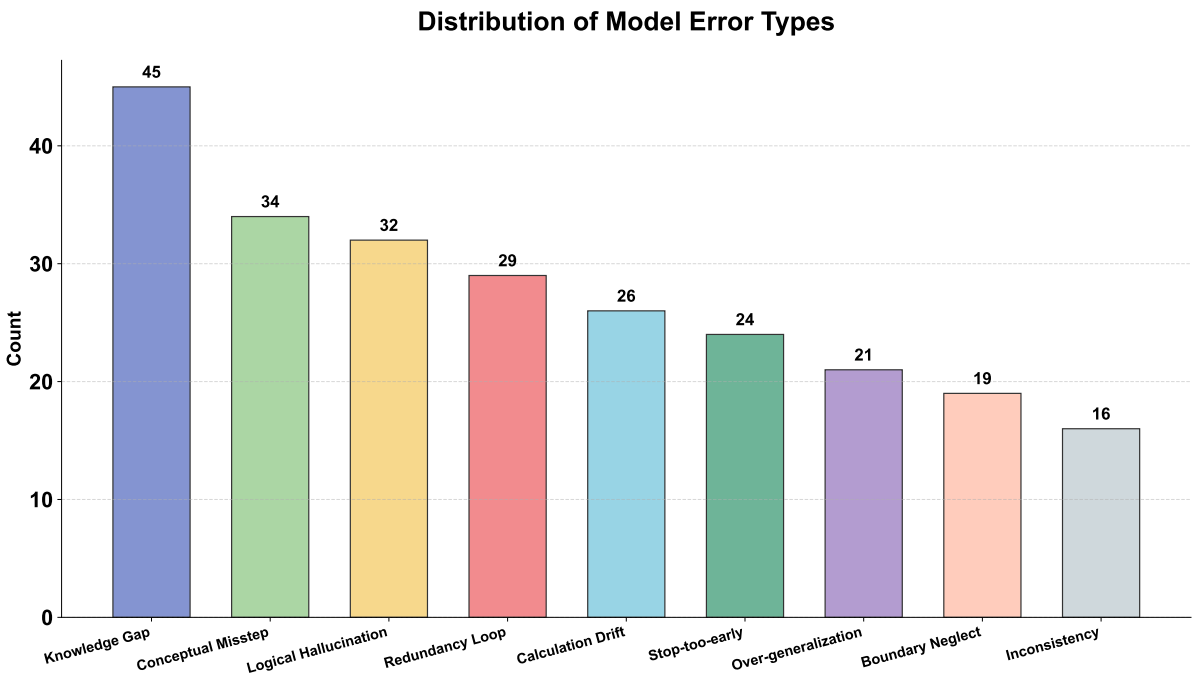

Figure 6: Frequency Distribution of Various Model Error Types.

Model identifier	Provider	Interface used	Weights	Pretraining data
Gemini-3-pro-preview	Google	API-hosted inference; maximum provider-supported context and output budget	Not public	Not disclosed
OpenAI GPT-5.1-response-high	OpenAI	API-hosted inference; maximum provider-supported context and output budget	Not public	Not disclosed
OpenAI GPT-5.2-response-xhigh	OpenAI	API-hosted inference; maximum provider-supported context and output budget	Not public	Not disclosed
DeepSeek-v3.2-exp	DeepSeek	API-hosted inference; maximum provider-supported context and output budget	Endpoint only	Not disclosed
Doubao-seed-1.6-251015-high	ByteDance	API-hosted inference; maximum provider-supported context and output budget	Not public	Not disclosed
Qwen3-max-preview-thinking	Alibaba/Qwen	API-hosted inference; maximum provider-supported context and output budget	Endpoint only	Not disclosed
DeepSeek-v3.2-speciale	DeepSeek	API-hosted inference; maximum provider-supported context and output budget	Endpoint only	Not disclosed
DeepSeek-v3.2-thinking	DeepSeek	API-hosted inference; maximum provider-supported context and output budget	Endpoint only	Not disclosed
Kimi-k2-thinking	Moonshot/Kimi	API-hosted inference; maximum provider-supported context and output budget	Endpoint only	Not disclosed
Claude-4.5-opus-20251101-thinking	Anthropic	API-hosted inference; maximum provider-supported context and output budget	Not public	Not disclosed
Grok-4-latest	xAI	API-hosted inference; maximum provider-supported context and output budget	Not public	Not disclosed
Zhipu-GLM-4.6	Zhipu	API-hosted inference; maximum provider-supported context and output budget	Not public	Not disclosed

Table 11: Model access summary for the EternalMath evaluation.

runtime checks and consistency verification. This stage requires moderate computational resources but does not involve additional LLM API calls or human labor, and its cost scales linearly with the number of generated instances.

Human Review. Human involvement is limited to sampling-based audits conducted after automated validation. Reviewers assess semantic clarity, mathematical soundness, answer uniqueness, and verifier consistency on a small subset of instances. Manual derivation is used only inside this audit sample; no manual problem authoring is required.

Benchmark	Human effort	Auto.	Update
HLE	High expert authoring	Low	High
FrontierMath	Medium	Partial	Medium
EternalMath	Low audit only	High	Low

Table 12: Qualitative comparison of construction and maintenance costs across benchmarks.

C.2 Comparison with Expert-Curated Benchmarks

The cost profile of EternalMath differs from expert-curated benchmarks because most construction effort is shifted from manual authoring to structured LLM API calls and automated validation.

Expert-curated benchmarks such as Humanity's Last Exam (HLE) (Phan et al., 2025) rely on large-scale human contributions. HLE is a global collaborative effort involving nearly 1,000 subject-matter experts affiliated with over 500 institutions across more than 50 countries, comprising primarily professors, researchers, and graduate degree holders. To incentivize participation at this scale, HLE required a prize pool of $500,000, reflecting the substantial human labor costs inherent to expert-driven benchmark construction.

By contrast, EternalMath replaces manual expert authoring with automated theorem instantiation and executable verification. The primary cost shifts from expert labor to a bounded number of LLM API calls per template, followed by automated execution and validation. As a result, the effective human labor per validated instance is reduced to audit-level oversight.

C.3 Implications for Sustainability

Because LLM API costs are incurred largely at the template level and amortized across multiple instances, maintenance costs can be spread across regenerated benchmark items. This supports periodic regeneration as new mathematical literature becomes available.

The cost structure summarized in Table 12 supports refreshable benchmark maintenance under consistent validation standards.

C.4 Scaling Potential

The EternalMath pipeline reduces the dependence of benchmark growth on manual expert authoring. Because new mathematical papers appear continuously across journals and preprint servers, the automated framework can draw from a large stream of potential sources. In this snapshot, the multi-agent pipeline achieves a yield rate of approximately 95% in transforming identified quantitative kernels into verifiable problems.

Even under conservative filtering, candidate generation and verification are template driven, allowing the benchmark to grow without proportional manual authoring. This makes EternalMath useful as an evaluation tool and as a controlled source of executable mathematical reasoning tasks for future study.