AGENT-GUIDED PROCESS VERIFICATION FOR MATHEMATICAL SELF-CORRECTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models have made substantial progress in mathematical reasoning, yet in complex multi-step reasoning, an error at any step may propagate along an incorrect path, and the resulting solution may omit necessary branches. Existing natural-language reflection methods often struggle to locate critical errors precisely, while outcome verification typically checks only whether the available candidate answers satisfy the problem conditions and can therefore overlook necessary branches. To address these limitations, we propose Agent-Guided Process Verification (AGPV) for mathematical self-correction. AGPV decomposes the current solution into an ordered sequence of key claims, verifies them individually while invoking a shared mathematical toolbox when needed, and stops upon detecting the first error. A summarization agent converts the failure evidence into a diagnosis that guides a correction agent to re-derive the subsequent solution from that point; AGPV then adapts the next verification iteration to the revised solution. AGPV also verifies solution completeness. We further prove that AGPV guarantees solution correctness and completeness when key claims provide sufficient coverage and local verification is reliable, provided that all key claims pass verification. The implementation compresses long working contexts and reuses successful tool results within each problem. We evaluate AGPV on five challenging mathematical reasoning datasets with four models from DeepSeek, Qwen, and GPT. AGPV achieves an average accuracy of 63.8%, outperforming the strongest among the representative baselines evaluated (50.3%) by 13.5 percentage points, with higher repair coverage than the programmatic-verification baselines.

1 INTRODUCTION

Large language models have made substantial progress in mathematical reasoning (Wei et al., 2022; Lewkowycz et al., 2022; Guo et al., 2025). However, in complex multi-step mathematical problems, an error at one step may cause the subsequent reasoning to proceed along an incorrect path and ultimately produce a wrong answer (Uesato et al., 2022; Lightman et al., 2024). Effective error checking and self-correction are therefore essential to improving the reliability of mathematical reasoning in large language models. Existing approaches primarily include natural-language reflection, iterative refinement, structured checking, and program- or tool-augmented verification (Pan et al., 2024; Gou et al., 2024a). For example, Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023) use natural-language feedback to support reflection and iterative refinement, while CheckList (Cook et al., 2024) organizes evaluation and correction through generated checklists. ProgCo (Song et al., 2025), by contrast, employs a programmatic verification mechanism to perform outcome verification from candidate answers.

These approaches nevertheless have two limitations. First, holistic natural-language reflection has difficulty locating errors precisely. Even when it identifies several suspicious points, it may fail to find the first critical error that actually invalidates the subsequent reasoning, leading to unfocused, wholesale rewriting (Huang et al., 2024; Tyen et al., 2024). Second, outcome verification starting from candidate answers typically focuses on whether the available results satisfy the problem conditions (Uesato et al., 2022; Lightman et al., 2024). However, it does not explicitly represent whether all valid solutions and necessary branches have been covered. It may therefore miss incompleteness errors such as incomplete solution sets or omitted branches. As shown in Fig. 1, the problem asks for all real roots of an equation. Substituting the available results $\{2, -1\}$ into the original equation

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

verifies both roots, yet the solution set remains incomplete. In contrast, checking the key claim "whether all solutions to $x^2 = 4$ have been found" along the solution process reveals the omission and precisely localizes the error.

The first critical error in mathematical reasoning resembles the first falling domino: subsequent errors often emerge through a cascading effect. It is therefore important to localize the first critical error in the solution process promptly and begin correction from that point. Tyen et al. show that providing an error location can facilitate correction (Tyen et al., 2024). Motivated by this, AGPV constructs and sequentially verifies key claims along the solution process to localize the first claim that compromises the correctness or completeness of subsequent reasoning. Step-level process verification provides fine-grained signals over intermediate reasoning (Lightman et al., 2024; Wang et al., 2024). AGPV uses this granularity to check whether necessary branches are complete as the reasoning unfolds, thereby verifying completeness within the same process.

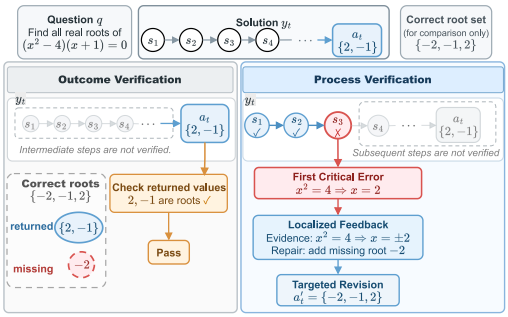

Figure 1: Comparison between outcome verification and AGPV process verification on an incomplete-root-set example. Here, $s_1, s_2, \ldots$ denote the ordered solution steps in the current solution y_t.

Accordingly, we propose AGPV. Here, process verification means converting the current solution into an ordered sequence of locally verifiable key claims and checking them in their original reasoning order. We call the method agent-guided because specialized agents dynamically construct and verify solution-specific key claims, summarize the first failed claim together with supporting evidence, and use this diagnosis to guide localized correction of the subsequent reasoning. Within the AGPV framework, a generation agent first parses the current solution and constructs an ordered sequence of key claims. A verification agent then checks these claims sequentially and, when necessary, invokes a shared mathematical toolbox to obtain mathematical evidence such as equation-solving and constraint-checking results. Upon encountering the first key claim that fails verification, the framework stops searching and initiates correction. A summarization agent then generates feedback from the failure evidence, guiding a correction agent to re-derive the subsequent solution from that point. Based on whether the answers before and after correction are equivalent, the framework regenerates or adapts the key claims for the revised solution and repeats this process until all key claims pass verification or the iteration limit is reached. We further prove that, under ideal conditions in which the key claims provide sufficient coverage and local verification is reliable, AGPV guarantees the correctness and completeness of the final solution, provided that all key claims pass verification.

We evaluate AGPV on five challenging mathematical reasoning datasets with four models from DeepSeek, Qwen, and GPT. AGPV achieves an average accuracy of 63.8%, outperforming the strongest evaluated baseline (50.3%) by 13.5 percentage points. Correction analysis shows that it repairs more initially incorrect answers, with particularly pronounced gains on problems that the initial models solve less reliably.

Our contributions are as follows: (1) We propose AGPV, which uses process verification to localize the first key claim that fails verification and guides the model to re-derive the subsequent solution from that point; (2) AGPV naturally incorporates solution-completeness checking into process verification without requiring a separate completeness-checking stage, enabling it to identify incompleteness errors. We further provide a corresponding conditional reliability proof; and (3) We develop a multi-agent implementation of AGPV with a shared mathematical toolbox, iterative key-claim adaptation, working-context compression, and within-problem reuse of successful tool results, and validate its effectiveness across multiple mathematical datasets and model settings.

2 Related Work

This work relates to three research areas: self-correction in large language models, process verification, and tool-augmented mathematical reasoning.

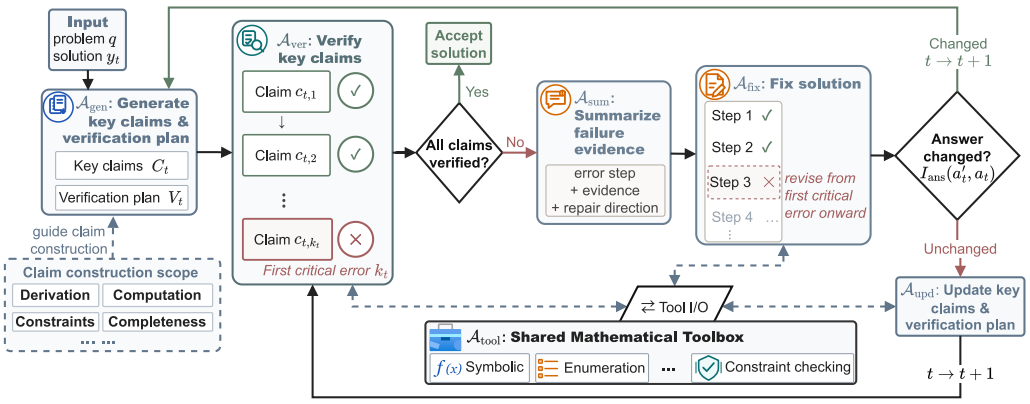

Figure 2: The AGPV workflow: sequential key-claim verification, first-error localization, failure-evidence summarization, solution correction, and verification-plan regeneration or updating.

Self-Correction in Large Language Models. Prior work improves test-time outputs through natural-language feedback, reflection, structured checking, and multi-path sampling. Self-Refine (Madaan et al., 2023) iteratively generates feedback and revises its responses; Reflexion (Shinn et al., 2023) uses linguistic feedback to summarize prior failures; TICK (Cook et al., 2024) organizes evaluation and correction with automatically generated checklists; and Self-Consistency (Wang et al., 2023) selects a consistent answer across multiple reasoning paths. Recent studies further improve self-verification and self-correction through training. S^2R (Ma et al., 2025) uses reinforcement learning to train models for self-verification and correction, while SPOC (Zhao et al., 2025) interleaves solution generation and verification in a single inference pass to trigger spontaneous correction. Collectively, these studies show that feedback and test-time iteration can improve large language models' reasoning outcomes.

Process Verification. Verifiers commonly evaluate, rank, or filter candidate solutions. Early mathematical reasoning work used outcome verifiers (Cobbe et al., 2021) to select more reliable outputs from multiple candidates. Uesato et al. (Uesato et al., 2022) compared outcome-based and process-based feedback for mathematical reasoning. Lightman et al. (Lightman et al., 2024) introduced PRM800K with step-level annotations and used process supervision to train process reward models. Math-Shepherd (Wang et al., 2024) automated process-supervision data construction, assigned rewards to intermediate reasoning steps, and applied them to candidate ranking and reinforcement learning. For test-time self-correction, ProgCo (Song et al., 2025) applies program-guided verification to candidate solutions and generates corrective feedback from execution results. Together, these studies extend verification from final outcomes to intermediate reasoning and provide a foundation for identifying process errors.

Tool-Augmented Mathematical Reasoning. Tool-augmented reasoning combines programs, code interpreters, and specialized mathematical tools to improve the reliability of language models' symbolic derivations and exact computation. PAL (Gao et al., 2023) translates problems into programs executed by a Python interpreter; Program-of-Thoughts (Chen et al., 2023) separates computation from natural-language reasoning; and ToRA (Gou et al., 2024b) alternates between natural-language reasoning and external tool calls to solve complex mathematical problems. ReAct (Yao et al., 2023) coordinates reasoning with external actions, while Toolformer (Schick et al., 2023) enables language models to select and invoke tools autonomously. Recent work such as AgentMath (Luo et al., 2026) and R1-Code-Interpreter (Liu et al., 2026b) further integrates code interpreters and mathematical tools with agent training for complex mathematical reasoning.

Overall, prior work improves mathematical reasoning through test-time self-correction, process supervision, and tool augmentation. AGPV builds on these directions by constructing ordered key claims from the current solution and problem objective, locating the first failed claim, checking completeness claim by claim, and using tool evidence to focus correction on the subsequent solution.

3 METHOD

Figure 2 presents the overall AGPV framework. Given a problem q, the current solution y_t at iteration t, and its extracted answer a_t, AGPV generates an ordered sequence of locally verifiable key claims and checks them in an order aligned with the original reasoning process. If all claims pass, the system accepts the current answer. Otherwise, it identifies the first failed claim as the first critical error, summarizes the failure evidence, and corrects the subsequent solution from that point. After correction, the answer-equivalence controller compares the answers before and after correction to decide whether the next iteration should regenerate or update the key-claim sequence and verification plan. Key-claim verification, solution correction, and updates to the key claims and verification plan may invoke the shared mathematical toolbox for exact computation. The following subsections describe key-claim generation and verification, solution correction, and verification-plan updates. Algorithm 1 in Appendix A provides the complete AGPV control procedure.

3.1 GENERATING KEY CLAIMS

AGPV verifies a sequence of key claims jointly determined by the current solution and the problem objective. Given problem q, current solution y_t, and extracted answer a_t at iteration t, $\mathcal{A}_{\text{gen}}$ generates the key-claim sequence and its verification plan:

$$(C_t, V_t) = \mathcal{A}_{\text{gen}}(q, y_t, a_t), \qquad C_t = (c_{t,1}, c_{t,2}, \ldots, c_{t,n_t}). \tag{1}$$

Here, the ordered key-claim sequence C_t specifies what to verify, and V_t specifies how to verify it along the current solution. The term $c_{t,i}$ is the i-th verifiable mathematical claim at iteration t, and n_t is the number of key claims in that iteration. These claims may come from explicit reasoning steps or be induced by the problem objective, including algebraic simplification, equation transformation, numerical computation, constraint derivation, and completeness requirements.

Key claims are distinct from solution steps. Solution steps are the original reasoning units that appear in their written order in the current solution y_t, whereas a key claim $c_{t,i}$ is an independently verifiable mathematical proposition extracted or constructed from one or more solution steps and the problem objective. Their mapping need not be one-to-one: one step may yield multiple claims, several steps may support one claim, and the problem objective may induce completeness claims that are not stated explicitly in the current solution. AGPV orders key claims by the original solution process, while V_t specifies their verification. The verification plan V_t specifies the order in which the key claims are verified, their local verification objectives, the tool types that may be required, and the evidence to retain during verification. AGPV decomposes the current solution into key claims ordered by the original reasoning process, enabling subsequent verification to localize errors to specific reasoning stages rather than issuing a single judgment on the final answer.

AGPV naturally incorporates solution-completeness checks into process verification. For problems requiring all solutions, complete counting, exhaustive enumeration, a complete set, or a global optimum, it converts each requirement into a key claim in C_t. These requirements thus enter the same process-verification chain as local computations, constraint derivations, and intermediate conclusions.

3.2 VERIFYING KEY CLAIMS

AGPV verifies the current solution claim by claim and invokes the shared mathematical toolbox agent when exact computation is required, producing traceable tool evidence. Given the key-claim sequence C_t and verification plan V_t from the preceding subsection, $\mathcal{A}_{\text{ver}}$ returns

$$(r_t, R_t, k_t) = \mathcal{A}_{\text{ver}}(q, y_t, a_t, C_t, V_t). \tag{2}$$

Here, $r_t \in \{0, 1\}$ indicates whether verification succeeds at iteration t; R_t stores the examined key claims, corresponding tool evidence, and pass/fail decisions; and k_t is the position of the first failed claim, with $k_t = \varnothing$ when all claims pass. Under the ideal conditions that the key claims sufficiently cover the current solution process, preserve the original reasoning order, and are verified reliably, the first failed claim corresponds to the first critical error. We use this term below. Specifically, $\mathcal{A}_{\text{ver}}$ checks the claims in C_t in the order specified by V_t. For each claim, $\mathcal{A}_{\text{ver}}$ determines whether the problem conditions and previously verified steps support it. $\mathcal{A}_{\text{ver}}$ directly returns pass or fail

for simple logical or expression-consistency checks. For algebraic simplification, equation solving, enumeration, counting, constraint checking, or exact numerical computation, $\mathcal{A}_{\text{ver}}$ sends a structured request to the shared mathematical toolbox agent and uses the result as verification evidence.

Shared Mathematical Toolbox Agent. Let $\mathbb{T}$ denote the available tools for symbolic simplification, equation solving, enumeration and counting, constraint checking, and exact computation. When reliable computation is needed during key-claim verification, solution correction, or updates to the key claims and verification plan, the corresponding agent generates a structured request τ_t^s, where $s \in \{\text{ver}, \text{fix}, \text{upd}\}$. $\mathcal{A}_{\text{tool}}$ selects a tool and generates an executable call:

$$\kappa_t^s = \mathcal{A}_{\text{tool}}(\tau_t^s; \mathbb{T}), \qquad z_t^s = \text{Run}(\kappa_t^s). \tag{3}$$

Here, κ_t^s is the generated tool call, Run executes it on the tool backend, and z_t^s is the returned result. Equation (3) may be repeated as needed within a stage. Verification results enter R_t; correction results support new computations or derivations; and update results help inspect the updated verification targets. $\mathcal{A}_{\text{sum}}$ does not call the toolbox directly but summarizes the tool evidence already stored in R_t. Our implementation supports SymPy (Meurer et al., 2017), Z3 (de Moura & Bjørner, 2008), and Python tools for exact numerical computation and enumeration.

If all key claims pass verification, $r_t = 1$ and $k_t = \varnothing$, and the system accepts the current solution and extracted answer. Otherwise, AGPV identifies the first failed claim as the first critical error:

$$k_t = \min\{i \mid c_{t,i} \text{ fails verification}\}, \qquad r_t = 0. \tag{4}$$

Verification stops immediately, and R_t retains the verification trace from the first claim through the first failed claim. Stopping at the first critical error is necessary because later reasoning may already depend on an erroneous premise, and continuing to inspect subsequent claims can obscure the actual source of the error.

3.3 Correcting the Solution and Updating the Verification Plan

When $r_t = 0$, AGPV enters feedback-based correction, involving $\mathcal{A}_{\text{sum}}$, $\mathcal{A}_{\text{fix}}$, $\mathcal{A}_{\text{upd}}$, and an answer-equivalence controller. The agents respectively summarize failure evidence into feedback, correct the solution, and update the key-claim sequence and verification plan when the answers are equivalent. Based on answer equivalence, the controller directs $\mathcal{A}_{\text{gen}}$ to regenerate the key claims and verification plan for the next iteration or $\mathcal{A}_{\text{upd}}$ to update them.

First, $\mathcal{A}_{\text{sum}}$ uses the current verification plan V_t, the verification trace R_t, and the position k_t of the first failed claim to summarize the failure evidence; $\mathcal{A}_{\text{fix}}$ then corrects y_t using F_t:

$$F_t = \mathcal{A}_{\text{sum}}(q, y_t, a_t, V_t, R_t, k_t), \tag{5}$$

$$y_t' = \mathcal{A}_{\text{fix}}(q, y_t, a_t, F_t). \tag{6}$$

Here, F_t is natural-language feedback identifying the first critical error, the relevant tool evidence in R_t, why the key claim fails, and where correction should begin. Because the evidence is read directly from R_t, no separate execution-record variable is required. Here, y_t' is the corrected candidate solution, and a_t' is its extracted answer. Guided by the error location in F_t, $\mathcal{A}_{\text{fix}}$ re-derives the subsequent solution from the first critical error rather than regenerating the entire solution. When recomputation, constraint checking, or verification of a new derivation is required, it may invoke $\mathcal{A}_{\text{tool}}$ and use the returned result.

After correction, the answer-equivalence controller compares a_t' and a_t, where $I_{\text{ans}}(x, y) = 1$ denotes equivalence and $I_{\text{ans}}(x, y) = 0$ non-equivalence. If $I_{\text{ans}}(a_t', a_t) = 0$, the changed answer may make C_t and V_t inapplicable. The system sets $y_{t+1} = y_t'$ and $a_{t+1} = a_t'$, clears C_t, V_t, and invokes $\mathcal{A}_{\text{gen}}$ in the next iteration to regenerate C_{t+1}, V_{t+1} from q, y_{t+1}, a_{t+1}.

If $I_{\text{ans}}(a_t', a_t) = 1$, the answers are equivalent, but the revised solution may no longer align with C_t, V_t. Direct reuse risks inconsistency, whereas full regeneration may discard the previous failure diagnosis. Therefore, $\mathcal{A}_{\text{upd}}$ updates the key-claim sequence and verification plan using $y_t', a_t', C_t, V_t, R_t, F_t$. If V_t contains missing or invalid tool tasks or conflicts with the corrected solution, $\mathcal{A}_{\text{upd}}$ may invoke $\mathcal{A}_{\text{tool}}$ before producing:

$$(C_{t+1}, V_{t+1}) = \mathcal{A}_{\text{upd}}(q, y_t', a_t', C_t, V_t, R_t, F_t). \tag{7}$$

The implementation compares extracted answer strings; differently formatted but mathematically equivalent answers may therefore trigger regeneration rather than adaptation. In both branches, $y_{t+1} = y'_t$ and $a_{t+1} = a'_t$, so, if another iteration remains, it verifies the corrected solution. When the iteration limit is reached, AGPV returns the latest corrected candidate without further verification. In the equivalent-answer branch, $\mathcal{A}_{\mathrm{upd}}$ uses the evidence in R_t and the diagnosis in F_t to update C_{t+1} and V_{t+1}, keeping them consistent with y'_t.

Working-Context Compression and Tool-Result Reuse. The implementation maintains a working context for continued verification and correction. When it reaches the configured token threshold, a compression agent summarizes older working content while retaining the fixed instructions and recent interaction context. The compression prompt requests preservation of key computations, tool evidence, failed approaches, and unfinished checks; the full execution records remain available separately. Before feedback summarization and verification-plan updates, long execution traces can likewise be replaced by compact working summaries. For repeated mathematical-tool requests within the same problem, the executor reuses an earlier successful, nonempty result for the same request text (after trimming surrounding whitespace), avoiding repeated code generation and Python execution. Failed or empty results are not cached, and the cache is not shared across problems. These mechanisms support the verification workflow; their individual effects are not isolated by the experiments reported here.

Conditional Reliability. The result assumes sufficient coverage of the correctness and completeness requirements and reliable verification of each claim against the problem conditions and previously accepted claims. Intuitively, the argument follows mathematical induction over the ordered claims. The first claim follows from the problem conditions. If the first $i-1$ claims have been established, reliable local verification shows that the i-th claim also follows from the problem conditions and those preceding claims. Repeating this step establishes all claims, and sufficient coverage then implies final-answer correctness and completeness. Let P_q denote the problem conditions and $C_t = (c_{t,1}, \ldots, c_{t,n_t})$ the ordered key-claim sequence. Suppose all key claims pass reliable local verification; that is, for every $i \in \{1, \ldots, n_t\}$,

$$P_q \wedge \bigwedge_{j=1}^{i-1} c_{t,j} \models c_{t,i}, \tag{8}$$

where $\models$ denotes semantic entailment. For $i = 1$, Eq. (8) yields $P_q \models c_{t,1}$; induction over the ordered claims then gives

$$P_q \models \bigwedge_{i=1}^{n_t} c_{t,i}. \tag{9}$$

Further suppose the problem objective contains completeness requirements $h_1, \ldots, h_m$, and every h_ℓ is semantically entailed by some key claim $c_{t,j(\ell)}$. Equation (9) and transitivity then yield

$$P_q \models \bigwedge_{\ell=1}^{m} h_\ell. \tag{10}$$

Thus, if the key claims sufficiently cover final-answer correctness and completeness and local verification is reliable, a single process-verification chain supports all these requirements. Complete definitions and proofs are provided in Appendix B.1.

4 Experiments

Our experiments address three questions: (1) whether AGPV improves final-answer accuracy; (2) whether its gains primarily arise from correcting initially incorrect answers rather than from indiscriminate rewriting; and (3) whether its performance remains stable across backbone models and settings with different initial capabilities.

4.1 Experimental Setup

Datasets We evaluate the methods on five hard-test mathematical reasoning datasets comprising 170 problems: AIME 2024, AIME 2025, BRUMO 2025, and HMMT February 2025 from Math-

Method	DeepSeek V4.1-Flash	DeepSeek V4-Pro (0813)	Qwen3.6 35B-A3B-FP8	GPT-5.6 Luna	Mean
Initial	52.9	38.8	53.5	23.5	42.2
Reflex	60.0 $_{+7.1}$	37.6 $_{-1.2}$	54.7 $_{+1.2}$	31.2 $_{+7.6}$	45.9 $_{+3.7}$
Self-Refine	61.8 $_{+8.8}$	51.2 $_{+12.4}$	57.6 $_{+4.1}$	30.6 $_{+7.1}$	50.3 $_{+8.1}$
Self-Reflection	55.3 $_{+2.4}$	40.0 $_{+1.2}$	57.1 $_{+3.5}$	32.9 $_{+9.4}$	46.3 $_{+4.1}$
CheckList	55.3 $_{+2.4}$	41.2 $_{+2.4}$	57.1 $_{+3.5}$	30.0 $_{+6.5}$	45.9 $_{+3.7}$
CoT-Tool	54.1 $_{+1.2}$	40.0 $_{+1.2}$	41.8 $_{-11.8}$	22.4 $_{-1.2}$	39.6 $_{-2.6}$
ProgCo	58.2 $_{+5.3}$	41.2 $_{+2.4}$	57.1 $_{+3.5}$	25.9 $_{+2.4}$	45.6 $_{+3.4}$
ProgCo-Py	62.4 $_{+9.4}$	48.2 $_{+9.4}$	58.8 $_{+5.3}$	28.2 $_{+4.7}$	49.4 $_{+7.2}$
AGPV	**72.4** $_{+19.4}$	**66.5** $_{+27.6}$	**64.1** $_{+10.6}$	**52.4** $_{+28.8}$	**63.8** $_{+21.6}$

Table 1: Final-answer accuracy (%). Model scores pool 170 problems across five datasets; Mean averages four models (680 model–problem instances). Subscripts give changes from Initial in percentage points, computed from unrounded counts (green/red: increases/decreases). Bold marks column bests; per-dataset results appear in the appendix.

Arena (Balunovic et al., 2025), together with AMO Bench (Liu et al., 2026a). These datasets cover algebra, combinatorics, number theory, geometry, and integrated reasoning in competition mathematics. Dataset sizes and basic characteristics are provided in Table 5. **Backbone Models.** The experiments use DeepSeek-V4.1-Flash and DeepSeek-V4-Pro (0813) (DeepSeek, 2026), Qwen3.6-35B-A3B-FP8 (Qwen Team, 2026), and GPT-5.6-Luna (OpenAI, 2026). Short model names follow the retained experiment records; the complete mapping is given in Table 3.

Baselines We compare AGPV with three representative classes of baselines. Initial Score denotes the backbone model's initial response without additional correction. The first class comprises natural-language correction methods: Reflex, which uses natural-language reflection; Self-Refine (Madaan et al., 2023), which uses iterative self-feedback; Self-Reflection, which uses reflective feedback; and CheckList (Cook et al., 2024), which performs checklist-guided correction. The second class is the reasoning-enhanced method CoT-Tool (Abedi Firouzjaei, 2025), which samples multiple chain-of-thought paths (Wei et al., 2022) and selects an answer using Self-Consistency (Wang et al., 2023). The third class comprises the verification-enhanced methods ProgCo (Song et al., 2025) and ProgCo-Py. ProgCo uses program-driven verification and correction, whereas ProgCo-Py enables Python-tool feedback based on the public ProgCo implementation. AGPV instead performs process verification along the key-claim sequence of the current solution and uses the first critical error and tool evidence to guide subsequent correction.

Experimental Protocol All self-correction methods use $\texttt{max_turn} = 3$ and follow their respective stopping rules. This setting does not imply matched model-call or token budgets. All self-correction methods start from the same pregenerated responses. We analyze intermediate answers retrospectively from these runs rather than from independent $\texttt{max_turn} = 1$ experiments. Empty outputs or outputs without a valid extractable answer are incorrect; when execution falls back to a saved answer, that returned answer is evaluated. Accuracy always uses the original fixed benchmark denominator. **Implementation Parameters.** The generation temperature is set to 0.0 for all methods. Following its own mechanism, CoT-Tool independently generates three chain-of-thought paths and determines its output by the majority among the extracted answers; its budget is therefore not interpreted as correction turns. Complete model identifiers, reasoning modes, method-level budgets, tool-execution limits, and answer-judgment rules are provided in Appendix C.1. **Evaluation Metrics.** Final-answer accuracy is our primary metric. Each model's overall score is computed over all 170 problems, weighting datasets by their sample counts. Mean averages these four model scores, equivalently pooling 680 model–problem instances, not 680 distinct questions. For correction analysis, C and W denote correct and incorrect answers. Thus, $W \to C$ denotes an initially incorrect answer becoming correct and $C \to W$ denotes an initially correct answer becoming incorrect. We additionally report the average number of recorded states as a description of the saved processes, not a measure of total inference cost.

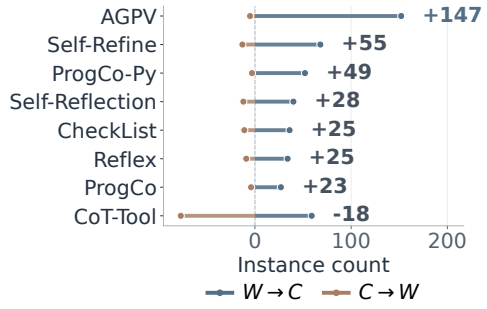

Figure 3: Initial-to-final repair and damage counts. Blue/right: $W \to C$ repairs; orange/left: $C \to W$ damage. Labels and descending order use net repairs, $N_{W\to C} - N_{C\to W}$.

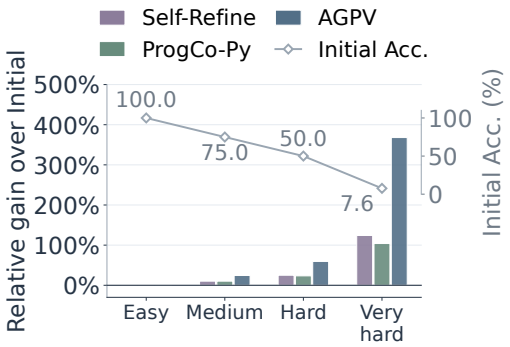

Figure 4: Relative accuracy gains (bars, left axis) and initial accuracy (gray line, right axis) across empirical-difficulty groups.

4.2 Main Results

AGPV leads all evaluated baselines under each of the four models (Table 1). Mean accuracy reaches 63.8%, exceeding Self-Refine, the strongest evaluated baseline, by 13.5 percentage points. Gains span models with different initial capabilities, indicating that the benefit is not restricted to one backbone. Additional robustness analyses are provided in Appendix C.3.

Per-Dataset Results. Tables 6 and 7 retain the complete breakdown over the five datasets. Overall improvements are consistent across the four models, although AGPV does not lead every individual dataset. We next examine the sources and distribution of these gains.

4.3 Correction-Behavior Analysis

To identify the source of AGPV's gains, we further measure each method's transitions between initial and final correctness. The main text focuses on $W \to C$ repairs and $C \to W$ damage; complete state transitions and derived statistics are reported in Table 9. Fig. 3 shows that AGPV's gains primarily come from broader error repair: it corrects 152 initially incorrect answers while damaging 5 initially correct answers. Repair benefits substantially outweigh these regressions, producing the largest net improvement among the evaluated methods.

Table 2 compares the disjoint overlap of repairs by ProgCo, ProgCo-Py, and AGPV. Each combination identifies methods that repair the same initially incorrect instances. AGPV alone repairs 111 instances left unrepaired by both baselines, showing that its advantage extends beyond their shared repair coverage.

4.4 Performance by Difficulty

To examine whether AGPV's gains concentrate in particular difficulty ranges, we estimate empirical difficulty from initial-answer correctness across multiple backbone-model settings. Specifically, a problem is considered empirically more difficult when fewer of the four models answer it correctly at initialization. We label problems solved initially by all four models as Easy, by three models as Medium, by two models as Hard, and by zero or one model as Very hard. These groups contain 30, 26, 32, and 82 distinct problems, respectively.

Fig. 4 compares Self-Refine, ProgCo-Py, and AGPV across these groups. Relative gain is defined as $(\mathrm{Acc}_{\mathrm{Final}} - \mathrm{Acc}_{\mathrm{Initial}})/\mathrm{Acc}_{\mathrm{Initial}}$. Because low initial accuracy magnifies this ratio, the figure also shows Initial Acc.

AGPV exceeds both baselines on Medium, Hard, and Very hard problems, with an absolute gain of 28.0 percentage points over Initial in the Very hard group. Its advantage therefore concentrates in problems that the initial models solve less reliably. Easy problems exhibit a small regression under AGPV, while both baselines preserve their initial accuracy.

Repairing method(s)	Share (%)
Disjoint repair groups	
AGPV only	62.7
AGPV and ProgCo-Py	15.3
AGPV and ProgCo	2.8
All three methods	5.1
ProgCo-Py only	6.8
ProgCo only	5.1
ProgCo and ProgCo-Py	2.3
Total coverage (overlapping)	
ProgCo	15.3
ProgCo-Py	29.4
AGPV	85.9

Table 2: Repair coverage (%) among 177 model–problem instances repaired by at least one method. Upper groups are exclusive: "and" means only the named, independently run methods repair an instance, not joint execution. Lower totals overlap. Percentages are rounded.

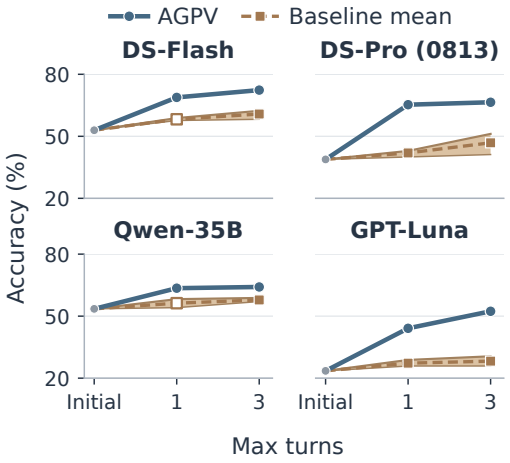

Figure 5: Accuracy at reconstructed stopping points. The dashed line and shaded band show the mean and min–max range of ProgCo, ProgCo-Py, and Self-Refine, respectively. Hollow markers indicate means containing imputed values.

4.5 Effect of Iteration Turns

AGPV verifies and corrects the current solution through feedback. To examine how accuracy changes during this process, Fig. 5 compares AGPV with ProgCo, ProgCo-Py, and Self-Refine across all four models. We reconstruct stopping-point answers for limits of one and three turns using each method's stopping rule, retaining an earlier terminal answer after stopping. Missing intermediate scores are completed with the same answer-evaluation procedure, without regenerating responses; unavailable intermediate answers use the recorded fallback. Reconstructed stopping-point accuracies are provided in Table 12.

Across all four models, AGPV exceeds the best of the three baselines at both reported post-initial checkpoints, not only their mean. Most gains are already present at the first reported stopping point, with further improvements by the final point. Table 9 describes recorded-state counts; neither these counts nor saved checkpoints imply equal computation budgets. Appendix C.5 reports the observed reduction in request-context length when compression is triggered, rather than an overall token-efficiency comparison.

Computational Cost. We report recorded input and output token costs for tool-code generation and context compression in Appendix C.5. This accounts for observed token expenditure in these components, separately from request-length reductions. Full per-call usage is unavailable across AGPV, ProgCo-Py, and Self-Refine; therefore, the accuracy gains do not establish equal-budget or lower-cost performance.

5 Conclusion

AGPV verifies ordered key claims, locates the first failed claim, and uses failure evidence to revise subsequent reasoning and update the verification plan. Embedding completeness requirements in these claims unifies correctness and completeness checks; both are conditionally guaranteed with sufficient claim coverage, reliable local verification, and all claims passing. Experiments demonstrate improved mathematical-reasoning accuracy, error correction, and repair coverage. Future work will extend process verification to other domains with traceable intermediate processes, including code generation, scientific reasoning, and data analysis, using domain knowledge and specialized tools to construct well-scoped claims verifiable with reliable evidence.

AI USE STATEMENT

Generative AI tools were used for language polishing, literature retrieval, discussions of research design, drafting portions of the experimental descriptions, and assistance with code and figure preparation. The core research ideas were developed by the authors, who reviewed the final content and take responsibility for the paper.

REFERENCES

Hassan Abedi Firouzjaei. Cogitator: A python toolkit for chain-of-thought prompting, 2025. URL https://github.com/habedi/cogitator.

Mislav Balunovic, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating LLMs on uncontaminated math competitions. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/1d27c01ebd3e3aebe226b44fc970d803-Abstract-Datasets_and_Benchmarks_Track.html.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023. URL https://openreview.net/forum?id=YfZ4ZPt8zd.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Jonathan Cook, Tim Rocktäschel, Jakob Foerster, Dennis Aumiller, and Alex Wang. TICKing all the boxes: Generated checklists improve LLM evaluation and generation. In *NeurIPS 2024 Workshop on Language Gamification*, 2024. URL https://neurips.cc/virtual/2024/106582.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24.

DeepSeek. DeepSeek API Documentation. https://api-docs.deepseek.com/, 2026. Accessed: 2026-09-25.

Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pp. 19–32, 1967.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10764–10799. PMLR, 2023. URL https://proceedings.mlr.press/v202/gao23f.html.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, 2024a. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/fef126561bbf9d4467dbb8d27334b8fe-Abstract-Conference.html.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu Chen. ToRA: A tool-integrated reasoning agent for mathematical problem solving. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=Ep0TtjVoap.

Daya Guo et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645:633–638, 2025. doi: 10.1038/s41586-025-09422-z. URL https://doi.org/10.1038/s41586-025-09422-z.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12 (10):576–580, 1969. doi: 10.1145/363235.363259.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/8b4add8b0aa8749d80a34ca5d941c355-Abstract-Conference.html.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/18abbeef8cfe9203fdf9053c9c4fe191-Abstract-Conference.html.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=v8L0pN6EOi.

Junlin Liu, Shengnan An, Shuang Zhou, Dan Ma, Yehao Lin, Xinxuan Lv, Xuanlin Wang, Xiaoyu Li, Ziwen Wang, Xuezhi Cao, and Xunliang Cai. AMO-bench: Large language models still struggle in high school math competitions. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 2120–2137, San Diego, California, United States, July 2026a. Association for Computational Linguistics. doi: 10.18653/v1/2026.findings-acl.101. URL https://aclanthology.org/2026.findings-acl.101/.

Yueying Liu, Yongchao Chen, Junwei Zhou, Yilun Hao, Jingquan Wang, Yang Zhang, Na Li, and Chuchu Fan. R1-code-interpreter: LLMs reason with code via supervised and multi-stage reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=FNlNH0iFOx.

Haipeng Luo, Huawen Feng, Qingfeng Sun, Can Xu, Kai Zheng, Yufei Wang, Tao Yang, Han Hu, and Yansong Tang. Agentmath: Empowering mathematical reasoning for large language models via tool-augmented agent. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=e2s7YHeVZW.

Ruotian Ma, Peisong Wang, Cheng Liu, Xingyan Liu, Jiaqi Chen, Bang Zhang, Xin Zhou, Nan Du, and Jia Li. S^2R: Teaching LLMs to self-verify and self-correct via reinforcement learning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22632–22654, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1104. URL https://aclanthology.org/2025.acl-long.1104/.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Stepán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. SymPy: Symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017. doi: 10.7717/peerj-cs.103. URL https://peerj.com/articles/cs-103/.

Open Logic Project. Soundness of natural deduction. https://builds.openlogicproject.org/content/intuitionistic-logic/soundness-completeness/soundness-nd.pdf, n.d. Accessed: 2026-07-03.

OpenAI. GPT-5.6 luna model. https://developers.openai.com/api/docs/models/gpt-5.6-luna, 2026. Accessed: 2026-09-25.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. Automatically correcting large language models: Surveying the landscape of diverse automated correction strategies. *Transactions of the Association for Computational Linguistics*, 12:484–506, 2024. doi: 10.1162/tacl_a_00660. URL https://aclanthology.org/2024.tacl-1.27/.

Benjamin C. Pierce et al. Software foundations: Hoare logic, part i. https://softwarefoundations.cis.upenn.edu/plf-current/Hoare.html, n.d. Accessed: 2026-07-03.

Qwen Team. Qwen3.6-35B-A3B-FP8 model card. https://huggingface.co/Qwen/Qwen3.6-35B-A3B-FP8, 2026. Accessed: 2026-09-25.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, pp. 68539–68551, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Xiaoshuai Song, Yanan Wu, Weixun Wang, Jiaheng Liu, Wenbo Su, and Bo Zheng. ProgCo: Program helps self-correction of large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 944–959, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-short.73. URL https://aclanthology.org/2025.acl-short.73/.

Gladys Tyen, Hassan Mansoor, Victor Carbune, Peter Chen, and Tony Mak. LLMs cannot find reasoning errors, but can correct them given the error location. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 13894–13908, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.826. URL https://aclanthology.org/2024.findings-acl.826/.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback, 2022. URL https://arxiv.org/abs/2211.14275.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 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)*, pp. 9426–9439. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.510. URL https://aclanthology.org/2024.acl-long.510/.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Xutong Zhao, Tengyu Xu, Xuewei Wang, Zhengxing Chen, Di Jin, Liang Tan, Yen-Ting Lin, Zishun Yu, Zhuokai Zhao, Yun He, Sinong Wang, Han Fang, Sarath Chandar, and Chen Zhu. Boosting LLM reasoning via spontaneous self-correction. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=9DCQAGBoII.

Appendix

A Overall AGPV Algorithm

Algorithm 1 Overall AGPV Procedure

Require: Problem q, initial solution y_0, extracted answer a_0, maximum zero-based iteration index T

Ensure: Final solution y and answer a

Initialize $y \leftarrow y_0,\ a \leftarrow a_0,\ C \leftarrow \varnothing,\ V \leftarrow \varnothing$
for $t = 0$ to T **do**
  if $V = \varnothing$ **then**
    $(C, V) \leftarrow \mathcal{A}_{\text{gen}}(q, y, a)$ {Generate the key-claim sequence and its verification plan}
  end if
  $(r, R, k) \leftarrow \mathcal{A}_{\text{ver}}(q, y, a, C, V)$ {Verify key claims; invoke $\mathcal{A}_{\text{tool}}$ as needed; locate the first critical error}
  if $r = 1$ **then**

    return y, a {Verification passes; accept the current answer}
  end if
  $F \leftarrow \mathcal{A}_{\text{sum}}(q, y, a, V, R, k)$ {Summarize failure evidence}
  $y' \leftarrow \mathcal{A}_{\text{fix}}(q, y, a, F)$ {Correct from the first critical error; invoke $\mathcal{A}_{\text{tool}}$ as needed}
  $a' \leftarrow \text{Extract}(y')$ {Extract the corrected answer}
  if $y' = \varnothing \ \vee \ a' = \varnothing$ **then**

    return y, a {No valid correction generated; retain the current candidate}
  end if
  if $I_{\text{ans}}(a', a) = 0$ **then**
    $y \leftarrow y',\ a \leftarrow a',\ C \leftarrow \varnothing,\ V \leftarrow \varnothing$ {Answers differ; regenerate C, V next turn}
  else
    $(C, V) \leftarrow \mathcal{A}_{\text{upd}}(q, y', a', C, V, R, F)$ {Answers are equivalent; update C, V; invoke tools as needed}
    $y \leftarrow y',\ a \leftarrow a'$
  end if
end for

return y, a {Iteration limit reached; return the latest candidate without further verification}

B Theoretical Supplement

B.1 Conditional Reliability Proof for Process Verification

To formalize how completeness is embedded in process verification, we denote the completeness requirements in the problem objective by

$$\mathcal{H}(q, y_t) = \{h_1, h_2, \ldots, h_m\}, \tag{11}$$

Here, $\mathcal{H}(q, y_t)$ is the set of completeness requirements induced by the problem objective and instantiated under the current solution y_t; $\ell \in \{1, \ldots, m\}$ indexes these requirements, and m is their total number. Each h_ℓ denotes a completeness property or proposition required by the problem objective rather than the target object itself. For example, h_ℓ may state that "the target object given by the current solution equals the complete target object implied by the problem conditions." The key-claim sequence C_t covers these completeness requirements if, for every $h_\ell \in \mathcal{H}(q, y_t)$, there exists a key claim $c_{t,j(\ell)}$, where $j(\ell) \in \{1, \ldots, n_t\}$ is the position of the key claim covering the ℓ-th completeness requirement, such that the claim semantically entails the corresponding requirement:

$$c_{t,j(\ell)} \models h_\ell. \tag{12}$$

Here, $\models$ denotes semantic entailment, and $c_{t,j(\ell)} \models h_\ell$ states that the $j(\ell)$-th key claim semantically entails the corresponding completeness requirement. This definition embeds completeness into the process-verification chain when the key-claim sequence is constructed.

Example: Instantiating a Completeness Requirement.
For the problem $x^2 = 4$, suppose the current solution gives only $x = 2$. Checking only whether 2 satisfies the equation is insufficient. The problem objective requires the complete set of real solutions, and solving under the problem conditions yields the complete target object

$$S^* = \{x \in \mathbb{R} : x^2 = 4\} = \{-2, 2\}. \tag{13}$$

If S_y is the set returned by the current solution, the corresponding completeness requirement can be written as

$$h : \quad S_y = S^*. \tag{14}$$

AGPV constructs a key claim associated with this objective, such as whether the solution set returned by the current solution is the complete real solution set. The set returned by the current solution is

$$S_y = \{2\}. \tag{15}$$

Since

$$S_y \neq S^*, \tag{16}$$

the key claim fails verification, allowing AGPV to detect the missing solution.

Conditional-Reliability Intuition AGPV's process verification admits an idealized conditional-reliability interpretation. To simplify notation, we omit the turn subscript t below and denote the current solution by y. Let P_q be the problem conditions and let the key-claim sequence be

$$C = (c_1, c_2, \ldots, c_n). \tag{17}$$

where n is the total number of key claims after omitting the turn subscript. For $i \in \{1, \ldots, n\}$, define the verified context preceding the i-th claim as

$$\Gamma_{i-1} = P_q \wedge c_1 \wedge c_2 \wedge \cdots \wedge c_{i-1}, \tag{18}$$

where $\Gamma_0 = P_q$. Ideally, if the verifier is reliable, then whenever the i-th key claim passes verification,

$$\Gamma_{i-1} \models c_i. \tag{19}$$

Here, $\models$ denotes semantic entailment. The relation $\Gamma_{i-1} \models c_i$ means that the current key claim c_i holds in every interpretation satisfying the problem conditions and the preceding verified claims. Thus, the i-th key claim follows from the problem conditions and the preceding verified claims.

Proposition 1: Conditional Reliability of Process Verification Let P_q be the problem conditions and let $C = (c_1, \ldots, c_n)$ be the ordered key-claim sequence constructed by AGPV from the current solution and the problem objective. Suppose that all key claims pass verification and that the verifier is reliable; that is, for every i,

$$P_q \wedge c_1 \wedge \cdots \wedge c_{i-1} \models c_i, \tag{20}$$

where the set of preceding claims is empty when $i = 1$. Then

$$P_q \models c_1 \wedge c_2 \wedge \cdots \wedge c_n. \tag{21}$$

Proof We proceed by induction on i.

Base case: when $i = 1$, verifier reliability gives

$$P_q \models c_1. \tag{22}$$

Therefore, the proposition holds for $i = 1$.

Induction hypothesis: suppose that, for some $i - 1 \geq 1$,

$$P_q \models c_1 \wedge c_2 \wedge \cdots \wedge c_{i-1}. \tag{23}$$

That is, the first $i - 1$ key claims are all supported by the problem conditions P_q.

Inductive step: because the i-th key claim passes verification and the verifier is reliable,

$$P_q \wedge c_1 \wedge \cdots \wedge c_{i-1} \models c_i. \tag{24}$$

From the induction hypothesis and $P_q \models P_q$, conjunction introduction gives $P_q \models \Gamma_{i-1}$. Because $\Gamma_{i-1} \models c_i$, transitivity of semantic entailment yields $P_q \models c_i$. Applying conjunction introduction again to this conclusion and the induction hypothesis gives

$$P_q \models c_1 \wedge \cdots \wedge c_i. \tag{25}$$

Mathematical induction therefore establishes Eq. (21). This completes the proof.

Corollary 1: Completeness Is Covered by the Same Process-Verification Chain Suppose the construction of the key-claim sequence C covers the completeness requirements $\mathcal{H}(q, y) = \{h_1, \ldots, h_m\}$ associated with the current solution; that is, for every $h_\ell \in \mathcal{H}(q, y)$, there exists a key claim $c_{j(\ell)}$ such that

$$c_{j(\ell)} \models h_\ell, \tag{26}$$

Then, under the conditions of Proposition 1,

$$P_q \models h_1 \wedge h_2 \wedge \cdots \wedge h_m. \tag{27}$$

Proof Proposition 1 gives $P_q \models c_1 \wedge \cdots \wedge c_n$, and hence $P_q \models c_{j(\ell)}$ for every $j(\ell)$. Combining this result with Eq. (26), transitivity of semantic entailment yields $P_q \models h_\ell$. This conclusion holds for every $h_\ell \in \mathcal{H}(q, y)$; conjunction introduction then gives Eq. (27). This completes the proof.

The proposition and corollary clarify the meaning of embedded completeness in AGPV. When a problem requires all solutions, complete counting, exhaustive enumeration, or a global optimum, these completeness requirements enter the same process-verification chain through the construction of the key-claim sequence. If the key claims provide complete coverage and the corresponding verification is reliable, the completeness conclusions are also supported by the problem conditions.

This conditional-reliability intuition is consistent with soundness theorems in formal logic: if a conclusion follows from a set of premises through sound inference rules, then the conclusion is a semantic consequence of those premises. Soundness proofs for natural deduction commonly proceed by induction over derivations (Open Logic Project, n.d.). The same idea is also related to Floyd–Hoare-style program verification, in which intermediate assertions organize local checks and composition rules support global properties (Floyd, 1967; Hoare, 1969; Pierce et al., n.d.).

AGPV is not a formal proof checker. The proposition and corollary characterize how process verification can embed completeness requirements under the ideal conditions that the key claims provide complete coverage, their order matches the original reasoning process, and local verification is reliable. Under these conditions, the first key claim that fails verification corresponds to the first critical error in the current solution. In practice, performance may still be affected by incomplete key-claim extraction, completeness requirements omitted from the key-claim sequence, incorrectly constructed verification targets, or tool-execution failures.

C Supplementary Experimental Results

This appendix provides additional information for the experiments in the main paper, including experimental settings and reproduction protocols, a benchmark overview, complete model results, supplementary correction-behavior analyses, and descriptive statistics of recorded states. The main paper reports the primary results most relevant to the central narrative, the core correction-behavior analysis, and the difficulty-conditioned analysis; this appendix documents the full configurations, complete numerical results, complete model-setting results, and additional statistical analyses.

C.1 Experimental Settings and Reproduction Protocol

To ensure that results are comparable and reproducible across methods and model settings, this section reports the experimental configuration from four perspectives: model and decoding configurations, method budgets, AGPV tool-execution limits, and unified answer evaluation. Unless

otherwise specified, all methods under the same model setting use the same model interface and reasoning mode, with the generation temperature set to 0.0. Provider-specific parameters that are not uniformly overridden by the experimental scripts retain the configurations of their respective API wrappers; interface parameters with different semantics are not forcibly aligned.

Model and Decoding Configurations Table 3 maps the four model settings to their API identifiers or model checkpoint. All four use the non-thinking wrapper configuration. The label 0813 identifies the retained DeepSeek-Pro experiment tag; it is not an additional model setting. The same model-specific initial responses are reused for every correction method.

Model setting	API identifier / checkpoint	Reasoning configuration
DeepSeek-V4.1-Flash	`deepseek-flash`	Thinking disabled
DeepSeek-V4-Pro (0813)	`deepseek-v4-pro`	Thinking disabled
Qwen3.6-35B-A3B-FP8	`Qwen/Qwen3.6-35B-A3B-FP8`	`enable_thinking=False`
GPT-5.6-Luna	`gpt-5.6-luna`	`reasoning_effort=none`

Table 3: Model identifiers and decoding configurations for the retained four-model experiments.

Method Budgets and Fair Comparison Table 4 summarizes the initial-response source and execution budget for each method. ProgCo, ProgCo-Py, Reflex, Self-Refine, Self-Reflection, CheckList, and AGPV start from the same pregenerated responses. CoT-Tool instead generates three independent chain-of-thought paths. All retained correction runs use $\texttt{max_turn} = 3$ under method-specific stopping rules. The shared setting does not imply identical model calls, token usage, or runtime. For example, Self-Refine generates feedback and a revision within one turn, whereas CheckList generates a checklist before correction.

The iteration analysis uses saved intermediate answers from these retained runs; it does not compare independent $\texttt{max_turn} \in \{1, 2, 3\}$ experiments or perform a parameter sweep. Generation temperature is fixed at 0 when exposed by the API. Other method-specific settings are retained rather than forcibly aligned across different execution mechanisms.

Method	Initial response	Main-experiment budget and execution mechanism
ProgCo	Shared	Up to three correction turns; program-driven verification and correction.
ProgCo-Py	Shared	Up to three correction turns; Python-tool feedback enabled within the ProgCo workflow.
Reflex	Shared	Up to three correction turns; one reflection-and-rewrite step per turn.
Self-Refine	Shared	Up to three correction turns; feedback is generated before each feedback-guided revision.
Self-Reflection	Shared	Up to three correction turns; one self-reflective correction per turn.
CheckList	Shared	One checklist is generated first, followed by up to three checklist-guided correction turns.
CoT-Tool	Independently generated	Independently generates three chain-of-thought paths and selects the output by majority vote over the extracted answers.
AGPV	Shared	$\texttt{max_turn} = 3$; locates the first failed key claim, generates feedback, and regenerates or adapts the verification plan after correction.

Table 4: Main-experiment budgets and execution mechanisms for all methods.

For reproducibility, every correction method receives the same stored initial response and candidate answer under a given model–problem setting. Reported accuracies use one retained run per model–method–problem combination. Existing answers and final-answer scores are preserved; only missing intermediate-answer judgments are completed for the retrospective trajectory analysis.

AGPV Tool Execution and Runtime Stability AGPV's verification, correction, and verification-plan update agents can invoke the shared mathematical toolbox dynamically. The backends support symbolic simplification, equation solving, enumeration, constraint checking, and exact or numerical computation. The four retained AGPV runs set a language-model-call timeout of 960 seconds and a Python execution timeout of 120 seconds; ordinary and stopping-decision calls have retry limits of 10 and 20, respectively. A soft warning and hard stop are configured at 100 and 120 generated Python snippets per sample. The working-context compression trigger is 131,072 tokens, subject to the runtime's model-capacity checks. Successful nonempty results are cached by problem identifier and trimmed tool-request text; cache hits return the previous evidence instead of generating and executing Python again. These are implementation settings, not an ablation or a measured token-saving result. Failed executions and their returned fallback answers remain in the fixed-denominator evaluation.

Unified Answer Evaluation All methods use the same three-stage answer-evaluation pipeline: normalized exact matching, directly decidable numerical comparison, and a model-based equivalence judgment when needed. Parsed numerical answers are considered equivalent when their absolute error is below 10^{-3}. Expressions, sets, and other unresolved answer forms are judged with the shared DeepSeek-Flash fallback at temperature 0.0. Set-valued answers must contain all required elements, and order is checked when the problem requires an ordered answer. Empty or unextractable answers are incorrect. If execution returns an earlier saved answer after a failure, we score that actual returned answer; no sample is removed. The fixed denominators are 30 each for AIME 2024, AIME 2025, BRUMO 2025, and HMMT February 2025, and 50 for AMO Bench, totaling 170 distinct problems per model.

Code and Data Availability Code and data are provided in the anonymous supplementary material.

C.2 Benchmark Overview

Table 5 summarizes the five hard-test mathematical reasoning datasets used in this work. AIME 2024, AIME 2025, BRUMO 2025, and HMMT February 2025 are included in MathArena (Balunovic et al., 2025), with 30 problems in each dataset; AMO Bench (Liu et al., 2026a) contains 50 problems. Together, the five datasets comprise 170 problems spanning algebra, number theory, combinatorics, geometry, and integrated reasoning in competition mathematics.

Dataset	Short	#Problems	Notes
AIME 2024	A24	30	competition math; integer answers
AIME 2025	A25	30	competition math; integer answers
BRUMO 2025	B25	30	hard contest-style reasoning
HMMT Feb. 2025	HMM	30	contest math; mixed topics
AMO Bench	AMO	50	advanced math reasoning; mixed answer forms

Table 5: Overview of the five hard-test mathematical reasoning datasets.

C.3 Complete Results

Tables 6 and 7 give the full five-dataset breakdown for the same four models as the main table. All scores use fixed denominators, including empty outputs and recorded failure fallbacks. AGPV has the highest overall accuracy for each model, but it is not the best method on every individual dataset; for example, Self-Refine is stronger on HMMT under DeepSeek-V4.1-Flash.

	DeepSeek-V4.1-Flash						DeepSeek-V4-Pro (0813)					
Method	A24	A25	B25	HMM	AMO	Avg	A24	A25	B25	HMM	AMO	Avg
Initial	73.3	73.3	86.7	53.3	8.0	52.9	63.3	46.7	66.7	26.7	10.0	38.8
Reflex	83.3	80.0	90.0	60.0	16.0	60.0 +7.1	63.3	43.3	63.3	30.0	8.0	37.6 -1.2
Self-Refine	**86.7**	83.3	83.3	**70.0**	16.0	61.8 +8.8	70.0	60.0	86.7	56.7	10.0	51.2 +12.4
Self-Reflection	80.0	70.0	90.0	60.0	8.0	55.3 +2.4	**76.7**	46.7	66.7	26.7	6.0	40.0 +1.2
CheckList	76.7	76.7	86.7	53.3	12.0	55.3 +2.4	66.7	46.7	66.7	33.3	12.0	41.2 +2.4
CoT-Tool	83.3	73.3	76.7	56.7	10.0	54.1 +1.2	66.7	46.7	66.7	30.0	10.0	40.0 +1.2
ProgCo	**86.7**	73.3	86.7	60.0	14.0	58.2 +5.3	70.0	50.0	70.0	26.7	10.0	41.2 +2.4
ProgCo-Py	83.3	76.7	90.0	63.3	24.0	62.4 +9.4	70.0	56.7	83.3	43.3	12.0	48.2 +9.4
AGPV	83.3	**96.7**	**96.7**	60.0	**44.0**	**72.4** +19.4	73.3	**80.0**	**93.3**	**76.7**	**32.0**	**66.5** +27.6

Table 6: Per-dataset accuracy (%) for DeepSeek-V4.1-Flash and DeepSeek-V4-Pro (0813). Avg is sample-weighted over 170 problems. Changes beside Avg are percentage-point differences from the shared Initial score, calculated from unrounded counts. Bold marks the best result in each column, including ties.

	Qwen3.6-35B-A3B-FP8						GPT-5.6-Luna					
Method	A24	A25	B25	HMM	AMO	Avg	A24	A25	B25	HMM	AMO	Avg
Initial	73.3	60.0	73.3	56.7	24.0	53.5	40.0	36.7	40.0	16.7	0.0	23.5
Reflex	70.0	60.0	73.3	66.7	24.0	54.7 +1.2	56.7	40.0	56.7	20.0	2.0	31.2 +7.6
Self-Refine	**80.0**	66.7	73.3	63.3	26.0	57.6 +4.1	53.3	43.3	46.7	23.3	4.0	30.6 +7.1
Self-Reflection	73.3	60.0	76.7	63.3	30.0	57.1 +3.5	56.7	40.0	53.3	30.0	4.0	32.9 +9.4
CheckList	**80.0**	63.3	73.3	63.3	26.0	57.1 +3.5	53.3	40.0	43.3	23.3	6.0	30.0 +6.5
CoT-Tool	60.0	56.7	56.7	43.3	12.0	41.8 -11.8	50.0	26.7	33.3	16.7	0.0	22.4 -1.2
ProgCo	**80.0**	70.0	73.3	60.0	24.0	57.1 +3.5	53.3	36.7	43.3	13.3	0.0	25.9 +2.4
ProgCo-Py	**80.0**	**73.3**	76.7	60.0	26.0	58.8 +5.3	46.7	40.0	46.7	23.3	2.0	28.2 +4.7
AGPV	**80.0**	70.0	**80.0**	**70.0**	**38.0**	**64.1** +10.6	**66.7**	**66.7**	**80.0**	**50.0**	**20.0**	**52.4** +28.8

Table 7: Per-dataset accuracy (%) for Qwen3.6-35B-A3B-FP8 and GPT-5.6-Luna. Avg is sample-weighted over 170 problems. Subscripts show percentage-point changes from the shared Initial score, calculated from unrounded counts. Bold marks the best result in each column, including ties.

Robustness Analysis Using the retained runs, we compute 95% paired bootstrap confidence intervals with 50,000 resamples stratified by dataset, resampling problems jointly across the four models. We also recompute accuracy gains after excluding each dataset in turn.

Baseline	Mean gain	95% CI	Minimum gain after excluding one dataset
Self-Refine	+13.5	[9.9, 17.4]	+11.0
ProgCo-Py	+14.4	[10.9, 18.1]	+13.1
ProgCo	+18.2	[14.6, 22.1]	+16.9

Table 8: Accuracy gains of AGPV over three baselines, in percentage points. Intervals quantify problem-sampling uncertainty conditional on the retained runs; minimum gains are taken over the five leave-one-dataset-out evaluations.

AGPV retains a positive average advantage over all three baselines after excluding any single dataset.

C.4 Supplementary Correction-Behavior Analysis

Table 9 summarizes the complete correction-state transitions and associated repair, damage, final-accuracy, and recorded-state statistics for all methods.

State transitions (%)

Method	$C \to C$	$C \to W$	$W \to C$	$W \to W$
Reflex	40.9	1.3	5.0	52.8
Self-Refine	40.3	1.9	10.0	47.8
Self-Reflection	40.4	1.8	5.9	51.9
CheckList	40.6	1.6	5.3	52.5
CoT-Tool	30.9	11.3	8.7	49.1
ProgCo	41.6	0.6	4.0	53.8
ProgCo-Py	41.8	0.4	7.6	50.1
AGPV	41.5	0.7	22.4	35.4

Repair, damage, and outcome statistics

Method	Repair (%)	Damage (%)	Net gain (pp)	Final Acc. (%)	Avg States
Reflex	8.7	3.1	+3.7	45.9	3.9
Self-Refine	17.3	4.5	+8.1	50.3	3.8
Self-Reflection	10.2	4.2	+4.1	46.3	3.9
CheckList	9.2	3.8	+3.7	45.9	3.9
CoT-Tool	15.0	26.8	-2.6	39.6	1.0
ProgCo	6.9	1.4	+3.4	45.6	1.8
ProgCo-Py	13.2	1.0	+7.2	49.4	1.9
AGPV	38.7	1.7	+21.6	63.8	1.9

Table 9: Correction-state transitions and repair/damage statistics. State transitions are percentages of 680 model–problem instances; C and W denote correct and incorrect answers, and $X \to Y$ denotes the initial-to-final transition. Repair and Damage use initially incorrect and initially correct samples as denominators, respectively. Net gain is final minus initial accuracy in percentage points (pp), computed before rounding. Avg States is the mean number of saved `turns` entries, not a computation budget.

AGPV achieves substantially broader repair coverage than ProgCo and ProgCo-Py, which accounts for most of its net gain despite a slightly higher damage count. For CoT-Tool, transitions compare its independently generated final answer with the shared Initial reference; they do not describe an iterative repair procedure.

Recorded-state conventions differ by method. Natural-language baselines generally include the initial state and subsequent revisions; verification methods log attempted verification states, and a final corrected answer need not create another verified-state entry. CoT-Tool records an aggregate result rather than one state per generation path. These counts do not normalize model calls, tokens, tool calls, or runtime and therefore do not establish an efficiency advantage.

C.5 Computational Cost and Context Compression

Recorded AGPV Component Token Costs Table 10 reports backend-returned token usage for AGPV's tool-code generation and successful context-compression calls. Values are averaged over all 170 problems per model, including problems that do not trigger either component. Tool-code generation is counted once from the timing records, without adding the duplicate entries stored in tool-execution records; compression usage is counted once per recorded successful event.

	Tool-code generation		Context compression	
Backbone	Input	Output	Input	Output
DS-Flash	8.08	0.96	0.00	0.00
DS-Pro (0813)	21.54	2.31	15.50	0.08
Qwen-35B	3.33	0.39	3.54	0.01
GPT-Luna	6.27	0.59	0.00	0.00

Table 10: Recorded AGPV token-cost components, in thousands of tokens per problem. Input and output use the respective backends' reported token counts. Zeros for Flash and Luna compression mean that no compression was triggered. These are component costs, not total inference costs.

This accounting excludes other verification, correction, plan-generation, summarization, and answer-extraction calls, shared initial-answer generation, evaluation, and any unrecorded attempts. The retained ProgCo-Py and Self-Refine outputs do not provide matching token-usage records; their missing costs are not treated as zero. Thus, the table quantifies observable AGPV overhead but does not support a total-cost ranking against these baselines. Compression itself consumes tokens, so the request-length reductions below cannot be interpreted as net token savings.

Python Execution Requests Table 11 separately compares Python execution requests in the retained AGPV and ProgCo-Py runs. These counts measure tool usage, not total inference cost.

Method	DS-Flash	DS-Pro (0813)	Qwen-35B	GPT-Luna
AGPV	4.065	11.035	1.494	3.206
ProgCo-Py	39.788	10.471	3.124	0.006

Table 11: Mean Python execution requests per problem, averaged over all 170 problems per model. Requests resulting in execution errors or timeouts are included; startup checks and cache hits are excluded.

Request-Context Compression Figure 6 compares request-context lengths immediately before and after compression over all recorded compression events in the retained runs. The DeepSeek-Pro (0813) run records 22 events across 6 problems, and the Qwen run records 5 events across 5 problems. DeepSeek-Flash and GPT-Luna are not shown because compression was not triggered in their retained runs. Repeated events on the same problem are included. Bars show event-level mean lengths, and the reduction ratio is the total length reduction divided by the total pre-compression length.

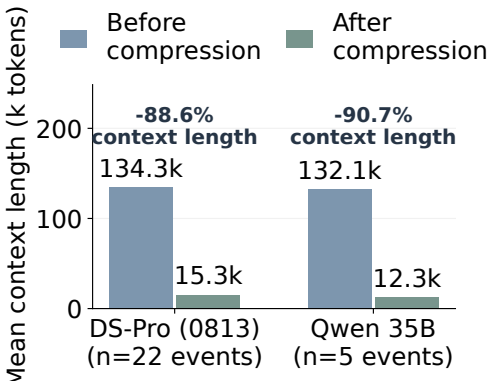

Figure 6: Request-context lengths before and after compression, measured in thousands of tokens. All recorded compression events are included; n denotes events, not distinct problems. These reductions measure request length, not total inference-token savings.

Compression reduces aggregate request-context length by 88.6% for DeepSeek-Pro (0813) and 90.7% for Qwen, leaving more compact inputs for continued verification. These observations characterize the executed compression operations, not an accuracy ablation or a total-cost comparison with ProgCo-Py or Self-Refine. Complete per-call token usage is not available across these methods, and compression itself incurs model-call cost; therefore, these context reductions should not be interpreted as net inference-token savings.

C.6 SAVED-CHECKPOINT ACCURACY

Model	Method	Initial	max_turn = 1	max_turn = 2	max_turn = 3
DS-Flash	AGPV	52.9	68.8	70.0	72.4
DS-Flash	ProgCo	52.9	57.6	57.6	58.2
DS-Flash	ProgCo-Py	52.9	58.8	62.4	62.4
DS-Flash	Self-Refine	52.9	58.2	60.6	61.8
DS-Pro (0813)	AGPV	38.8	65.3	66.5	66.5
DS-Pro (0813)	ProgCo	38.8	40.0	41.2	41.2
DS-Pro (0813)	ProgCo-Py	38.8	42.9	45.9	48.2
DS-Pro (0813)	Self-Refine	38.8	42.9	45.9	51.2
Qwen-35B	AGPV	53.5	63.5	64.1	64.1
Qwen-35B	ProgCo	53.5	56.5	57.1	57.1
Qwen-35B	ProgCo-Py	53.5	58.2	58.8	58.8
Qwen-35B	Self-Refine	53.5	54.1	56.5	57.6
GPT-Luna	AGPV	23.5	44.1	50.0	52.4
GPT-Luna	ProgCo	23.5	25.9	25.3	25.9
GPT-Luna	ProgCo-Py	23.5	27.1	28.2	28.2
GPT-Luna	Self-Refine	23.5	28.8	30.6	30.6

Table 12: Accuracy (%) on 170 problems per model at initialization and reconstructed stopping points (max_turn=1, 2, 3).

C.7 COMPARATIVE CASE ANALYSIS

Table 13 presents four illustrative successful corrections from the retained four-model runs, with model names and sample identifiers stated explicitly. The table summarizes the saved diagnoses and answers; it is not a frequency estimate of error types or a formal proof of every generated derivation.

Dataset	Problem-objective summary	Initial error	ProgCo / ProgCo-Py behavior	AGPV key evidence and correction	Characteristic
HMMT February 2025	Count circular arrangements of $1, \ldots, 7$ with adjacent products at most 20; rotations and reflections are distinct.	DeepSeek-Pro (0813), hmmt_feb_2025_10: the initial answer is 28, not 56. After fixing 7, the counting argument wrongly excludes arrangements where 6 is adjacent to 1.	Both methods retain 28.	Enumeration returns 56 valid ordered arrangements and confirms that 7's neighbors are $1, 2$. Feedback identifies the incomplete count; the corrected argument includes the omitted placements and returns 56.	Counting-coverage diagnosis and correction.
AIME 2024	Compute a product over the thirteenth roots of unity and return its remainder modulo 1000.	GPT-Luna, aime_2024_19: the initial answer is 65, not 321. The factorization is valid, but the sign of $(1+i)^{13}$ is wrong.	Both methods retain 65.	Verification identifies $(1+i)^{13} = -64 - 64i$, rather than $64 + 64i$. Recomputing the conjugate product gives 8321, hence remainder 321.	Local arithmetic-error diagnosis.
AIME 2025	Find the expected number of regions formed by two perpendicular diameters and 25 random cross-quadrant chords.	DeepSeek-Flash, aime_2025_12: the initial answer is $487/3$, not 204. It assumes random chords intersect with probability $1/3$, ignoring the quadrant-conditioned sampling.	Both methods retain $487/3$.	The saved feedback gives the conditioned probability $17/36$; the correction trace checks it numerically and recomputes the expected intersections and regions, returning 204.	Intermediate-probability correction.
AMO Bench	Find all possible remainders modulo 32 of $\lfloor n!/((n+1)(n+2)) \rfloor$.	DeepSeek-Pro (0813), amo_bench_6: the initial set is $\{0, 2, 4, 6, 12, 18\}$, omitting valid values.	ProgCo returns a set containing an extra 21; ProgCo-Py returns an incomplete seven-element set.	The saved verification enumerates $1 \le n \le 1000$ and finds missing even values. AGPV returns the reference-matching set $\{0, 2, 4, \ldots, 30\}$, explicitly enumerated in the saved answer.	Missing-value detection from finite witnesses.

Table 13: Illustrative corrections from the current four-model runs. Problem identifiers follow the stored benchmark files. In the AMO case, finite enumeration supplies witnesses of omitted values; it is not by itself a proof of exhaustiveness for all n.

CASE STUDY: CORRECTING AN ERROR CAUSED BY INCOMPLETE COUNTING COVERAGE

We examine the DeepSeek-V4-Pro (0813) run on `hmmt_feb_2025_10`. The following is a condensed account of its saved verification, feedback, and correction trace, not a verbatim transcript.

Problem and Initial Solution.
Problem. Compute the number of ways to arrange $1, 2, 3, 4, 5, 6, 7$ around a circle such that every adjacent product is at most 20. Rotations and reflections count as different arrangements.
Answer Status. The initial extracted answer is 28; the reference answer is 56. ProgCo and ProgCo-Py both return 28 on this model–problem instance.
Initial Reasoning. The initial solution correctly observes that 7 can be adjacent only to $1, 2$, and 6 only to $1, 2, 3$. It fixes 7, considers the two orientations of its neighbors, and counts only two arrangements per orientation. The error is the assertion that 6 cannot be adjacent to 1. This excludes feasible placements and leads to $7 \cdot (2 + 2) = 28$.
Generated Verification Plan. The saved plan constructs the complete set of arrangements satisfying the adjacent-product condition, computes its cardinality N, and checks the structural claim that the neighbors of 7 are exactly $1, 2$. Verification requires both the structural claim and agreement between N and the candidate count.

First Verification Cycle: Checking the Count.
Tool Task. The verifier requests the complete count and the structural check together. Ordered permutations $p = (p_1, \ldots, p_7)$ are retained exactly when

$$p_i p_{i+1} \le 20 \quad (i = 1, \ldots, 6), \qquad p_7 p_1 \le 20.$$

Counting ordered permutations respects the problem's rule that rotations and reflections are distinct. The computation can be summarized as:

```
V = permutations satisfying every adjacent-product bound
N = len(V)
Claim7 = every sequence in V has neighbors(7) = {1, 2}
return (N, Claim7)
```

Returned Evidence and Diagnosis. The saved Python result is $\{N : 56,\ \mathrm{Claim7} : \mathrm{True}\}$. Thus, the structural constraint is supported but the candidate count 28 fails. Verification returns False. The feedback retains the correct adjacency observation and directs correction to recompute the count after fixing 7, rather than treating the incomplete set as exhaustive.

Feedback-Guided Correction and Reverification.
The correction stage enumerates the arrangements with 7 fixed, obtains eight possibilities, and identifies the omitted placements where 6 is adjacent to 1. Its revised argument is as follows.
For the orientation

$$7 - 1 - a - b - c - d - 2 - 7,$$

the four remaining values are $3, 4, 5, 6$. The value 6 can occupy either endpoint a or d. If $a = 6$, then $b = 3$, with $4, 5$ in either order. If $d = 6$, then $c = 3$, again with $4, 5$ in either order. The four valid arrangements are

$$(7, 1, 6, 3, 4, 5, 2), \quad (7, 1, 6, 3, 5, 4, 2),$$
$$(7, 1, 4, 5, 3, 6, 2), \quad (7, 1, 5, 4, 3, 6, 2).$$

The other orientation of $1, 2$ around 7 contributes four further arrangements. Restoring the seven rotational positions gives

$$7 \cdot (4 + 4) = 56.$$

No division by two is made because reflections are distinct. The first cycle therefore revises the answer from 28 to 56; the next cycle verifies the revised answer and terminates successfully.
What This Case Demonstrates. The trace illustrates using a complete-target computation to reject an incomplete count, retaining valid structural evidence, and directing the subsequent correction to the missing cases. It also shows that the verifier's diagnosis and the correction agent's more detailed explanation are distinct parts of the process.