ARCHER: From Rationale Imitation to Executable Reasoning in Small Language Models

Anonymous EMNLP submission

Abstract

Small language models are attractive for local, low-cost, and privacy-sensitive deployment, but they remain unreliable on mathematical reasoning tasks that require decomposition, exact computation, tool use, and verification. We propose **ARCHER**: **A**ction-trace **R**einforcement for **C**alculator-grounded **H**ierarchical **E**xecutable **R**easoning, a two-stage framework for training compact models as executable mathematical reasoners. ARCHER first converts chain-of-thought rationales into typed action traces with calculator calls, runtime observations, verification actions, and final answers, teaching the model to control a tool-grounded reasoning process rather than imitate free-form explanations. It then refines this policy with asymmetric hierarchical reinforcement learning, using grouped and conditionally gated trajectory rewards over correctness, tool grounding, process discipline, and trace alignment. Experiments with `Falcon-H1R-7B` show that ARCHER improves GSM-8K accuracy from 77.40 to 85.20 while improving tool recall, observation consistency, trace validity, and process quality. Transfer and baseline comparisons show consistent gains on mathematical reasoning benchmarks, with more limited transfer to broader general-reasoning tasks. These results suggest that compact mathematical reasoners benefit from executable supervision and reward-guided behavioral refinement rather than free-form rationale imitation alone.

1 Introduction

Small language models (SLMs) are attractive for local, low-cost, and privacy-sensitive deployment, but they remain unreliable on mathematical reasoning tasks that require decomposition, exact computation, tool use, and verification. In these tasks, fluent explanations are insufficient: the model must identify relevant quantities, select operations, use intermediate results consistently, and terminate with a well-formed answer. Larger systems can often compensate through scale, stronger reasoning priors, or tool access, but compact models must learn a more disciplined reasoning procedure under limited capacity. This motivates training SLMs not merely to imitate explanations, but to produce reasoning trajectories that can be executed, inspected, and improved.

A common way to improve smaller models is rationale distillation, where a student model is trained on chain-of-thought explanations from datasets or stronger teachers (Wei et al., 2022). However, free-form rationales are a brittle supervision target for compact mathematical reasoners. They are verbose, non-canonical, and often mix reasoning decisions with implicit arithmetic, making it difficult to distinguish what the model should decide from what should be computed exactly. As a result, a student may learn the style of a solution without learning a reliable execution policy. We argue that mathematical reasoning for SLMs should instead be framed as calculator-grounded executable reasoning, where the model controls the solution process and exact arithmetic is delegated to a deterministic runtime.

Existing approaches address parts of this problem but leave an important gap. Rationale distillation transfers explanatory text, but does not convert reasoning into an executable control process. Tool-use methods provide access to calculators or external APIs, but often treat tool invocation as an interaction pattern rather than as a supervised and reward-optimized trajectory structure. In this work, we target this missing middle-ground: it converts existing rationales into executable action traces, treats calculator outputs as deterministic runtime observations, and trains the model to generate complete trajectories whose intermediate steps can be executed, checked, and refined.

We propose **ARCHER**: **A**ction-trace **R**einforcement for **C**alculator-grounded

[0]Our code is available at this link.

Hierarchical **E**xecutable **R**easoning. ARCHER is a two-stage framework for training compact models as executable mathematical reasoners. In Stage 1, **TRACE** (**T**ool-augmented **R**easoning via **A**ction-**C**ompiled **E**xecution) converts legacy chain-of-thought rationales into typed action traces with reasoning steps, calculator calls, runtime observations, verification actions, and final answers. In Stage 2, **HiGRA** (**Hi**erarchically **G**ated **R**eward **A**ggregation) refines the TRACE-trained policy using priority-ordered and conditionally gated trajectory rewards over answer correctness, tool grounding, process discipline, and trace alignment. Unlike flat reward mixtures, HiGRA activates auxiliary signals only when they are relevant to the trajectory, reducing noisy or conflicting supervision during reinforcement learning.

We evaluate ARCHER using `Falcon-H1R-7B` as the primary student model, with GSM-8K as the main benchmark and additional transfer evaluations on mathematical and general reasoning tasks. ARCHER improves GSM-8K accuracy from 77.40 to 85.20 while also improving tool recall, observation consistency, trace validity, and process quality. Across mathematical transfer benchmarks, ARCHER consistently improves over the base model, while broader general-reasoning transfer is more mixed. Error and diagnostic analyses show that the gains are not only answer-level improvements: ARCHER reduces missing tool calls, wrong tool inputs, malformed trajectories, and early reasoning failures.

Contributions. ARCHER makes three contributions. First, it formulates small-model mathematical reasoning as an executable policy-learning problem, where the model emits typed reasoning actions and delegates exact arithmetic to a deterministic calculator runtime instead of imitating unconstrained rationales. Second, ARCHER introduces TRACE, a compilation procedure that turns legacy chain-of-thought solutions into calculator-grounded action traces with explicit tool calls, runtime observations, verification steps, and final answers. Third, ARCHER introduces HiGRA, a hierarchical reinforcement-learning refinement stage that uses priority-ordered and conditionally gated reward families to jointly optimize correctness, tool grounding, process discipline, and trace alignment. Together, these components improve mathematical reasoning accuracy while making intermediate reasoning trajectories more valid, auditable, and tool-grounded.

2 Related Work

Reasoning supervision and distillation for smaller models. Chain-of-thought prompting showed that intermediate reasoning can improve multi-step task performance (Wei et al., 2022), motivating methods that transfer reasoning behavior into smaller models. Prior work studies rationale supervision and stepwise distillation (Hsieh et al., 2023; Shridhar et al., 2023), rationale faithfulness and consistency (Wang et al., 2023), improved distillation objectives for generative students (Gu et al., 2023), more efficient transfer through rationale chunks (Chen et al., 2025), and hybrid small-large reasoning collaboration (Zhao et al., 2025). ARCHER differs from this line of work by changing the unit of supervision: the student is not trained to reproduce explanatory text, but to generate typed executable actions whose computations are delegated to a deterministic runtime.

Tool use and externalized computation. Tool-augmented reasoning methods show that models can improve reliability by delegating exact computation or external operations to specialized runtimes. Program-aided approaches separate natural-language reasoning from executable computation (Gao et al., 2023; Chen et al., 2023), while faithful reasoning frameworks argue that generated reasoning should correspond to an executable process (Lyu et al., 2023). More general tool-use methods interleave reasoning with actions or API calls (Yao et al., 2023; Schick et al., 2023). For mathematics, calculator-centered systems and tool-feedback methods demonstrate the value of explicit computation traces and iterative tool interaction (Kadlčík et al., 2023; Chen et al., 2024). ARCHER differs from prior tool-use methods by making tool use part of the training target itself: legacy rationales are compiled into canonical action traces with paired calculator observations, and the resulting policy is refined using trajectory-level rewards over correctness, tool grounding, and process validity.

Reinforcement learning and process-aware supervision. Recent work shows that reasoning can improve when training uses verifiable outcomes and process-level feedback rather than next-token imitation alone. Process supervision methods reward intermediate reasoning quality instead of only final answers (Lightman et al., 2023; Wang

et al., 2024a), while recent RL-based reasoners show that structured or verifiable objectives can induce stronger reasoning behavior (Guo et al., 2025; Team et al., 2025). At the same time, tool-integrated and stable RL work highlights the difficulty of reward design: sparse, noisy, or overly fragmented signals can make optimization brittle, especially for compact models and longer trajectories (Feng et al., 2025; Dai et al., 2025). HiGRA differs from flat process-reward mixtures by organizing feedback into priority-ordered reward families and activating auxiliary rewards only when they are relevant to the executed trajectory.

3 Methodology

The ARCHER framework, illustrated in Figure 1 has two stages. Stage 1, TRACE, compiles legacy chain-of-thought rationales into calculator-grounded action traces and trains the student by supervised fine-tuning. Stage 2, HiGRA, refines the TRACE policy with grouped and conditionally gated trajectory rewards. TRACE teaches the model the executable action language; HiGRA improves how reliably the model uses that language to solve problems. The key distinction is that ARCHER does not merely expose the model to tools or rationales; it converts rationale supervision into an executable control interface and then optimizes the model over completed, runtime-executed trajectories.

3.1 Executable Reasoning Policy

Each training example is a tuple $(x, r, y^\star)$, where x is the input problem, r is a legacy rationale, and $y^\star$ is the gold answer. ARCHER trains a policy π_θ that maps x to an executable trajectory τ and a final answer $\hat{y}$. A trajectory is a sequence of states $\tau = (s_1, \ldots, s_T)$ and $s_t = (a_t, u_t, o_t)$, where a_t is an action type, u_t is a model-emitted payload, and o_t is an optional runtime observation. We use a compact action vocabulary:

$$\mathcal{A} = \left\{ \begin{array}{l} \texttt{READ, PLAN, CALL_CALC,} \\ \texttt{OBSERVE, VERIFY, ANSWER} \end{array} \right\}$$

The model generates reasoning actions and calculator-call payloads. Calculator observations are produced by a deterministic runtime rather than hallucinated by the model. The final prediction is extracted from the terminal answer action, $\hat{y} = \mathrm{Ans}(\tau)$.

3.2 Stage 1: TRACE

TRACE converts free-form rationales into executable action traces. The compiler $\mathcal{C}$ maps an example $(x, r, y^\star)$ to a gold executable trajectory $\tau^\star = \mathcal{C}(x, r, y^\star)$. The compiler separates textual reasoning spans from arithmetic spans. Textual spans are mapped to actions such as `READ`, `PLAN`, and `VERIFY`. Arithmetic spans are normalized into calculator expressions and emitted as paired tool actions:

$$\texttt{CALL_CALC}(e_t) \rightarrow \texttt{OBSERVE}(f_{\mathrm{calc}}(e_t)),$$

where f_{calc} is the calculator runtime. For GSM-8K, the compiler uses the dataset's inline calculator annotations to identify arithmetic spans, executes each expression, and inserts the resulting observation into the trace. The trace is always terminated with a normalized gold answer.

The student is then trained by supervised fine-tuning on compiled traces:

$$\mathcal{L}_{\mathrm{TRACE}}(\theta) = -\sum_{t \in \mathcal{M}(\tau^\star)} \log \pi_\theta(a_t^\star, u_t^\star \mid x, s_{<t}^\star),$$

where $\mathcal{M}(\tau^\star)$ denotes model-generated positions. Runtime observations are included in the context but are not treated as free-form tokens the model must hallucinate. This teaches the student to control the reasoning process while leaving exact arithmetic to the calculator runtime.

3.3 Stage 2: HiGRA

TRACE gives the student a structured executable interface, but supervised imitation alone can still produce missing tool calls, wrong calculator inputs, malformed traces, ignored observations, or premature answers. HiGRA addresses these failures by refining the TRACE policy using trajectory-level reinforcement learning. The goal is not only to increase final-answer accuracy, but also to improve the reliability of the executable reasoning process that leads to the answer.

A central design choice in HiGRA is its asymmetric distributed hierarchical reward structure. Rather than treating all feedback signals as a flat collection of equally weighted micro-rewards, HiGRA separates trajectory quality into reward families with different responsibilities and priorities. Outcome correctness remains the dominant signal because the final answer must be correct. Tool grounding is the next most important signal because mathematical reasoning often fails when the

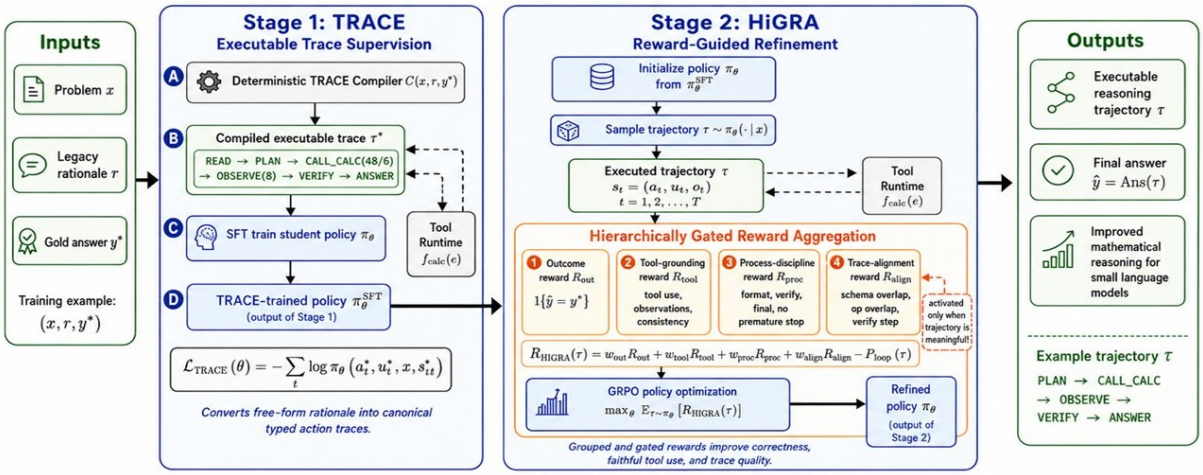

Figure 1: Overview of ARCHER. Modules in Blue are trainable components, Light Gray are non-trainable or deterministic, Orange are Reward or evaluation, and Green are Data/traces/observations.

model omits a needed calculator call, submits the wrong expression, or ignores the returned observation. Process discipline provides a structural signal that rewards valid, complete, concise, and verifiable trajectories. Trace alignment is used as the weakest signal: it regularizes the rollout toward the compiled TRACE structure without forcing exact imitation of the supervised trace.

This asymmetry is important because not every reward component should influence every trajectory in the same way. For example, alignment is useful only when the trajectory is already correct or sufficiently tool-grounded; otherwise, rewarding similarity to the compiled trace can reinforce superficial structure even when the computation is wrong. Similarly, tool-use feedback is meaningful primarily when the problem requires executable computation. HiGRA therefore gates auxiliary rewards conditionally, so each reward family contributes only when it is relevant to the current trajectory. This makes HiGRA different from ordinary outcome-only reinforcement learning and from flat process-reward mixtures: it distributes credit across correctness, tool use, process quality, and trace compatibility, but applies these signals asymmetrically according to their reliability and importance.

For a sampled trajectory τ, HiGRA computes a grouped reward:

$$R_{\text{HiGRA}}(\tau) = w_{\text{out}} R_{\text{out}}(\tau) + w_{\text{tool}} R_{\text{tool}}(\tau) + w_{\text{proc}} R_{\text{proc}}(\tau) + w_{\text{align}} \widetilde{R}_{\text{align}}(\tau) - P_{\text{len}}(\tau). \quad (1)$$

The outcome reward measures final-answer correctness. The tool reward measures whether calculator calls are needed, present, executable, and paired with correct observations. The process reward measures whether the trajectory is valid, complete, concise, and verified. The gated alignment reward softly regularizes the rollout toward the compiled TRACE structure. The penalty term discourages repetitive or overly long trajectories. Full mathematical definitions of the reward families, gates, and penalty are provided in Appendix A.

We optimize the TRACE-initialized policy with Group Relative Policy Optimization (GRPO). For each prompt, we sample a group of trajectories, execute tool calls through the calculator runtime, score each trajectory with R_{HiGRA}, and update the policy using group-relative advantages. This allows ARCHER to optimize non-differentiable trajectory properties such as answer extraction, calculator consistency, schema validity, and trace alignment without training a separate value model.

3.4 Inference

At inference time, ARCHER runs the model in the same executable loop used during training. The model emits one action at a time. Non-tool actions are appended directly to the transcript. When the model emits `CALL_CALC`, the calculator executes the expression and appends an `OBSERVE` action. The process stops when the model emits `ANSWER` or reaches a step budget. Thus, the model is evaluated as a calculator-grounded executable reasoning policy rather than as a free-form text generator.

Full implementation details, reward definitions,

Model	Acc.↑	Tool Score↑	Tool Rec.↑	Proc. Score↑	Valid Trace↑	Obs. Cons.↑
Falcon-H1R-7B	77.40	0.00	0.00	0.52	41.2	0.00
ARCHER w/o HiGRA	79.06	0.46	0.58	0.67	68.7	0.84
ARCHER	**85.20**	**0.76**	**0.79**	**0.79**	**82.4**	**0.93**

Table 1: Main GSM-8K results. ARCHER improves final-answer accuracy and trajectory-quality metrics over the base model. The intermediate row removes Stage 2 HiGRA and reports the Stage 1 TRACE-only variant. Tool Rec. denotes tool recall, and Obs. Cons. denotes observation consistency.

and the GRPO objective are provided in Appendix A.

4 Experimental Setup

Models and training stages. The primary student backbone is `Falcon-H1R-7B`. We evaluate three variants: the base model, `ARCHER w/o HiGRA`, and `ARCHER`. The base model is evaluated without ARCHER training. `ARCHER w/o HiGRA` applies only Stage 1, corresponding to TRACE supervised fine-tuning on compiled executable traces. `ARCHER` is the full two-stage system, initialized from the TRACE checkpoint and refined with HiGRA.

Training data and execution. TRACE uses examples $(x, r, y^\star)$, where x is the problem, r is a legacy rationale, and $y^\star$ is the gold answer. The compiler converts each rationale into a typed trajectory with reasoning actions, calculator calls, runtime observations, verification steps, and a final answer. For GSM-8K, arithmetic spans are identified from the dataset's inline calculator annotations, normalized, executed with a deterministic calculator runtime, and inserted into the trace. During HiGRA rollouts and inference, calculator calls emitted by the model are executed by the same runtime, and returned values are appended as `OBSERVE` actions.

Benchmarks and baselines. GSM-8K is the primary benchmark (Cobbe et al., 2021). We evaluate mathematical transfer on MATH (Hendrycks et al., 2021), SVAMP (Patel et al., 2021), Omni-MATH (Gao et al., 2025a), AIME-2024 (Balunović et al., 2025), and MM, the mathematical subset of MMLU (Hendrycks et al., 2020). We also evaluate GPQA-Diamond (Rein et al., 2023), MMLU-Pro (Wang et al., 2024b), and HLE (Phan et al., 2025) to test broader reasoning transfer. For comparisons with prior methods, we report overlapping results against CoT-FSP, CoT-FT, Toolformer, and CoA, with the compact main-body comparison focused on CoA and the full table in the appendix.

Baseline comparison protocol. For the LLaMA-2-7B and LLaMA-2-Chat-7B backbones, we compare ARCHER with prior methods on GSM-8K, SVAMP, MM, MATH, and AIME-2024. The CoT and Toolformer results are taken directly from Gao et al. (2025b). To validate comparability with CoA, we reproduced the released CoA code and evaluated it in our environment. In Tables 4 and 5, values outside parentheses are from our reproduced CoA runs, while values in parentheses are those reported in the original CoA paper. The reproduced and reported CoA scores are closely aligned, which supports comparing ARCHER against the remaining baselines reported by Gao et al. (2025b) without rerunning every method.

Decoding and metrics. Each model produces one response or executable trajectory per example. For TRACE and ARCHER, the answer is extracted from the terminal `ANSWER` action; invalid or missing answer actions are counted as incorrect. For base and non-ARCHER baselines, the final answer is extracted using the same normalization rules. Final-answer accuracy is the primary metric. We also report tool score, tool precision, tool recall, observation consistency, process score, valid trace rate, trace edit distance, first-error position, and reward-family diagnostics to separate answer correctness from executable reasoning quality.

5 Results

We evaluate whether ARCHER improves both final-answer accuracy and executable reasoning behavior. GSM-8K is the primary benchmark because it directly tests multi-step arithmetic reasoning with verifiable intermediate computations. We compare the base `Falcon-H1R-7B` model with `ARCHER w/o HiGRA`, which uses only Stage 1 executable trace supervision, and `ARCHER`, the full two-stage system.

Primary GSM-8K results. Table 1 shows that Stage 1 of ARCHER improves the base model from 77.40 to 79.06 accuracy. Although this variant gives a modest answer-level gain, it substantially

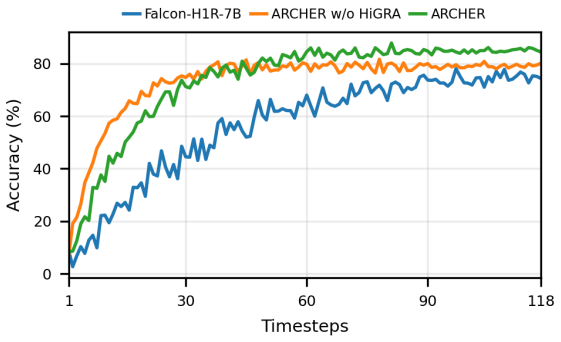

Figure 2: Training dynamics for the base model, ARCHER without Stage 2 HiGRA, and full ARCHER. Stage 1 improves early learning, while the full ARCHER model continues improving after the Stage 1-only curve begins to plateau.

Benchmark	Base	w/o HiGRA	ARCHER	Δ
GSM-8K	77.40	79.06	**85.20**	+7.80
MM	57.38	55.93	**67.25**	+9.87
MATH	74.38	76.31	**83.79**	+9.41
SVAMP	93.76	94.20	**97.56**	+3.80
Omni-MATH	21.48	20.82	**26.38**	+4.90
AIME-2024	88.39	89.46	**93.67**	+5.28

Table 2: Mathematical transfer results on the Falcon-H1R-7B backbone. *w/o HiGRA* denotes the Stage 1-only ARCHER variant.

Benchmark	Base	w/o HiGRA	ARCHER	Δ
GPQA-D	60.48	60.35	**61.76**	+1.28
MMLU-Pro	**72.21**	69.24	71.63	-0.58
HLE	11.18	11.42	**13.65**	+2.47

Table 3: General reasoning transfer results on the Falcon-H1R-7B backbone.

changes the model's behavior: valid trace rate increases from 41.2 to 68.7, tool recall rises from 0.00 to 0.58, and observation consistency reaches 0.84. The full ARCHER model provides the larger gain, reaching 85.20 accuracy, which is 6.14 points above the Stage 1-only variant and 7.80 points above the base model. Tool recall rises to 0.79, observation consistency rises to 0.93, and valid trace rate reaches 82.4. These improvements show that ARCHER improves not only final-answer accuracy, but also the intermediate behavior that makes the answer auditable.

Training dynamics. Figure 2 compares training curves for the three variants. The Stage 1-only ARCHER curve rises faster than the base model, indicating that executable trace supervision provides a stronger learning target than unconstrained response generation. Full ARCHER is learning a complex reward set, which makes it's growth slower and noisier in the beginning. Full ARCHER continues improving after the Stage 1-only curve begins to saturate, showing that reward-guided refinement improves the executable policy learned during Stage 1.

Transfer across benchmarks. Table 2 reports mathematical transfer on the Falcon-H1R-7B backbone. The Stage 1-only variant improves GSM-8K, MATH, SVAMP, and AIME-2024, but drops on MM and Omni-MATH. This suggests that executable trace supervision improves the reasoning interface, but can introduce some distribution mismatch when transferred beyond the compiled training setting. Full ARCHER corrects this brittleness and improves over the base model on every mathematical benchmark, with the largest gains on MM, MATH, and GSM-8K.

Table 3 shows that broader reasoning transfer is more mixed. ARCHER improves GPQA-D and HLE, while MMLU-Pro remains slightly below the base model. This is expected because ARCHER is designed around mathematical decomposition, calculator-grounded execution, and answer verification. The general reasoning results therefore support a focused conclusion: ARCHER is strongest in tasks where intermediate computation and structured verification are central.

Comparison with prior baselines. Tables 4 and 5 compare ARCHER with prior rationale-distillation and tool-augmented baselines. Table 4 includes GSM-8K and SVAMP because these benchmarks contain results for the full set of compared methods. Across all three backbones, ARCHER obtains the best GSM-8K and SVAMP scores among the listed methods. This shows that ARCHER is not only better than the base Falcon model, but also competitive with strong tool-augmented and rationale-distillation baselines.

Table 5 focuses on MM, MATH, and AIME-2024, where the cleanest comparison is between CoA and ARCHER. On LLaMA-2-7B and LLaMA-2-Chat-7B, ARCHER improves over CoA on MM and AIME-2024, while CoA remains higher on MATH. On Falcon-H1R-7B, the main backbone used in this paper, ARCHER improves over CoA on MATH and AIME-2024, while CoA is slightly higher on MM. Overall, these comparisons show that ARCHER is strongest on arithmetic and competition-style mathematical reasoning bench-

Backbone	Method	GSM-8K	SVAMP
LLaMA-2 7B	CoT-FSP	16.38	38.40
	CoT-FT	35.33	48.20
	Toolformer	17.59	37.10
	CoA	37.08 (37.83)	51.70 (50.93)
	ARCHER	**39.25**	**56.21**
LLaMA-2 Chat-7B	CoT-FSP	24.03	51.30
	CoT-FT	35.41	46.90
	CoA (no Tool)	35.03	51.50
	Toolformer	23.65	48.80
	Toolformer-Math	36.01	47.60
	CoA	38.14 (38.29)	53.72 (54.20)
	ARCHER	**39.47**	**58.69**
Falcon H1R-7B	CoA	84.16	95.81
	ARCHER	**85.20**	**97.56**

Table 4: Comparison with rationale-distillation and tool-augmented baselines on GSM-8K and SVAMP. CoT and Toolformer results are taken from Gao et al. (2025b). For CoA, values outside parentheses are from our reproduced runs, while parenthesized values are reported in the original CoA paper.

Backbone	Method	MM	MATH	AIME
LLaMA-2 7B	CoA	65.93	**42.85**	52.70
	ARCHER	**68.32**	40.66	**54.18**
LLaMA-2 Chat-7B	CoA	67.31	**44.19**	53.94
	ARCHER	**69.42**	42.96	**55.73**
Falcon H1R-7B	CoA	**68.42**	82.17	91.58
	ARCHER	67.25	**83.79**	**93.67**

Table 5: Comparison with CoA on additional mathematical benchmarks. Values outside parentheses are from our reproduced CoA runs when available, while parenthesized values are reported in the original CoA paper. AIME denotes AIME-2024.

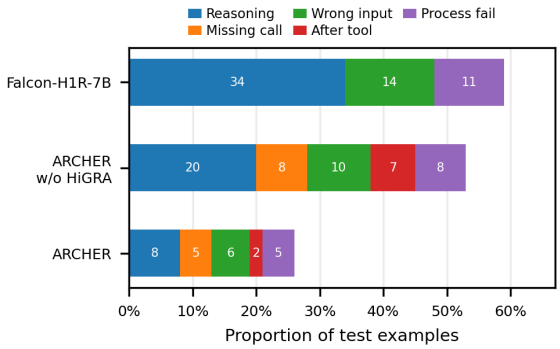

Figure 3: Distribution of dominant error types across model variants. ARCHER's first stage makes intermediate failures more observable, while full ARCHER reduces tool-use and process failures.

marks, while leaving room for improvement on some broader mathematical subsets.

6 Ablations and Error Analysis

We next test which parts of the framework contribute to the observed gains and whether the gains correspond to better reasoning behavior rather than only better final-answer distributions. To keep the main text focused, we report one consolidated ablation table and one error-type figure. Detailed structural metrics, first-error analysis, reward correlations, compiler statistics, and additional baseline tables are moved to Appendix B.

Training target and reward design. Table 6 shows that CoT-SFT only slightly improves over the base model, from 77.40 to 78.12, and barely changes tool or process scores. ARCHER w/o HiGRA improves accuracy to 79.06 while producing much larger gains in tool and process behavior than CoT-SFT. This supports the central motivation for ARCHER's first stage: the key benefit is not simply adding rationales, but changing the supervision target into an executable action format. Reward-guided refinement improves substantially beyond the Stage 1-only variant. Outcome-only GRPO reaches 82.35, and flat micro-reward GRPO reaches 83.10, but full ARCHER performs best at 85.20. This supports the use of HiGRA's grouped macro-rewards with conditional gating rather than a flat reward mixture.

Component contributions. Removing the tool reward causes the largest accuracy drop among the ARCHER reward components, from 85.20 to 81.74. This indicates that explicit tool grounding is central to the improvement, not merely an auxiliary diagnostic. Removing the process reward reduces the process score from 0.79 to 0.61 and lowers accuracy to 82.18, showing that disciplined trajectory structure is also important. Removing alignment has the smallest effect, reducing accuracy to 84.12, which is consistent with its intended role as a weak stabilizer rather than the main learning signal.

The gating ablation is especially important. Removing gating lowers accuracy to 82.95 and reduces both tool and process scores. This supports ARCHER's hierarchical reward design: reward families should be activated only when they are relevant to the trajectory. Without gating, auxiliary rewards can add noise or conflicting pressure, especially for a compact model.

Error distribution. Figure 3 summarizes the dominant failure mode for incorrect trajectories. The base model is dominated by broad reasoning and process failures, which are difficult to audit because the reasoning is not consistently exposed

Model Variant	Acc.↑	Tool Score↑	Proc. Score↑
Falcon-H1R-7B	77.40	0.00	0.52
CoT-SFT	78.12	0.03	0.55
ARCHER w/o HiGRA	79.06	0.46	0.67
ARCHER w/ outcome-only GRPO	82.35	0.55	0.69
ARCHER w/ flat micro-reward GRPO	83.10	0.64	0.72
ARCHER w/o tool reward	81.74	0.51	0.76
ARCHER w/o process reward	82.18	0.71	0.61
ARCHER w/o alignment reward	84.12	0.74	0.77
ARCHER w/o gating	82.95	0.69	0.72
ARCHER	**85.20**	**0.76**	**0.79**

Table 6: Main ablation study on GSM-8K. CoT-SFT tests whether ordinary rationale imitation is sufficient. ARCHER w/o HiGRA isolates the Stage 1 TRACE-only variant. Outcome-only and flat micro-reward GRPO test whether the grouped and gated HiGRA design is necessary. Component removals isolate the contributions of tool grounding, process discipline, alignment, and gating.

as executable structure. ARCHER w/o HiGRA reduces some broad reasoning failures, but it also makes tool-stage errors more visible because the model now emits explicit calculator calls and observations. Full ARCHER produces the strongest error profile, reducing missing-call, wrong-tool-input, and process-failure categories relative to the Stage 1-only variant. This shift is important for interpreting the accuracy gains. The improvement is not only that the final answer is more often correct. The trajectory itself becomes more reliable: fewer failures occur before computation, fewer calculator interactions are malformed or missing, and more solutions terminate through valid tool-grounding.

7 Conclusion and Future Work

We presented **ARCHER**, a two stage framework that combines executable trace supervision with reward-guided refinement: Stage 1 compiles free-form rationales into typed action traces with calculator calls, runtime observations, verification actions, and final answers, while Stage 2 refines this policy with grouped and conditionally gated rewards over correctness, tool grounding, process discipline, and trace alignment. Experiments with `Falcon-H1R-7B` show that ARCHER improves GSM-8K accuracy from 77.40 to 85.20 while also improving tool recall, observation consistency, valid trace rate, and process quality. Transfer and baseline comparisons further show that ARCHER is competitive with strong tool-augmented methods on mathematical reasoning benchmarks, supporting the value of executable supervision and reward-guided refinement for small mathematical reasoners.

Future work can extend ARCHER in three directions. First, the action space can be expanded beyond calculator calls to include symbolic solvers, unit converters, code execution, retrieval, and proof checking, enabling richer forms of executable reasoning. Second, TRACE can be generalized from rule-based arithmetic trace compilation to broader trace construction pipelines that handle more diverse rationale formats and reasoning domains. Third, HiGRA can be further improved through automatic reward calibration, adaptive gating, and model-specific reward weighting. These extensions would test whether the same principle behind ARCHER, aligning reasoning supervision with executable intermediate computation, can improve SLMs across a wider range of tasks.

Limitations

ARCHER is evaluated primarily in mathematical reasoning settings where intermediate computations can be executed and final answers can be checked reliably. This makes the setting well suited for studying calculator-grounded reasoning, but it also means that the current results should be interpreted most directly for tasks with verifiable numerical or symbolic structure. Current implementation uses a deterministic TRACE compiler and manually designed HiGRA reward families, so broader domains may require richer action schemas, additional tools, and task-specific reward calibration. We view these as natural extensions of the framework rather than fundamental restrictions, since ARCHER is designed around the general idea of training compact models to produce executable and auditable reasoning trajectories.

Disclosure of AI Assistant Use

The authors used LLM-based writing assistants to revise and improve portions of the manuscript prose based on text originally written by the authors. The authors retained full responsibility for the scientific content of the paper, including the research ideas, technical approach, experiments, results, and conclusions. All AI-assisted revisions were carefully inspected by the authors for accuracy, and the manuscript was manually checked to ensure that no hallucinated, fabricated, or unsupported content was introduced. GPT-Codex and Claude Code Coding Agents were used to assist with implementation and debugging for some parts of the experimental codebase.

References

Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. 2025. MathArena: Evaluating LLMs on Uncontaminated Math Competitions. In *NeurIPS Datasets and Benchmarks Track*.

Nuo Chen, Hongguang Li, Baoyuan Wang, and Jia Li. 2024. From good to great: Improving math reasoning with tool-augmented interleaf prompting. In *Proceedings of the 2nd Workshop on Natural Language Reasoning and Structured Explanations (@ACL 2024)*, pages 64–79, Bangkok, Thailand. Association for Computational Linguistics.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.

Xiaoshu Chen, Sihang Zhou, Ke Liang, Xiaoyu Sun, and Xinwang Liu. 2025. Skip-thinking: Chunk-wise chain-of-thought distillation enable smaller language models to reason better and faster. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

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

Muzhi Dai, Shixuan Liu, and Qingyi Si. 2025. Stable reinforcement learning for efficient reasoning. *arXiv preprint arXiv:2505.18086*.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. 2025. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*.

Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Zhengyang Tang, et al. 2025a. Omni-math: A universal olympiad level mathematic benchmark for large language models. In *International Conference on Learning Representations*, volume 2025, pages 100540–100569.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International conference on machine learning*, pages 10764–10799. PMLR.

Silin Gao, Jane Dwivedi-Yu, Ping Yu, Xiaoqing Ellen Tan, Ramakanth Pasunuru, Olga Golovneva, Koustuv Sinha, Asli Celikyilmaz, Antoine Bosselut, and Tianlu Wang. 2025b. Efficient tool use with chain-of-abstraction reasoning. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 2727–2743.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. 2023. Minillm: Knowledge distillation of large language models. *arXiv preprint arXiv:2306.08543*. Version 5 (revised Nov 2025).

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*.

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. *arXiv preprint arXiv:2103.03874*.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. In *Findings of the Association for Computational Linguistics: ACL 2023*.

Marek Kadlčík, Michal Štefánik, Ondrej Sotolár, and Vlastimil Martinek. 2023. Calc-x and calcformers: Empowering arithmetical chain-of-thought through interaction with symbolic systems. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 12101–12108.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. In *The twelfth international conference on learning representations*.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 305–329.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are nlp models really able to solve simple math word problems? In *Proceedings of the 2021 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 2080–2094.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, et al. 2025. Humanity's last exam. *arXiv preprint arXiv:2501.14249*.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. 2023. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551.

Kumar Shridhar, Alessandro Stolfo, and Mrinmaya Sachan. 2023. Distilling reasoning capabilities into smaller language models. In *Findings of the Association for Computational Linguistics: ACL 2023*.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. 2025. Kimi k1. 5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*.

Peifeng Wang, Zhengyang Wang, Zheng Li, Yifan Gao, Bing Yin, and Xiang Ren. 2023. SCOTT: Self-consistent chain-of-thought distillation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5546–5558, Toronto, Canada. Association for Computational Linguistics.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024a. 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.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. 2024b. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37:95266–95290.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Kun Zhao, Bohao Yang, Chen Tang, Siyuan Dai, Haoteng Tang, Chenghua Lin, and Liang Zhan. 2025. Dre: An effective dual-refined method for integrating small and large language models in open-domain dialogue evaluation. *arXiv preprint arXiv:2506.04516*.

A Full ARCHER Methodology

This appendix gives the full technical description of ARCHER: **A**ction-trace **R**einforcement for **C**alculator-grounded **H**ierarchical **E**xecutable **R**easoning. ARCHER consists of two stages. TRACE constructs executable supervision from legacy rationales and trains the initial policy. HiGRA refines the TRACE policy with grouped and gated trajectory rewards.

A.1 Problem Setting

Each training example is represented as $(x, r, y^\star)$, where x is the input problem, r is a free-form rationale or chain-of-thought solution, and $y^\star$ is the gold final answer. The goal is to train a compact student policy π_θ that generates a structured trajectory τ and final answer $\hat{y}$:

$$\pi_\theta : x \mapsto (\tau, \hat{y}).$$

Unlike ordinary rationale distillation, ARCHER does not train the student to imitate raw explanatory text. Instead, it trains the student to operate in an executable action language with explicit calculator calls and deterministic runtime observations.

A.2 Executable Action Language

A trajectory is a sequence of structured states:

$$\tau = (s_1, s_2, \ldots, s_T), \qquad s_t = (a_t, u_t, o_t).$$

Here, a_t is an action type, u_t is the model-emitted payload, and o_t is the observation returned by the runtime when the action invokes a tool. ARCHER uses the action vocabulary

$$\mathcal{A} = \left\{ \begin{array}{l} \texttt{READ},\ \texttt{PLAN},\ \texttt{CALL_CALC}, \\ \texttt{OBSERVE},\ \texttt{VERIFY},\ \texttt{ANSWER} \end{array} \right\}$$

The final prediction is extracted from the terminal answer action:

$$\hat{y} = \mathrm{Ans}(\tau).$$

A.3 Stage 1: TRACE

TRACE is the supervised compilation stage of ARCHER. It converts a free-form rationale into a calculator-grounded executable trace:

$$\tau^\star = \mathcal{C}(x, r, y^\star),$$

where $\mathcal{C}$ is the TRACE compiler.

Action	Role
READ	Restates or identifies the relevant information in the problem.
PLAN	Specifies the next reasoning step or decomposition.
CALL_CALC	Submits an arithmetic expression to the calculator runtime.
OBSERVE	Records the deterministic output returned by the runtime.
VERIFY	Checks whether the intermediate or final result is consistent.
ANSWER	Emits the final normalized answer.

Table A.1: ARCHER action vocabulary. The model generates reasoning actions and calculator-call payloads; calculator observations are produced by the runtime.

A.3.1 Trace Compilation

The compiler segments the rationale into textual reasoning spans and arithmetic spans. Textual spans are mapped to reasoning actions such as READ, PLAN, and VERIFY. Arithmetic spans are normalized into executable calculator expressions. For each extracted expression e_t, TRACE emits:

$$\begin{aligned} s_t^\star &= (\texttt{CALL_CALC}, e_t, \varnothing), \\ s_{t+1}^\star &= (\texttt{OBSERVE}, \varnothing, f_{\text{calc}}(e_t)). \end{aligned}$$

Here, f_{calc} is the deterministic calculator runtime. For example, a rationale fragment containing the operation $48/6 = 8$ is converted into:

$$\texttt{CALL_CALC}(48/6) \rightarrow \texttt{OBSERVE}(8).$$

For GSM-8K, the compiler uses the dataset's inline calculator format, « `expr = result` », to identify arithmetic spans. Each expression is normalized, executed, and paired with the resulting observation. Text outside calculator annotations is retained as reasoning content and mapped into structured actions.

A.3.2 Compiler Policy

The TRACE compiler is deterministic and rule-based. It does not use an external LLM to rewrite or repair rationales. The compiler follows a retain-and-annotate policy:

- Examples with valid calculator spans are converted into executable traces.
- Examples without calculator spans are still converted into structured traces when possible.
- Arithmetic mismatches are repaired at the observation level when exact execution succeeds.

- Unsupported calculator spans are retained with compiler metadata.
- Each trace is terminated with the normalized gold answer.

This policy avoids discarding useful training examples while preserving metadata for later filtering and analysis.

A.3.3 Calculator Runtime

The calculator runtime separates numerical execution from reasoning control. If the model emits a calculator call with payload e_t, the runtime returns:

$$o_t = f_{\text{calc}}(e_t).$$

The observation is inserted into the transcript and becomes part of the context for subsequent model actions. The model therefore learns what to compute and how to use the result, while the runtime performs exact arithmetic.

A.3.4 Student Policy

The student policy generates action types and payloads autoregressively:

$$\pi_\theta(a_t, u_t \mid x, s_{<t}).$$

The trajectory distribution is:

$$\pi_\theta(\tau \mid x) = \prod_{t \in \mathcal{M}(\tau)} \pi_\theta(a_t, u_t \mid x, s_{<t}),$$

where $\mathcal{M}(\tau)$ is the set of model-generated positions. Runtime observations are not sampled from the language model and are not treated as model decisions.

A.3.5 TRACE Supervised Fine-tuning

Given a compiled gold trace $\tau^\star = (s_1^\star, \ldots, s_{T^\star}^\star)$, TRACE trains the student with:

$$\mathcal{L}_{\text{TRACE}}(\theta) = -\sum_{t \in \mathcal{M}(\tau^\star)} \log \pi_\theta(a_t^\star, u_t^\star \mid x, s_{<t}^\star).$$

Runtime observations are included in the teacher-forced context, but the loss is applied only to model-generated positions. This preserves the distinction between decisions made by the policy and observations produced by external execution.

A.3.6 TRACE Inference Loop

At inference time, the TRACE-trained policy runs iteratively. At step t, the model emits an action and payload:

$$(a_t, u_t) \sim \pi_\theta(\cdot \mid x, s_{<t}).$$

The runtime observation is:

$$o_t = \begin{cases} f_{\text{calc}}(u_t), & \text{if } a_t = \texttt{CALL_CALC}, \\ \varnothing, & \text{otherwise.} \end{cases}$$

The resulting state is appended to the trajectory. The process continues until the model emits `ANSWER` or reaches the maximum step budget.

A.4 Stage 2: HiGRA

HiGRA is the reward-guided refinement stage of ARCHER. It starts from the TRACE policy and improves it using trajectory-level rewards. The purpose of HiGRA is to correct failures that supervised trace imitation alone may not remove, including missing calculator calls, wrong expressions, ignored observations, malformed traces, and premature answers.

A.4.1 Reward Families

HiGRA groups fine-grained feedback into four macro-reward families:

$$R_{\text{out}}, \qquad R_{\text{tool}}, \qquad R_{\text{proc}}, \qquad R_{\text{align}}.$$

The outcome reward measures final-answer correctness. The tool reward measures calculator use and execution faithfulness. The process reward measures trajectory validity and discipline. The alignment reward softly encourages compatibility with the compiled TRACE supervision.

A.4.2 Outcome Reward

The outcome reward is:

$$R_{\text{out}}(\tau) = \mathbf{1}[\hat{y} = y^\star].$$

This is the primary task reward and anchors optimization to answer correctness.

A.4.3 Tool-grounding Reward

Let $c^\star$ be the number of calculator calls in the compiled gold trace, and let m be the number of calculator calls emitted by the model. Define:

$$u_{\text{need}} = \mathbf{1}[c^\star > 0], \qquad u_{\text{used}} = \mathbf{1}[m > 0].$$

For emitted calls, define:

$$u_{\text{obs}} = \frac{1}{\max(m,1)} \sum_{i=1}^{m} \mathbf{1}[\exists\, o_i \text{ paired with call } i],$$

and

$$u_{\text{cons}} = \frac{1}{\max(m,1)} \sum_{i=1}^{m} \mathbf{1}[o_i = f_{\text{calc}}(e_i)].$$

Here, e_i is the expression submitted in the ith calculator call and o_i is the paired observation. The tool reward is:

$$R_{\text{tool}}(\tau) = \begin{cases} \alpha_1 u_{\text{used}} + \alpha_2 u_{\text{obs}} + \alpha_3 u_{\text{cons}}, & \text{if } u_{\text{need}} = 1, \\ 1 - \beta u_{\text{used}}, & \text{if } u_{\text{need}} = 0. \end{cases}$$

The coefficients satisfy:

$$\alpha_1, \alpha_2, \alpha_3 \geq 0, \quad \alpha_1 + \alpha_2 + \alpha_3 = 1, \quad \beta \in [0,1].$$

This reward encourages calculator use when required and discourages unnecessary calculator use when computation is not needed.

A.4.4 Process-discipline Reward

Process discipline evaluates whether the trajectory is well formed and complete. Define:

$$\begin{aligned} p_{\text{fmt}} &\in [0,1], \\ p_{\text{verify}} &= \mathbf{1}[\texttt{VERIFY} \in \tau], \\ p_{\text{short}} &= \mathbf{1}[T \leq T_{\max}], \\ p_{\text{final}} &= \mathbf{1}[\texttt{ANSWER} \in \tau], \\ p_{\text{prem}} &= \mathbf{1}[\text{premature final answer}]. \end{aligned}$$

The raw process reward is:

$$R_{\text{proc}}(\tau) = \gamma_1 p_{\text{fmt}} + \gamma_2 p_{\text{verify}} + \gamma_3 p_{\text{short}} + \gamma_4 p_{\text{final}} - \gamma_5 p_{\text{prem}}.$$

It is clipped to keep it bounded:

$$R_{\text{proc}}(\tau) \leftarrow \min(1, \max(0, R_{\text{proc}}(\tau))).$$

A.4.5 Trace-alignment Reward

Trace alignment softly regularizes the rollout toward the compiled executable trace without requiring exact imitation. Let $\mathcal{S}(\tau)$ and $\mathcal{S}^\star$ denote the generated and gold action schemas, and let $\mathcal{O}(\tau)$ and $\mathcal{O}^\star$ denote the generated and gold operation sequences. Define:

$$a_{\text{schema}} = \frac{|\mathcal{S}(\tau) \cap \mathcal{S}^\star|}{|\mathcal{S}^\star| + \varepsilon},$$

$$a_{\text{op}} = \frac{|\mathcal{O}(\tau) \cap \mathcal{O}^\star|}{|\mathcal{O}^\star| + \varepsilon},$$

and

$$a_{\text{verify}} = \mathbf{1}[\text{gold trace expects verification}] \cdot \mathbf{1}[\texttt{VERIFY} \in \tau].$$

The alignment reward is:

$$R_{\text{align}}(\tau) = \delta_1 a_{\text{schema}} + \delta_2 a_{\text{op}} + \delta_3 a_{\text{verify}}.$$

This term is intentionally weaker than outcome, tool, and process rewards.

A.4.6 Hierarchical Gating

HiGRA gates auxiliary rewards so that irrelevant reward dimensions do not contribute uniformly to every trajectory. Tool grounding is naturally gated by tool necessity through u_{need}. Alignment is also gated:

$$\widetilde{R}_{\text{align}}(\tau) = g_{\text{align}}(\tau) R_{\text{align}}(\tau),$$

where

$$g_{\text{align}}(\tau) = \mathbf{1}[R_{\text{out}}(\tau) = 1 \vee R_{\text{tool}}(\tau) > \eta].$$

Thus, alignment is emphasized only when the trajectory is already correct or shows sufficient tool-grounded structure.

A.4.7 Length and Loop Penalty

To discourage repetitive or overly long trajectories, HiGRA uses a soft length penalty:

$$P_{\text{len}}(\tau) = \lambda_{\text{len}} \max\left(0, \frac{T - T_{\max}}{T_{\max}}\right).$$

This is smoother than a hard failure condition and is better suited to compact policies.

A.4.8 Final HiGRA Reward

The final HiGRA reward is:

$$R_{\text{HiGRA}}(\tau) = w_{\text{out}} R_{\text{out}}(\tau) + w_{\text{tool}} R_{\text{tool}}(\tau) + w_{\text{proc}} R_{\text{proc}}(\tau) + w_{\text{align}} \widetilde{R}_{\text{align}}(\tau) - P_{\text{len}}(\tau).$$

The weights follow the priority ordering:

$$w_{\text{out}} > w_{\text{tool}} > w_{\text{proc}} > w_{\text{align}}.$$

In our experiments, we use:

$$w_{\text{out}} = 2.0, \quad w_{\text{tool}} = 1.0, \quad w_{\text{proc}} = 0.5, \quad w_{\text{align}} = 0.25.$$

A.5 Policy Optimization with GRPO

HiGRA starts from the TRACE-supervised policy π_θ^{SFT}. Given an input x, the current policy samples a group of G trajectories:

$$\{\tau^{(1)}, \ldots, \tau^{(G)}\} \sim \pi_{\phi_{\text{old}}}(\cdot \mid x).$$

Each trajectory is executed through the calculator runtime and scored:

$$r_i = R_{\text{HiGRA}}(\tau^{(i)}).$$

The group mean and standard deviation are:

$$\bar{r} = \frac{1}{G}\sum_{i=1}^{G} r_i, \qquad \sigma_r = \sqrt{\frac{1}{G}\sum_{i=1}^{G}(r_i - \bar{r})^2 + \epsilon}.$$

The group-relative advantage is:

$$A_i = \frac{r_i - \bar{r}}{\sigma_r}.$$

For token $z_{i,t}$ in trajectory $\tau^{(i)}$, define the importance ratio:

$$\rho_{i,t} = \frac{\pi_\phi(z_{i,t} \mid x, z_{i,<t})}{\pi_{\phi_{\text{old}}}(z_{i,t} \mid x, z_{i,<t})}.$$

The clipped ratio is:

$$\bar{\rho}_{i,t} = \text{clip}(\rho_{i,t}, 1-\varepsilon, 1+\varepsilon).$$

The GRPO loss is:

$$\mathcal{L}_{\text{GRPO}}(\phi) = -\mathbb{E}_x\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|\tau^{(i)}|}\sum_{t=1}^{|\tau^{(i)}|} \min\left(\rho_{i,t}A_i, \bar{\rho}_{i,t}A_i\right)\right].$$

This objective increases the likelihood of trajectories whose HiGRA reward is higher than the group average while clipping overly large policy updates.

A.6 Why ARCHER Uses Two Stages

TRACE and HiGRA play complementary roles. TRACE converts raw rationale supervision into an executable action language and gives the compact model an initial tool-using policy. HiGRA then improves that policy with rewards that evaluate complete executed trajectories. This separation is important because reward optimization from an unstructured policy is unstable, while supervised imitation alone can overfit to compiled traces and leave tool-use failures unresolved. ARCHER combines both: TRACE teaches the model how to produce executable reasoning traces, and HiGRA improves whether those traces solve the problem reliably.

B Additional ARCHER Diagnostic Results

This appendix reports additional diagnostics for ARCHER that support the main results but are not required in the main body. The goal is to separate answer-level gains from improvements in executable reasoning behavior, including trace validity, calculator use, observation consistency, first-error position, and reward-family behavior. In these diagnostic tables, *w/o HiGRA* denotes the Stage 1-only ARCHER variant.

Structural and tool-use metrics. Table B.1 reports whether the model produces executable traces and uses calculator calls faithfully. Accuracy measures final-answer correctness. NED_{act} and NED_{op} are normalized edit distances to the compiled gold action and operation sequences, so lower values indicate closer alignment with the target executable trace. Valid Trace is the percentage of outputs that follow the expected action format. Tool precision measures whether emitted tool calls are appropriate, tool recall measures whether needed calls are made, and observation consistency measures agreement between recorded observations and deterministic calculator outputs.

Metric	Falcon	ARCHER w/o HiGRA	ARCHER
Acc.↑	77.40	79.06	**85.20**
NED_{act} ↓	0.48	0.29	**0.21**
NED_{op} ↓	0.55	0.34	**0.26**
Valid Trace↑	41.2	68.7	**82.4**
Tool Prec.↑	0.00	0.71	**0.83**
Tool Rec.↑	0.00	0.58	**0.79**
Obs. Cons.↑	0.00	0.84	**0.93**

Table B.1: Structural and tool-use diagnostics on GSM-8K. TRACE substantially improves trace validity and tool use over the base model, while HiGRA further improves both executable structure and tool faithfulness.

The main inference is that TRACE changes the form of the model's reasoning: valid traces increase from 41.2 to 68.7, and tool recall rises from 0.00 to 0.58. HiGRA then improves reliability within that format, raising valid trace rate to 82.4, tool recall to 0.79, and observation consistency to 0.93. Thus, the final accuracy gain is accompanied by cleaner execution behavior.

First-error position. Table B.2 shows where trajectories first become incorrect. No Error is the percentage of examples without a detected reasoning failure. Early Error refers to failures during problem understanding or planning. Tool-Step Error refers to failures in calculator-call construction

or observation handling. Post-Tool Error refers to failures after a correct tool observation has already been produced. Mean First-Error Pos. is normalized from 0 to 1, where larger values mean the first error occurs later in the trajectory.

Metric	Falcon	ARCHER w/o HiGRA	ARCHER
No Error↑	77.40	79.06	**85.20**
Early Error↓	12.8	8.6	**5.1**
Tool-Step Error↓	4.9	7.1	**4.3**
Post-Tool Error↓	**4.9**	5.2	5.4
Mean First-Error Pos.↑	0.38	0.51	**0.64**

Table B.2: First-error position on GSM-8K. TRACE shifts errors later in the trajectory, and HiGRA further reduces early and tool-step failures.

The key pattern is that TRACE reduces early failures and moves the first error later, but it also exposes more tool-step errors because the model now attempts explicit calculator use. HiGRA reduces both early errors and tool-step errors, and the mean first-error position increases from 0.38 to 0.64. This indicates that ARCHER improves not only final correctness but also the durability of the reasoning process.

Reward-family correlation. Table B.3 measures how strongly each reward family is associated with final-answer correctness. Correlation measures linear association with correctness, while AUC measures how well a reward score separates correct from incorrect trajectories. R_{out} has perfect association because it directly encodes answer correctness. The auxiliary rewards are not expected to be perfect predictors, but they should still correlate with better trajectories.

Reward	Corr.↑	AUC↑
R_{out}	1.00	1.00
R_{tool}	0.54	0.78
R_{proc}	0.37	0.69
R_{align}	0.29	0.63
R_{HiGRA}	**0.68**	**0.84**

Table B.3: Association between reward-family scores and final-answer correctness. Auxiliary rewards provide useful but incomplete signals, while the combined HiGRA reward is the strongest non-trivial predictor.

The auxiliary rewards follow the intended ordering. Tool grounding is the strongest auxiliary signal, process discipline is moderately predictive, and trace alignment is weaker but still useful as a stabilizer. The combined HiGRA reward has higher correlation and AUC than any individual auxiliary reward, supporting the grouped reward design.

Objective comparison. Table B.4 compares different training objectives. Acc. is final-answer accuracy, Tool is the aggregate tool-use score, Proc. is the process-discipline score, Valid is valid trace rate, and Obs. is observation consistency.

Variant	Acc.	Tool	Proc.	Valid	Obs.
Falcon-H1R-7B	77.40	0.00	0.52	41.2	0.00
CoT-SFT	78.12	0.03	0.55	44.6	0.05
TRACE-SFT	79.06	0.46	0.67	68.7	0.84
Outcome-only GRPO	82.35	0.55	0.69	71.8	0.86
Flat micro-reward GRPO	83.10	0.64	0.72	75.3	0.88
ARCHER	**85.20**	**0.76**	**0.79**	**82.4**	**0.93**

Table B.4: Detailed objective comparison on GSM-8K. Executable trace supervision is stronger than free-form CoT-SFT, and grouped HiGRA reward optimization outperforms outcome-only and flat micro-reward alternatives.

The numbers show that ordinary CoT-SFT provides only a small gain over the base model and does not meaningfully improve tool behavior. TRACE-SFT improves trace validity and observation consistency, confirming that the executable supervision format matters. Outcome-only GRPO improves accuracy, but full HiGRA performs best across all metrics because it rewards correctness together with tool grounding and process discipline.

TRACE compiler outcomes. Table B.5 summarizes compiler outcomes on the GSM-8K training split. The `ok` category indicates examples with successfully compiled executable traces. `calc_only` indicates examples where the compiler primarily recovered calculator structure. `no_calc_blocks` indicates examples without explicit GSM-8K calculator annotations. `unsupported_exec` indicates examples with calculator spans that could not be executed by the current runtime.

Compiler Output	Count	Percent
`ok`	6,982	93.0
`calc_only`	118	1.6
`no_calc_blocks`	373	5.0
`unsupported_exec`	47	0.6
Total prepared examples	7,520	100.0

Table B.5: TRACE compiler status counts on the GSM-8K training split. Most examples are successfully compiled, while a small fraction is retained with metadata for transparency.

The compiler statistics show that TRACE covers most of the training split: 93.0 percent of pre-

pared examples compile successfully. The remaining cases are small but useful to report because they clarify that the compiler follows a retain-and-annotate policy rather than silently dropping difficult examples.

C TRACE Compiler in ARCHER: Operational Details

This appendix details the TRACE compiler used in Stage 1 of ARCHER. The high-level description of TRACE as a "trace compiler" is intentionally simple, but the actual compilation pipeline is fully deterministic and rule-based in the current implementation. It takes a prepared GSM-8K row and converts it into a structured `TraceExample` containing:

$$\big(\texttt{id},\ \texttt{split},\ x,\ r_{\text{raw}},\ y^\star,\ \mathcal{S}^\star,\ \tau^\star,\ \texttt{compiler_status},\ \texttt{metadata}\big).$$

where:

- x is the question,
- r_{raw} is the original answer string from GSM-8K,
- $y^\star$ is the parsed gold final answer,
- $\mathcal{S}^\star$ is the set of inferred schema tags,
- $\tau^\star$ is the compiled action trace.

C.1 Compiler Type: Rule-Based Rather than Model-Assisted

The current compiler is **rule-based**, not model-assisted. It does not invoke an external LLM to repair or reinterpret rationales. Instead, it uses deterministic parsing rules, regular expressions, whitespace normalization, numeric canonicalization, and exact calculator execution. This design choice is deliberate: the project aims to reduce dependence on large teachers and to produce executable supervision from existing dataset rationales alone.

Formally, the compiler is a deterministic mapping

$$\mathcal{C} : (x, r_{\text{raw}}, y^\star) \mapsto \tau^\star,$$

implemented through symbolic preprocessing and exact execution rather than learned sequence transduction.

C.2 Detection of Arithmetic Spans

Arithmetic spans are detected through the GSM-8K convention of inline calculator blocks:

`« expr = result ».`

The parser uses a regular expression of the form

`« expr = result »`

to locate each arithmetic block inside the rationale. If a match is found, the compiler extracts:

- the raw arithmetic expression *expr*,
- the rationale-provided numeric result *result*,
- the span boundaries in the original rationale.

Let the rationale be r. The parser decomposes it into an ordered sequence of pieces:

$$\text{Split}(r) = (p_1, p_2, \ldots, p_M),$$

where each p_j is either:

- a `text` segment, or
- a `calc` segment corresponding to one matched arithmetic block.

Text segments become `reason` actions, while calculator segments become paired

`tool_call` $\rightarrow$ `tool_observation`

actions.

Limitation of the current detector. Only arithmetic explicitly marked in the GSM-8K-style inline format is guaranteed to be externalized. Arithmetic implied only in prose but not wrapped in ‘ $<< \cdot >>$ ‘ is not converted into executable tool calls in the current compiler. Such spans remain plain reasoning text. As a result, some examples may receive the status `no_calc_blocks` even if arithmetic is conceptually present.

C.3 Expression and Number Normalization

Before execution, the compiler normalizes both extracted expressions and numeric results.

Whitespace normalization. All extracted arithmetic expressions are compacted by collapsing repeated whitespace:

$$e = \text{CompactWS}(e_{\text{raw}}).$$

This ensures that syntactic variants such as

`48 / 6,  48/6,  48 / 6`

map to a stable expression form.

Numeric normalization. Rationale-provided results and final answers are canonicalized using decimal parsing. Let v denote a numeric string. The normalization function

$$\mathrm{NormNum}(v)$$

performs the following:

1. remove commas,
2. strip trailing decimal points,
3. parse the value using decimal arithmetic,
4. if the value is integral, render it as an integer string,
5. otherwise remove redundant trailing zeros.

For example:

$$\texttt{"1,200.0"} \mapsto \texttt{"1200"}, \qquad \texttt{"8.500"} \mapsto \texttt{"8.5"}.$$

This same normalization is used for final answer parsing and for rationale-provided arithmetic results, which ensures consistent comparison across compiled traces, tool observations, and evaluation outputs.

C.4 Handling of Unsupported or Ambiguous Rationale Spans

The current compiler is conservative. It does not attempt to infer hidden arithmetic from ambiguous prose, nor does it rewrite unsupported spans into executable expressions using a learned model.

Instead, unsupported or ambiguous spans are handled as follows:

1. if a rationale segment is plain text, it is retained as a `reason` action,
2. if an arithmetic block is present but the expression cannot be executed exactly by the calculator, the compiler still emits a `tool_call` with the extracted payload and stores the rationale-provided result in the observation,
3. the observation metadata is marked with a status of the form

$$\texttt{calc_error:ExceptionType}.$$

Thus, the compiler never drops the surrounding example solely because one arithmetic block is unsupported. Instead, it degrades gracefully by preserving the action structure and annotating the observation status.

C.5 Repair of Rationale Arithmetic

TRACE explicitly checks whether the rationale-provided result agrees with exact execution. Let e_t be an extracted expression and let

$$r_t^{\mathrm{exp}}$$

be the result written in the rationale. The compiler computes

$$r_t^{\mathrm{act}} = f_{\mathrm{calc}}(e_t).$$

If execution succeeds and the `repair_equation_result` option is enabled, then:

$$r_t^{\mathrm{final}} = \begin{cases} r_t^{\mathrm{act}}, & \text{if } r_t^{\mathrm{act}} \neq r_t^{\mathrm{exp}}, \\ r_t^{\mathrm{exp}}, & \text{otherwise.} \end{cases}$$

When a mismatch occurs, the observation metadata is marked as

$$\texttt{status} = \texttt{"repaired"}.$$

This means that the compiler can correct arithmetic slips inside the legacy rationale while preserving the overall reasoning skeleton. In other words, TRACE does not merely copy rationale text. It checks arithmetic spans against an exact executor and repairs them when possible.

C.6 Compiler Status and Compilation Outcomes

Every prepared row is compiled into a serialized trace example. The current implementation therefore treats compilation as a *total transformation* rather than a binary success-or-failure filter. Instead of dropping rows, it assigns a categorical compiler status:

- `ok`: at least one calculator block and at least one reasoning text span were extracted,
- `calc_only`: calculator blocks were extracted but there were no remaining text spans,
- `no_calc_blocks`: no inline calculator block was found in the rationale.

Hence, all prepared examples produce an output trace, but the `compiler_status` records the degree to which the rationale could be externalized into tool-using form.

This has two implications for reporting:

1. the number of "successfully compiled" examples should be reported by status counts rather than by a single binary success count,

2. training-time filtering may optionally restrict supervision to rows whose status is `ok` or `calc_only`, especially in early experiments.

Let

$$N_{\text{ok}}, \quad N_{\text{calc_only}}, \quad N_{\text{no_calc}}$$

denote the counts of these categories on the train split. Then the paper can report:

$$N_{\text{total}} = N_{\text{ok}} + N_{\text{calc_only}} + N_{\text{no_calc}}.$$

Because the compiler emits one trace per prepared input row, there is no separate dropped-example count in the current deterministic pipeline unless an additional filtering policy is introduced downstream.

C.7 Dropped, Repaired, or Retained Examples

The current compiler follows a **retain-and-annotate** policy:

- examples are **retained** even if no calculator block is found,
- arithmetic mismatches are **repaired** when exact execution succeeds and repair is enabled,
- unsupported execution spans are **retained with error metadata** rather than removed.

Therefore:

- failed compilations are *not* dropped by default,
- ambiguous spans are *not* rewritten by a learned repair model,
- arithmetic mismatches are repaired only at the observation level, not by re-authoring the entire trace.

C.8 Gold Final Answer Handling

The final answer in the compiled trace is **forced to match the gold answer** parsed from the GSM-8K annotation. Let the raw GSM-8K answer string be r_{raw}. The gold answer is extracted from the standard marker

$$\texttt{\#\#\#\#}\ y^{\star},$$

and then normalized numerically:

$$y^{\star} = \text{ParseFinal}(r_{\text{raw}}).$$

The compiler strips this terminal answer marker from the rationale text before creating intermediate actions, but the final compiled trace always ends with

$$\texttt{final_answer} = y^{\star}.$$

In addition, when the verify step is included, the compiler inserts the gold answer into verification metadata:

$$\texttt{metadata}[\texttt{gold_final_answer}] = y^{\star}.$$

Thus, the compiled supervision does not ask the student to recover the terminal answer purely from the legacy rationale string. Instead, it presents the student with a cleaned executable trace whose final target answer is always the gold normalized answer.

C.9 Practical Reporting Recommendation

For reproducibility, we recommend reporting the following compiler statistics in the paper:

1. total number of prepared examples,
2. counts and percentages for `ok`, `calc_only`, and `no_calc_blocks`,
3. number of repaired arithmetic observations,
4. average number of calculator steps per compiled trace,
5. average number of reasoning spans per trace,
6. optional filtered subset size if training excludes `no_calc_blocks` rows.

These statistics make it clear how much of the original rationale corpus was successfully externalized into executable tool-using supervision and how much remained purely textual.

C.10 Role of TRACE in the ARCHER Pipeline

Within ARCHER, TRACE is responsible for converting legacy rationale supervision into the executable action format that HiGRA later refines. This separation is important because TRACE defines the control interface, while HiGRA optimizes how reliably the trained policy uses that interface during executed rollouts. TRACE and HiGRA operate at different stages of the pipeline. TRACE

is the *compilation and supervised imitation layer*; it converts raw dataset rationales into executable structured supervision and uses them to initialize the student policy. HiGRA is the *reinforcement refinement layer*; it optimizes the SFT-initialized policy using grouped and gated trajectory-level rewards.

Analogically, TRACE provides the student with a structured reasoning language and explicit tool-use behavior. HiGRA then improves the quality of that behavior by optimizing for correctness, faithful tool use, process discipline, and trace compatibility without requiring a flat and unstable reward over many fine-grained labels.

D ARCHER Evaluation Metrics

This appendix defines the metrics used to evaluate ARCHER as an executable reasoning policy rather than as a free-form text generator. Let $\tau = (s_1, \ldots, s_T)$ denote a model-generated trajectory and let $\tau^\star = (s_1^\star, \ldots, s_{T^\star}^\star)$ denote the gold compiled TRACE trajectory. Each state contains an action type, an optional action payload, and an optional tool observation. The predicted answer extracted from τ is denoted by $\hat{y}$, and the gold answer is denoted by $y^\star$.

D.1 Final-Answer Accuracy

Final-answer accuracy measures whether the extracted prediction matches the gold answer after normalization:

$$\mathrm{Acc} = \frac{1}{|\mathcal{D}|} \sum_{(x,y^\star)\in\mathcal{D}} \mathbf{1}[\hat{y} = y^\star]. \tag{D.1}$$

For TRACE and HiGRA, $\hat{y}$ is extracted from the terminal `ANSWER` action. If no valid `ANSWER` action is present, the trajectory is counted as incorrect.

D.2 Valid Trace Rate

A generated trajectory is considered valid if it follows the required structured format, contains parseable action tags, has well-formed tool calls, pairs tool observations with tool calls, and terminates with a valid answer action. Let $V(\tau)$ be the validity indicator:

$$V(\tau) = \mathbf{1}[\tau \text{ is a valid structured trace}]. \tag{D.2}$$

The valid trace rate is

$$\mathrm{ValidTrace} = \frac{1}{|\mathcal{D}|} \sum_{\tau\in\mathcal{D}} V(\tau). \tag{D.3}$$

D.3 Normalized Levenshtein Trace Distance

To measure structural alignment with the compiled executable trace, we compute normalized Levenshtein edit distance over action sequences. Let $A(\tau)$ denote the action-type sequence extracted from τ:

$$A(\tau) = (a_1, a_2, \ldots, a_T).$$

The action-level normalized edit distance is

$$\mathrm{NED}_{\mathrm{act}}(\tau, \tau^\star) = \frac{\mathrm{Lev}(A(\tau), A(\tau^\star))}{\max(|A(\tau)|, |A(\tau^\star)|, 1)}, \tag{D.4}$$

where $\mathrm{Lev}(\cdot,\cdot)$ is Levenshtein distance.

We also compute an operation-level distance over the sequence of arithmetic operations. Let $O(\tau)$ denote the sequence of calculator operation types extracted from the generated trajectory, such as addition, subtraction, multiplication, and division. The operation-level normalized edit distance is

$$\mathrm{NED}_{\mathrm{op}}(\tau, \tau^\star) = \frac{\mathrm{Lev}(O(\tau), O(\tau^\star))}{\max(|O(\tau)|, |O(\tau^\star)|, 1)}. \tag{D.5}$$

Lower values indicate stronger alignment with the compiled executable reasoning structure.

D.4 Tool-Use Precision, Recall, and Observation Consistency

Let $C(\tau)$ be the set of tool calls emitted in a generated trajectory and let $C^\star(\tau^\star)$ be the set of required tool calls in the gold compiled trace. Let $\mathrm{match}(c, c^\star)$ indicate that a generated call c matches a required gold call $c^\star$ in relevant quantities, operation type, and trajectory position.

Tool precision measures whether emitted tool calls are appropriate:

$$M_{\mathrm{prec}}(c, \tau^\star) = \mathbf{1}[\exists c^\star \in C^\star(\tau^\star) : \mathrm{match}(c, c^\star)]. \tag{D.6}$$

$$\mathrm{Prec}_{\mathrm{tool}} = \frac{\sum_{c\in C(\tau)} M_{\mathrm{prec}}(c, \tau^\star)}{\max(|C(\tau)|, 1)}. \tag{D.7}$$

Tool recall measures whether required calls are actually made:

$$M_{\mathrm{rec}}(c^\star, \tau) = \mathbf{1}[\exists c \in C(\tau) : \mathrm{match}(c, c^\star)]. \tag{D.8}$$

$$\mathrm{Rec}_{\mathrm{tool}} = \frac{\sum_{c^\star\in C^\star(\tau^\star)} M_{\mathrm{rec}}(c^\star, \tau)}{\max(|C^\star(\tau^\star)|, 1)}. \tag{D.9}$$

Observation consistency measures whether recorded tool observations agree with executable calculator results. For the i-th generated tool call, let e_i be the submitted expression and o_i be the recorded observation. The calculator output is $f_{\text{calc}}(e_i)$. We define

$$\text{ObsCons} = \frac{\sum_{i=1}^{|C(\tau)|} \mathbf{1}[o_i = f_{\text{calc}}(e_i)]}{\max(|C(\tau)|, 1)}. \quad \text{(D.10)}$$

D.5 Position of First Error

To analyze where trajectories fail, we identify the earliest causally relevant error in the generated reasoning process. Let $E_t(\tau)$ be an indicator that step t is the first step where the trajectory becomes incorrect or unreliable. This can include an incorrect decomposition, missing tool call, wrong tool input, inconsistent observation, premature final answer, or malformed trace transition. The first error step is

$$t_{\text{err}}(\tau) = \min\{t : E_t(\tau) = 1\}. \quad \text{(D.11)}$$

If no error is identified, we set $t_{\text{err}}(\tau) = \varnothing$ and report the trajectory as a no-error case.

For erroneous trajectories, the normalized first-error position is

$$\text{PFE}(\tau) = \frac{t_{\text{err}}(\tau)}{T}, \qquad t_{\text{err}}(\tau) \neq \varnothing. \quad \text{(D.12)}$$

Higher values indicate that the trajectory remains correct for a larger fraction of the reasoning process before failing.

We also group first errors into phases:

$$\phi_{\text{err}}(\tau) \in \{\textsc{Plan}, \textsc{Tool}, \textsc{Observe}, \textsc{Verify}, \textsc{Answer}, \textsc{None}\}. \quad \text{(D.13)}$$

D.6 Reward-Family Scores

HiGRA defines four macro-reward families:

$$R_{\text{out}}, \qquad R_{\text{tool}}, \qquad R_{\text{proc}}, \qquad R_{\text{align}}.$$

The full training reward is defined in Eq. 1. For reporting in the main tables, we use a diagnostic HiGRA score that summarizes the three most directly comparable reward components:

$$S_{\text{HiGRA}}(\tau) = 2R_{\text{out}}(\tau) + R_{\text{tool}}(\tau) + 0.5R_{\text{proc}}(\tau). \quad \text{(D.14)}$$

This reported score excludes trace alignment and the loop penalty so that it can be compared consistently across the base model, TRACE, and HiGRA. The full HiGRA training reward still includes alignment and loop penalties as described in Section A.

For a dataset $\mathcal{D}$, the average diagnostic score is

$$\overline{S}_{\text{HiGRA}} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} S_{\text{HiGRA}}(\tau). \quad \text{(D.15)}$$

D.7 Reward-Family Contribution

For each reward family $k \in \{\text{out}, \text{tool}, \text{proc}, \text{align}\}$, we define its weighted contribution over a dataset $\mathcal{D}$ as

$$\text{Contrib}_k = \frac{\sum_{\tau \in \mathcal{D}} w_k R_k(\tau)}{Z + \varepsilon}, \quad \text{(D.16)}$$

where

$$Z = \sum_{\tau \in \mathcal{D}} \Big(w_{\text{out}} R_{\text{out}}(\tau) + w_{\text{tool}} R_{\text{tool}}(\tau) + w_{\text{proc}} R_{\text{proc}}(\tau) + w_{\text{align}} \widetilde{R}_{\text{align}}(\tau) \Big). \quad \text{(D.17)}$$

The constant ε prevents division by zero. For gated rewards, R_k refers to the gated version when applicable.

D.8 Correlation with Correctness

To test whether a reward family is associated with final-answer correctness, we compute its Pearson correlation with the binary correctness indicator. Let

$$z(\tau) = \mathbf{1}[\hat{y} = y^\star].$$

For reward family R_k, define centered variables

$$\Delta R_k(\tau) = R_k(\tau) - \overline{R}_k, \qquad \Delta z(\tau) = z(\tau) - \overline{z}. \quad \text{(D.18)}$$

The correlation is

$$\rho_k = \frac{\sum_{\tau \in \mathcal{D}} \Delta R_k(\tau) \Delta z(\tau)}{\sqrt{\sum_{\tau \in \mathcal{D}} \Delta R_k(\tau)^2} \sqrt{\sum_{\tau \in \mathcal{D}} \Delta z(\tau)^2}}. \quad \text{(D.19)}$$

A higher value indicates that the reward family is more predictive of final-answer correctness.

D.9 AUC of Reward Families

We also compute the area under the ROC curve by treating each reward-family score as a scalar predictor of final-answer correctness. For each reward family R_k, trajectories are ranked by $R_k(\tau)$ and evaluated against $z(\tau)$. Higher AUC indicates that the reward family better separates correct trajectories from incorrect ones.

E ARCHER Error Taxonomy and Analysis

This appendix provides the full error taxonomy and annotation protocol used to analyze ARCHER failures in the tool-augmented reasoning setting. The goal of this analysis is twofold. First, we use it for *behavior diagnosis*, where we identify how and where an executable trajectory fails. Second, we use it for *reward auditing*, where we check whether the HiGRA reward decomposition assigns credit and blame in a way that agrees with human judgment. Unlike final-answer accuracy, this analysis distinguishes between reasoning failures, tool-use failures, post-tool failures, and structural trace failures.

E.1 Unit of Analysis

Each evaluated example produces a trajectory

$$\tau = (s_1, s_2, \ldots, s_T),$$

where each state may contain a structured action, an action payload, a tool call, a tool observation, or a final answer. The annotation unit is the full generated trajectory, not only the final answer. For each trajectory, we inspect the input problem, the gold answer, the generated trace, tool calls and observations, the predicted answer, and the corresponding gold compiled trace when available.

A trajectory may contain several mistakes. To avoid over-fragmenting the analysis, each example receives one primary error label and, when useful, one secondary error label. The primary label is assigned to the earliest dominant causal failure. For example, if a model calls the calculator with the wrong expression and then gives an incorrect final answer, the primary label is `wrong_tool_input`, not merely `answer_failure`. If a correct tool result is obtained but later misused, the primary label is `wrong_after_correct_tool`. If the trace is malformed in a way that prevents reliable parsing or execution, the primary label is assigned to the relevant process failure category.

E.2 Sampling Setup

We analyze two complementary GSM-8K test-set samples. The first is an unbiased random sample, used to estimate realistic aggregate error proportions. The second is a diagnostic stratified sample, used to inspect targeted failure types and debug reward behavior. Table E.1 summarizes the two samples.

Sample A: Random
Construction: Fixed-seed random draw from the GSM-8K test set without filtering. **Bias:** Unbiased. **Use:** Estimate error proportions and report aggregate statistics.
Sample B: Stratified
Construction: Stratified by answer correctness, tool necessity, and reasoning length. **Bias:** Diagnostic. **Use:** Inspect failure modes and debug reward behavior.

Table E.1: Analysis samples used for manual inspection and reward auditing.

E.3 Per-Example Analysis Views

For each sampled example, we inspect five complementary views.

Outcome view. We record the gold answer, predicted answer, and final-answer correctness. We also determine whether the final error is primary or downstream from an earlier reasoning, tool-use, or process failure.

Tool-use view. We check whether tool use was needed, whether the model attempted a tool call, whether the tool call had a paired observation, whether the observation matches executable output, and whether the final answer correctly uses the tool result.

Process-discipline view. We inspect whether the generated transcript follows the expected structured format. This includes checking for required tags, malformed action ordering, duplicated or missing actions, premature final answers, excessive verbosity, and non-productive loops.

Trace-alignment view. We compare the generated trajectory against the compiled gold trace using schema tags, operation tags, calculator-call structure, and verify-step compatibility. This view is used as a soft diagnostic signal rather than as a strict correctness requirement.

Reward-audit view. We compare the human primary error label with the HiGRA reward decomposition. This allows us to identify whether the dominant automatic penalty agrees with the main human-identified failure.

E.4 Expanded Error Taxonomy

Table E.2 gives the full diagnostic taxonomy. These labels are used during manual inspection before

Table E.2: Expanded manual error taxonomy for structured tool-augmented mathematical reasoning.

Error label	Definition	Typical diagnostic signal	Primary reward family affected
reasoning_error	The model decomposes the problem incorrectly, selects the wrong operation, misses a relevant quantity, or makes an invalid inference before or independent of tool execution.	Incorrect plan, wrong equation setup, wrong mathematical relation, or unsupported intermediate conclusion.	R_{out}, sometimes R_{align}
wrong_tool_input	The model invokes a tool, but the submitted expression or arguments are incorrect.	Calculator call uses the wrong operation, wrong quantity, wrong order of arguments, or incomplete expression.	R_{tool}, R_{out}
wrong_tool_observation	The generated observation is inconsistent with the executable result of the tool call.	The trace records a value that does not match the calculator output for the submitted expression.	R_{tool}
missing_tool_call	A tool call is needed according to the gold trace or task structure, but the model does not invoke the tool.	The trajectory performs arithmetic internally or jumps to an answer without an expected calculator call.	R_{tool}, R_{proc}
spurious_tool_call	The model invokes a tool when tool use is unnecessary or irrelevant.	Calculator call appears for a simple non-computational step or for an expression unrelated to the problem.	R_{tool}, R_{proc}
wrong_after_correct_tool	The tool call and observation are correct, but the model misuses the result in later reasoning or answer synthesis.	Correct calculator output appears in the trace, but the final answer or next reasoning step contradicts it.	R_{out}, R_{proc}
premature_final_answer	The model terminates before sufficient reasoning, tool use, or verification has occurred.	Final answer is emitted immediately after reading or planning, before required evidence is established.	R_{proc}, R_{out}
format_or_trace_error	The trace is malformed, incomplete, duplicated, or violates the canonical action structure.	Missing final-answer tag, malformed tool-call tag, observation without call, repeated incompatible actions, or unparsable transcript.	R_{proc}, R_{align}
loop_or_verbosity_error	The trajectory is excessively long, repetitive, or trapped in non-productive reasoning.	Repeated plans, repeated calculator calls with no progress, or reasoning that exceeds the rollout budget.	R_{proc}, P_{loop}
gold_trace_mismatch	The generated schema or operator pattern diverges substantially from the compiled gold trace, even if some local steps are plausible.	Missing expected action types, wrong operation sequence, or incompatible verify-step structure.	R_{align}
mixed_or_ambiguous	No single failure type is clearly dominant, or several failures are tightly entangled.	Multiple errors occur at nearly the same point, or the first causal failure cannot be determined reliably.	Case dependent

being collapsed into the top-level reporting buckets in Table E.4.

E.5 Manual Annotation Schema

Table E.3 shows the annotation schema used for each inspected example. The primary and secondary labels use the taxonomy in Table E.2. The auxiliary fields are used to support reward auditing and qualitative analysis.

E.6 Collapsed Reporting Buckets

For aggregate reporting, we collapse the detailed labels into a smaller set of top-level error buckets. This makes comparisons across model variants easier while retaining the distinctions most relevant to tool-augmented reasoning. Table E.4 defines the reporting buckets.

E.7 Reward-Audit Protocol

For each trajectory, we compute the four HiGRA macro-reward families:

$$R_{\text{out}}, \qquad R_{\text{tool}}, \qquad R_{\text{proc}}, \qquad R_{\text{align}}.$$

We then compare the human primary error label with the dominant automatic reward penalty. Agreement means that the reward family receiving the largest meaningful penalty corresponds to the human-identified failure type. For example, if the human label is `wrong_tool_input`, the reward audit agrees when the tool-grounding reward is the dominant degraded component. If the human label is `format_or_trace_error`, the reward audit agrees when the process-discipline or trace-alignment reward accounts for the main penalty.

Table E.5 shows the expected relationship between error labels and HiGRA reward families.

The audit is useful because outcome accuracy alone cannot identify whether a wrong answer

Field	Meaning
Primary and secondary manual error labels	
`reasoning_error`	Incorrect decomposition, inference, or mathematical reasoning before or independent of tool execution.
`wrong_tool_input`	Tool was called, but the submitted expression or arguments were incorrect.
`wrong_tool_observation`	Tool observation in the trace is inconsistent with executable tool output.
`missing_tool_call`	A needed tool call is absent despite tool necessity in the gold trace.
`spurious_tool_call`	Tool was invoked unnecessarily on an example that does not require it.
`wrong_after_correct_tool`	Tool use is correct, but later reasoning or answer synthesis is incorrect.
`premature_final_answer`	The model terminates with a final answer before sufficient support is established.
`format_or_trace_error`	Trace is malformed, incomplete, duplicated, or violates the canonical action structure.
`loop_or_verbosity_error`	Reasoning is excessively repetitive, long, or trapped in a non-productive loop.
`gold_trace_mismatch`	Generated schema or operator pattern diverges substantially from the compiled gold trace.
`mixed_or_ambiguous`	No single failure type is clearly dominant, or multiple failures are tightly entangled.
Auxiliary annotation fields	
`first_wrong_step`	Earliest step at which the trajectory becomes incorrect or unreliable.
`recoverable`	Whether the trajectory could still plausibly recover after the first wrong step.
`reward_agreement`	Whether the dominant HiGRA penalty matches the main human diagnosis.
`notes`	Optional free-form explanation for edge cases or disagreement cases.

Table E.3: Manual annotation schema used for error diagnosis and reward auditing.

Bucket	Definition
Answer failure	Final answer is incorrect, regardless of upstream cause.
Tool omission	A needed tool call is missing.
Tool faithfulness failure	Tool input or tool observation is inconsistent with executable tool behavior.
Post-tool reasoning failure	Tool use is correct, but later reasoning or answer synthesis is wrong.
Premature stopping	Final answer is produced before sufficient evidence or intermediate support.
Process failure	Trace is malformed, incomplete, duplicated, or excessively verbose.
Trace mismatch	Generated schema or operator pattern diverges from the compiled gold trace.

Table E.4: Top-level error buckets used for aggregate reporting.

Manual error label	Expected dominant reward signal
`reasoning_error`	R_{out}, sometimes R_{align}
`wrong_tool_input`	R_{tool} and R_{out}
`wrong_tool_observation`	R_{tool}
`missing_tool_call`	R_{tool}
`spurious_tool_call`	R_{tool}
`wrong_after_correct_tool`	R_{out} and R_{proc}
`premature_final_answer`	R_{proc} and R_{out}
`format_or_trace_error`	R_{proc} and R_{align}
`loop_or_verbosity_error`	R_{proc} and P_{loop}
`gold_trace_mismatch`	R_{align}
`mixed_or_ambiguous`	Case dependent

Table E.5: Expected mapping between manual error labels and HiGRA reward-family penalties.

came from poor decomposition, missing tool use, incorrect tool input, misuse of a correct observation, or malformed trace structure. HiGRA is intended to preserve this diagnostic information while avoiding direct optimization over a large set of unstable micro-rewards.

E.8 Analysis Workflow

The full workflow consists of five steps. First, we run each model variant on the evaluation set and store the generated trace, tool calls, tool observations, and final answer. Second, we compute automatic outcome correctness, tool-use metadata, process-validity indicators, and HiGRA macro-reward scores. Third, we construct the random and stratified samples described in Table E.1. Fourth, we manually annotate each sampled trajectory using the schema in Table E.3. Fifth, we compare the manual labels against automatic buckets and reward-family decompositions to identify agreement and disagreement cases.

Disagreement cases are especially important. They reveal situations where the automatic reward may over-penalize a secondary issue, under-penalize a causal error, or fail to separate tool input construction from downstream answer synthesis. These cases are used to revise reward weights, gating thresholds, and process penalties.

E.9 Observed Error Patterns

The main-body error distribution figure shows a consistent shift across model variants. The base Falcon model is dominated by broad reasoning errors and process failures. This indicates that the base model often produces plausible natural-language reasoning but lacks reliable control over multi-step execution.

TRACE changes the error profile. After supervised trace training, some broad reasoning errors are reduced, while tool-related failures become more explicit. This is expected because TRACE forces the model to expose tool decisions, calculator inputs, observations, verification steps, and final answers in a typed format. As a result, failures that were previously hidden inside generic free-form rationales become easier to localize. For example, an incorrect solution can now be diagnosed as a missing tool call, wrong tool input, wrong observation, or post-tool synthesis error.

HiGRA produces the most favorable profile. Relative to TRACE alone, it reduces missing tool calls, wrong tool inputs or observations, post-tool reasoning failures, and process failures. This supports the central design choice of HiGRA: grouped and conditionally activated macro-rewards provide useful learning pressure without exposing the small model to a large, brittle set of independent micro-rewards.

E.10 Interpretation

For ARCHER, the central question is not only whether the final answer improves, but whether failures become later, more localized, and more attributable to specific executable actions. The taxonomy supports the paper's main claim that TRACE and HiGRA play complementary roles. TRACE improves observability and structure by converting free-form rationales into executable traces. This makes model failures more inspectable, even when the model is still wrong. HiGRA then improves the quality of those structured trajectories by reducing tool-use errors, stabilizing process behavior, and aligning optimization with the behaviors that make final answers reliable.

Thus, the observed accuracy improvements are not merely answer-level gains. They are accompanied by a shift in the internal failure structure of the model: fewer broad reasoning failures, fewer ungrounded tool interactions, fewer malformed traces, and more trajectories that terminate with verified tool-grounded answers.

F Implementation Details

This appendix summarizes implementation settings details for ARCHER.

Variant	Description
`Falcon-H1R-7B`	Base model evaluated without ARCHER training.
`ARCHER w/o HiGRA`	TRACE supervised fine-tuning on compiled executable traces.
`ARCHER`	TRACE initialization followed by HiGRA reward-guided refinement.

Table F.1: Primary ARCHER model variants.

Action	Implementation role
`READ`	Identify or restate relevant information.
`PLAN`	Specify decomposition or the next reasoning step.
`CALL_CALC`	Submit an arithmetic expression to the calculator.
`OBSERVE`	Insert the deterministic calculator output.
`VERIFY`	Check intermediate or final consistency.
`ANSWER`	Emit the normalized final answer.

Table F.2: ARCHER action vocabulary used by the implementation.

Compiler setting	Value
Include fact extraction action	`true`
Include verification action	`true`
Maximum reasoning span length	220 characters
Repair arithmetic result with runtime output	`true`
Default schema label	`arithmetic_word_problem`

Table F.3: Default TRACE compiler settings.

Status	Meaning
`ok`	Reasoning text and calculator spans are both detected.
`calc_only`	Calculator spans are detected with little or no remaining reasoning text.
`no_calc_blocks`	No inline calculator annotations are detected.
`unsupported_exec`	A calculator span cannot be executed by the current runtime.

Table F.4: TRACE compiler status labels.

Operation	Supported syntax
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Integer division	`//`
Modulo	`%`
Exponentiation	`**`, normalized from `^`
Functions	`sqrt`, `abs`, `round`, `max`, `min`

Table F.5: Arithmetic operations supported by the calculator runtime.

Hyperparameter	Value
Base model	`Falcon-H1R-7B`
Training method	LoRA SFT
LoRA rank	8
LoRA alpha	16
LoRA dropout	0.05
Target modules	`q_proj`, `k_proj`, `v_proj`, `o_proj`
Train batch size per device	1
Eval batch size per device	1
Gradient accumulation steps	16
Learning rate	2.0×10^{-5}
Epochs	2
Warmup ratio	0.03
Scheduler	cosine
Weight decay	0.01
Maximum sequence length	256
Precision	fp16
Activation checkpointing	`true`

Table F.6: TRACE supervised fine-tuning configuration.

Hyperparameter	Value
Initial policy	TRACE checkpoint
Base model	`Falcon-H1R-7B`
Quantized loading	4-bit enabled
Train batch size per device	1
Gradient accumulation steps	4
Learning rate	5.0×10^{-6}
Maximum RL steps	120
Maximum prompt length	384 tokens
Maximum completion length	256 tokens
Generations per prompt	4
Sampling temperature	0.8
KL coefficient β	0.0
Precision	fp16
Gradient checkpointing	`true`
Loss type	`dr_grpo`
Reward scaling	none
Aggregation	`sum_then_normalize`
Mask truncated completions	`true`

Table F.7: HiGRA RL configuration.

Reward family	Weight
Outcome correctness	2.0
Tool grounding	1.0
Process discipline	0.5
Trace alignment	0.25

Table F.8: HiGRA reward-family weights.

Coefficient	Value
Tool-use presence α	0.2
Observation presence α	0.2
Observation consistency α	0.6
Spurious tool-use penalty β	0.2
Alignment gate threshold η	0.6
Format score γ	0.4
Verification γ	0.2
Shortness γ	0.2
Final-answer γ	0.2
Premature-answer penalty γ	0.5
Schema alignment δ	0.4
Operator alignment δ	0.4
Verification alignment δ	0.2
Length penalty λ	0.25
Preferred reasoning actions	8

Table F.9: HiGRA reward coefficients used in the implementation.

Inference setting	Value
Maximum executable steps	12
Single-action generation limit	96 new tokens
Executable-loop decoding	greedy
Active GSM-8K tool	`calc`
Direct generation length	512 new tokens
Direct-generation decoding	greedy
Sampling by default	disabled

Table F.10: ARCHER inference settings.

Benchmark	Role
GSM-8K	Main in-domain mathematical reasoning benchmark.
MATH	Out-of-domain mathematical reasoning.
SVAMP	Out-of-domain arithmetic word problems.
Omni-MATH	Broader mathematical reasoning.
AIME-2024	Contest-style mathematical reasoning.
MMLU math subset	Multiple-choice mathematical reasoning.
GPQA-Diamond	Hard scientific reasoning transfer.
MMLU-Pro	Harder general multiple-choice reasoning.
HLE	Broad high-difficulty evaluation.

Table F.11: Benchmarks used for ARCHER evaluation.

Answer type	Normalization
Numeric	Remove commas, normalize decimals, compare canonical numbers.
Math	Prefer boxed answer or final-answer phrase; otherwise extract the last number.
Multiple choice	Compare answer text, choice index, and letter aliases.
Text QA	Lowercase, strip punctuation and articles, normalize whitespace.

Table F.12: Answer normalization rules.

Step	CoA math verification operation
1	Extract bracketed equations.
2	Parse each equation as `lhs = yN`.
3	Convert placeholders `y1`, `y2`, ... into symbolic variables.
4	Solve equations left-to-right or as a symbolic system.
5	Substitute solved placeholder values into the trace.
6	Extract the final answer from the reified trace.
7	Keep the sample only if the final answer matches the gold answer.

Table F.13: CoA math trace verification procedure used for reproduction.

Dataset	Count	Role
GSM-8K test	1,319	In-domain math.
SVAMP	1,000	Out-of-domain math.
ASDiv test	2,305	Single-step math.
MAWPS	2,065	Out-of-domain math.
MAWPS AddSub	395	Subset.
MAWPS SingleEq	508	Subset.
MAWPS SingleOp	562	Subset.
MAWPS MultiArith	600	Subset.

Table F.14: CoA math reproduction benchmark counts from the original setup.

Randomness. The default random seed is 42.