METABENCH-HARNESS: UNLOCKING END-TO-END OPTIMIZATION OF BENCHMARK HARNESSES

Anonymous authors
Paper under double-blind review

ABSTRACT

Rapid progress in Large Language Models (LLMs) is saturating static benchmarks faster than they can be designed. While existing automated evolution frameworks attempt to generate harder questions by perturbing individual tasks, they remain constrained by rigid, hard-coded generation rules. Moving beyond the evolution of isolated tasks, we propose to optimize the benchmark generation workflow itself end to end with METABENCH-HARNESS, a dual-loop search framework. Specifically, the inner loop utilizes a benchmark harness to generate a new benchmark in each round, while the outer meta-harness orchestration layer iteratively refines and searches over harness implementations based on historical evolution trajectories. By applying METABENCH-HARNESS to the competitive programming CodeContests and Olympiad mathematics AIME-2024 datasets, we demonstrate that the evolved benchmarks are challenging and discriminative for frontier models. Trajectory and quality analyses verify that METABENCH-HARNESS enables multi-dimensional evolution, steadily improving evolution reasonableness, benchmark competency, and evaluator robustness across successive rounds. Furthermore, case studies reveal its effective utilization of diverse difficulty levers to reframe problems and elevate required capabilities. Ultimately, this work provides a solution to the pressing challenge of benchmark saturation.

1 INTRODUCTION

Recent breakthroughs in Large Language Models (LLMs) (Achiam et al., 2023) have showcased exceptional capabilities across diverse applications, from creative text generation to intricate problem-solving. Consequently, dynamic and rigorous evaluation frameworks have emerged as a vital research domain (Lou et al., 2025; Chang et al., 2024). Effective assessment frameworks offer deep insights into model strengths and limitations, assist practitioners in selecting suitable LLMs (Zheng et al., 2023), and provide directional guidance for future architectural and algorithmic improvements, as highlighted in recent taxonomies addressing the evaluation trilemma (Tiwari et al., 2026).

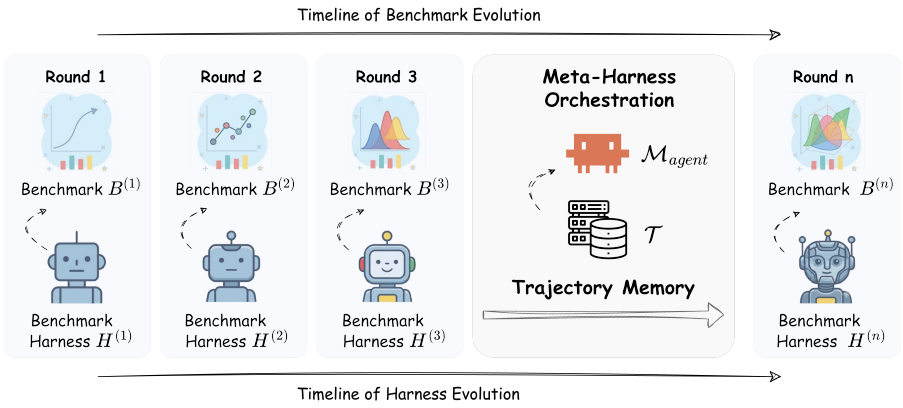


Figure 1: Overview of the METABENCH-HARNESS framework.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

While various benchmarks have been introduced to evaluate LLMs, traditional static datasets are increasingly inadequate due to the rapid evolution of model capabilities and widespread data contamination (Cheng et al., 2025), which occurs when public test sets inadvertently infiltrate training corpora and result in skewed evaluations. Although existing methods attempt to address these challenges through perplexity-based metrics or fine-grained semantic detection (Di et al., 2026), they remain limited by an over-reliance on static rules or poor task generalization. Consequently, while dynamic paradigms have emerged (Shi et al., 2026), preventing benchmark saturation through continuous and robust evolution remains a critical and largely unresolved challenge.

To meet this need, recent efforts have increasingly turned toward automated and self-evolving benchmarks designed to renew evaluation. However, current approaches still exhibit several limitations: frameworks like Benchmark Self-Evolving (Wang et al., 2025) and AutoEvoEval (Wu & Liu, 2025) apply predefined atomic operations that primarily probe robustness rather than genuinely scaling task complexity; specialized solutions like AdamMeme (Chen et al., 2025) are restricted to narrow domains such as meme harmfulness; EvoCodeBench (Li et al., 2024) relies on fixed ingestion schedules rather than model-driven exploration; and TRACE (Guo et al., 2026) is confined to test-time exploration within specific real environments. However, reliance on these rigid and domain-specific mechanisms fails to provide a comprehensive solution to the ongoing risk of benchmark saturation.

Overcoming these fragmented limitations requires moving beyond ad-hoc solutions. A practical benchmark generation framework must be grounded in three core pillars that directly address the aforementioned gaps: (1) **Evolution**, replacing rigid fixed processes and environment-restricted limits with dynamic, model-driven controllers that continuously adapt as target LLMs advance; (2) **Discriminability**, ensuring generated queries effectively test underlying capabilities rather than merely increasing difficulty, moving beyond superficial robustness checks; and (3) **Diversity**, ensuring that the evolved benchmarks and problems encompass diverse problem-solving techniques, target varying capability dimensions across tasks, and dynamically utilize different difficulty levers, rather than relying on uniform or narrow modifications. Our main contributions are threefold:

- We propose METABENCH-HARNESS (Figure 1), a framework that operates over dual evolving timelines of benchmarks and harnesses. It orchestrates multi-round harness generation by feeding accumulated trajectory memory back into the search process, dynamically controlling access to prior code and execution traces to improve LLM benchmark quality.
- Applying METABENCH-HARNESS to the CodeContests and AIME-2024 benchmarks demonstrates its superior effectiveness compared to previous static evolution methods in benchmark generation (Table 1), while consistently producing challenging and discriminative tasks that induce performance degradation in prominent LLM systems (Table 2, 3).
- Through comprehensive analyses across trajectories, quality assessment, and quantitative case studies, we reveal that our framework not only steadily improves benchmark competency and evaluator robustness, but also effectively utilizes difficulty levers to reframe problems. Ultimately, it provides a solution to benchmark saturation.

2 RELATED WORK

Static vs. Dynamic Benchmark Generation. Recent developments in dynamic evaluation seek to counter the swift saturation of static datasets. Early efforts like Benchmark Self-Evolving (Wang et al., 2025) and AutoEvoEval (Wu & Liu, 2025) use predefined atomic operations for NLP reasoning and multiple-choice QA, testing surface-level robustness rather than scaling task complexity. Specialized frameworks target specific niches, such as AdamMeme (Chen et al., 2025) leveraging multi-agent collaboration for safety probing, or EvoCodeBench (Li et al., 2024) refreshing software benchmarks at rigid temporal intervals without active model feedback. While advanced test-time exploration approaches like TRACE (Guo et al., 2026) pioneer autonomous task generation, they remain bound by predefined generative pathways. Overcoming these restrictions requires a fully dynamic pipeline. Through an outer-loop meta-harness orchestration that selectively controls the proposer's access to historical code and execution traces, METABENCH-HARNESS enables a truly dynamic benchmark generation pipeline that continuously adapts to target LLM advancements.

Adaptive Utilization of External Knowledge. Several studies demonstrate that models benefit from treating large repositories or lengthy contexts as dynamically accessed external assets rather

than consuming them in a single pass. While early paradigms include retrieval-augmented generation (Lewis et al., 2020) and alternating retrieval-reasoning (Trivedi et al., 2023), recent advances further introduce active associative discovery (Li et al., 2026) for fragmented evidence and targeted delta planning (Chou et al., 2026) for information gaps. Together with memory-enabled agents (Packer et al., 2023) and recursive models (Zhang et al., 2025a), these methods highlight a shift toward adaptive context interaction. METABENCH-HARNESS applies this retrieval pattern within a benchmark-driven harness engineering loop, granting the proposer unrestricted access to historical code, scores, and execution traces to discover and improve end-to-end context handling procedures.

Search Over Executable Code. Modern approaches increasingly explore executable code spaces to discover functions, workflows, or agent architectures. Early investigations integrated large models as mutation and crossover engines within evolutionary program search (Lehman et al., 2024). Subsequent techniques advanced to evolving specific functions inside fixed code frameworks (Romera-Paredes et al., 2024), deploying meta-agents to synthesize new agents from historical findings (Hu et al., 2025), or optimizing workflow graphs for agentic systems (Zhang et al., 2025b). Another research direction focuses on memory structures that enable continual-learning agents to retain information across multiple tasks (Zhang et al., 2026; Xiong et al., 2026). Closely related is Meta-Harness (Lee et al., 2026), which searches over domain-specific task harnesses using a minimal outer loop with unrestricted filesystem access. While Meta-Harness focuses on solving individual downstream tasks for leaderboard performance, our work shifts the objective toward the automated discovery and evolution of benchmark generation pipelines. By leveraging an open-ended search strategy, METABENCH-HARNESS enables a comprehensive exploration of full benchmark harness implementations, allowing the generation framework itself to continuously self-evolve.

3 METABENCH-HARNESS: HARNESS FOR BENCHMARK GENERATION

This section details METABENCH-HARNESS, our proposed method for automated benchmark generation. As shown in Figure 1, the proposed design integrates three core components into a pipeline: the benchmark harness, which acts as an executable program that dynamically transforms and constructs the target benchmark from seeds; a trajectory memory that records historical optimization trajectories; and a meta-harness orchestration that drives the iterative refinement process.

Objective. Our framework employs an LLM agent $\mathcal{M}_{\text{agent}}$ to iteratively propose benchmark harnesses. The objective is straightforward: to find the harness that poses challenging questions to the underlying proxy models while maintaining discriminability among the models. To guide the optimization, we maintain a trajectory memory $\mathcal{T}$ that stores past iterations of proposed harnesses, generated benchmarks, and their feedback. At each iteration, the agent conditions on this memory to generate a new candidate harness $H \sim p_{\mathcal{M}}(\mathcal{T})$. The objective of our benchmark harness optimization is to find the optimal harness H^* that maximizes the expected reward:

$$H^* = \arg\max_{H} \mathbb{E}_{B \sim H(B_{\text{seed}})} \left[r(s_1, s_2) \right]$$

where B_{seed} is the initial seed benchmark, and $B \sim H(B_{\text{seed}})$ denotes the target benchmark instantiated by executing H as a stateful program. The variables s_1 and s_2 represent the respective validation scores of two proxy models of varying capabilities, denoted as $\mathcal{M}_1$ and $\mathcal{M}_2$, when validated on B. Here, $r(s_1, s_2)$ serves as the *extrinsic execution reward*, an empirical metric that scores the generated benchmark, where it jointly maximizes the score discrepancy between models to ensure discriminability while minimizing their average performance to guarantee challenge, so that a higher r signifies a more ideal benchmark harness. To realize this optimization objective, our framework introduces a dual-level workflow (Figure 2). The system operates through two interconnected stages: an outer loop for Meta-Harness Search Orchestration (Figure 2a), and an inner loop for Benchmark Generation Governed by Agentic Harness (Figure 2b).

Meta-Harness Orchestration Guided by Trajectory Memory. To drive the iterative evolution of the benchmark harness over time, the meta-harness orchestration mechanism employs an LLM-based agent $\mathcal{M}_{\text{agent}}$ conditioned on an accumulated trajectory memory $\mathcal{T}$ to propose candidate harness updates. Unlike prior systems relying on manual search heuristics to drive improvements, we delegate diagnosis and harness generation directly to the agent through a cohesive four-step workflow. First, acting as the *proposer*, the agent inspects uncompressed raw artifacts from $\mathcal{T}$, including past harness code, generated benchmarks, and execution traces, to generate candidate updates to the

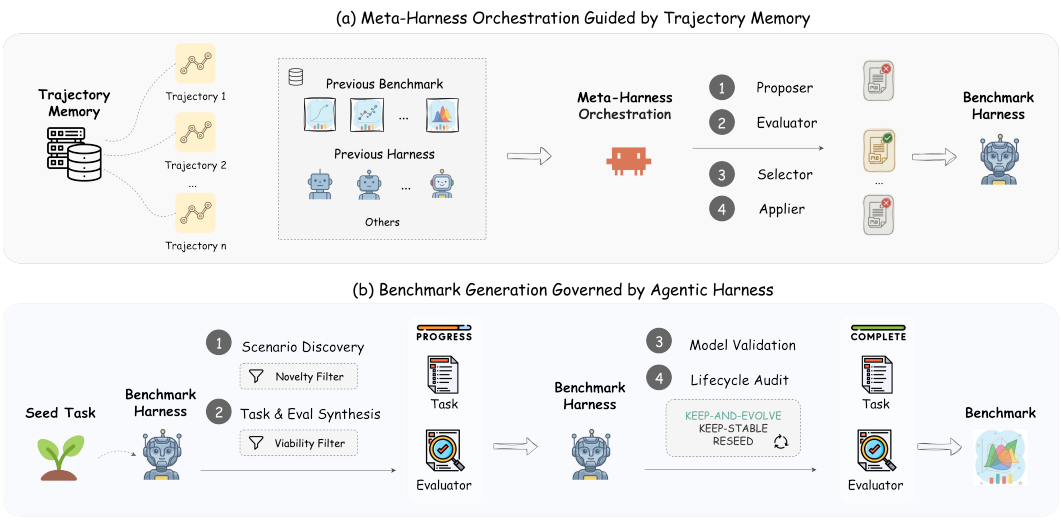


Figure 2: The dual-loop procedure of METABENCH-HARNESS. (a) Meta-Harness Orchestration: The outer loop where the LLM agent, conditioned on trajectory memory, proposes harness updates. (b) Benchmark Generation: The inner loop where the evolved harness drives benchmark synthesis.

incumbent harness $H^{(t-1)}$. Second, an *evaluator* simply scores and ranks these proposals to help identify promising candidates. Third, a *selector* chooses the best benchmark harness candidate from the rankings. Finally, the *applier* applies the approved change to instantiate the evolved benchmark harness $H^{(t)}$, which subsequently drives the inner execution loop to generate the new benchmark checkpoint $B^{(t)}$. Relying on the uncompressed, accumulated trajectory memory, our meta-harness orchestration enables precise error diagnosis, prevents redundant search paths, and effectively guides the selection toward optimal benchmark harness evolution across generations.

Benchmark Generation Governed by Agentic Harness. To generate high-quality benchmarks using the evolving benchmark harness, the inner execution loop runs once per optimization round t using the updated harness $H^{(t)}$, a seed benchmark B_{seed}, and proxy models $\mathcal{M}_1$ and $\mathcal{M}_2$. Operated through a coding agent with shell-level access, this agentic harness workflow executes a four-stage process to produce the benchmark $B^{(t)}$ and its validation- and audit-derived telemetry for subsequent optimization rounds. Overall, structured into two core phases, the pipeline balances generative creation with rigorous governance. The first part, encompassing *Scenario Discovery* and *Task & Eval Synthesis*, systematically generates novel tasks paired with deterministic evaluators. Subsequently, the second part, comprising *Model Validation* and *Lifecycle Audit*, executes these tasks and governs their lifecycles to produce critical execution telemetry, feeding this data directly back into the outer loop of meta-harness orchestration to drive subsequent iterations. First, during *Scenario Discovery*, the agent explores new problem directions for task evolution by mutating selected seed items into diverse candidate scenarios, with redundant copies and invalid replays excluded by the novelty filter. Second, building upon these candidate scenarios, *Task & Eval Synthesis* synthesizes concrete task families paired with shared deterministic evaluators, while an internal validity filter discards failing tasks. Third, *Model Validation* executes the generated benchmark suite across the proxy models to capture responses, intermediate artifacts, and objective metrics, thereby producing the execution telemetry required by the outer loop optimization. Fourth, building directly upon these validation outcomes, the final stage, *Lifecycle Audit*, assigns discrete scenario-level management states (KEEP-AND-EVOLVE, KEEP-STABLE, RESEED) to each scenario, sustainably steering the benchmark's evolution and maximizing search efficiency rather than simply discarding search budgets. Ultimately, this continuous cycle of generation, validation, and curation feeds directly into the meta-harness, driving progressive improvements across successive optimization rounds.

End-to-End Optimization Procedure and Deliverables. To clearly illustrate the dual-loop operation, we formalize the procedure as pseudocode in Algorithm 1. The procedure begins by initializing the trajectory memory using a predefined benchmark harness and a seed dataset. The optimization then executes over T rounds, with each round comprising two main stages: harness proposal and

Algorithm 1 The METABENCH-HARNESS benchmark harness optimization procedure.

Require: Agent $\mathcal{M}_{\text{agent}}$, Engine R_G, Seed dataset B_{seed}, Proxy models $(\mathcal{M}_1, \mathcal{M}_2)$, Max round T

Initialize: $H^{(0)}$; $B^{(0)} \leftarrow R_G(H^{(0)}, B_{\text{seed}})$; $\mathcal{T} \leftarrow \{(H^{(0)}, B^{(0)}, \text{ValidateAndAudit}(B^{(0)} \mid \mathcal{M}_1, \mathcal{M}_2))\}$
for $t = 1 \dots T$ **do**
  $\mathcal{M}_{\text{agent}}$ proposes candidates $\{\widetilde{H}_1, \dots, \widetilde{H}_K\}$ based on $H^{(t-1)}$ ▷ Outer loop harness proposal
  for $i = 1 \dots K$ **do**
    if $\widetilde{H}_i$ passes interface validation **then**
      $v_i \leftarrow \text{Evaluate}(\widetilde{H}_i \mid \mathcal{T})$ ▷ Harness proposal evaluation
  $i^* \leftarrow \arg\max_i v_i$; $H^{(t)} \leftarrow \widetilde{H}_{i^*}$ ▷ Select best benchmark harness proposal
  $B^{(t)} \leftarrow \emptyset$
  for each scenario $s \in B^{(t-1)}$ **do**
    $s^{(t)} \leftarrow R_G(H^{(t)}, s, B_{\text{seed}})$ ▷ Inner loop benchmark generation
    $B^{(t)} \leftarrow B^{(t)} \cup \{s^{(t)}\}$
  $r^{(t)} \leftarrow \text{ValidateAndAudit}(B^{(t)} \mid \mathcal{M}_1, \mathcal{M}_2)$; $\mathcal{T} \leftarrow \mathcal{T} \cup \{(H^{(t)}, B^{(t)}, r^{(t)})\}$ ▷ Validation and Audit
return $(B^{(T)}, H^{(T)})$

benchmark generation. During the harness proposal stage, the LLM agent generates K candidate configurations, and an LLM-as-a-judge mechanism evaluates these candidates to select the optimal benchmark harness. In the subsequent generation stage, the inner loop leverages this selected harness and seed data to evolve prior scenarios, accumulating them into the updated benchmark suite. This suite then undergoes a *ValidateAndAudit* phase, where proxy models execute the benchmark to provide empirical feedback for subsequent evolution. This validation score, alongside the round's harness and benchmark states, is appended to the trajectory memory to guide future search iterations. Upon completing all T rounds, the procedure outputs the fully optimized evaluation benchmark and its corresponding harness configuration. Note that although the trajectory memory comprehensively logs all execution artifacts in practice, its representation is simplified here for clarity.

Module Instantiation of METABENCH-HARNESS. Within our framework, the core modules are operationalized by an LLM agent ($\mathcal{M}_{\text{agent}}$) guided by structured Markdown scaffolds. These templates explicitly define the expected functionality, inputs, outputs, and operational constraints for each distinct module. Furthermore, because our optimization objective is to discover a harness that yields challenging yet highly discriminative questions, we employ two distinct LLMs to serve as the underlying proxy models ($\mathcal{M}_1$ and $\mathcal{M}_2$) during the `ValidateAndAudit` phase. Specific model choices, system prompts, and detailed empirical setups are deferred to Section 4 and Appendix A.

4 EXPERIMENTS

In our implementation of METABENCH-HARNESS, we instantiate the central agent ($\mathcal{M}_{\text{agent}}$) with `Claude-Sonnet-5` (Anthropic, 2026a), and employ both `Claude-Sonnet-5` and `GPT-5.4` (OpenAI, 2026) as the proxy models ($\mathcal{M}_1$ and $\mathcal{M}_2$) to provide the necessary difficulty and discrimination signals during the search phase. Driven by this setup, we assess the framework's efficacy by evolving benchmarks in the domains of coding and mathematics. Specifically, we utilize CodeContests (Li et al., 2022) and AIME-2024 (Jia, 2024) as our foundational seed datasets, executing the optimization procedure for $T = 10$ and $T = 5$ rounds respectively, to yield the final suites denoted as MB-CodeContests and MB-AIME. Because the framework equips each newly generated task with a programmatic evaluator, a model's prediction is considered correct only upon passing this execution check; consequently, we adopt pass rate (accuracy) as our primary evaluation metric.

To verify that the evolved benchmarks remain challenging and discriminative beyond the inner-loop proxy models, we evaluate several held-out models, including `DeepSeek-V3.1` (DeepSeek-AI, 2024), `Kimi-K2.5` (Bai et al., 2026), and `Gemini-2.5-Flash` (Google, 2026), alongside the distilled reasoning models `DeepSeek-R1-Distill-Qwen-32B` (DeepSeek-AI, 2025) and `DeepSeek-R1-Distill-Qwen-7B` (DeepSeek-AI, 2025). For fair comparison, all held-out evaluations use zero-temperature decoding under a uniform inference budget, enforced by a 32,000-token output cap and a 900-second timeout per call. Further details are in Appendix B.

Table 1: Model performance on benchmarks evolved via different evolution methods.

Evolution Methods	Note	DeepSeek	Kimi	Gemini	Qwen-32B	Qwen-7B
Baseline	AIME-2024	0.900	0.900	0.833	0.700	0.533
TRACE	Reported	N/A	N/A	N/A	0.533	0.393
AutoEvoEval	Reproduced	0.700	0.633	0.500	0.400	0.367
Naïve Prompting	Ablation	1.000	1.000	1.000	0.875	0.750
Fixed Harness	Ablation	0.817	0.767	0.633	0.600	0.133
METABENCH-HARNESS	Our	0.322 ±0.038	0.544 ±0.217	0.144 ±0.019	0.089 ±0.019	0.044 ±0.051

* The specific versions of the held-out models evaluated here are DeepSeek-V3.1, Kimi-K2.5, Gemini-2.5-Flash, DeepSeek-R1-Distill-Qwen-32B, DeepSeek-R1-Distill-Qwen-7B.

Table 2: Model performance across evolved benchmarks. Models are grouped into reasoning (R), non-reasoning (NR), and adaptive (AR) types. "Involve." denotes whether the LLM was involved in the evolution loop; "Base." is the seed benchmark, with "Early/Late" as different evolution rounds.

Model	Type	Loop	MB-CodeContests			MB-AIME		
			Base.	Early	Late	Base.	Early	Late
Claude-Sonnet-5	AR	✓	0.835	0.948	0.839	0.733	0.750	0.633
GPT-5.4	NR	✓	0.586	0.591	0.293	0.067	0.125	0.100
DeepSeek-V3.1	NR	×	0.736	0.845	0.396	0.667	0.125	0.133
Kimi-K2.5	NR	×	0.694	0.834	0.739	0.833	0.333	0.167
Gemini-2.5-Flash	NR	×	0.458	0.484	0.278	0.767	0.167	0.100
DeepSeek-V3.1	R	×	0.788	0.946	0.770	0.900	0.625	0.333
Kimi-K2.5	R	×	0.935	0.948	0.852	0.900	0.667	0.533
Gemini-2.5-Flash	R	×	0.799	0.728	0.585	0.833	0.167	0.167

4.1 MAIN RESULT: PERFORMANCE OF HELD-OUT MODELS ON EVOLVED BENCHMARKS

We evaluate METABENCH-HARNESS in terms of evolution strategies (Table 1) and different evolution rounds (Table 2), analyzing how different approaches push task complexity and discriminate model capabilities. Additional model and experiment configurations are available in Appendix B.1.

Model Performance Across Different Evolution Methods. We compare model performance across benchmark variants evolved from the AIME-2024 dataset, primarily focusing on METABENCH-HARNESS and its two ablated baselines, alongside AutoEvoEval (Wu & Liu, 2025) and TRACE (Guo et al., 2026). The Baseline refers to the original, unevolved AIME-2024 evaluation dataset. Naïve Prompting consists solely of *Scenario Discovery* and *Task & Eval Synthesis* (Figure 2b), omitting execution feedback entirely. Fixed Harness omits the outer-loop meta-harness orchestration (Figure 2a) and relies only on a static harness (Figure 2b). For AutoEvoEval, we select code- and math-related execution units for benchmark evolution, whereas for TRACE, we report their published results. Evaluation results summarized in Table 1 highlight the distinct impacts of these design choices. Specifically, while Baseline evaluations suffer from severe saturation (yielding up to 0.900 accuracy on DeepSeek), alternative methods offer only marginal difficulty scaling. Naïve Prompting even exacerbates this issue: without execution feedback to verify viability, generation collapses into trivial or flawed tasks where top models score a perfect 1.000. In contrast, METABENCH-HARNESS systematically drives down performance across all models, dropping DeepSeek to 0.322 and Qwen-7B to 0.044. This demonstrates that both inner-loop execution feedback and outer-loop meta-harness orchestration are essential to eliminate performance saturation and expose fine-grained model capabilities.

Model Performance Across Different Evolution Rounds. We evaluate proxy (in-loop) and held-out models across METABENCH-HARNESS's base, early, and late evolution rounds. As shown in Table 2, while proxy models typically peak early and decline as late-stage complexity increases, this

Table 3: Comparison of model reasoning lengths under the base and late evolution datasets.

Model	Type	MB-CodeContests		MB-AIME	
		Base → Late	Δ Tokens	Base → Late	Δ Tokens
`DeepSeek-V3.1`	R	11,223 → 13,382	(+2,159)	10,208 → 15,744	(+5,536)
`Kimi-K2.5`	R	15,879 → 16,732	(+853)	14,218 → 28,982	(+14,764)
`Gemini-2.5-Flash`	R	15,537 → 16,562	(+1,025)	11,254 → 17,020	(+5,766)

Table 4: Harness-Level Evolutionary Changes.

	MB-CodeContests	MB-AIME
Meta-Harness Orchestrated Proposals		
Optimization rounds	10	5
Proposals applied	20/23 (87%)	10/12 (83%)
Edits Applied by the Harness		
Measurement	11 (39%)	2 (20%)
Generation	9 (32%)	6 (60%)
Execution	5 (18%)	2 (20%)
Telemetry	3 (11%)	0 (0%)

Table 5: Benchmark Evolution Statistics.

	#. Ori.	#. Evo.
MB-CodeContests		
Total Questions	165	100
Solving Techniques	5.02	5.49
Capability	3.55	3.90
MB-AIME		
Total Questions	30	30
Solving Techniques	3.67	4.73
Capability	3.30	3.40

evolution ultimately enhances benchmark quality: it better distinguishes model capabilities on MB-CodeContests, and escalates difficulty on MB-AIME while preserving strong discriminative power. For held-out models, late-stage difficulty scaling highlights a stark contrast between non-reasoning (NR) and reasoning (R) models. On the highly challenging MB-AIME, NR models suffer a near-total collapse, dropping to a narrow 0.100–0.167 accuracy band. Conversely, R models demonstrate strong resilience; top reasoners like `Kimi-K2.5` and `DeepSeek-V3.1` maintain scores of 0.533 and 0.333, respectively. Finally, Table 3 substantiates this increased complexity through reasoning lengths. By the late stage on MB-AIME, `Kimi-K2.5` and `DeepSeek-V3.1` require significantly more computational effort, generating an additional $14,764$ and $5,536$ tokens, respectively. Ultimately, the preserved discriminative power and increased reasoning effort prove that METABENCH-HARNESS generates genuinely reasonable benchmarks that test advanced models, rather than merely encouraging unintended shortcuts or flawed heuristics.

4.2 BENCHMARK HARNESS EVOLUTION ANALYSIS AND DATA QUALITY ASSESSMENT

The previous section demonstrated that METABENCH-HARNESS effectively escalates benchmark difficulty while preserving strong model discriminability. Building on these findings, this section examines the underlying dynamics of the evolution procedure, assesses the quality of the generated benchmark, and analyzes the specific model capabilities required by the evolved tasks.

How does the meta-harness adapt to different domains? As shown in Table 4, our framework maintains a high acceptance rate for harness proposals across both coding and math scenarios. However, the meta-harness reveals distinct adaptation trends tailored to each domain's inherent challenges. For coding tasks in CodeContests, the system heavily modifies *Measurement* (39%). Unlike math, programming tasks cannot be judged by a single final answer alone; they require different unit tests, edge-case checks, and performance tracking. Consequently, the meta-harness frequently updates measurement tools to ensure rigorous evaluation. Furthermore, code execution generates rich intermediate data, necessitating robust *Telemetry* edits (11%) to capture execution logs, runtime errors, and trace signals for subsequent evolution adjustments. Conversely, math task evolution in MB-AIME concentrates primarily on *Generation* (60%). Because AIME problems have clear, deterministic numerical answers, the evaluation mechanism is straightforward, placing minimal burden on measurement. Instead, the core challenge lies in evolving the task itself. The meta-harness directs most of its modifications toward task generation and structural mutation, continuously altering constraints and problem settings to prevent models from relying on memorized templates and to push the limits of advanced mathematical reasoning. Lastly, the details of Table 4 can be found in Appendix B.2, and a round-by-round qualitative analysis is provided in Appendix B.2.2.

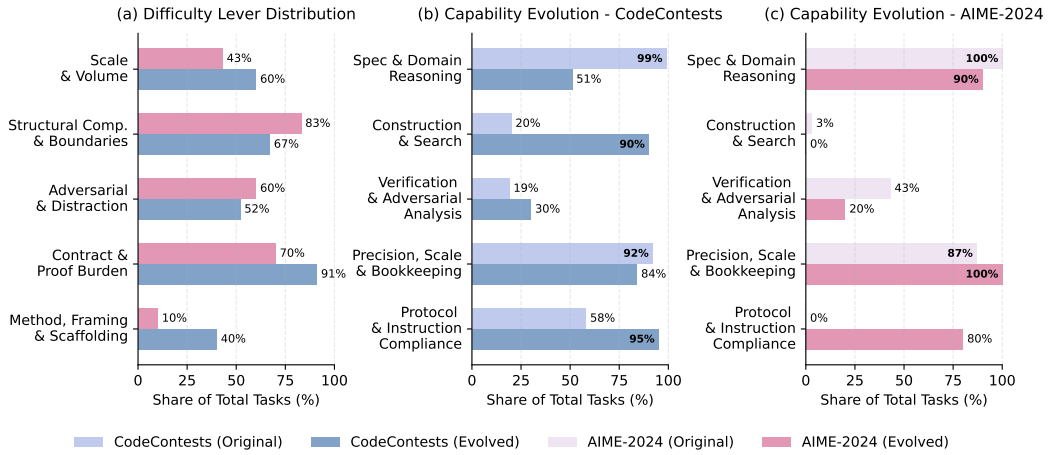


Figure 3: Quantitative analysis of evolved benchmarks across MB-CodeContests and MB-AIME.

Which dimensions of the benchmark's capabilities are evolved? Having detailed the mechanical edits applied by the meta-harness, this section now examines how these changes fundamentally transform the tasks within the evolved benchmarks. Table 5 compares the original datasets with their counterparts evolved by METABENCH-HARNESS. Across both domains, the evolved tasks demand more capability stages, indicating a substantial increase in overall task complexity. Evolution elevates task complexity by engaging more capability stages across the solving process. To analyze how task complexity scales, Figure 3 utilizes `Claude-Fable-5` (Anthropic, 2026c) to compare evolved tasks with their original seed questions across two dimensions (difficulty levers and capability dimensions). By sampling a subset of examples for human validation, we confirm high Human–AI agreement of 97.1% and 96.1% for the two dimensions, respectively. Figure 3 (a) shows that both domains evolve primarily by tightening answer criteria and compounding constraints. While mathematical tasks lean toward structural composition, coding tasks increase the scale of required work. Figures 3 (b) and (c) further illustrate the shift in required capabilities following evolution across the two datasets. Mathematical tasks emphasize precise computation, multi-case bookkeeping, and structured output compliance, while coding tasks shift toward constructive solution generation under strict evaluation constraints. See Appendix B.2.3 for a detailed breakdown.

Does the benchmark quality improve through evolution? To ensure the evolved benchmarks are meaningful, we pairwise compare evolved tasks against their seeds using a more powerful LLM judge, `Claude-Opus-5` (Anthropic, 2026b), as summarized in Table 6. For each dimension, the judge outputs a binary validation, with the reported success rates reflecting the proportion of tasks that meet the criteria. To validate the reliability of the LLM judge, we also measure Human–AI agreement on a sampled subset (reported as the Human column). First, *Reasonableness* evaluates whether the evolved task remains valid and well-formed, preserving a recognizable lineage while introducing meaningful changes. Second, *Competency* verifies that the task faithfully measures the intended capability, ensuring that it demands genuine reasoning rather than adding superficial complexity. Lastly, *Robustness* examines whether the evolved evaluator can reliably distinguish valid from invalid solutions, maintaining strict resistance to formatting artifacts, implementation errors, and adversarial exploitation. Crucially, the *Overall* metric demands simultaneous success across all three dimensions, with this joint pass rate in the final round substantially outperforming the cross-round average. This consistent upward trend demonstrates that our iterative pipeline progressively converges toward highly valid and structurally robust tasks. (See Appendix B.2.4 for details.)

Table 6: Evolution Quality Assessment Results: Final and overall average success rates. "Human" denotes the Human–AI Agreement, measuring consistency between human and agent.

Metric	MB-CC*	MB-AIME	Human
Reasonableness	95% / 91%	90% / 83%	85%
Competency	78% / 61%	80% / 78%	68%
Robustness	90% / 62%	93% / 69%	85%
Overall	70% / 44%	70% / 49%	80%

*MB-CC denotes MB-CodeContests.

Table 7: METABENCH-HARNESS Evolution Case Study: From Initial Seed to Deepening

Part I: Case Study of MB-CodeContests Evolution

Seed (Codeforces 1582D)	**Round 5** (Evolved Emergence)	**Round 9** (Evolved Deepening)
Task Formulation		
Constructive Math: Given an array a ($a_i \neq 0$), construct b such that $\sum a_i b_i = 0$ and $\sum \lvert b_i \rvert \leq 10^9$.	**Single Gap Audit**: Exploit a planted gap (`range(n-1)`) to trick a Python checker with an invalid (a, b) pair.	**Compound Audit**: Trace complex logic to exploit a vulnerability gated by 3 conditions ($n > 1000$, duplicate a, odd n).
Evolution Summary		
Capability: 3 **Technique:** 3	**Capability:** 3 **Technique:** 5	**Capability:** 4 **Technique:** 6
Lever: Baseline specification	**Key Lever:** Meta level shift, Output contract tightening	**Key Lever:** Compound conjunction, Scaffolding removal
Baseline: The root foundation originating from a standard coding task.	**Moderate:** Shifts to basic code auditing and static inspection.	**High:** Mandates multi-technique fusion without hints.

Part II: Case Study of MB-AIME Evolution

Seed (AIME 2024-II-9)	**Round 1** (Evolved Emergence)	**Round 5** (Evolved Deepening)
Task Formulation		
Combinatorial Counting: Count placements on a 5×5 grid where no further chip can be added (*answer*: 902).	**Pre-Blocked Obstacle**: Count maximal placements on a 4×4 board with 2 permanently blocked cells (*answer*: 150).	**Post-Hoc Filter**: Count maximal placements on a 3×5 board, then filter out those hitting 4 flagged cells (*answer*: 14).
Evolution Summary		
Capability: 2 **Technique:** 4	**Capability:** 3 **Technique:** 3	**Capability:** 3 **Technique:** 4
Lever: Baseline specification	**Key Lever:** Compound conjunction	**Key Lever:** Distractor injection
Baseline: The root foundation originating from a standard olympiad task.	**Moderate:** Forms compound constraints by adding permanent obstacles to maximal placement.	**High:** Requires a multi-step deduction to filter out injected distractors without hints.

From Seed to Spark: Case Studies. Table 7 shows two task evolution trajectories that highlight our core evolution pattern: the benchmark harness evolved by METABENCH-HARNESS systematically incorporates difficulty levers to drive multi-dimensional evolution across required capabilities and solving techniques. In Part I (MB-CodeContests), the system transforms the problem type, shifting from a standard constructive math seed to a complex code-auditing task. By applying levers like *meta level shift* and *scaffolding removal*, it forces multi-technique fusion, expanding capabilities to include static code analysis. Similarly, Part II (MB-AIME) evolves a counting problem via *compound conjunction* and *distractor injection*, shifting the required skills from pure domain reasoning to include instruction compliance and distractor resistance. Rather than merely scaling difficulty along a single axis, the evolved harness uses diverse levers to formulate questions that demand additional capabilities and solving techniques. Detailed case studies are in Appendix B.2.5.

5 CONCLUSION

In this paper, we propose METABENCH-HARNESS, a dual-loop framework designed for the end-to-end optimization of benchmark generation workflows. By combining an inner loop for task generation with an outer meta-harness layer that refines harness implementations based on historical trajectories, we transform benchmark creation from a static process into a dynamic evolution. Using CodeContests and AIME-2024 as seed tasks, our experiments show that the framework generates benchmarks that are both challenging and discriminative for models. Supported by trajectory and quality analyses, we verify that it enables multi-dimensional evolution, and steadily improves evolution reasonableness, benchmark competency, and evaluator robustness across successive evolution rounds. Furthermore, case studies quantitatively show that it not only increases the required capabilities and problem-solving techniques, but also uses different difficulty levers to reframe the problems. More broadly, this work provides a solution to static benchmark saturation.

REPRODUCIBILITY STATEMENT

To ensure high reproducibility, our pipeline automatically generates a matching code-based evaluator for every new task. This guarantees that future evaluations on our dataset remain consistent and fair, avoiding the instability of subjective LLM-as-a-judge scoring. Furthermore, all seed datasets, prompt templates, and the complete codebase will be publicly released upon paper acceptance.

THE USE OF LARGE LANGUAGE MODELS

In this work, Large Language ¡odels (LLMs) play multiple roles. First, as the core module of our research, they act as agents within the METABENCH-HARNESS framework for autonomous task generation, evaluator synthesis, and meta-harness orchestration, with their generation quality rigorously assessed through human validation. Second, they serve as the target models evaluated in our experiments. Third, during manuscript preparation, we utilized LLMs strictly as writing assistants. This usage was strictly limited to language refinement, including grammar correction, clarity improvement, and avoiding ambiguous expressions, without generating any scientific claims or core ideas. We explicitly emphasize that all scientific claims, experimental designs, logical arguments, and core contributions were entirely conceived and proposed by human authors in this paper.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*, 2023. URL https://arxiv.org/abs/2303.08774.

Anthropic. Introducing Claude Sonnet 5. Anthropic News, June 2026a. URL https://www.anthropic.com/news/claude-sonnet-5. Accessed: August 31, 2026.

Anthropic. Introducing Claude Opus 5. Anthropic News, July 2026b. URL https://www.anthropic.com/news/claude-opus-5. Accessed: August 31, 2026.

Anthropic. Claude Fable 5 and Claude Mythos 5. Anthropic News, June 2026c. URL https://www.anthropic.com/news/claude-fable-5-mythos-5. Accessed: August 31, 2026.

Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Ziwei Chai, Y Charles, HS Che, Cheng Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026. URL https://arxiv.org/abs/2602.02276.

Yupeng Chang, Xu Wang, Jindong Wang, et al. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol.*, 15(3), March 2024. ISSN 2157-6904. doi: 10.1145/3641289. URL https://doi.org/10.1145/3641289.

Zixin Chen, Hongzhan Lin, Kaixin Li, Ziyang Luo, Zhen Ye, Guang Chen, Zhiyong Huang, and Jing Ma. AdamMeme: Adaptively probe the reasoning capacity of multimodal large language models on harmfulness. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4234–4253. Association for Computational Linguistics, July 2025. doi: 10.18653/v1/2025.acl-long.213. URL https://aclanthology.org/2025.acl-long.213/.

Yuxing Cheng, Yi Chang, and Yuan Wu. A survey on data contamination for large language models. *arXiv preprint arXiv:2502.14425*, 2025. URL https://arxiv.org/abs/2502.14425.

Wei-Chieh Chou, Xuanjun Chen, Jian-Ren Lin, Claire Lin, Hung yi Lee, and Jyh-Shing Roger Jang. Only Ask What You Don't Know: Grounded delta planning for efficient multi-step RAG. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=lD07FqX0YY.

DeepSeek-AI. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437*, 2024. URL https://arxiv.org/abs/2412.19437.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. URL https://arxiv.org/abs/2501.12948.

Yifeng Di, Xuliang Huang, and Tianyi Zhang. TRACER: A semantic-aware framework for fine-grained contamination detection in code LLMs. *arXiv preprint arXiv:2605.24079*, 2026. URL https://arxiv.org/abs/2605.24079.

Google. Gemini 2.5 Flash. https://ai.google.dev/gemini-api/docs/models/gemini-2.5-flash, 2026. Large Language Model.

Dadi Guo, Tianyi Zhou, Dongrui Liu, Chen Qian, Qihan Ren, Shuai Shao, Zhiyuan Fan, Yi R. Fung, Kun Wang, Linfeng Zhang, and Jing Shao. Towards Self-Evolving Agent Benchmarks: Validatable agent trajectory via test-time exploration. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=2H03gm4Rq6.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

Maxwell Jia. AIME 2024 dataset. Hugging Face, 2024. URL https://huggingface.co/datasets/Maxwell-Jia/AIME_2024. 30 problems from AIME 2024 I and II. Accessed: August 18, 2026.

Yoonho Lee, Roshen Sanjay Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=tmbOUyFx3R.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution through large models. In *Handbook of Evolutionary Machine Learning*, pp. 331–366. Springer Nature Singapore, Singapore, 2024. ISBN 978-981-99-3814-8. doi: 10.1007/978-981-99-3814-8_11. URL https://doi.org/10.1007/978-981-99-3814-8_11.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020. URL https://dl.acm.org/doi/abs/10.5555/3495724.3496517.

Cheng-Yen Li, Xuanjun Chen, Claire Lin, Wei-Yu Chen, Wenhua Nie, Hung-Yi Lee, and Jyh-Shing Roger Jang. CodaRAG: Connecting the dots with associativity inspired by complementary learning. *arXiv preprint arXiv:2604.10426*, 2026. URL https://arxiv.org/abs/2604.10426.

Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. EvoCodeBench: An evolving code generation benchmark with domain-specific evaluations. In *Advances in Neural Information Processing Systems*, volume 37, pp. 57619–57641. Curran Associates, Inc., 2024. doi: 10.52202/079017-1837. URL https://dl.acm.org/doi/10.5555/3737916.3739753.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. URL https://www.science.org/doi/10.1126/science.abq1158.

Renze Lou, Hanzi Xu, Sijia Wang, Jiangshu Du, Ryo Kamoi, Xiaoxin Lu, Jian Xie, Yuxuan Sun, Yusen Zhang, Jihyun Janice Ahn, Hongchao Fang, Zhuoyang Zou, Wenchao Ma, Xi Li, Kai Zhang, Congying Xia, Lifu Huang, and Wenpeng Yin. AAAR-1.0: Assessing AI's potential to assist research. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=RHAWcjIyl2.

OpenAI. Introducing GPT-5.4. OpenAI Blog, March 2026. URL https://openai.com/index/introducing-gpt-5-4/. Accessed: August 31, 2026.

OpenRouter. OpenRouter: The unified interface for every model. OpenRouter, 2026. URL https://openrouter.ai/. Accessed: August 31, 2026.

Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. URL https://arxiv.org/abs/2310.08560.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. URL https://www.nature.com/articles/s41586-023-06924-6.

Zhichao Shi, Xuhui Jiang, Chengjin Xu, Cangli Yao, Shengjie Ma, Yinghan Shen, Zixuan Li, Jian Guo, and Yuanzhuo Wang. JudgeAgent: Beyond static benchmarks for knowledge-driven and dynamic LLM evaluation. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 13004–13030, 2026. URL https://aclanthology.org/2026.findings-acl.634/.

Vaibhavi Tiwari, Ramy Othman, Ola Suaifan, and Dajin Wang. Rethinking LLM Evaluation: A comprehensive survey, taxonomy, and the evaluation trilemma. *2026 IEEE World AI IoT Congress (AIIoT)*, pp. 0320–0329, 2026. URL https://api.semanticscholar.org/CorpusID:289490388.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10014–10037, 2023. URL https://aclanthology.org/2023.acl-long.557/.

Siyuan Wang, Zhuohan Long, Zhihao Fan, Xuanjing Huang, and Zhongyu Wei. Benchmark Self-Evolving: A multi-agent framework for dynamic LLM evaluation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 3310–3328. Association for Computational Linguistics, January 2025. URL https://aclanthology.org/2025.coling-main.223/.

JiaRu Wu and Mingwei Liu. AutoEvoEval: An automated framework for evolving close-ended LLM evaluation data, 2025. URL https://arxiv.org/abs/2506.23735.

Yiming Xiong, Shengran Hu, and Jeff Clune. Learning to continually learn via meta-learning agentic memory designs. *arXiv preprint arXiv:2602.07755*, 2026. URL https://arxiv.org/abs/2602.07755.

Alex L Zhang, Tim Kraska, and Omar Khattab. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025a. URL https://arxiv.org/abs/2512.24601.

Guibin Zhang, Haotian Ren, Chong Zhan, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng YAN. MemEvolve: Meta-evolution of agent memory systems. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=qpkG0eKx4v.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=z5uVAKwmjf.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P Xing, Hao Zhang, Joseph E Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, 2023. URL https://dl.acm.org/doi/10.5555/3666122.3668142.

APPENDIX

A THE IMPLEMENTATION DETAILS OF METABENCH-HARNESS

This section details the concrete implementation and module design of METABENCH-HARNESS. Expanding upon the module instantiation introduced in Section 3, we explain how the primary LLM agent ($\mathcal{M}_{\text{agent}}$) operationalizes the structured Markdown scaffolds. We begin with a high-level overview of the system's core mechanics, covering Master Initialization, Module Instantiation, and the overarching Dual-Loop Evolution workflow.

Master Initialization. Serving as the foundational entry point, this initial setup establishes the operational constraints and global configuration for a METABENCH-HARNESS run. Executed by $\mathcal{M}_{\text{agent}}$ (operating as a coding agent with shell access) through a single master configuration file, it specifies the seed dataset, item formatting conventions, overarching objectives, stage sequencing, dual-loop workflow, and the mandatory on-disk locations for all generated artifacts and scripts.

Module Instantiation. Guided by the master configuration, $\mathcal{M}_{\text{agent}}$ executes the workflow's distinct modules by sequentially traversing the Markdown scaffolds. Treating each prompt as a unit of work, the agent reads prior artifacts, executes its objective, and writes the required outputs (e.g., JSON, Python) to mandated locations. A stage is complete only when these files exist on disk; this rigorous file-passing discipline, rather than a persistent context, robustly carries state through the round.

Dual-Loop Evolution. This self-evolving workflow organizes the system's modules into two alternating cycles. The outer loop drives the system's meta-optimization, where $\mathcal{M}_{\text{agent}}$ analyzes archived telemetry (e.g., score distributions and failure analyses) to propose, evaluate, and apply edits directly to the benchmark harness's Markdown scaffolds, effectively rewriting its underlying instructions. Under this updated setup, the inner loop acts as the execution engine for the harness itself. It runs the revised sequence of generation and measurement modules to generate the evolved benchmarks, while collecting fresh telemetry for the next update. This benchmark evolutionary cycle repeats until the stop controller terminates the execution.

To unpack the METABENCH-HARNESS workflow, the following subsections detail its three primary phases: Master Initialization (initial setup), Meta-Harness Orchestration (outer loop), and Benchmark Generation (inner loop). To highlight the core logic of each module, the Markdown scaffolds presented below are condensed summaries rather than exhaustive full prompts.

A.1 MASTER INITIALIZATION

This phase establishes the global constraints and foundational anchors for the entire execution. The agent first ingests the master configuration to strictly bind the pipeline to the target dataset and define the baseline evaluation thresholds for all subsequent inner-loop operations.

```
--- Master Initialization ---                          [00_master_start.md]
1. Seed Anchor:  pipeline.yaml and 00_master_start.md solely define
   dataset-level facts; no other prompt may hard-code seed details.
2. Evolution Rule:  every scenario and task must be seed-derived with
   explicit seed_lineage; transformation, never copying.
3. Difficulty Defaults:  too-easy task (mean of at least 0.85 or
   universal pass); too-easy scenario (mean of at least 0.80, failure
   rate of at most 0.20); non-discriminative (spread of at most 0.10);
   healthy band between 0.25 and 0.65.
4. Hard Constraints:  deterministic programmatic verification per
   task; strict artifact/script layout (data/rounds/{id}/, scripts/).
```

A.2 OUTER LOOP: META-HARNESS ORCHESTRATION

Following Section 3, we detail the meta-harness that *evolves* the harness. Its four prompts operate exclusively on archived round telemetry instead of raw model answers, keeping optimization strictly decoupled from individual evaluations to ensure robust and unbiased self-improvement.

A.2.1 STAGES 1: TELEMETRY REVIEW AND HARNESS PROPOSER

--- Proposal Workflow (shared) --- [mh/01_harness_proposer.md]

1. **Telemetry Review**: read the round's summaries, score distributions, failure analyses, rejection logs, and lifecycle decisions.
2. **Candidate Generation**: emit 3--5 scoped candidate revisions to specific stage prompts or configs (difficulty escalation, evaluator reliability, lifecycle policy), each with evidence, expected score direction, risks, and a rollback plan.
3. **Seed Guardrails**: preserve seed lineage, deterministic scoring, and 00_master_start.md as the single source of truth.
4. **Output**: proposal.md/.json per candidate, plus an index.

A.2.2 STAGE 2: HARNESS EVALUATOR

--- Evaluation Workflow (shared) --- [mh/02_harness_evaluator.md]

1. **Ten-Dimension Scoring**: rate each candidate 1--5 on difficulty pressure, discrimination, evaluator reliability, task validity, novelty, seed-evolution fidelity, lifecycle quality, reproducibility, cost control, and implementation risk.
2. **Hard Rejection**: reject candidates that weaken deterministic verification, remove seed-lineage requirements, hard-code seed items, enable from-scratch generation, or add difficulty via ambiguity or brittle formatting.
3. **Weighted Judgment**: compute overall_harness_score via weights (difficulty pressure, evaluator reliability, task validity, seed fidelity, reproducibility weighted highest), not a blind average.
4. **Output**: per-candidate evaluation.json/.md and a summary.

A.2.3 STAGE 3: HARNESS SELECTOR

--- Selection Workflow (shared) --- [mh/03_harness_selector.md]

1. **Ranking**: aggregate evaluation results with the persistent cross-round leaderboard.json.
2. **Selection Rule**: select at most one candidate per round, unless two are complementary and low-risk; selecting none is valid.
3. **Invariant Check**: re-verify that the selected candidate preserves the seed-evolution contract and the configuration source of truth.
4. **Output**: updated leaderboard, selected_candidate.json, and a report.

A.2.4 STAGE 4: HARNESS APPLIER

--- Application Workflow (shared) --- [mh/04_harness_applier.md]

1. **Scoped Edits**: apply the selected revision directly to live files (e.g., stage prompts $\mathcal{P}$, pipeline.yaml / difficulty configuration $\mathcal{L}$, or lifecycle policy Π_{life}), strictly limited to those in selected_candidate.json.
2. **Integrity Check**: confirm the seed-evolution contract and master configuration remain intact after editing.
3. **No-Op Reporting**: if nothing was selected, write an apply report stating that the next round runs under the unchanged harness.
4. **Output**: apply_report.md/.json enumerating every modified file, so that round $t{+}1$ verifiably runs on $\mathcal{H}^{(t+1)}$.

A.3 INNER LOOP: BENCHMARK GENERATION

Having specified how the harness *evolves*, we now detail the inner-loop pipeline that each harness state instantiates to generate a round's benchmark. This execution strictly follows a four-stage sequence: Scenario Discovery, Task & Evaluator Synthesis, Model Validation, and a Lifecycle Audit.

A.3.1 STAGE 1: SCENARIO DISCOVERY

--- Discovery Workflow (shared) --- [01_scenario_discovery.md]

1. **Seed Selection:** load the configured dataset or local manifest; select one seed item supporting deterministic evolution; save selected_seed_item.json and a seed-selection report.
2. **Over-Generation:** produce $(B - K) \times \mu$ candidate scenarios (100 in round 1, where K=0), each a capability-centered seed evolution.
3. **Lineage Contract:** every candidate carries seed_lineage and an explicit account of why it is a transformation, not a copy.
4. **Output:** candidate JSONs under scenarios_raw/.

--- Filtering and Selection Workflow (shared) --- [02-03_*.md]

1. **Lineage Validation:** confirm seed lineage; flag seed_copy_risk; score novelty, feasibility, evaluator reliability, and degree of seed evolution (1--5 each); write novelty_reports/{scenario_id}.json with a keep/revise/drop recommendation.
2. **Slot Refill:** carry over kept scenarios; select only enough survivors to restore the active budget B=10.
3. **Diversity Enforcement:** the selected set must cover distinct transformation types, not near-duplicate variants of one.
4. **Output:** scenarios_selected/ plus a selection report documenting diversity and seed-evolution rationale.

Domain instantiation. The CodeContests novelty checker additionally enumerates valid programming-domain transformation axes (reasoning burden, edge-case distribution, testing strategy, output contract, solution-debugging objective) as acceptable evolutions of a competitive-programming seed.

A.3.2 STAGE 2: TASK AND EVALUATOR SYNTHESIS

--- Task and Evaluator Synthesis Workflow (shared) --- [04-06_*.md]

1. **Difficulty-Aware Generation:** for families whose previous mean $\geq$ 0.85 or all models passed, same-difficulty siblings forbidden; escalation only via validity-preserving levers (constraint composition, counterfactuals, boundary cases), never ambiguity.
2. **Evaluator Construction:** emit eval.py implementing evaluate(model_output, item); execute three validation probes (ground-truth must pass, wrong-answer must fail, malformed-output must raise a format error) and assign a reliability grade; grades below 4 are unusable.
3. **QA Gates:** accept only if the evaluator (validated eval.py, reliability $\geq$ 4), seed-evolution (valid lineage, low copy risk), and empirical-difficulty (reject near-equivalents of previous tasks with mean $\geq$ 0.85 or universal pass) gates all hold.
4. **Output:** survivors in tasks_filtered/, rejections logged with reasons, evaluators in scripts/evaluators/.

Domain instantiation (AIME-2024). Each math task requires an `answer_verification` object: an agent-written deterministic checker must reproduce the expected answer with run evidence. Ground truth that cannot be re-derived is rejected at QA as evaluator-fragile. CodeContests needs no such object, as its evaluator directly checks correctness.

A.3.3 STAGE 3: MODEL VALIDATION

--- Execution Workflow (shared) --- [07_model_runner.md]

1. **Reproducible Script**: generate a per-round runner under scripts/model_runners/; load models and credentials from models.yaml and .env via safe loaders; never print or persist secrets.
2. **Token-Budget Discipline**: request the largest safe per-model output budget; retry once at a lower accepted value on rejection.
3. **Truncation Handling**: detect token-limit finish reasons, continue once when safely supported, and record truncation explicitly rather than silently scoring incomplete answers.
4. **Output**: structured JSONL outputs with token-limit metadata, plus an execution report.

--- Scoring Workflow (shared) --- [08_scorer.md]

1. **Programmatic Only**: a generated per-round scoring script dynamically imports each scenario's validated eval.py and scores every output; no external APIs, no informal judgment.
2. **Structured Failure**: missing outputs, import failures, and truncated generations are scored 0 with explicit error types, never dropped.
3. **Difficulty Signals**: compute per-task means and cross-model spreads; flag items as *non-discriminative* (spread ≤ 0.10) or *too-easy* (mean ≥ 0.85 or all pass) for audit.
4. **Output**: per-model score records in scores/, aggregated summaries, and a scoring report.

A.3.4 STAGE 4: LIFECYCLE AUDIT

--- Lifecycle Audit Workflow (shared) --- [09-12_*.md]

1. **Failure Analysis**: classify scenarios against the healthy band $[0.25, 0.65]$; separate confound-driven scores from genuine capability signal; isolate high-variance families as capability boundaries.
2. **Lifecycle Decision**: assign each scenario one of KEEP-AND-EVOLVE, KEEP-STABLE, RESEED; kept-but-easy scenarios carry explicit escalation instructions (*must-be-harder-than* task IDs) into next_round_scenario_plan.json.
3. **Stop Decision**: continue while valid harder tasks remain producible; termination is triggered only by novelty collapse, evaluator instability, or repeated failure to produce valid harder tasks, never by high scores alone.
4. **Round Summary**: consolidate metrics, audit every executable component (evaluators, runner, scorer) for strict reproducibility, and distill improvement signals for the outer loop.

A.4 Implementation Details of MetaBench-Harness

We instantiate the core autonomous agent model ($\mathcal{M}_{\text{agent}}$) of MetaBench-Harness using `Claude-Sonnet-5` to drive the entire workflow end-to-end. The agent's behavior is governed by specific skills and workflows tailored to each of the aforementioned evolutionary stages. Execution begins by downloading the necessary data and parsing the dataset, project scope, and workflow. Within the evolution loop, the proxy models ($\mathcal{M}_1, \mathcal{M}_2$) are accessed via API, with all requests routed through OpenRouter (OpenRouter, 2026). To ensure resource efficiency, the agent's generation is constrained by a 16,000-token output budget and a 600-second timeout per call. A small fraction of API calls yielding errors (e.g., empty responses, timeouts, or budget overflows) are robustly handled, with token counts and scores recorded exactly as returned.

B Additional Experimental Details and Results

This appendix provides foundational background and supplementary configurations to offer a more comprehensive understanding of the experimental setups presented in Section 4. It is structured into two primary parts: Appendix B.1 details the evaluation protocols and configurations for the LLMs, while Appendix B.2 outlines the experimental setup used to analyze the framework's evolutionary dynamics and rigorously assess the evolved benchmark's quality.

B.1 Experimental Setup for Main Result of Held-out Model Evaluation

Benchmark Baselines and Different Evolution Rounds. To establish a comparative baseline that quantifies the difficulty introduced by MetaBench-Harness, we first evaluate models on the original, unevolved seed datasets (e.g., the original CodeContests and AIME-2024 benchmark) which serve as our baselines. To demonstrate how the subsequent evolution process enhances benchmark quality by increasing difficulty and better differentiating model capabilities (results presented in Table 2), we analyze specific Early and Late milestone rounds. For MB-CodeContests, we select rounds 2 and 10 to represent the Early and Late stages, respectively. For MB-AIME, rounds 1 and 5 are chosen as the corresponding Early and Late stages.

Evolved Benchmarks via Different Evolution Methods. To rigorously validate the effectiveness of our approach, MetaBench-Harness, and isolate the performance gains attributed to the evolution method (results presented in Table 1), we compare the MetaBench-Harness-evolved benchmark against datasets generated by several alternative methodologies, all of which are evolved from AIME-2024. For external baselines, we generate a comparison dataset using the `AutoEvoEval` (Wu & Liu, 2025) framework by reimplementing its official specifications. Specifically, we retain its seven question-level operations and rule-based character injection, and randomly sample and perturb scenarios across rounds. Note that we exclude some evolution units that are not related to math or coding. Regarding the `TRACE` (Guo et al., 2026) methodology, due to its private codebase, we directly report performance metrics from the original paper rather than generating a separate dataset. Furthermore, we construct two internal ablation datasets to evaluate specific evolution components of MetaBench-Harness. First, we produce a dataset using a `Naïve Prompting` strategy by entirely removing the outer loop and disabling candidate viability mechanisms (i.e., the model-validation and lifecycle-audit stages), meaning candidates are accepted without intrinsic evaluation. Second, we generate a dataset via a `Fixed Harness` configuration. While this setting also eliminates the outer loop (preventing prompt and configuration edits between rounds), it maintains identical seed items and evaluation logic, thereby isolating the specific impact of harness-level evolution.

Proxy and Held-Out Model Setup. To ensure comprehensive coverage of current LLM capabilities and strictly separate the models used for benchmark generation from those used for evaluation (with evaluation results presented in Table 2), we categorize our models into proxy and held-out sets. For the **proxy models**, which drive the in-loop evolution to ensure the generated candidates successfully meet our quality and viability expectations, we utilize `Claude-Sonnet-5` as an Adaptive Reasoning model (exhibiting dynamic reasoning and tool-use self-correction) and `GPT-5.4` as a Standard Instruct (Non-Reasoning) model. Conversely, for the **held-out models**, which are evaluated on the resulting benchmarks to verify that the evolved problems effectively enhance discriminative power across unseen architectures, we employ two distinct behavioral classes: Standard

Instruct models operating under standard inference budgets (e.g., `DeepSeek-V3.1`, `Kimi-K2.5`, `Gemini-2.5-Flash`), and Test-Time Scaling (Reasoning) models (e.g., `DeepSeek-V3.1`, `Kimi-K2.5`, `Gemini-2.5-Flash`, `DeepSeek-R1-Distill-Qwen-7B/32B`,), which are specialized for long-horizon thinking, self-reflection, and chain-of-thought scaling. As introduced in Section 4.1, all held-out model evaluations employ standard generation constraints. Notably, the two Qwen reasoning models are evaluated across two strictly budgeted phases: a reasoning phase capped at 96,000 tokens, followed by an answer-extraction phase capped at 32,000 tokens, with the final answer derived exclusively from the second phase.

B.2 Experimental Setup for Evolutionary Dynamics and Quality Analysis

This section analyzes the internal dynamics of our evolutionary framework and the intrinsic quality of the resulting benchmarks. We begin by examining the meta-harness's outer-loop adaptations in Appendix B.2. Next, Appendices B.2.1 and B.2.3 explore task generation qualitatively and quantitatively. To ensure data integrity, Appendix B.2.4 rigorously assesses benchmark quality, focusing on reasonableness, competency, and robustness. Finally, Appendix B.2.5 grounds these analyses with detailed case studies, illustrating the trajectory from seed problems to fully evolved tasks.

B.2.1 Harness-Level Evolutionary Dynamics Analysis

While Section 4.2 analyzes how benchmark harness edits drive domain-specific adaptations, this appendix provides the extended details that underpin the benchmark harness-edit audit (Table 4). Specifically, our audit draws on three archived artifacts per candidate, namely the machine-readable proposals, per-round leaderboards, and apply reports, to categorize harness modifications into four functional groups. These groups comprise *measurement* updates for evaluators and scorers, *generation* modifications for task evolution and seed selectors, *execution* adjustments governing runners and configurations, and *telemetry* refinements for analyzers and early-stopping conditions. Beyond the domain-adaptive trends discussed in Section 4.2, this fine-grained granular breakdown reveals that a steady minority of edits is actively dedicated to maintaining the instrument itself. Crucially, these maintenance updates introduce execution-reliability fixes like robust timeout handling and retry safety mechanisms, alongside telemetry rules designed to filter out anomalous or confounded scores from lifecycle decisions. Ultimately, these adjustments highlight an organic synergy within the dual-loop workflow. By continuously reinforcing the evaluation environment and filtering unreliable feedback, the outer harness-editing loop ensures that the inner loop can explore increasingly complex problem spaces, thereby driving a stable and continuous evolution.

B.2.2 Qualitative Analysis on Task Evolution

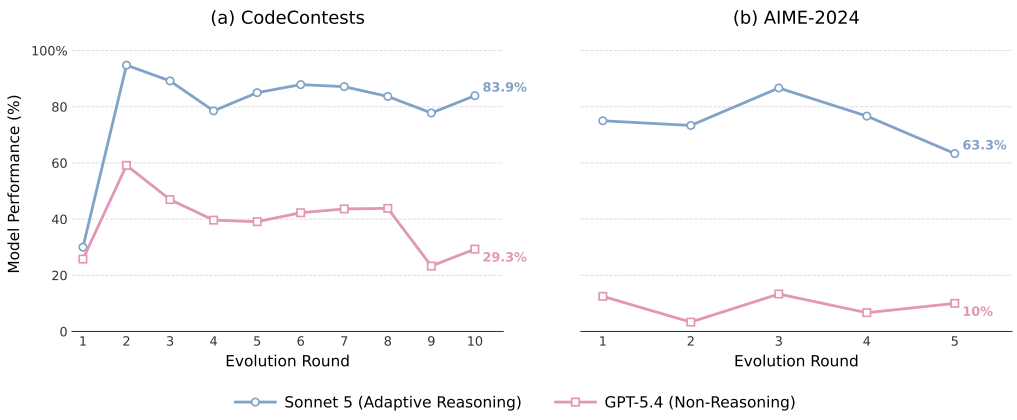


Figure 4: Round-by-round model validation scores along the evolution of MetaBench-Harness.

While the statistical breakdowns in Section 4.2 identify which harness components evolved, the main purpose of this section is to examine the model validation feedback to understand how the system simultaneously escalates task difficulty and preserves model discriminability. Specifically, we aim to demonstrate that the meta-harness does not merely apply random mutations, but dynamically

reacts to the proxy models' behaviors by adjusting constraints, fixing formats, and pushing capability ceilings round by round. To capture this interactive process, Figure 4 tracks the longitudinal in-loop validation scores for our proxy models (`Claude-Sonnet-5` and `GPT-5.4`), illustrating two distinct evolutionary trajectories for CodeContests and AIME-2024.

Coding Problems. The 10-round CodeContests evolution (Figure 4a) reveals a dynamic shift from evaluator correction to difficulty escalation, unfolding in four distinct phases:

- *Round 1 (Identifying Measurement Defects):* The evolution initially records artificially low performance, with `Claude-Sonnet-5` and `GPT-5.4` scoring 0.300 and 0.258. The failure analyzer reveals that out of 180 model-task pairs, 121 fail due to output formatting (e.g., answers wrapped in explanatory text) rather than flawed reasoning, masking the models' true capabilities.
- *Round 2 (Evaluator Correction):* Recognizing that the defect lies in the evaluator rather than the task, the meta-harness strictly corrects the measurement logic by mandating explicit answer markers. Without altering actual task difficulty, scores surge to 0.948 and 0.591, proving that the seed tasks are actually too easy once superficial formatting confounds are removed.
- *Rounds 3–4 (Complexity Escalation):* Having secured a robust evaluation mechanism, the framework shifts to escalating task complexity, successfully driving performance down to 0.785 and 0.396. This targeted escalation establishes a clean capability split across three specific scenarios, which `Claude-Sonnet-5` solves perfectly while `GPT-5.4` fails entirely.
- *Rounds 5–10 (Refinement and Final Separation):* Although Rounds 5–7 experience a temporary stall, the system learns from ineffective mutations (e.g., asking a model to certify logic it has already mastered, which inadvertently simplifies the task) and retires scenarios that fail to increase difficulty after two consecutive attempts. Leveraging these insights, Rounds 8–9 intensify the remaining tasks. Ultimately, by Round 10, the performance gap between the two models widens significantly to 0.546 (0.839 vs. 0.293), compared to a mere 0.042 margin in Round 1, achieving the dual objective of escalating difficulty while preserving discriminability.

Mathematical Problems. Unlike CodeContests, the 5-round AIME trajectory (Figure 4b) bypasses initial measurement defects to directly target mathematical reasoning, unfolding as follows:

- *Round 1 (Immediate Capability Separation):* The initial round separates the models (0.750 vs. 0.125). Unlike CodeContests, `GPT-5.4`'s failures stem from incorrect numeric derivations rather than format rejections, allowing the framework to proceed to difficulty escalation.
- *Rounds 2–3 (Isolating Reasoning Gaps):* Round 2 (0.733 vs. 0.033) highlights a specific problem cluster where both models score 0.0 due to a shared misconception regarding an evolved combinatorial family (rigorously verified against ground truth). In Round 3 (0.867 vs. 0.133), a performance divergence becomes the informative event: `Claude-Sonnet-5` overcomes the misconception, whereas `GPT-5.4` repeats the identical error. This effectively localizes the performance gap to a specific reasoning step rather than general difficulty.
- *Round 4 (Handling Confounds):* Recording the peak performance divergence of the study (0.767 vs. 0.067), this round encounters infrastructure-level confounds (e.g., programmatic execution timeouts). Instead of absorbing these anomalies into the final scores, the evaluation loop correctly identifies the issue and pauses difficulty escalation on the affected scenarios.
- *Round 5 (Correcting Instruction Drift & Final Bounds):* Delivering the first fully clean run since Round 3 (0.633 vs. 0.100), the system diagnoses an apparent capability ceiling as "instruction drift" (where a previously proven step-by-step reasoning constraint had been silently dropped). By restoring this constraint, the framework avoids mistaking prompt saturation for inherent task difficulty. Across the entire trajectory, `GPT-5.4` never exceeds a score of 0.133, while `Claude-Sonnet-5` consistently remains above 0.633.

Together, these round-by-round case studies concretely illustrate the dual-loop synergy introduced in Appendix B.2.1. By adaptively distinguishing between basic measurement errors (e.g., evaluator flaws, instruction drift) and actual mathematical or algorithmic reasoning limits, the outer harness-editing loop continuously stabilizes the evaluation environment. This targeted filtering of unreliable feedback ensures that the inner loop can safely escalate task complexity without being derailed by superficial confounds, ultimately generating capability gaps that accurately reflect the models' true performance boundaries.

B.2.3 Quantitative Analysis of Evolved Benchmark Characteristics

Building on Section 4.2 regarding which dimensions of the benchmark's capabilities are evolved, this section provides a fine-grained quantitative analysis of the resulting tasks. Specifically, we evaluate the benchmarks along three complementary dimensions: *Difficulty Levers*, *Capabilities*, and *Solving Techniques*. These taxonomies characterize the evolution at different angles, capturing the specific evolutionary mechanisms chosen to escalate difficulty (Figure 5), and tracking the distributional shifts from original to evolved tasks in both evaluated capabilities (Figure 6) and underlying solving techniques (Figure 7). Difficulty levers denote the operational methods used to mutate seed problems, whereas the other two dimensions compare the original and evolved benchmarks.

Distribution of Difficulty Levers. Figure 5 illustrates how the meta-harness applies domain-specific mechanisms to escalate task difficulty. To actively drive complexity, the framework relies on structural levers such as "Output Contract Tightening," "Compound Conjunction," and "Distractor Injection." The AIME-2024 trajectory aggressively utilizes these active mutations: mathematical difficulty is primarily driven by "Output Contract Tightening" (70%) and "Compound Conjunction" (70%), alongside adversarial reframing like "Distractor Injection" (40%). Similarly, in CodeContests, "Output Contract Tightening" (91%) and "Compound Conjunction" (53%) serve as the primary structural drivers, alongside "Scale Up" (33%) and "Meta Level Shift" (30%). However, a domain disparity emerges: mathematical evolution relies more heavily on structural and adversarial reframing, whereas coding evolution spreads its difficulty across a broader range of mechanisms, including "Multi-case Batching" (24%) and "Adversarial Input Seeding" (22%). This indicates that both domains introduce substantive changes beyond mere instance regeneration, while relying on distinct mechanisms to escalate complexity.

Shift in Evaluated Capabilities. Figure 6 illustrates the evolutionary shift in the breadth of capabilities required by the benchmarks. To escalate cognitive burden, the framework actively enforces exhaustive state-tracking and strict formatting constraints. In CodeContests, the system pivots toward rigorous algorithmic implementation, driving "Construction Search" to surge from 20% to 90% and "Output Contract" compliance to reach 95%; consequently, the original reliance on pure "Domain Reasoning" (96.4%) drastically drops. Similarly, AIME-2024 aggressively amplifies multi-step execution demands: "Multi-case Bookkeeping" skyrockets from 30% to 100% and "Output Contract" emerges at 70%. While AIME maintains its foundational "Domain Reasoning" (90%), it decisively sheds basic "Spec Parsing" (dropping from 83.3% to 30%). Across both domains, this dynamic demonstrates that successful benchmark evolution builds toward exhaustive search, complex execution, and strict protocol compliance, thereby outgrowing the need to test basic conceptual parsing.

Evolution of Solving Techniques. Figure 7 details the shift in the underlying techniques required to crack individual tasks. To actively push models to their limits, the framework enforces rigorous combinatorial searches and survival against adversarial constraints. In CodeContests, evolution creates a massive demand for "Delimiter Protocol Compliance" (96%) and explicit "Magnitude Bound Control" (50%), completely replacing the original dependency on standard "Complexity Budgeting" (which vanishes from 72.7%). Similarly, evolved mathematical tasks in AIME-2024 heavily enforce grueling logical searches dominated by "Combinatorial Enumeration" (90%) and "Maximality Saturation Reasoning" (80%), alongside strict "JSON Schema Compliance" (70%). To make room for these advanced demands, the trajectory decisively abandons standard textbook approaches: initial requirements like "Closed Form Derivation" (63.3%) disappear entirely, while "Exact Integer Arithmetic" plummets from 76.7% to just 10%. Ultimately, by forcing models to master complex casework, distractor resistance, and unyielding protocol adherence, the framework systematically evolves both domains away from straightforward heuristic problem-solving.

Overall, the distributions confirm that METABENCH-HARNESS actively drives domain-specific evolution to increase complexity. In CodeContests, the framework enforces a constraint-based evolution, maximizing difficulty through strict protocol requirements, compound constraints, and meta-level reframing. In contrast, AIME-2024 undergoes active structural evolution, where the system injects distractors and domain-specific structural mutations (e.g., "Modular Reduction Step") and demands heavy combinatorial casework to build upon the core mathematical foundation. Despite these differences, both domains show that METABENCH-HARNESS systematically enriches task composition rather than simply lengthening reasoning, adaptively tailoring its mechanisms to each domain's natural demands.

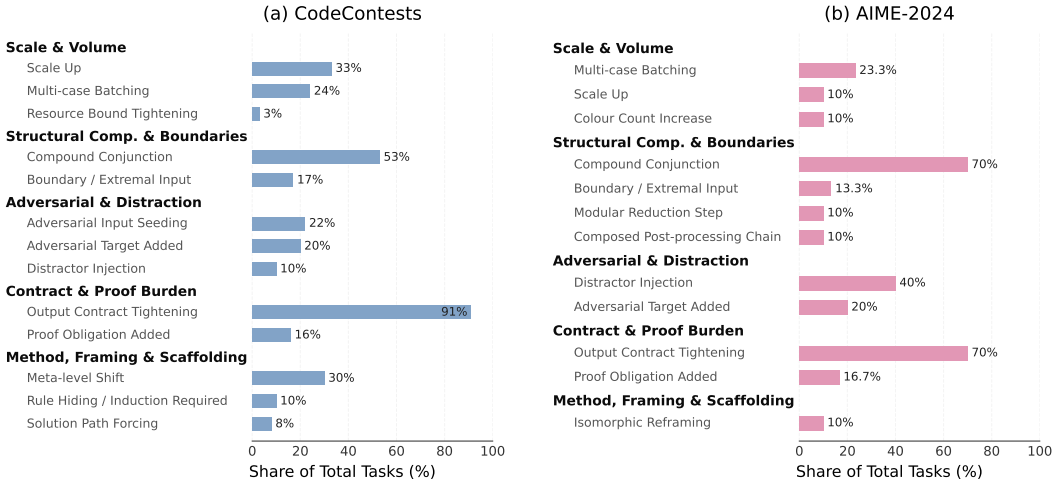


Figure 5: A detailed quantitative analysis of the distribution of difficulty levers.

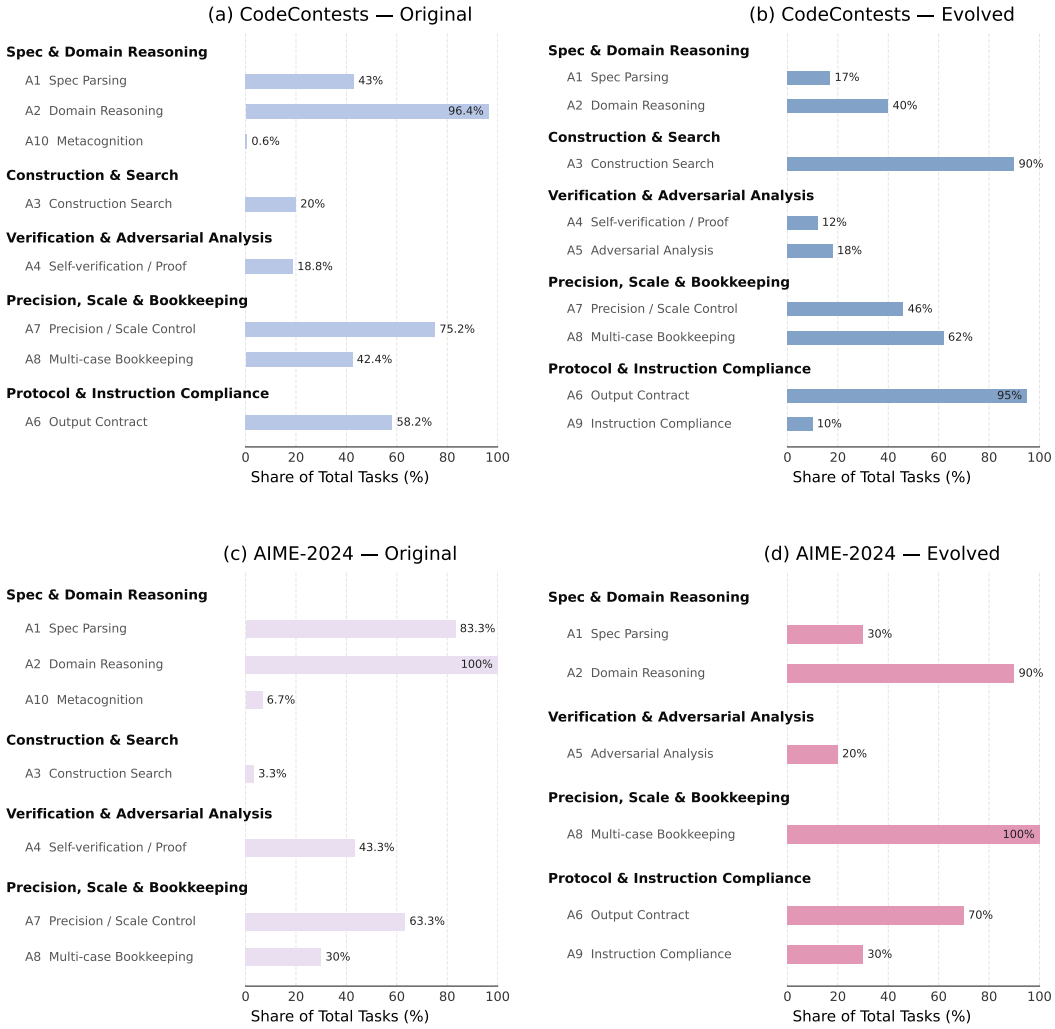


Figure 6: A detailed quantitative analysis of the distribution of the capability distribution.

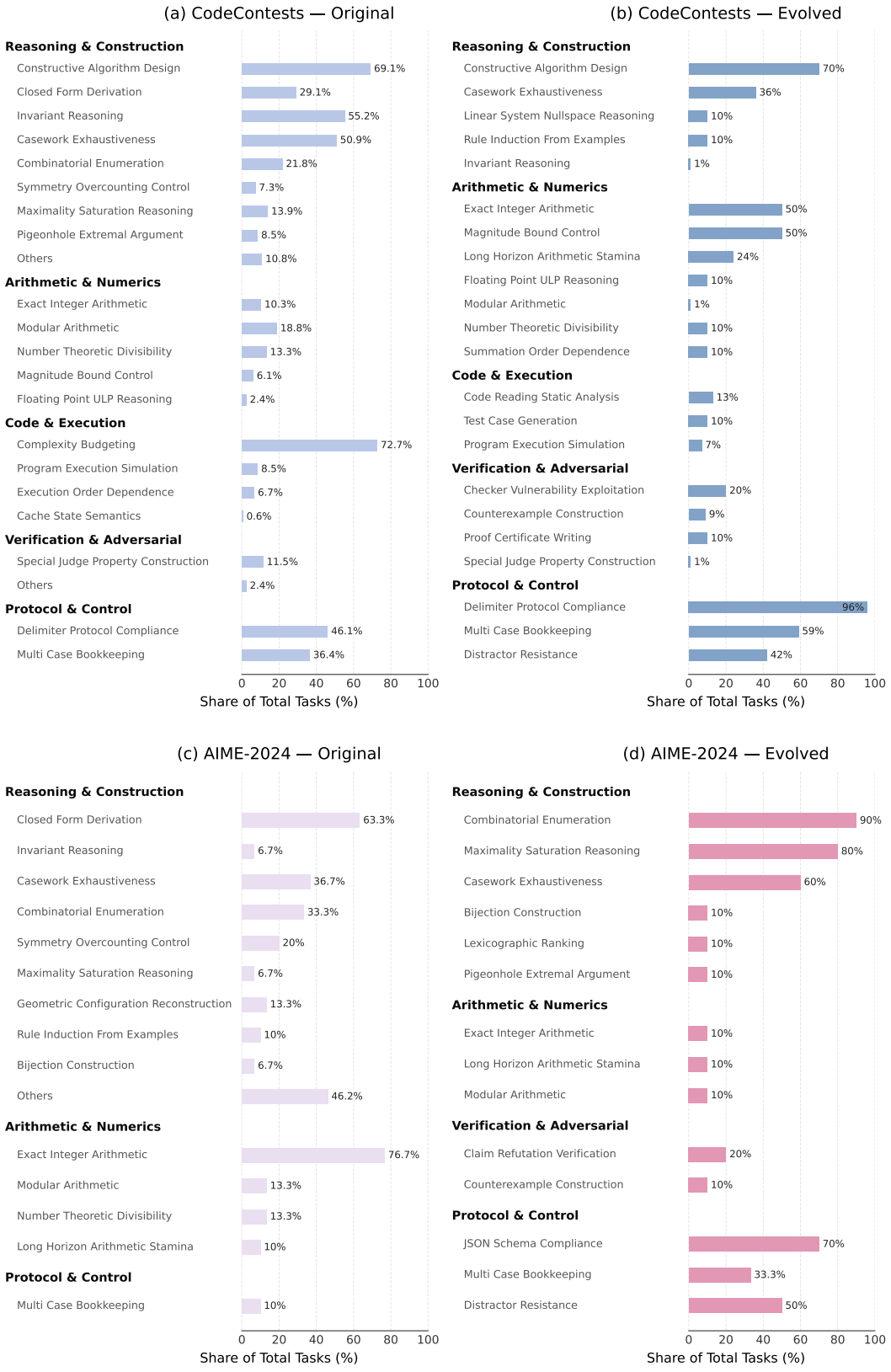


Figure 7: A detailed quantitative analysis of the distribution of the problem-solving technique.

B.2.4 Quality Assessment of the Evolved Benchmarks

Building on Section 4.2, this section evaluates task quality across every evolutionary round rather than focusing solely on late-round outcomes. An isolated LLM judge (`Claude-Opus-5`) assesses each evolved task against its seed across Reasonableness, Competency, and Robustness (Table 6). Figure 8 provides the simplified version system prompt, detailing the essential evaluation criteria while omitting operational overhead like I/O formatting. Figure 9 reveals benchmark evolution as an aggressively exploratory, non-monotonic process. While "Evolutionary Reasonableness" remains consistently high, pushing difficulty boundaries causes severe fluctuations in stricter metrics. Sudden metric drops, such as the collapse in "Evaluator Robustness" during CodeContests Round 8 and AIME Round 4, do not indicate systemic failure; rather, they highlight phases where extreme structural mutations temporarily break solver reliability. Crucially, the system exhibits strong self-correction. Following these exploratory crashes, both trajectories recover entirely, converging to robust final metrics (e.g., $> 90\%$ Evaluator Robustness) and a strict 70% overall acceptance rate ("All Three Yes"). This volatile trajectory proves the framework actively risks temporary quality degradation to discover deeper complexities before successfully stabilizing.

Role & Objective

You are an expert in benchmark design, capability evaluation, and testing. Act as an independent external judge to evaluate the evolved task T against its original seed problem P_0. All benchmark-internal claims must be treated as hypotheses to verify rather than evidence of correctness.

The Evaluation Rubrics

Evaluate the task strictly on the following three axes:

- **1. Evolutionary Reasonableness:** Is T a semantically sound, materially transformed, and internally consistent evolution of P_0? (Reject contradictions, impossible requirements, or unverifiable claims).
- **2. Benchmark Competency:** Does T genuinely measure the intended capability? (Reject tasks where trivial shortcuts bypass the intended reasoning, or the prompt leaks the solution method).
- **3. Evaluator Robustness:** Is the task evaluator deterministic, semantically valid, and resistant to adversarial submissions? (Ensure it fails safely and evaluates semantics rather than surface format).

Decision Framework & Output

Every axis must receive exactly Yes or No (no intermediate verdicts).

- Classify identified issues as **Minor** (cosmetic/mechanically removable) or **Major** (affecting correctness, solvability, competency, or security).
- A Yes permits only harmless Minor issues. Any Major issue requires a strict No.

Output the final verdict for each axis, supported by directly observed or tested evidence, and specify required fixes for any No verdict.

Figure 8: The simplified version system prompt for the quality evaluation using LLM-as-a-judge.

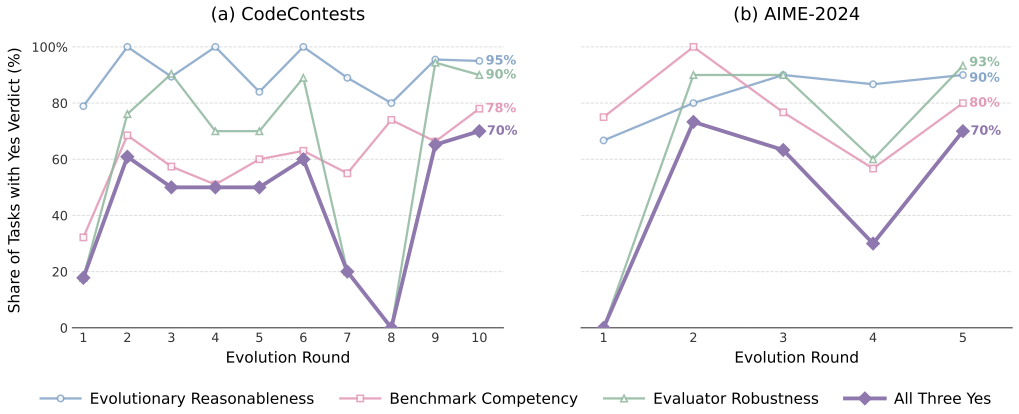


Figure 9: Quality assessment of evolved benchmarks across iterative rounds.

B.2.5 DETAILED CASE STUDIES: FROM SEED TASKS TO EVOLVED TASKS

While Table 7 summarizes the overall evolution, Table 8 provides a comprehensive, round-by-round breakdown for both the MB-CodeContests and MB-AIME scenarios. It explicitly tracks the transformation of the Task Formulation from the initial seed to the deepened stages. Additionally, the table unpacks the progressive expansion of required Capabilities and Solving Techniques to illustrate how new cognitive demands are introduced. Finally, it enumerates the specific Applied Difficulty Levers used at each round to systematically drive this multi-dimensional evolution.

Table 8: METABENCH-HARNESS Evolution Case Study: From Initial Seed to Deepening

Part I: Case Study of MB-CodeContests Evolution

Seed (Codeforces 1582D)	**Round 5** (Evolved Emergence)	**Round 9** (Evolved Deepening)
Task Formulation		
Constructive Math: Given an array a ($a_i \neq 0$), construct b such that $\sum a_i b_i = 0$ and $\sum \lvert b_i \rvert \leq 10^9$.	**Single Gap Audit**: Exploit a planted gap (`range(n-1)`) to trick a Python checker with an invalid (a, b) pair.	**Compound Audit**: Trace complex logic to exploit a vulnerability gated by 3 conditions ($n > 1000$, duplicate a, odd n).
Capabilities and Solving Techniques		
Capabilities: Construction search, Domain reasoning, Precision scale control **Solving Techniques:** Constructive algorithm design, Exact integer arithmetic, Magnitude bound control	**Capabilities:** Adversarial analysis, Construction search, Output contract **Solving Techniques:** Code reading static analysis, Checker vulnerability exploitation, Counterexample construction, Program execution simulation, Protocol compliance	**Capabilities:** Adversarial analysis, Construction search, Precision scale control, Output contract **Solving Techniques:** Code reading static analysis, Checker vulnerability exploitation, Counterexample construction, Long horizon arithmetic stamina, Casework exhaustiveness, Protocol compliance
Applied Difficulty Levers		
Baseline	Meta level shift, Adversarial target added, Compound conjunction, Output contract tightening	Meta level shift, Adversarial target added, Output contract tightening, Compound conjunction, Scale up, Scaffolding removal

Part II: Case Study of MB-AIME Evolution

Seed (AIME 2024-II-9)	**Round 1** (Evolved Emergence)	**Round 5** (Evolved Deepening)
Task Formulation		
Combinatorial Counting: Count placements on a 5×5 grid where no further chip can be added (*answer*: 902).	**Pre-Blocked Obstacle**: Count maximal placements on a 4×4 board with 2 permanently blocked cells (*answer*: 150).	**Post-Hoc Filter**: Count maximal placements on a 3×5 board, then filter out those hitting 4 flagged cells (*answer*: 14).
Capabilities and Solving Techniques		
Capabilities: Domain reasoning, Multi-case bookkeeping **Solving Techniques:** Maximality saturation reasoning, Combinatorial enumeration, Casework exhaustiveness, Symmetry overcounting control	**Capabilities:** Domain reasoning, Multi-case bookkeeping, Spec parsing **Solving Techniques:** Combinatorial enumeration, Maximality saturation reasoning, Casework exhaustiveness	**Capabilities:** Domain reasoning, Multi-case bookkeeping, Instruction compliance **Solving Techniques:** Combinatorial enumeration, Maximality saturation reasoning, Distractor resistance, Casework exhaustiveness
Applied Difficulty Levers		
Baseline	Compound conjunction	Distractor injection, Solution path forcing