LEARNING WHAT TO EXECUTE AND WHEN: REASONING PROTOCOLS FOR LLM INFERENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Inference-time methods that extend a single language-model call use hand-designed *procedures*: sample and vote, re-solve with a hint, critique, write and run a program, or verify and revise. Learned multi-agent topologies change only who talks to whom, but under one evaluation protocol they only match hand-designed graphs. We introduce REPROT to learn what to execute and when. A *reasoning protocol* is a stateful program over typed reasoning operations. It specifies the *visibility* of earlier results, *composition*, and *trigger conditions* on runtime state that determine when each problem receives extra computation. The language represents six widely used procedure families. A synthesizer reads search-set execution traces and proposes *state-conditioned patches*, each comprising a diagnostic, a trigger condition and a sub-protocol with its visibility. Acceptance requires a net gain in problems fixed over problems broken on the full search set that passes a sign test, within a budget on mean calls. Selection pools search and disjoint validation data; the final protocol is fixed before held-out test evaluation. On eight benchmarks against seventeen baselines reproduced under one protocol and the original ADAS and AFlow, the learned protocols have the highest average accuracy, 84.2 at 3.9 calls per question. The top tier of hand-designed and learned-topology methods scores 79.7–80.3 at 5.2–11.0 calls, and AFlow scores 82.6. Paired comparisons favour REPROT over AFlow on four benchmarks and tie on four, with no losses. The largest margins over the seventeen reproduced baselines are on TheoremQA (83.5 vs. 77.7) and MuSR (79.1 vs. 64.2), where the learned change is an instruction in a protocol using one or three calls; AFlow scores 82.7 and 77.2 there. Free-code search with the same synthesizer finds a similar answer-format change on OlympiadBench. Condition overrides change cost by up to 2.6×. Forcing the first branch disables LiveCodeBench escalation and reduces accuracy by 9.7 points.

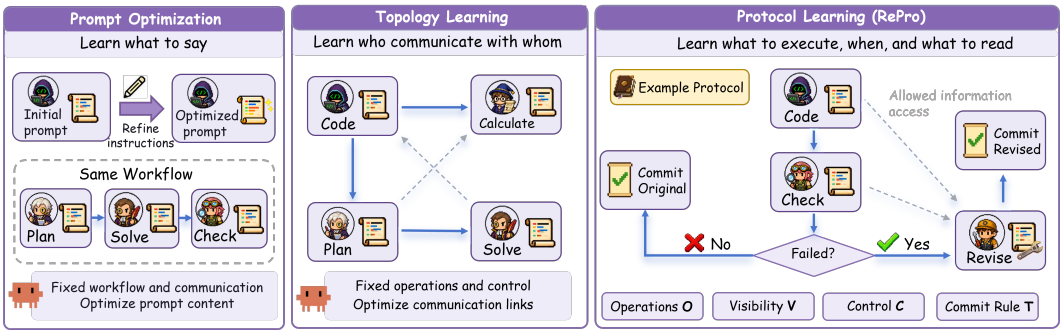

Figure 1: Three objects of learning. Prompt optimization learns what each step says within a fixed workflow. Topology learning learns who communicates with whom among agents that run fixed operations. A reasoning protocol $P = (O, V, C, T)$ learns what to execute, when to execute it and what each step may read.

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.

1 INTRODUCTION

Inference-time methods for large language models (LLMs) specify procedures that extend a single call. They sample several derivations and vote (Wang et al., 2022), feed a previous answer back as a hint (Zheng et al., 2023), let copies of the model critique one another (Du et al., 2023), write and run a program (Chen et al., 2022), or verify and revise a candidate (Madaan et al., 2023; Weng et al., 2023). Their designers manually choose which operations run, what each may see and when to spend extra computation. Multi-agent methods make the second choice learnable: GPTSwarm, G-Designer, AgentPrune, DyLAN and CARD learn *who talks to whom* among agents that run the same computation (Zhuge et al., 2024; Zhang et al., 2024; 2025b; Liu et al., 2024; Wu et al., 2026). In our reproduction of seventeen methods on eight benchmarks under one backbone and evaluation protocol (Table 1), the strongest learned topologies perform similarly to hand-designed graphs. The two strongest learned-topology systems average 80.3 and 80.1, compared with 80.2 for a complete graph and 79.8 for a random one. Different baselines lead on different benchmarks.

We learn the procedure itself (Figure 1). A *reasoning protocol* $P = (O, V, C, T)$ has typed reasoning operations O, a visibility map V specifying which earlier artifacts each operation may read, a control structure C (sequence, parallel, conditional, bounded repetition) and a commit rule T. Conditions depend only on runtime state, such as agreement, check outcomes, execution status and verifier confidence; they have no access to labels. The language includes instances of six prior procedure families. A role is a parameter of an operation, and a topology is a visibility pattern.

Prior work also learns inference programs. ADAS uses a meta agent to write agent code (Hu et al., 2025), AFlow searches operator graphs (Zhang et al., 2025c), and Archon searches configurations of inference-time techniques (Saad-Falcon et al., 2024). Their updates are unconditional: a rewritten program or an added node runs on every input. Accepted changes therefore incur new computation on all problems, and these designs rarely repay their cost (El et al., 2025). Candidate scores also hide which correct answers become incorrect. REPROT instead learns a *state-conditioned patch* $\delta = (\ell, d, g, Q, V_Q)$. At location ℓ, it runs a diagnostic d and executes sub-protocol Q with visibility V_Q when trigger condition g holds on the observable state. A synthesizer proposes patches from search-set execution traces. Each patch must pass a *sign-test gate* on the *full* search set: the excess of fixes over breaks must pass a sign test, and mean calls must stay within budget B.

Contributions. Our protocol representation enforces visibility and runtime-only triggers and includes instances of six prior procedure families (Section 3). State-conditioned synthesis jointly learns a trigger and sub-protocol, with a sign-test gate on net gain under budget and size constraints. Final selection pools search and disjoint validation data (Section 4). We evaluate eight benchmarks against seventeen baselines reproduced under one protocol, with five controls (Section 5). The learned protocols have the highest average, 84.2 at 3.9 calls, compared with 79.7–80.3 at 5.2–11.0 calls for a tier of nine baselines and 82.6 for the original AFlow. They have the highest mean on seven of eight benchmarks and no significant per-benchmark losses in paired comparisons. The controls identify large gains from learned instructions and a substantial trigger effect on LiveCodeBench. Free-code search finds a similar answer-format change.

2 RELATED WORK

Learned communication topologies (GPTSwarm (Zhuge et al., 2024), G-Designer (Zhang et al., 2024), AgentPrune (Zhang et al., 2025b), DyLAN (Liu et al., 2024), CARD (Wu et al., 2026), MacNet (Qian et al., 2025)) vary V with fixed O and C. REPROT varies all four components and conditions execution on the runtime state, without using the problem text as the trigger. Prompting, sampling and verification methods supply the operations and control patterns (Wei et al., 2022; Kojima et al., 2022; Wang et al., 2022; Zheng et al., 2023; Zhou et al., 2022; Chen et al., 2022; Yao et al., 2023; Madaan et al., 2023; Weng et al., 2023; Chen et al., 2024; Gou et al., 2024; Shinn et al., 2023; Du et al., 2023; Jiang et al., 2023). Adaptive test-time compute (Snell et al., 2024; Chen et al., 2023a; Brown et al., 2024) studies when additional computation pays off; REPROT also learns which computation to add. Automated design methods (ADAS (Hu et al., 2025), AFlow (Zhang et al., 2025c), Archon (Saad-Falcon et al., 2024), A^{2}Flow (Zhao et al., 2026), AgentSquare (Shang et al., 2025), MaAS (Zhang et al., 2025a), MASS (Zhou et al., 2026), MoA (Wang et al., 2025)) search agent code, operator graphs or modular spaces through execution. Prompt optimisers (Khattab et al.,

2023; Yuksekgonul et al., 2024; Yang et al., 2024) tune prompts within a fixed pipeline. REPROT also searches by execution. Its updates use typed programs with enforced visibility, couple a trigger to a sub-protocol, and pass a sign-test gate on fixed-minus-broken counts. We compare directly with ADAS-style free-code search (Section 5.4; extended discussion in Appendix A).

3 REASONING PROTOCOLS

3.1 STATE AND EXECUTION

Fix a backbone $\mathcal{M}$ and a problem q with reference answer $y^\star(q)$. A protocol runs on a *state* $s = (q, \mathcal{A}, b)$, where $\mathcal{A}$ maps identifiers to *artifacts*. Each artifact has a kind κ (solution, program, subproblems, checkspec, evidence, verdict, aggregate), full text x, extracted answer y, evidence summary e, outcome flag ok and confidence γ. The remaining call budget is b. An operation o sees the state only through its declared *reads* $V(o)$ (the problem, an artifact's full text, only its answer or evidence, or a group's answers), which the executor renders into its prompt. Executing P yields the committed answer $\hat{y}$, a trace z of every executed operation and branch decision, and the cost c; correctness is $R(P, q) = \mathbb{1}[\hat{y} \equiv y^\star(q)]$ under the benchmark's scorer (formal semantics in Appendix C.1).

3.2 PROTOCOLS AS PROGRAMS

A reasoning protocol $P = (O, V, C, T)$ consists of typed operations O that *create* a candidate (`solve`, `program`, `revise`), *structure* the problem (`decompose`, `build_check`), *evaluate* a candidate (`run_check`, `verify`) or *select* (`aggregate`, `select`) (Table 3); their reads V; a control program C; and a priority list T of identifiers from which the answer is committed. Control programs and trigger conditions are generated by

$$C ::= o \mid \mathrm{Seq}(C_1, \ldots, C_n) \mid \mathrm{Par}_\iota(C_1, \ldots, C_n) \mid \mathrm{If}(g, C_1, C_2) \mid \mathrm{Rep}_k(C, g) \mid \mathrm{Commit}(T), \tag{1}$$

$$g ::= \mathrm{agree}(I) \mid \mathrm{disagree}(I) \mid \mathrm{ndistinct}(I) \geq k \mid \mathrm{passed}(\iota) \mid \mathrm{failed}(\iota) \mid \mathrm{exec_ok}(\iota) \mid \mathrm{exec_err}(\iota) \mid \mathrm{empty}(\iota) \mid \gamma_\iota < \theta \mid \gamma_\iota \geq \theta \mid b \geq n \mid t < k \mid g \wedge g \mid g \vee g \mid \neg g \mid \top, \tag{2}$$

with $n \leq 6$ branches per Par, $k \leq 3$ iterations per Rep and at most $L = 24$ nodes. Every predicate is a function of the state alone; the grammar has no access to $y^\star$. Par runs all branches from the same snapshot, preventing reads of siblings' artifacts. $\mathrm{Commit}(T)$ commits the first listed artifact that exists and has an answer. If an identifier belongs to an untaken branch, execution proceeds to the next entry of T,

$$\hat{y} = y\big(\mathcal{A}[\iota_{j^\star}]\big), \qquad j^\star = \min\{\, j : \iota_j \in \mathrm{dom}\,\mathcal{A},\ y(\mathcal{A}[\iota_j]) \neq \varnothing \,\}; \tag{3}$$

this makes conditional sub-protocols composable. On code tasks the answer is a program, agreement compares programs by their outputs on the public examples, and `run_check`[exec] runs a candidate on those examples at no LLM cost (Appendix B).

3.3 EXPRESSIVENESS

Table 4 instantiates six prior procedure families. These simplified seeds differ from the reproduced baselines: CoT and SC use one-call solves, and PHP repeatedly hints with the initial answer. The language also represents three further families. A *role* is a persona parameter of `solve`. A *topology* over N agents consists of `solve` operations with a visibility pattern: agent i reads its in-neighbours' full text, and successive Par groups implement rounds. Learning the topology changes only V. A *tool* is an operation with an external executor (`program`, `run_check`[exec]). The language also expresses protocols absent from prior methods. The *PoT-check* seed, found in a pilot search on competition mathematics, writes a program and derives conditions that a correct answer must satisfy. It checks the program's answer against those conditions with an executed checker and revises only if the check fails. The trigger keeps its cost near three calls.

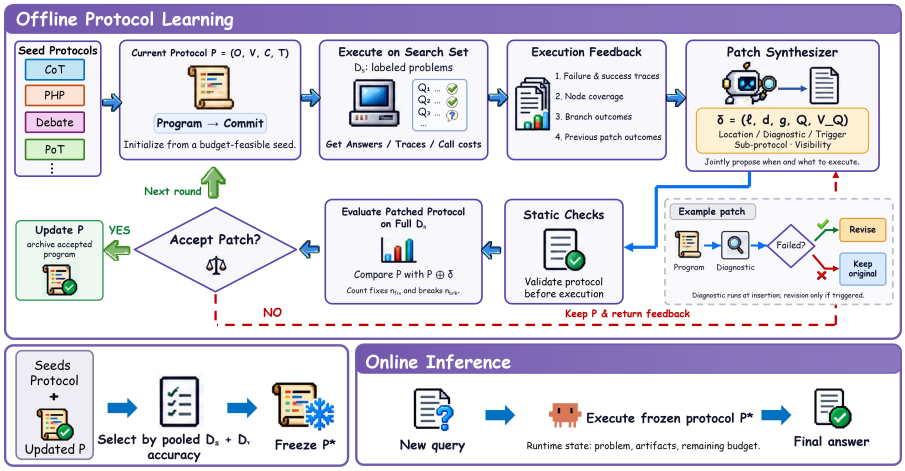

Figure 2: REPROT: execute on the search set, synthesize state-conditioned patches from traces and coverage, keep those that pass the sign-test gate under the budget, select on pooled search–validation accuracy, and freeze before held-out test evaluation.

4 STATE-CONDITIONED PROTOCOL SYNTHESIS

4.1 EXECUTION TRACES AS FAILURE EVIDENCE

Figure 2 shows the search loop. Executing the current protocol on the labelled search set $\mathcal{D}_s$ produces a digest for each problem,

$$\zeta(q) = \Big(q,\ y^\star(q),\ \hat{y},\ R(P,q),\ c,\ \big\langle(\iota_o,\ V(o),\ y_o,\ \mathrm{ok}_o,\ \gamma_o,\ e_o)\big\rangle_{o\in z_q},\ \big\langle(g,\ \mathrm{taken})\big\rangle_{\mathrm{If}\in z_q}\Big), \quad (4)$$

the only place labels enter the method. Labels identify failures; the trace records which operations ran, what they read and answered, and how their outputs were judged. The synthesizer sees the program with its search-set accuracy, mean cost and the budget, $n_f = 10$ random failure digests, $n_s = 4$ success digests marked "do not break these", the outcomes of the last twelve patches tried, and a *node-coverage* summary over the whole search set: for every operation o and condition g of P,

$$\kappa(o) = \big(|\{q : o \in z_q\}|,\ \mathrm{Acc}(P \mid o \in z_q)\big), \qquad \kappa(g) = \big(|\{q : g(s_q)\}|,\ \mathrm{Acc}(P \mid g),\ \mathrm{Acc}(P \mid \neg g)\big). \quad (5)$$

Digests describe individual failures, while coverage identifies operations and branches with many failures. Acceptance is tested over the whole search set (Section 4.4), so a trigger that fires on only a handful of problems cannot pass. Coverage helps the synthesizer target larger groups of failures (full prompt in Appendix E).

4.2 PATCHES

A patch $\delta = (\ell, d, g, Q, V_Q)$ has a location ℓ (an operation or Par identifier, or the program's end), an optional diagnostic d (typically `build_check` then `run_check`, or `verify`), a trigger condition g that may reference d, and a sub-protocol Q with fresh identifiers and reads V_Q. Applying it rewrites the control program and the commit list,

$$P \oplus \delta: \qquad C_\ell \leftarrow \mathrm{Seq}\big(C_\ell,\ d,\ \mathrm{If}(g,\ Q,\ \varnothing)\big), \qquad T \leftarrow (\iota_Q,\ \iota_1, \ldots, \iota_m), \quad (6)$$

so d runs whenever ℓ is reached, Q only when g holds, and Q's answer takes priority with the previous commit target as fallback. For example, the exec-fix seed is CoT, `s1 = solve(problem)`, patched at ℓ = `s1` with d: `c1 = run_check[exec](s1)`, g = failed(`c1`) and Q: `r1 = revise(problem, s1, c1.evidence)`, committing `r1` before `s1`. Degenerate patches replace an operation, re-point T or write the *instruction slot*: at most 800 characters appended to the fixed prompt of one operation, a Par group or every answer-producing operation. An answer-format rule

Algorithm 1 REPROT: execution-feedback protocol search

Require: backbone $\mathcal{M}$, synthesizer $\mathcal{M}_{\text{syn}}$, seeds $\{P_1, \dots, P_m\}$, $\mathcal{D}_{\text{s}}$, $\mathcal{D}_{\text{v}}$, budget B, size bound L, rounds R, patches per round K

$P \leftarrow \arg\max\{\text{Acc}_{\mathcal{D}_{\text{s}}}(P_i) : \hat{c}_{\mathcal{D}_{\text{s}}}(P_i) \le B/2\}$; archive $\mathcal{H} \leftarrow \{P_1, \dots, P_m\}$ ▷ Eq. (13)
for $r = 1, \dots, R$ **do**
  $\{\zeta(q)\}_{q \in \mathcal{D}_{\text{s}}} \leftarrow$ digests of $\text{Exec}_{\mathcal{M}}(P, q)$ ▷ Eq. (4)
  $\{\delta_1, \dots, \delta_K\} \leftarrow \mathcal{M}_{\text{syn}}\big(P,\ \text{failures},\ \text{successes},\ \text{coverage},\ \text{history},\ B\big)$ ▷ Eqs. (5), (16)
  for $j = 1, \dots, K$ **do**
    $P'_j \leftarrow P \oplus \delta_j$; **if** invalid **or** $\text{dead}(P'_j)$ meets δ_j **then** record and **continue** ▷ Eqs. (6), (10)
    execute P'_j on $\mathcal{D}_{\text{s}}$; compute $n_{\text{fix}}, n_{\text{brk}}, p, \hat{c}$ against P; $\mathcal{H} \leftarrow \mathcal{H} \cup \{P'_j\}$ ▷ Eqs. (17), (18)
  end for
  $P \leftarrow$ best accepted P'_j by $(\text{Acc}_{\mathcal{D}_{\text{s}}}, -\hat{c})$ if any, else unchanged ▷ Eq. (9)
end for
$\mathcal{H}_B \leftarrow$ top-6 of $\{P \in \mathcal{H} :\ P$ a seed or accepted, $\hat{c}_{\mathcal{D}_{\text{s}}}(P) \le B\}$ by $(\text{Acc}_{\mathcal{D}_{\text{s}}}, -\hat{c})$, run on $\mathcal{D}_{\text{v}}$
return $P^\star = \arg\max_{P \in \mathcal{H}_B} \big(\text{Acc}_{\text{s+v}}(P), -\hat{c}_{\mathcal{D}_{\text{v}}}(P)\big)$ ▷ Eq. (19)

or checklist therefore adds no calls and must pass the same test as a structural change. Invalid patches and *dead computation* (a new candidate that nothing commits, reads or conditions on) are rejected statically. A patch can change outcomes only where its trigger fires, so its fixed-minus-broken count and its cost are both properties of the pair (g, Q).

4.3 JOINT TRIGGER AND SUB-PROTOCOL LEARNING

Write $A_P(q) = R(P, q)$ and let $s_\ell(q)$ be the state at ℓ. A patch can change the outcome only on its *trigger set* $G_\delta = \{q : g(s_\ell(q)) = 1\}$, so its effect on accuracy and cost decomposes as

$$\text{Acc}(P \oplus \delta) - \text{Acc}(P) = \mathbb{E}_q\Big[\mathbb{1}[g(s_\ell(q))]\ \Big(\mathbb{1}[\neg A_P(q) \wedge A_{P\oplus\delta}(q)] - \mathbb{1}[A_P(q) \wedge \neg A_{P\oplus\delta}(q)]\Big)\Big],$$
$$\mathbb{E}[c_{P\oplus\delta}] = \mathbb{E}[c_P] + c_d + \Pr[g(s_\ell) = 1] \cdot \mathbb{E}\big[c_Q \mid g(s_\ell) = 1\big]. \tag{7}$$

The first term counts fixes minus breaks; both terms depend on the pair (g, Q). With $g \equiv \top$, the sub-protocol incurs c_Q on every problem and can overturn any correct answer. With $g = \text{ndistinct}(I) \ge 2$, the disagreement rule of agreement-gated methods, it runs only when candidates differ. Failures on which all candidates agree are left unchanged. A diagnostic trigger ($\text{failed}(d)$, $\text{exec_err}(d)$, $\gamma_d < \theta$) can reach these failures, but the diagnostic is worth its cost only if Q uses the evidence. The synthesizer therefore proposes (d, g, Q, V_Q) jointly. Section 5.4 holds Q fixed and replaces g with fixed triggers.

4.4 OBJECTIVE AND ACCEPTANCE

The learning problem is

$$P^\star = \arg\max_P\ \mathbb{E}_q\big[R(P, q)\big] \quad \text{s.t.} \quad \mathbb{E}_q\big[c(P, q)\big] \le B, \qquad |P| \le L, \tag{8}$$

with a budget on *mean* calls (a conditional sub-protocol is charged only where it fires). On $\mathcal{D}_{\text{s}}$, a patch changes accuracy by exactly $(n_{\text{fix}} - n_{\text{brk}})/|\mathcal{D}_{\text{s}}|$. A neutral patch whose flips are coin tosses has $n_{\text{fix}} > n_{\text{brk}}$ with probability near one half. We therefore gate acceptance with a one-sided sign test on the flipped problems, $p(\delta) = \Pr[X \ge n_{\text{fix}}]$ with $X \sim \text{Binomial}(n_{\text{fix}} + n_{\text{brk}}, \frac{1}{2})$:

$$\text{accept}(\delta) \iff n_{\text{fix}}(\delta) - n_{\text{brk}}(\delta) > 0\ \wedge\ p(\delta) \le \alpha\ \wedge\ \hat{c}_{\mathcal{D}_{\text{s}}}(P \oplus \delta) \le B\ \wedge\ |P \oplus \delta| \le L, \tag{9}$$

with $\alpha = 0.2$, which rejects $(n_{\text{fix}}, n_{\text{brk}}) = (2, 0)$ or $(3, 1)$ and accepts $(3, 0)$ or $(6, 2)$; counts are paired and taken on the full search set (Appendix C.6).

4.5 SEARCH AND SELECTION

Each benchmark has three disjoint pools. Patches are proposed and accepted on $\mathcal{D}_{\text{s}}$; $\mathcal{D}_{\text{v}}$ enters only final selection. All methods share the test set, evaluated once per seed after freezing the protocol.

Search uses a benchmark-specific subset of the ten-seed library (Appendix D). The root P_0 is the best seed costing at most $B/2$, leaving budget for learned structure. Each of R rounds proposes K patches and keeps the best accepted one (Algorithm 1). Selection considers only seeds and accepted programs within budget; choosing a rejected candidate that tops the search set would incur winner's-curse bias. The six programs with the highest search-set accuracy are evaluated on $\mathcal{D}_{\mathrm{v}}$, and $P^{\star}$ maximises pooled search–validation accuracy (Eq. 19). A seed that remains best is returned unchanged (data discipline and cost accounting in Appendix C.7).

5 EXPERIMENTS

5.1 SETUP

Benchmarks and baselines. We evaluate on MATH-500 (Hendrycks et al., 2021), OlympiadBench (He et al., 2024) and TheoremQA (Chen et al., 2023b) for mathematics; LiveCodeBench (Jain et al., 2025) (v6, January–April 2025) for code; GPQA-Diamond (Rein et al., 2023), SuperGPQA-hard (Du et al., 2026) and MMLU-Pro (Wang et al., 2024) for science and expert QA; and MuSR (Sprague et al., 2024) for narrative reasoning. The metric is accuracy (execution pass@1 on LiveCodeBench), averaged with equal weight over benchmarks. MuSR, TheoremQA and OlympiadBench have no training split. MuSR is partitioned by story: all questions sharing a story belong to the same search, validation or test set, with 400 test items. For TheoremQA and OlympiadBench, a seed-42 sample of 247 and 181 items is held out for calibration (OlympiadBench adds 93 multi-answer rows outside its evaluation set), leaving 500 and 400 test items. Every method uses the same test partition for each benchmark (Appendix G). The seventeen baselines fall into four groups: *single-agent* (Vanilla, CoT, Self-Consistency, PHP), *fixed topologies* (Chain, Star, Tree, Complete and Random graph (Qian et al., 2025)), *frameworks* (LLM-Debate (Du et al., 2023), LLM-Blender (Jiang et al., 2023), AutoGen (Wu et al., 2023)) and *learned structure* (DyLAN, GPTSwarm, G-Designer, AgentPrune, CARD).

Protocol and settings. All methods use one backbone (`gpt-5.6-luna`; up to 16,384 output tokens; temperature as in each method's paper) and a shared adapter for question text, answer extraction and scoring. Learned-structure systems run from pinned commits of their official repositories, using their own training loops on pools disjoint from the test items (Appendix F). Every main-table cell is the mean over three evaluation seeds (42, 1, 2; for learned-structure baselines the seed also spans training). The backbone is not deterministic at temperature 0: the seed s.d. of the average is 0.1–0.8 (CoT 1.2) against up to 4.2 for single columns, so the average is the primary comparator. REPROT uses $B = 6$ mean calls, $R = 10$ rounds of $K = 5$ patches, $\alpha = 0.2$ and synthesizer `gpt-5.6-sol` (full settings and pools in Appendices C.7 and G). The final configuration and selection rule are fixed before test evaluation. Programs are selected using only search and validation data (Eq. 19); the main table reports the subsequent three-seed evaluation of these fixed protocols.

5.2 MAIN RESULTS

Top tier. Nine baselines form a top tier at 79.7–80.3 (the five fixed topologies, LLM-Debate, G-Designer, AgentPrune, CARD). ADAS averages 80.00 and AFlow 82.56. REPROT averages 84.18 ± 0.50, about 3.9 points above G-Designer and 1.6 above AFlow. Paired sign tests on the 3,673 test items pooled over three seeds favour it over every baseline: against G-Designer there are 713 vs. 315 discordant pairs ($p<10^{-35}$), and against AFlow 458 vs. 289 ($p=6.7\times10^{-10}$). The difference persists without TheoremQA and MuSR (AFlow 308 vs. 174, $p=1.1\times10^{-9}$; G-Designer 277 vs. 183, $p<10^{-4}$). Per benchmark (Figure 4), REPROT has the highest mean on seven of eight columns and no significant losses. Its record against AFlow is 4–4–0, with wins on OlympiadBench, GPQA, SuperGPQA and MMLU-Pro and ties elsewhere. The largest margins over the seventeen reproduced baselines are on TheoremQA (83.5 vs. 77.7 for PHP) and MuSR (79.1 vs. 64.2 for LLM-Debate). The corresponding margins over AFlow do not reach significance (p=0.25, 0.10).

Accuracy–cost trade-off and seed spread. REPROT uses 3.9 mean calls, within B, against 5.2–11.0 for the top tier. Its average accuracy–call pair is not dominated by any baseline (Figure 3). AFlow uses fewer calls (3.2) with about 1.6 points lower accuracy, after spending more executor calls during search on seven of eight benchmarks (Appendix H.1). REPROT has the highest average

Method	MATH-500	Olympiad	TheoremQA	LCB	GPQA	SuperGPQA	MMLU-Pro	MuSR	Avg.↑	Calls↓
Self-Consistency	**96.1** ± 0.1	86.6 ± 0.4	72.5 ± 0.9	85.1 ± 2.0	85.9 ± 0.5	61.5 ± 0.1	85.3 ± 1.1	59.5 ± 1.1	79.06 ± 0.12	10.0
PHP	95.3 ± 0.2	84.5 ± 1.6	77.7 ± 0.8	86.7 ± 0.3	84.2 ± 2.8	61.3 ± 0.6	84.5 ± 0.5	59.4 ± 2.4	79.20 ± 0.61	2.1
CoT	94.8 ± 0.7	81.0 ± 0.4	72.5 ± 0.1	83.6 ± 4.2	85.5 ± 2.9	58.5 ± 0.5	84.7 ± 0.2	60.6 ± 2.6	77.65 ± 1.17	2.0
Vanilla	94.1 ± 0.8	82.2 ± 2.3	75.2 ± 0.7	81.3 ± 1.7	82.5 ± 0.8	59.2 ± 0.8	84.3 ± 0.9	60.0 ± 1.3	77.35 ± 0.44	1.0
Complete graph	**96.1** ± 0.3	87.3 ± 0.4	74.3 ± 0.9	87.8 ± 0.3	86.5 ± 1.5	62.6 ± 0.5	84.9 ± 0.8	62.2 ± 0.8	80.21 ± 0.13	11.0
Random graph	95.9 ± 0.2	87.8 ± 0.5	73.3 ± 0.8	87.2 ± 0.9	86.4 ± 0.0	62.3 ± 0.3	85.1 ± 1.0	60.6 ± 0.8	79.83 ± 0.09	6.0
Chain	95.9 ± 0.5	87.1 ± 0.6	74.5 ± 0.1	87.8 ± 0.3	86.5 ± 0.8	62.4 ± 0.1	84.8 ± 1.2	61.3 ± 1.4	80.04 ± 0.39	6.0
Star	95.9 ± 0.5	86.9 ± 0.3	73.7 ± 0.5	87.2 ± 0.9	87.2 ± 0.3	62.4 ± 0.5	85.1 ± 1.0	59.4 ± 2.8	79.73 ± 0.43	6.0
Tree	95.9 ± 0.2	86.4 ± 0.9	74.2 ± 0.6	86.7 ± 0.3	87.0 ± 0.8	62.4 ± 0.5	84.8 ± 0.2	60.8 ± 1.8	79.78 ± 0.34	6.0
LLM-Debate	95.7 ± 0.1	86.9 ± 0.3	72.2 ± 0.3	86.5 ± 1.4	86.6 ± 1.1	61.7 ± 0.7	85.7 ± 0.5	64.2 ± 0.5	79.94 ± 0.25	6.0
LLM-Blender	95.0 ± 0.2	81.2 ± 0.6	71.9 ± 0.8	81.7 ± 1.0	82.3 ± 2.3	60.4 ± 0.5	83.4 ± 1.4	63.0 ± 1.1	77.36 ± 0.07	5.0
AutoGen	94.5 ± 0.1	81.7 ± 1.0	72.0 ± 0.6	78.5 ± 2.2	83.5 ± 1.6	58.8 ± 0.8	83.9 ± 1.0	61.3 ± 1.8	76.78 ± 0.15	1.2
G-Designer	95.5 ± 0.2	86.6 ± 0.4	75.7 ± 0.4	86.3 ± 2.3	87.4 ± 1.5	62.2 ± 0.5	85.1 ± 0.3	63.5 ± 2.5	80.29 ± 0.81	5.3
AgentPrune	95.5 ± 0.1	86.2 ± 1.1	75.3 ± 1.0	85.9 ± 0.9	87.2 ± 1.3	62.4 ± 0.2	85.7 ± 0.8	62.8 ± 3.2	80.13 ± 0.54	5.2
CARD	95.4 ± 0.2	86.7 ± 0.6	75.6 ± 0.9	84.6 ± 0.6	86.4 ± 2.2	61.4 ± 0.7	84.9 ± 0.4	63.0 ± 2.0	79.75 ± 0.50	5.3
GPTSwarm	95.8 ± 0.4	83.8 ± 0.2	72.4 ± 1.2	87.0 ± 1.2	84.0 ± 1.2	61.3 ± 0.6	83.5 ± 2.2	60.6 ± 1.4	78.55 ± 0.54	4.4
DyLAN	94.0 ± 0.3	81.0 ± 0.9	71.1 ± 0.6	81.9 ± 0.9	87.5 ± 0.6	62.7 ± 0.3	85.6 ± 0.5	62.3 ± 2.9	78.26 ± 0.59	6.0
ADAS	94.2 ± 0.5	88.2 ± 0.9	75.5 ± 0.9	84.2 ± 2.7	86.5 ± 1.6	62.6 ± 0.4	84.9 ± 0.2	63.9 ± 1.9	80.00 ± 0.81	7.7
AFlow	95.9 ± 0.1	86.6 ± 0.3	82.7 ± 0.1	88.0 ± 1.7	84.3 ± 0.0	61.2 ± 0.1	84.6 ± 0.5	77.2 ± 1.6	82.56 ± 0.45	3.2
REPROT (ours)	95.9 ± 0.3	**88.9** ± 0.6	**83.5** ± 0.8	**89.0** ± 2.3	**87.7** ± 1.8	**63.1** ± 0.3	**86.2** ± 0.4	**79.1** ± 0.6	**84.18** ± 0.50	3.9

Best Second-best Rankings use reported means; ties receive the same highlight. Calls are not ranked.

Table 1: Main results on the test sets of Section 5.1 (OlympiadBench, TheoremQA and MuSR: 400, 500 and 400 items; MuSR split by story): accuracy (mean±s.d. over seeds 42, 1, 2), one backbone; "Calls": mean LLM calls per problem. Groups as in Section 5.1, then learned workflows and protocols. Shading marks the best and second-best mean per column over all rows, not a significance claim (paired tests in Section 5.2).

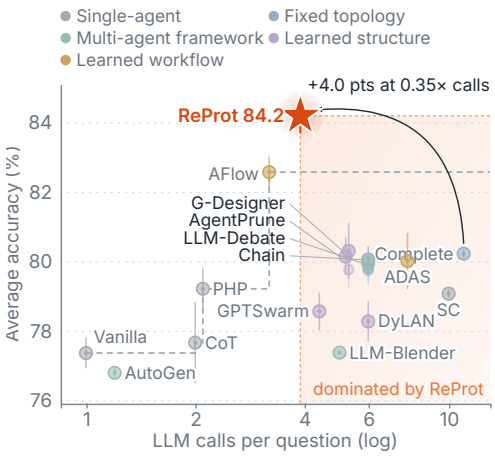

Figure 3: Average accuracy vs. mean calls (bars: seed s.d.). Dashed: baseline Pareto front; shaded: region REPROT dominates. SC: self-consistency. Faded: other fixed topologies and CARD. Gold: original ADAS and AFlow.

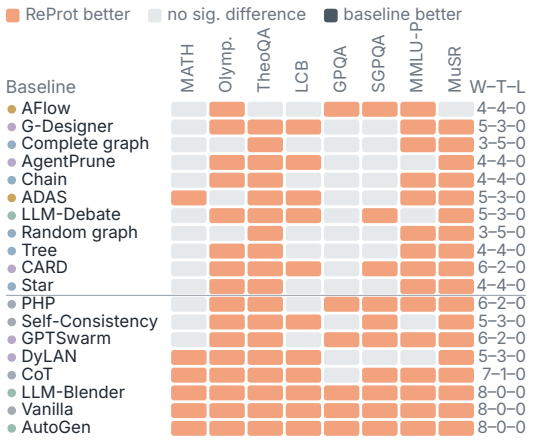

Figure 4: Head-to-head per benchmark: exact two-sided sign test on (seed, item) pairs pooled over three seeds (tie if $p \geq 0.05$). Rows ordered by the baseline's average (line: nine-baseline tier plus ADAS/AFlow above); W–T–L: REPROT's wins, ties, losses.

of all twenty methods on each evaluation seed (83.66–84.66), and its lowest seed exceeds AFlow's highest (83.04). On every benchmark it lies on the accuracy–call Pareto front among the seventeen reproduced baselines (Figure 5); its MMLU-Pro program remains the unpatched Debate seed at 6.0 calls.

5.3 SEARCH DYNAMICS

In each seed-42 search (Figure 6), the gate accepted 1–4 patches per benchmark and rejected 40–49 candidates; the budget rarely bound. Search-split gains exceed test gains: OlympiadBench rises from 64.0 to 84.8, mostly through the round-1 answer-format instruction. Its search pool contains

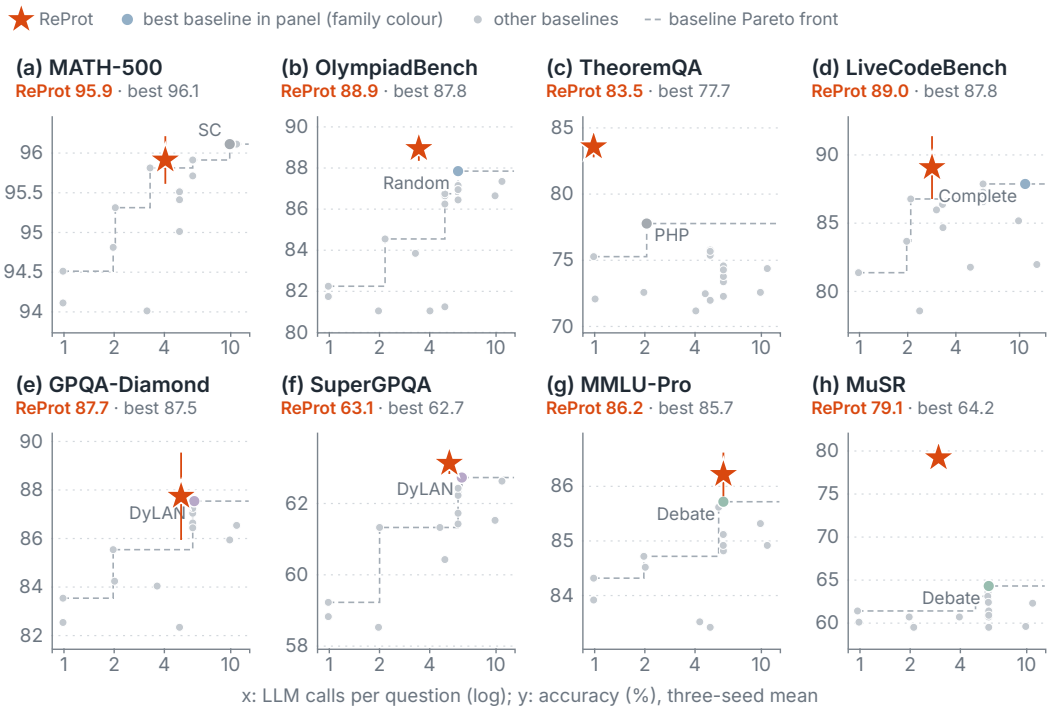

Figure 5: Accuracy against mean calls per benchmark (three-seed means; bars: s.d.). Dashed: Pareto front of the 17 reproduced baselines. Subtitles give REPROT's accuracy and the best of these 17, which is coloured by family and named. ADAS and AFlow are not drawn; their results and mean calls are reported in Table 1.

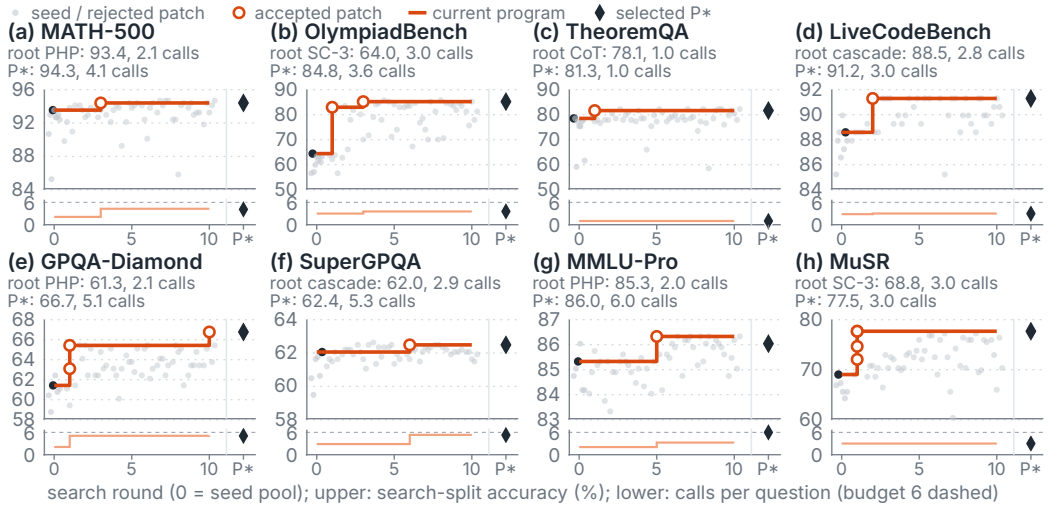

Figure 6: Search at seed 42. Round 0: the seed pool (root in black); rounds 1–10: rejected (gray) and accepted (ringed) patches; orange step line: the current program; diamond: the selected $P^\star$. Lower strip: calls per question of the current program, with the budget B=6 dashed. Accuracies are on the search split.

93 multi-answer rows absent from the test set, where accuracy moves 85.8 → 89.0. On MMLU-Pro, pooled selection favours the unpatched Debate seed over the cheaper patched program. The call strip shows the budget binding only on MMLU-Pro (6.0); TheoremQA stays at 1.0 call (Appendix H).

Variant	MATH-500	Olympiad	TheoremQA	LCB	GPQA	SuperGPQA	MMLU-Pro	MuSR
Root seed P_0	95.6 (2.0)	85.8 (3.0)	72.6 (1.0)	84.0 (2.8)	85.9 (2.1)	62.0 (2.9)	85.4 (2.1)	60.0 (3.0)
REPROT (learned)	96.0 (4.1)	89.0 (3.5)	84.4 (1.0)	89.1 (2.8)	86.4 (5.1)	63.2 (5.3)	86.6 (6.0)	78.8 (3.0)
Structure-only search†, 3 seeds	95.6 ± 0.2 (6.0)	86.5 ± 0.5 (3.0)	72.4 ± 0.5 (3.0)	85.9 ± 0.9 (2.9)	86.4 ± 1.0 (4.9)	61.7 ± 0.6 (6.0)	84.5 ± 0.2 (6.0)	59.2 ± 1.2 (4.1)
Free-code search†, 3 seeds	–	88.6 ± 0.5 (2.7)	–	–	–	–	84.4 ± 0.5 (2.3)	–
Trigger: first branch ($g \equiv \top$)	95.0 (10.0)	87.5 (5.0)	81.6 (1.0)*	79.4 (2.0)	83.8 (9.0)	62.5 (5.0)	84.6 (6.0)*	78.8 (3.0)*
Trigger: disagreement (ndistinct ≥ 2)	94.4 (4.2)	88.0 (3.5)	83.4 (1.0)*	83.4 (7.2)	85.4 (5.4)	62.1 (6.7)	84.8 (6.0)*	78.8 (3.0)*
Visibility: isolated	96.4 (4.1)	89.5 (3.5)*	82.8 (1.0)*	85.1 (3.0)*	86.4 (5.1)*	62.0 (5.3)*	84.0 (6.0)	78.8 (3.0)*
Visibility: open	96.0 (4.1)	87.5 (3.5)*	83.4 (1.0)*	84.6 (3.1)	86.9 (5.1)	62.4 (5.3)	85.4 (6.0)*	78.8 (3.0)*

Table 2: Test-set controls: accuracy (mean calls in parentheses); seed 42 unless marked "3 seeds" (mean±s.d.). Red and blue mark the highest and second-highest reported accuracies per column, including ties. †Search used an earlier configuration (R=8, K=4, earlier pools), not rerun under the main configuration; MuSR search predates the digest fix. *Identical to the learned program, so differences are single-seed noise (MuSR has no trigger or full-text read). "–": not run.

5.4 CONTROLS

Table 2 reports five controls (Appendix H.2). Seeds compete at selection; REPROT improves over P_0 on all eight benchmarks at seed 42, from 0.4 on MATH-500 to 18.8 on MuSR (11.8 on TheoremQA and 5.1 on LiveCodeBench). Free-code search† (ADAS-style (Hu et al., 2025), same synthesizer, seeds, gate and budget) reaches 88.6 ± 0.5 on OlympiadBench, 0.3 below REPROT. The operative rewrite is a strict answer-format instruction. No MMLU-Pro candidate passes the gate (84.4 ± 0.5). A prompt-level change can outweigh a structural one.

Without the instruction slot (structure-only†), search averages 79.03 against 84.18, differing most on MuSR (59.2 → 79.1) and TheoremQA (72.4 → 83.5). Rounds, pools and the MuSR digest fix differ, so this comparison bounds but does not isolate the slot's effect. Replacing all conditions, including seed conditions, with *first-branch* ($\top$) or *disagreement* (ndistinct ≥ 2) changes cost by up to 2.6×. LiveCodeBench accuracy falls from 89.1 to 79.4 and 83.4, respectively. Smaller effects elsewhere include rerun variation in unchanged programs. Replacing full-text with answer-only reads (*isolated*) or vice versa (*open*) changes accuracy by at most 4.5 points: the open-visibility loss on LiveCodeBench. Isolated visibility loses 2.6 on MMLU-Pro.

5.5 FIXES, BREAKS AND COSTS

At seed 42 (Figure 8 and Table 9 in Appendix H), we record which test problems P_0 and REPROT solve. Each P_0 error is a *selection* failure if any candidate is correct and a *generation* failure otherwise. Fixes exceed breaks on all eight benchmarks; five paired sign tests pass: OlympiadBench (18 vs. 5, p=0.011), LiveCodeBench (12 vs. 3, p=0.035), SuperGPQA (18 vs. 6, p=0.023), TheoremQA (66 vs. 7) and MuSR (106 vs. 31), the last two with $p<0.001$. Generation failures account for all 66 TheoremQA fixes and 85 of 106 MuSR fixes, which selection alone cannot correct. Of the 66 TheoremQA fixes, 52 have correct root values in forms the scorer rejects (e.g. a fraction; Appendix J.1). This gain mostly reflects answer-format compliance. Smaller net gains on MATH-500, GPQA and MMLU-Pro are statistically indistinguishable from noise ($p \geq 0.18$).

Learned triggers adapt cost on four benchmarks (Figure 7); LiveCodeBench uses 2 calls on 87% of items and 8 otherwise. Offline search takes 1.8–6.7 hours per benchmark (Appendix I); online call savings over the complete graph repay it after 700–13,500 problems. Savings over G-Designer are 1.4 calls, negative on MMLU-Pro.

6 DISCUSSION AND CONCLUSION

REPROT learns typed reasoning protocols, gating patches by a sign test and mean-call budget. With one backbone and evaluation protocol, it leads twenty methods at 84.18 average accuracy and 3.9 calls, about 1.6 points above AFlow (four paired wins and four ties). Controls identify large gains from learned instructions, mostly TheoremQA answer formatting, and substantial LiveCodeBench accuracy losses when triggers change. Frozen weights and prompts limit search to composing nine operations; small search sets cannot distinguish a net gain of two from noise. Each search evaluates $R \times K$ candidates with a stronger synthesizer, amortised only at scale (El et al., 2025). Transfer to other backbones remains untested. Accepted changes record fixes and breaks.

REPRODUCIBILITY STATEMENT

Every method, including REPROT, uses the same adapter for question text, answer extraction and scoring. Appendix F describes the adapter, scorer, evaluation-set sizes and fidelity records of the reproduced baselines (Table 6). All methods use the same fixed test partition for each benchmark. TheoremQA and OlympiadBench use seed-42 held-out calibration samples; MuSR is split by story, with no story shared between search, validation and test. Appendix G describes pool construction and final selection. The protocol language and static checks are in Appendix B, and the seed library is reproduced exactly as executed in Appendix D. Algorithm 1 gives the search loop; Appendix E provides the released executor and synthesizer prompt templates. Appendix I records every learned protocol: its root, accepted patches with $(n_{\text{fix}}, n_{\text{brk}}, p, \hat{c})$, learned instructions, selection record and offline search cost. Main-table cells are means over evaluation seeds 42, 1 and 2; other tables specify seed coverage. For learned-structure baselines, each seed also spans training. Tests run once per seed after the protocol is frozen, with the executor call cache disabled. Test scores do not enter configuration or program selection (Appendix G). Search hyperparameters (B, L, R, K, α, synthesizer and temperature) are given in Section 5.1. The code release contains the executor, synthesizer loop, shared adapter and scorer, baseline harness with pinned commits, held-out indices and per-item records behind every table.

AI USE STATEMENT

The method uses large language models: `gpt-5.6-luna` is the executor backbone for every method in Section 5, and `gpt-5.6-sol` synthesizes protocol patches (Section 4; system prompt in Appendix E). The learned protocols and instruction slots in Appendix I are therefore LLM-generated artifacts produced and selected by the pipeline described in this paper. Generative AI was also used to assist with writing and polishing the manuscript. The authors reviewed the final text and take responsibility for the paper's claims, results and artifacts.

REFERENCES

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023a.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Wenhu Chen, Ming Yin, Max Ku, Pan Lu, Yixin Wan, Xueguang Ma, Jianyu Xu, Xinyi Wang, and Tony Xia. Theoremqa: A theorem-driven question answering dataset. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7889–7901, 2023b.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024.

Xeron Du, Yifan Yao, Kaijing Ma, Bingli Wang, Tianyu Zheng, Minghao Liu, Yiming Liang, Xiaolong Jin, Zhenlin Wei, Chujie Zheng, et al. Supergpqa: Scaling llm evaluation across 285 graduate disciplines. *Advances in Neural Information Processing Systems*, 38, 2026.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*, 2023.

Batu El, Mert Yuksekgonul, and James Zou. Inefficiencies of meta agents for agent design. *arXiv preprint arXiv:2510.06711*, 2025.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Nan Duan, Weizhu Chen, et al. Critic: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, volume 2024, pp. 57734–57811, 2024.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3828–3850, 2024.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14165–14178, 2023.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic llm-powered agent network for task-oriented agent collaboration, 2024. URL https://arxiv.org/abs/2310.02170.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Chen Qian, Zihao Xie, YiFei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, et al. Scaling large language model-based multi-agent collaboration. In *International Conference on Learning Representations*, volume 2025, pp. 41488–41505, 2025.

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

Jon Saad-Falcon, Adrian Gamarra Lafuente, Shlok Natarajan, Nahum Maru, Hristo Todorov, Etash Guha, E Kelly Buchanan, Mayee Chen, Neel Guha, Christopher Ré, et al. Archon: An architecture search framework for inference-time techniques. *arXiv preprint arXiv:2409.15254*, 2024.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space. In *International Conference on Learning Representations*, volume 2025, pp. 3841–3865, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Zayne Sprague, Xi Ye, Kaj Bostrom, Swarat Chaudhuri, and Greg Durrett. Musr: Testing the limits of chain-of-thought with multistep soft reasoning. In *International Conference on Learning Representations*, volume 2024, pp. 14670–14728, 2024.

Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Y Zou. Mixture-of-agents enhances large language model capabilities. In *International Conference on Learning Representations*, volume 2025, pp. 33944–33963, 2025.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

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

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

Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Shengping Liu, Bin Sun, Kang Liu, and Jun Zhao. Large language models are better reasoners with self-verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 2550–2575, 2023.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Tongtong Wu, Yanming Li, Ziye Tang, Chen Jiang, Linhao Luo, Guilin Qi, Shirui Pan, and Gholamreza Haffari. Card: Towards conditional design of multi-agent topological structures. *arXiv preprint arXiv:2603.01089*, 2026.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. G-designer: Architecting multi-agent communication topologies via graph neural networks. *arXiv preprint arXiv:2410.11782*, 2024.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*, 2025a.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for llm-based multi-agent systems. In *International Conference on Learning Representations*, volume 2025, pp. 75389–75428, 2025b.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025c.

Mingming Zhao, Xiaokang Wei, Yuanqi Shao, Kaiwen Zhou, Lin Yang, Siwei Rao, Junhui Zhan, and Zhitang Chen. A^2flow: Automating agentic workflow generation via self-adaptive abstraction operators. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29930–29938, 2026.

Chuanyang Zheng, Zhengying Liu, Enze Xie, Zhenguo Li, and Yu Li. Progressive-hint prompting improves reasoning in large language models. *arXiv preprint arXiv:2304.09797*, 2023.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Arik. Multi-agent design: Optimizing agents with better prompts and topologies. In *International Conference on Learning Representations*, volume 2026, pp. 15844–15872, 2026.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Language agents as optimizable graphs. *arXiv preprint arXiv:2402.16823*, 2024.

A Extended Related Work

Learned communication topologies. GPTSwarm (Zhuge et al., 2024) optimises edge probabilities, G-Designer (Zhang et al., 2024) generates query-conditioned graphs, AgentPrune (Zhang et al., 2025b) prunes message graphs, DyLAN (Liu et al., 2024) rewires teams across rounds, CARD (Wu et al., 2026) conditions generation on execution context and MacNet (Qian et al., 2025) scales fixed topologies. All vary V over a fixed O and C. REPROT varies all four components, with trigger conditions on the state a problem reaches rather than on its text.

Prompting, sampling and verification. Our operations and control patterns draw on prompting, sampling, decomposition, program execution, verification and refinement methods (Wei et al., 2022; Kojima et al., 2022; Wang et al., 2022; Zheng et al., 2023; Zhou et al., 2022; Chen et al., 2022; Yao et al., 2023; Madaan et al., 2023; Weng et al., 2023; Chen et al., 2024; Gou et al., 2024; Shinn et al., 2023); debate (Du et al., 2023) and LLM-Blender (Jiang et al., 2023) aggregate candidates. Adaptive test-time compute (Snell et al., 2024; Chen et al., 2023a; Brown et al., 2024) studies when more computation pays. REPROT asks which computation to add and when.

Automated design of agentic systems. ADAS (Hu et al., 2025), AFlow (Zhang et al., 2025c), Archon (Saad-Falcon et al., 2024), A^2Flow (Zhao et al., 2026), AgentSquare (Shang et al., 2025), MaAS (Zhang et al., 2025a), MASS (Zhou et al., 2026) and Mixture-of-Agents (Wang et al., 2025) search or stack agent code, operator graphs or modular agent spaces by execution. We also search by execution and compare directly against an ADAS-style free-code search with the same synthesizer, seeds and budget (Section 5.4). Our update uses a typed program with enforced visibility, a patch that couples a trigger condition to a sub-protocol, and a sign-test gate based on fixed-minus-broken counts instead of a score. Prompt optimisers (DSPy (Khattab et al., 2023), TextGrad (Yuksekgonul et al., 2024), OPRO (Yang et al., 2024)) tune the prompts of a fixed pipeline. REPROT learns the pipeline and changes prompts only through a bounded, gated instruction slot.

B Protocol Language

Table 3 lists the operation set O of Section 3.2. The concrete syntax is JSON. The grammar below condenses the synthesizer's grammar, reproduced verbatim in Appendix E.3. The synthesizer does not receive the loop-iteration predicate.

```
PROGRAM:
  {"type":"op","id":ID,"op":OP,"reads":[REF,...], ...params}
  {"type":"seq","body":[P,...]}
  {"type":"par","id":ID,"body":[P,...]}          (branches cannot see each other)
  {"type":"if","cond":COND,"then":P,"else":P|null}
```

```
  {"type":"repeat","max":1..3,"until":COND,"body":P}
  {"type":"commit","ref":ID | [ID, ID, ...]}   answer = first listed artifact that exists
  A "par" carries an "id"; "<parId>.answers" exposes all branch answers to later ops.
  Limits: at most 24 nodes, 6 branches per par, 3 iterations per repeat.

OPS and params:
  solve        style: cot | direct | hint | critique | subproblems; temperature
   0.0/0.7/1.0
  program      writes and executes Python; answer = printed result; ok = ran without
   error
  decompose    -> sub-questions (kind subproblems)
  build_check  -> checklist of concrete conditions a correct answer must satisfy (
   checkspec)
  run_check    mode: llm | exec; candidate = first solution-like read; conditions = a
               checkspec read; sets ok=True/False and evidence
  verify       k: 1 | 3 isolated verifications of the candidate answer (sees ONLY problem
    +
               answer); sets conf 0-100 and ok
  revise       reads a solution (full text) + evidence; produces a corrected solution
  aggregate    rule: vote | evidence (prefer candidates whose check passed, then vote) |
   llm
  select       deterministic: first candidate whose attached check passed, else vote

READS (visibility -- the ONLY information the op sees besides the problem):
  "problem" | "<id>" (full text) | "<id>.answer" | "<id>.evidence" | "<parId>.answers"

CONDITIONS (runtime state only):
  {"agree":[ids]} {"disagree":[ids]} {"n_distinct":[ids],"ge":k} {"passed":id} {"failed":
   id}
  {"exec_ok":id} {"exec_err":id} {"empty":id} {"conf_lt":[id,x]} {"conf_ge":[id,x]}
  {"budget_ge":n} {"iter_lt":k} {"and":[..]} {"or":[..]} {"not":C} true

PATCH KINDS (each with a one-sentence "rationale"):
  {"kind":"insert_after","where":ID,"diagnostic":P|null,"cond":COND|null,"sub":P,
   "commit":ID|[ID,..]|null}     after node ID (any op or par id, also inside if-branches
    /
                                repeat bodies): run diagnostic, then if cond holds run
   sub
  {"kind":"wrap_root","diagnostic":P|null,"cond":COND|null,"sub":P,"commit":ID|null}
  {"kind":"replace","where":ID,"node":P}
  {"kind":"set_commit","ref":ID}
  {"kind":"set_instruction","where":ID|"*","instruction":"..."}
      set the INSTRUCTION SLOT (<= 800 chars, appended to the op's prompt) of one
      LLM-calling op, of every LLM op inside a par group, or ("*") of every
      answer-producing solve/program/revise op: a persona, precise answer-format
      rules, a checklist of pitfalls, a solving strategy.
      Any op node may also carry "instruction" when created inside a patch.

RULE: a new solution/aggregate artifact must be able to reach the answer -- list it in
  "commit", read it from a later op, or use it in a condition; otherwise the patch is
  rejected statically as dead computation.

ACCEPTANCE: a patch is kept only if it fixes clearly more problems than it breaks (one-
   sided
  sign test on fixed vs broken, p <= 0.2) within budget -- so target failure modes that
   recur
  across many problems, and gate new computation so that correct problems are left
   untouched.
```

Operation	Parameters	Typical reads $V(o)$	Produces	Calls
`solve`	style $\in$ {cot, direct, hint, critique, subproblems}; τ	problem (+ an answer / other solutions' text / a decomposition, by style)	solution (x, y)	1
`program`	τ	problem	program: y = printed result, ok = ran	1 + exec
`decompose`	–	problem	subproblems	1
`build_check`	–	problem	checkspec (conditions a correct answer must satisfy)	1
`run_check`	mode $\in$ {llm, exec}	problem, a candidate's answer, a checkspec	evidence: ok = PASS/FAIL, e	1 + exec
`verify`	$k \in \{1, 3\}$	problem, a candidate's answer *only*	verdict: ok, γ (mean of k isolated verdicts)	k
`revise`	–	problem, a solution's full text, evidence	solution	1
`aggregate`	rule $\in$ {vote, evidence, llm}	candidate answers (full texts for llm)	aggregate: y	0 / 0 / 1
`select`	–	candidates and their attached checks	aggregate: first checked-and-passed candidate, else vote	0

Table 3: The operation set O: one prompt (plus a sandboxed execution where marked) with exactly the declared reads as inputs. Prompts are fixed across benchmarks except for the answer-format line and an optional learned instruction slot ($\leq$ 800 characters; Section 4.2); τ: temperature.

Static validation. A program is rejected if any operation name, style, rule or mode is unknown; if an identifier is reused within a scope (identifiers may repeat only across the two branches of one `if`); if a read or a condition references an identifier not defined earlier in program order; if an aggregator has no candidate reads; if a loop bound exceeds 3; if the program exceeds 24 nodes; or if a commit reference is undefined. A `par` group's identifier becomes defined after its body. Before execution, a further check rejects any patch that introduces a *dead* solution or aggregate artifact: one that is neither committed, read by an operation, referenced by a condition, nor included in a Par group read as a whole ($\mathrm{Id}(\cdot)$: identifiers mentioned; mem: group members),

$$\mathrm{dead}(P) = \Big\{ \iota \text{ a candidate} : \ \iota \notin T \cup \textstyle\bigcup_{o \in O} \mathrm{Id}(V(o)) \cup \bigcup_{g \in C} \mathrm{Id}(g) \cup \bigcup_{\iota' \in \mathrm{used}} \mathrm{mem}(\iota') \Big\}, \quad (10)$$

and the patch is rejected if $\mathrm{dead}(P \oplus \delta)$ contains an identifier it introduces.

Operation prompts. Every operation uses a fixed system/user prompt pair. The user message contains the problem, the rendered reads and the benchmark's answer-format line (`Answer: <letter>`, `\boxed{}`, `Final answer:`, or a single `python` block). `program` asks for a self-contained Python program using exact arithmetic that prints only the final answer; the program runs in a subprocess with a 10-second timeout. `run_check`[exec] asks for a checker program that prints `CHECK_PASS` or `CHECK_FAIL`. `verify` instructs the model to re-derive the answer without assuming the proposal is right and to end with `VERDICT: YES|NO` and `CONFIDENCE: 0-100`; a NO verdict with confidence γ is scored $100 - \gamma$, and k samples are averaged. The exact prompts are in Appendix E.2.

Code tasks. On LiveCodeBench the answer is a program, and four benchmark-level definitions apply. Two candidate programs are equivalent under $\equiv$ iff they produce identical outputs on the problem's public examples; agree, $\mathrm{ndistinct}$ and the vote rules use this comparison. `run_check`[exec] executes the candidate on the public examples (the statement examples, in both the calibration pool and the test set) and returns the passed and failed cases as evidence without an LLM call, at cost 0. `build_check` asks for three to six additional test cases with hand-derived expected outputs, and `program` coincides with `solve`. A note in the synthesizer's prompt states these definitions. Hidden tests are never executed inside a protocol; they score only the committed program.

C Formal Semantics and Synthesis Details

This appendix expands the definitions, arguments and implementation details summarized in Sections 3 and 4.

C.1 State and Execution

Fix a backbone $\mathcal{M}$ and a problem q with reference answer $y^\star(q)$. A protocol operates on a *state*

$$s = (q, \mathcal{A}, b), \qquad \mathcal{A} : \ \mathrm{Id} \rightharpoonup \{(\kappa, x, y, e, \mathrm{ok}, \gamma)\}, \quad (11)$$

where $\mathcal{A}$ is a partial map from identifiers to *artifacts*. Each artifact contains a kind κ (solution, program, subproblems, checkspec, evidence, verdict, aggregate), the full text x, the extracted answer y, an evidence summary e, an outcome flag $\mathrm{ok} \in \{\top, \bot, \varnothing\}$ and a confidence $\gamma \in [0, 100] \cup \{\varnothing\}$. The remaining call budget is b. An operation o with identifier ι_o sees the state only through its declared *reads* $V(o) \subseteq \mathrm{Ref} = \{\texttt{problem}\} \cup \{\iota,\ \iota\texttt{.answer},\ \iota\texttt{.evidence},\ \iota\texttt{.answers} : \iota \in \mathrm{Id}\}$, which the executor renders into its prompt through a fixed function ρ:

$$(a, c) \sim [\![o]\!]_{\mathcal{M}}\big(q,\ \rho(V(o), \mathcal{A});\ \xi\big), \qquad \mathcal{A}' = \mathcal{A}[\iota_o \mapsto a], \qquad b' = b - c, \quad (12)$$

with sampling noise ξ and c calls consumed. A read ι exposes an artifact's full text; $\iota\texttt{.answer}$ and $\iota\texttt{.evidence}$ expose only the named field, and $\iota\texttt{.answers}$ exposes the answers of a group (Eq. 15). An operation reached with $b \le 0$ is skipped. Executing a protocol P from the initial state $(q, \varnothing, b_{\max})$ yields

$$\mathrm{Exec}_{\mathcal{M}}(P, q; \xi) = (\hat{y},\ z,\ c), \quad (13)$$

the committed answer $\hat{y}$ (Eq. 3), the trace z (the records $(\iota_o, V(o), y, \mathrm{ok}, \gamma, c)$ of every executed operation and every branch decision), and the total cost c. Correctness is $R(P, q) = \mathbb{1}[\hat{y} \equiv y^\star(q)]$ under the benchmark's scorer (Appendix F). The same $\equiv$ compares candidates inside a protocol.

C.2 Protocols as Programs

A *reasoning protocol* is a quadruple

$$P = (O, V, C, T), \qquad V : O \to 2^{\mathrm{Ref}}, \qquad T = (\iota_1, \ldots, \iota_m), \tag{14}$$

where O is a finite set of typed operations with parameters (Table 3 in Appendix B), V assigns each operation its reads, C is a control program over O from the grammar of Eq. (1), and T is a priority list of identifiers from which the answer is committed (JSON syntax in Appendix B). Operations are typed by what they produce: *create* a candidate (`solve`, `program`, `revise`), *structure* the problem (`decompose`, `build_check`), *evaluate* a candidate (`run_check`, `verify`) or *select* (`aggregate`, `select`). An evaluating operation attaches its verdict to the first candidate it reads.

C.3 Composition and Conditions

The grammar of Eq. (1) (Section 3) generates control programs with $n \leq 6$ branches per Par, $k \leq 3$ iterations per Rep and at most $L = 24$ nodes per program. Conditions come from the closed grammar of predicates on the runtime state in Eq. (2). Here, $I \subseteq \mathrm{Id}$ is a set of candidates compared under $\equiv$; $\mathrm{passed}/\mathrm{failed}$ and $\mathrm{exec_ok}/\mathrm{exec_err}$ read the ok flag of a check, verdict or program; γ_ι is a verifier's confidence; b is the remaining budget; and t is the iteration. Every predicate is a function $g : \mathcal{S} \to \{0, 1\}$ of the state alone. The grammar has no access to $y^\star$, and a static check rejects reads or conditions on identifiers not defined earlier in program order.

Two semantic points matter for learning. Par_ι runs its branches from a snapshot $\mathcal{A}_0$ of the state, so no branch can read a sibling's artifacts. It then merges the results into a named group,

$$\mathcal{A}' = \mathcal{A}_0 \cup \textstyle\bigcup_i \big(\mathcal{A}_i \setminus \mathcal{A}_0\big), \qquad \mathcal{A}'[\iota] = \mathrm{group}\big(\{\iota_j \in \mathcal{A}_i \setminus \mathcal{A}_0 : \ \iota_j \text{ a candidate}\}\big), \tag{15}$$

so a later read of ι`.answers` sees all branch answers, and a read of ι sees all branch texts. If runs one branch (identifiers may repeat across the two). $\mathrm{Rep}_k(C, g)$ repeats C while $\neg g$, $t < k$ and $b > 0$. $\mathrm{Commit}(T)$ commits the first listed artifact that exists and has an answer (Eq. (3)). This makes conditional sub-protocols composable: an identifier defined inside an untaken branch is absent, and the next entry of T is committed. On code tasks the answer is a program: `program` coincides with `solve`, agreement and votes compare programs by their outputs on the public examples, `run_check`[exec] runs the candidate on those examples at no LLM cost, and hidden tests are seen only by the final scorer (Appendix B).

C.4 Execution Traces as Failure Evidence

The digest and node coverage are defined in Eqs. (4) and (5) (Section 4.1). Coverage also lists commit-target counts. Each history entry gives the patch's $(n_{\mathrm{fix}}, n_{\mathrm{brk}}, p)$, the number of problems it changed and the verdict.

C.5 Patches

A patch is a tuple

$$\delta = (\ell,\ d,\ g,\ Q,\ V_Q), \tag{16}$$

with a location ℓ (an operation or Par identifier, or the program's end), an optional diagnostic d (typically `build_check` then `run_check`, or `verify`), a trigger condition g from Eq. (2) that may reference d, a sub-protocol Q from Eq. (1) with fresh identifiers, and the reads V_Q of its operations. Applying a patch rewrites the control program and the commit list as in Eq. (6). The diagnostic runs whenever ℓ is reached; Q runs only when g holds. The answer from Q takes priority, with the previous commit target as fallback. This rule applies at an operation (`insert_after`) and at the program's end (`wrap_root`), plus three degenerate patches: `replace` (swap one operation in place), `set_commit` (re-point T) and `set_instruction`. The last writes the *instruction slot*, at most 800 characters appended to the fixed prompt, for one operation, a Par group or every answer-producing operation. A persona, answer-format rule or checklist is thus a parameter of a named operation: it costs no calls and passes the same test as a structural change. Before execution, static validation checks grammar, scoping, bounds, and whether reads and conditions reference identifiers defined earlier in program order. A second static check rejects *dead computation*: a new candidate artifact that nothing commits, reads or conditions on can never reach the committed answer (Eq. 10, Appendix B). Invalid and dead patches are reported back to the synthesizer.

C.6 Objective and Acceptance

Eq. (8) defines the learning problem with a budget on *mean* calls (a conditional sub-protocol is charged only where it fires) and a size bound. On the search set, a patch's accuracy change is exactly fixed minus broken,

$$\mathrm{Acc}_{\mathcal{D}_\mathrm{s}}(P \oplus \delta) - \mathrm{Acc}_{\mathcal{D}_\mathrm{s}}(P) = \frac{n_{\mathrm{fix}}(\delta) - n_{\mathrm{brk}}(\delta)}{|\mathcal{D}_\mathrm{s}|}, \tag{17}$$

but net gain alone is too permissive. A *neutral* patch, whose every flipped outcome is a coin toss, passes $n_{\mathrm{fix}} - n_{\mathrm{brk}} > 0$ with probability near one half. A search over $R \times K$ candidates therefore accumulates changes that fit only the search set. We gate acceptance with a one-sided sign test on the flipped problems,

$$p(\delta) = \Pr\left[X \geq n_{\mathrm{fix}}\right] = 2^{-n} \textstyle\sum_{k=n_{\mathrm{fix}}}^{n} \binom{n}{k}, \qquad X \sim \mathrm{Binomial}\big(n, \tfrac{1}{2}\big), \quad n = n_{\mathrm{fix}} + n_{\mathrm{brk}}, \tag{18}$$

the probability that a neutral patch shows at least the observed number of fixes. Eq. (9) defines the *sign-test gate*, with $\alpha = 0.2$ in all experiments. This gate rejects $(n_{\mathrm{fix}}, n_{\mathrm{brk}}) = (2, 0)$ or $(3, 1)$ and accepts $(3, 0)$ or $(6, 2)$. Counts use the full search set, and the comparison is paired. During search, executor calls are cached on (model, prompt, temperature), so the unchanged prefix of $P \oplus \delta$ reproduces the artifacts of P. The round's best accepted patch (search-set accuracy, then cost) becomes the current protocol (Algorithm 1).

C.7 Optimization Details

Data discipline. Each benchmark has three disjoint pools: a search set $\mathcal{D}_\mathrm{s}$ for proposing and accepting patches, a validation set $\mathcal{D}_\mathrm{v}$ used only in the final selection, and the test set $\mathcal{D}_\mathrm{t}$ shared with every baseline. $\mathcal{D}_\mathrm{s}$ and $\mathcal{D}_\mathrm{v}$ come from the benchmark's training material, from a related source disjoint from the test items or, for TheoremQA and OlympiadBench, from a seeded subset held out from every method's test set. MuSR is grouped by story before partitioning, so $\mathcal{D}_\mathrm{s}$, $\mathcal{D}_\mathrm{v}$ and $\mathcal{D}_\mathrm{t}$ share no stories (Appendix G). Labels are used on $\mathcal{D}_\mathrm{s}$ (digests, coverage, $n_{\mathrm{fix}}, n_{\mathrm{brk}}$) and $\mathcal{D}_\mathrm{v}$ (selection). After the protocol is frozen, the test set is run once per seed with the call cache disabled.

Seeds and root. The library contains ten seed types (Appendix D): the encodings of Table 4, three-sample self-consistency, the agreement-gated cascade, and two check-and-revise programs, *exec-fix* (solve, executable check, revise on failure) and *PoT-check*. The benchmark-specific subsets have six seeds for multiple choice, nine for mathematics and eight for code. The root P_0 is the seed with the highest search-set accuracy among those costing at most $B/2$, leaving room for learned structure within the budget. Every seed enters the archive and competes at selection, so a seed that is already best is returned unchanged.

Pooled selection. After R rounds, the seeds and accepted programs under the budget are eligible for selection. Rejected candidates are excluded: a rejected patch that tops the search set is a winner's-curse pick that a small validation set cannot correct. The six eligible programs with the highest search-set accuracy are executed on $\mathcal{D}_\mathrm{v}$. The final protocol maximises accuracy pooled over search and validation items,

$$P^\star = \operatorname*{arg\,max}_{P \in \mathcal{H}_B} \Big(\mathrm{Acc}_{\mathrm{s+v}}(P),\ -\hat{c}_{\mathcal{D}_\mathrm{v}}(P)\Big), \qquad \mathrm{Acc}_{\mathrm{s+v}}(P) = \frac{|\mathcal{D}_\mathrm{s}|\,\mathrm{Acc}_{\mathcal{D}_\mathrm{s}}(P) + |\mathcal{D}_\mathrm{v}|\,\mathrm{Acc}_{\mathcal{D}_\mathrm{v}}(P)}{|\mathcal{D}_\mathrm{s}| + |\mathcal{D}_\mathrm{v}|}, \tag{19}$$

with ties broken by cost. The validation items alone are too few to rank programs that differ by a point or two. Because no patch was fitted to these items, however, they weigh against a chain that fits the search set.

Cost accounting. Offline learning cost and the online cost of the learned protocol are reported separately,

$$C_{\mathrm{off}} = N_{\mathrm{exec}} \cdot \bar{\tau}_{\mathrm{exec}} + R \cdot \bar{\tau}_{\mathrm{syn}}, \qquad c_{\mathrm{on}}(P^\star) = \mathbb{E}_q\big[c(P^\star, q)\big] \leq B, \tag{20}$$

where N_{exec} counts uncached executor calls during search and $\bar{\tau}$ denotes mean tokens per call. The synthesizer is called once per round; C_{off} is paid once per benchmark and c_{on} per deployed problem.

REPROT settings. We use a budget of $B = 6$ mean calls per problem, a hard cap of 30 calls, $L = 24$, $R = 10$ rounds of $K = 5$ patches, and $\alpha = 0.2$. The synthesizer is `gpt-5.6-sol` at temperature 0.7, and the executor is the backbone. We use the seed library in Appendix D, with exec-fix on mathematics and code and PoT and PoT-check on mathematics. Search/validation sizes $|\mathcal{D}_s|/|\mathcal{D}_v|$ are: MMLU-Pro and SuperGPQA $700/300$; GPQA $300/131$; MATH-500 $350/150$; OlympiadBench $178/96$; TheoremQA $160/87$; LiveCodeBench $148/80$. MuSR uses the story-disjoint pools described in Appendix G. Six programs enter validation; the final choice maximizes pooled search–validation accuracy, with ties broken by cost (Eq. 19).

D ENCODINGS OF PRIOR METHODS

Table 4 gives the six encodings of Section 3.3 in shorthand (reads in parentheses). The seed library contains the JSON programs of Table 4, exactly as executed (SC-3 is SC-5 with three branches). These are followed by the hand-written agreement-gated cascade (two calls when the two initial solutions agree, seven otherwise) and the two check-and-revise seeds, exec-fix (2–3 calls on mathematics, 1–2 on code) and PoT-check (mathematics).

Protocol Examples

```
CoT:
  seq{
    s1 = solve(reads=[problem]) {"style":"cot"}
    commit s1
  }

SC-5:
  seq{
    par sc {
      s1..s5 = solve(reads=[problem])
        {"style":"cot","temperature":0.7}
    }
    agg = aggregate(reads=[sc.answers]) {"rule":"vote"}
    commit agg
  }

PHP:
  seq{
    s1 = solve(reads=[problem]) {"style":"cot"}
    repeat(max=3) until {"agree":["s1","h1"]} {
      h1 = solve(reads=[problem, s1.answer]) {"style":"hint"}
    }
    commit h1 > s1
  }

Debate:
  seq{
    par r1 {
      a1,a2,a3 = solve(reads=[problem])
        {"style":"cot","temperature":1.0}
    }
    par r2 {
      b1 = solve(reads=[problem,a2,a3])
        {"style":"critique","temperature":1}
      b2 = solve(reads=[problem,a1,a3])
        {"style":"critique","temperature":1}
      b3 = solve(reads=[problem,a1,a2])
        {"style":"critique","temperature":1}
    }
    agg = aggregate(reads=[r2.answers]) {"rule":"vote"}
    commit agg
  }

PoT:
  seq{
    p1 = program(reads=[problem])
    commit p1
  }

Star:
  seq{
```

```
    hub = solve(reads=[problem])
      {"style":"cot","temperature":1.0}
    par spokes {
      k1..k4 = solve(reads=[problem, hub])
        {"style":"critique","temperature":1.0}
    }
    agg = aggregate(reads=[problem, hub, spokes]) {"rule":"llm"}
    commit agg
  }

Cascade:
  seq{
    par st1 {
      c1 = solve(reads=[problem]) {"style":"cot"}
      c2 = solve(reads=[problem])
        {"style":"cot","temperature":0.0}
    }
    if {"agree":["c1","c2"]} {
      agg1 = aggregate(reads=[st1.answers]) {"rule":"vote"}
    }
    else {
      par st2 {
        d1,d2,d3 = solve(reads=[problem])
          {"style":"cot","temperature":1.0}
      }
      v1 = verify(reads=[problem, c1.answer]) {"k":1}
      v2 = verify(reads=[problem, c2.answer]) {"k":1}
      agg1 = aggregate(reads=[st1.answers, st2.answers])
        {"rule":"evidence"}
    }
    commit agg1
  }

ExecFix:
  seq{
    s1 = solve(reads=[problem]) {"style":"cot"}
    c1 = run_check(reads=[s1]) {"mode":"exec"}
    if {"failed":"c1"} {
      r1 = revise(reads=[problem, s1, c1.evidence])
    }
    commit r1 > s1
  }

PoTCheck:
  seq{
    p1 = program(reads=[problem])
    b1 = build_check(reads=[problem])
    c1 = run_check(reads=[problem, p1, b1]) {"mode":"exec"}
    if {"failed":"c1"} {
      r1 = revise(reads=[problem, p1, c1.evidence])
    }
    commit r1 > p1
  }
```

Method	Protocol	Calls
CoT	`s1 = solve`[cot](problem); `commit s1`	1
Self-consistency	$\mathrm{Par}_{\mathrm{sc}}$: $\mathtt{s}_i$ = `solve`[cot, τ=0.7](problem), i=1..5; `agg = aggregate`[vote](sc.answers); `commit agg`	5
PHP	`s1 = solve`[cot](problem); Rep_3 until agree(**s1**, **h1**): `h1 = solve`[hint](problem, s1.answer); `commit h1 > s1`	2–4
Debate	$\mathrm{Par}_{\mathrm{r1}}$: $\mathtt{a}_i$ = `solve`[cot, τ=1](problem), i=1..3; $\mathrm{Par}_{\mathrm{r2}}$: $\mathtt{b}_i$ = `solve`[critique, τ=1](problem, $\{\mathtt{a}_j\}_{j\neq i}$); `agg = aggregate`[vote](r2.answers); `commit agg`	6
Star topology	`hub = solve`[cot, τ=1](problem); $\mathrm{Par}_{\mathrm{sp}}$: $\mathtt{k}_i$ = `solve`[critique, τ=1](problem, hub), i=1..4; `agg = aggregate`[llm](problem, hub, sp); `commit agg`	6
Program-of-thoughts	`p1 = program`(problem); `commit p1`	1

Table 4: Six prior methods (Kojima et al., 2022; Wang et al., 2022; Zheng et al., 2023; Du et al., 2023; Qian et al., 2025; Chen et al., 2022) as reasoning protocols (reads in parentheses). Appendix D gives the executed JSON and the remaining seeds (SC-3, cascade, exec-fix, PoT-check).

E PROMPTS

This appendix reproduces the code release's prompt templates: the benchmark task strings shared by every method (§E.1), the executor's operation prompts (§E.2), the synthesizer's system prompt

and message templates (§E.3), and one complete synthesizer response with the gate's verdict on each patch (§E.4). Text in `{braces}` marks a slot filled at run time; other text is literal within the applicable template version. Algorithm 1 (Section 4.5) gives the search loop that issues these calls.

E.1 Benchmark Task Strings

The shared adapter turns each test item into one task string. This fills the `{task}` slot of every prompt below and is passed unchanged to every baseline.

Task strings (one per benchmark family)

```
GPQA, SuperGPQA, MMLU-Pro:
{question}
Option A: {option A}
Option B: {option B}
...
Conclude with "Answer: <letter>".

MuSR:
{narrative}

{question}
Option A: {option A}
Option B: {option B}
...
Conclude with "Answer: <letter>".

MATH-500:
{problem}

Give the final answer in LaTeX inside \boxed{}.

OlympiadBench:
{question}

Give the final answer in LaTeX inside \boxed{}. Express the answer in {unit}.
                                                (unit sentence only if the item has a
   unit)
TheoremQA:
{question}

{Answer with a single number. | Answer True or False. | Answer with the option letter. |
 Answer with a comma-separated list of integers. | Answer with a comma-separated list of
   numbers.}
End with 'Final answer: <answer>'.                  (hint chosen by the item's answer
   type)

LiveCodeBench:
{question content}

Example input:
{public test 1 input}
Example output:
{public test 1 output}
Example input:
{public test 2 input}
Example output:
{public test 2 output}

{Write a complete Python program that reads from standard input and writes the answer to
 standard output.
 | Complete the following starter code; return the answer from the method (do not print).
   ```python
   {starter code}
   ```}
Put the final code in a single ```python block.
```

The calibration pools use the same answer lines on their own items (Appendix G): GSM8K questions end with the MATH-500 line, or with `Answer with a single number. End with 'Final answer: <answer>'.` for TheoremQA; HumanEval items read `Complete the following Python function so that it satisfies its docstring. Reply with one ```python code block containing the`

`complete function.` followed by the function stub. Baseline prompt sets that hard-code a benchmark receive only mechanical changes: the four-option wording (`A, B, C and D`, `4 answers`, `4 options`) becomes the benchmark's letter set and count; the GSM8K answer example `The answer is 140` becomes `The answer is \boxed{140}` (`Final answer: 140` on TheoremQA); and the HumanEval wording `a function signature and its docstring` becomes `a programming problem (statement, examples and the required output format).`

E.2 Operation Prompts

Every LLM-calling operation sends one system and one user message. The `{fmt}` slot is the benchmark's answer line: `Conclude with a final line exactly of the form: Answer: <letter>` (multiple choice), `Give the final answer in LaTeX inside \boxed{} on the last line.` (MATH-500, OlympiadBench), `End with a line exactly of the form: Final answer: <answer>` (TheoremQA) or `Put the complete solution in a single ```python code block at the end.` (LiveCodeBench). The `{context}` slot renders exactly the operation's reads, one block per read, and nothing else from the state:

Rendering of reads into context

```
"<id>.answer"    ->  [<id>] answer: {answer or (none)}
"<par>.answers"  ->  [<par>] candidate answers: <m1>: {answer}; <m2>: {answer}; ...
"<id>.evidence"  ->  [<id>] evidence: {evidence summary, or the last 800 characters of
  the text}
"<id>"           ->  [<id>] ({kind}):
                     {full output text}        (one block per member for a par group)
"problem"        ->  (the task string is always included; this read adds nothing)

Instruction slot, when the node carries one, appended to the system prompt:
{system prompt}

Additional instructions:
{instruction}
```

Operation prompts

```
solve, style=cot
  system: You are a careful expert problem solver. Reason step by step, then commit to
   one final
          answer.
  user:   Problem:
          {task}

          {fmt}

solve, style=direct
  system: Answer the problem as accurately as possible.
  user:   (as cot)

solve, style=hint
  system: You are a careful expert problem solver. A previous attempt is provided as a
   hint.
          Re-derive the answer from the problem; keep the hint only if your own
   derivation
          confirms it.
  user:   Problem:
          {task}

          Previous attempt(s):
          {context}

          {fmt}

solve, style=critique
  system: You are a careful expert problem solver. Other solvers' solutions are provided.
    Examine
```

```
          them critically -- they may be wrong -- then give your own updated, fully
   justified
          answer.
  user:   Problem:
          {task}

          Other solutions:
          {context}

          {fmt}

solve, style=subproblems
  system: You are a careful expert problem solver. A decomposition into sub-questions is
   provided.
          Solve the sub-questions in order, then combine them into the final answer.
  user:   Problem:
          {task}

          Decomposition:
          {context}

          {fmt}

program                          (on LiveCodeBench, program = solve with the same style)
  system: Solve the problem by writing a self-contained Python program. Do all
   computation in
          code, using sympy / fractions for EXACT arithmetic (never floats for exact
   answers;
          simplify with sympy.nsimplify and sympy.simplify). The program must print ONLY
   the final
          answer on the last line: an integer, an exact LaTeX expression produced by
          sympy.latex(...) (e.g. \frac{3}{4}, 4 + 4\sqrt{2}), or the option letter. Reply
    with one
          ```python block.
  user:   Problem:
          {task}

          Context:                       (only if the node has reads)
          {context}
  -> the last ```python block runs in a subprocess (10 s); the last printed line is the
   answer.

decompose
  system: Break the problem into 2-4 concrete sub-questions whose answers determine the
   final
          answer. Do NOT solve them. Output a numbered list, one sub-question per line.
  user:   Problem:
          {task}

build_check
  system (math): List the concrete conditions a correct final answer to this problem must
          satisfy: equations it must satisfy when substituted back, domain/sign/
   integrality
          constraints, units, edge cases. Do NOT solve the problem. Numbered list, 2-5
   checkable
          items.
  system (multiple choice): For this multiple-choice problem, write the concrete,
   checkable facts
          or computations that decide between the options (e.g. 'the value of X must
   equal ...',
          'option letters whose claim contradicts the given data'). Do NOT pick an option
   .
          Numbered list, 2-5 items.
  system (code): For this programming problem, write 3-6 additional concrete test cases
   as lines
          'input -> expected output', covering edge cases (empty/minimal input, maximum
          constraints, duplicates/ties, negative or boundary values). Work each expected
   output
          out by hand from the problem statement. Do NOT write the solution.
  user:   Problem:
          {task}

run_check, mode=llm
  system: You are a strict checker. Verify the candidate answer against each listed
   condition (or
          against the problem directly if no conditions are given) by actually carrying
   out the
```
```

```
          check -- substitute, recompute, compare. For each item write PASS or FAIL with
   a one-line
          justification. Finish with a line exactly 'OVERALL: PASS' or 'OVERALL: FAIL'.
  user:   Problem:
          {task}

          Candidate answer: {candidate}

          Conditions / context:          (only if the node has reads besides the
   candidate)
          {context}

run_check, mode=exec             (math; on LiveCodeBench: no LLM call, public examples
   are run)
  system: Write a self-contained Python program that CHECKS whether the candidate answer
          satisfies the problem's conditions (substitute it back, verify constraints
          numerically/symbolically). It must print CHECK_PASS if the candidate is
   consistent with
          every condition and CHECK_FAIL otherwise, followed by a one-line reason. Reply
   with one
          ```python block.
  user:   Problem:
          {task}

          Candidate answer: {candidate}

          Context:
          {context}

verify                           (k samples; temperature 0.7 if k > 1, else 0)
  system: You are a meticulous verifier. Independently determine whether the proposed
   answer is
          correct: re-derive the answer from the problem WITHOUT assuming the proposal is
    right.
          End with two lines:
          VERDICT: YES or NO
          CONFIDENCE: <0-100>
  user:   Problem:
          {task}

          Proposed answer: {candidate}

revise
  system: You are a careful expert problem solver. A previous solution and evidence about
    its
          problems are provided. Fix the identified issues (re-derive the affected steps;
    do not
          merely restate), then give the corrected final answer. If the evidence shows
   the
          previous answer is actually correct, keep it.
  user:   Problem:
          {task}

          {context}

          {fmt}

aggregate, rule=llm              (vote, evidence and select make no call)
  system: You are the final judge. Several candidate solutions are provided. Exactly one
   final
          answer must be committed. Weigh them critically (they may disagree or contain
   errors),
          check against the problem, and commit.
  user:   Problem:
          {task}

          Candidates:
          {context}

          {fmt}
```

Each node specifies its temperature (`0` by default). Output is capped at 1,024 tokens for `decompose` and `build_check`, 4,096 for `program`, `run_check`, `verify` and the judge, and 16,384 for the solving operations.
```

E.3 SYNTHESIZER PROMPTS

The synthesizer receives the grammar below and the current program in both notations, with its search-set accuracy, mean cost, budget and size relative to the 24-node bound. It also receives the node-coverage block (Eq. 5), a failure taxonomy (empty answers; selection failures, where some candidate held the reference answer; generation failures, where none did), ten failure digests, four success digests, and the last twelve patches tried with their gate statistics. On LiveCodeBench, it also receives the code-task note. Sampling uses temperature 0.7 with $8{,}192$ output tokens. A patch rejected by static validation receives one repair call at temperature 0.3 with the same system prompt.

Synthesizer system prompt

```
You are a research engineer improving a *reasoning protocol*: a program that organises
   how a fixed LLM solves a problem. You will see the current program, its accuracy/cost,
   and execution traces of failures and successes on a training set. Propose patches that
   add a NEW computation (diagnostic -> trigger condition -> sub-protocol) at a specific
   point, so that the protocol fixes a class of failures WITHOUT breaking the successes
   and within the call budget.

Principles:
- A patch is valuable only if its trigger condition identifies the states where the new
   computation helps; a sub-protocol that runs on every problem must itself be worth its
   cost on every problem.
- EFFECT SIZE MATTERS: acceptance is a sign test on fixed vs broken over the whole search
    set, so a patch whose condition fires on a handful of problems cannot be accepted. Use
    the NODE COVERAGE numbers: put new computation where many failures live (e.g. the
   agreeing-but-wrong majority, not only the rare disagreement branch), and spend the
   remaining budget on computation that changes candidates on a large fraction of problems
    (more independent solutions, program execution, hint/critique rounds) when the budget
   allows it.
- Prefer computations that can CHANGE or CREATE candidates (check-and-revise, program
   execution, decomposition, a differently-conditioned solve), not only select among
   existing ones, when failures show all candidates wrong.
- Prefer isolated verification / independent branches when failures show agents copying
   each other's errors.
- When many failures are systematic (wrong answer FORM, missing cases, a recurring
   misreading, a domain-specific pitfall), an instruction slot is the cheapest fix: it
   costs no calls and applies to every problem. Use "where":"*" so that every answer-
   producing op follows it. Good instructions are concrete: the exact output contract of
   this benchmark (what goes inside the box, how several values / ordered pairs /
   intervals / units / letters are written), a short checklist of the pitfalls seen in the
    FAILURE TAXONOMY, or a solving discipline ("enumerate all cases", "verify by
   substitution", "read the question stem twice"). Prefer one well-written instruction
   over several vague ones, and keep the program structure unless the taxonomy shows
   selection failures (then improve adjudication / verification) or generation failures on
    computable problems (then add program execution / executable checks).
- Conditions may only use runtime-observable state (agreement, check results, execution
   status, confidence, budget). Never assume access to the correct answer.
- Keep programs small; respect the budget; every op must read only what it needs.
Output ONLY a JSON array of patches (2-5 items), each with a one-sentence rationale.
```

Synthesizer user message

```
{GRAMMAR: verbatim, Listing "Grammar given to the synthesizer"}

{CODE NOTE: code tasks only}
CURRENT PROGRAM ({benchmark}; search-set accuracy {acc}%, {calls} calls/question, budget
   {B} calls/question):
{program, pretty-printed}

PROGRAM SIZE: {size} nodes; hard limit 24 (every op/seq/par/if/repeat/commit node counts)
   , so a
patch may add at most {24 - size} nodes. A larger patch is rejected without evaluation --
    to add more
structure, use "replace" to substitute an existing op or sub-block instead of inserting
   next to it.

{NODE COVERAGE block}

{FAILURE TAXONOMY block}
```

```
JSON:
{program, JSON}

FAILURE TRACES ({n_fail} failures total; {10} shown):
{10 failure digests}

SUCCESS TRACES (do not break these):
{4 success digests}

PATCHES ALREADY TRIED (do not repeat; net gain on the search set shown):
{last 12 history lines}

Propose {K} patches as a JSON array. Each must be syntactically valid under the grammar,
  use fresh ids for new ops, read only what the op needs, and keep expected calls/
  question <= {B}.
```

Grammar given to the synthesizer (the GRAMMAR slot)

```
PROGRAM GRAMMAR (JSON):
  {"type":"op","id":ID,"op":OP,"reads":[REF,...], ...params}
  {"type":"seq","body":[P,...]}        {"type":"par","id":ID,"body":[P,...]}   (branches
  cannot see each other)
  {"type":"if","cond":COND,"then":P,"else":P|null}
  {"type":"repeat","max":1..3,"until":COND,"body":P}
  {"type":"commit","ref":ID | [ID, ID, ...]}   final answer = first listed artifact that
  exists; ALWAYS list a
                                                fallback after any id defined inside an if
  -branch, e.g. ["r2","p1"]
  A "par" MUST carry an "id"; "<parId>.answers" then exposes all branch answers to later
  ops.

OPS and their params:
  solve        style: cot | direct | hint (reads a previous answer; re-derives, keeps
  only if confirmed)
                    | critique (reads other solvers' full solutions; gives updated
  answer)
                    | subproblems (reads a decompose artifact; solves the parts then
  combines)
               temperature: 0.0 (default) or 0.7/1.0 for diverse sampling
  program      writes+executes Python; answer = printed result; ok = executed without
  error   (math/code)
  decompose    -> sub-questions (artifact kind subproblems); read it via "<id>"
  build_check  -> a checklist of concrete conditions a correct answer must satisfy (kind
  checkspec)
  run_check    mode: llm (carries out each condition against the candidate) | exec (
  writes+runs a checker
               program; prints CHECK_PASS/CHECK_FAIL). Candidate = first solution-like
  read; conditions =
               a checkspec read. Sets ok=True/False and evidence.
  verify       k: 1|3 independent isolated verifications of the candidate answer (sees
  ONLY problem +
               candidate answer). Sets conf 0-100 and ok.
  revise       reads a solution (full text) + evidence (".evidence" of a run_check/verify
  /program);
               produces a corrected solution
  aggregate    rule: vote (majority over candidate answers, ties->earliest) | evidence (
  prefer candidates
               whose check passed, then vote) | llm (a judge reads the visible context
  and commits)
  select       deterministic: first candidate whose attached check passed, else vote

READS (visibility -- the ONLY information the op sees besides the problem):
  "problem" | "<id>" (full text of that artifact) | "<id>.answer" | "<id>.evidence" | "<
  parId>.answers"

CONDITIONS (runtime state only):
  {"agree":[ids]} {"disagree":[ids]} {"n_distinct":[ids],"ge":k} {"passed":id} {"failed":
  id}
  {"exec_ok":id} {"exec_err":id} {"empty":id} {"conf_lt":[id,x]} {"conf_ge":[id,x]} {"
  budget_ge":n}
  {"and":[..]} {"or":[..]} {"not":C} true

PATCH KINDS:
  {"kind":"insert_after","where":ID,"diagnostic":P|null,"cond":COND|null,"sub":P,"commit
  ":ID|[ID,..]|null,"rationale":"..."}
```

```
      after node ID (any op id or par-group id, also inside if-branches / repeat bodies;
   the block runs
      right after it, in place): run diagnostic (if any), then if cond holds run sub;
   optionally re-point the commit
   {"kind":"wrap_root","diagnostic":P|null,"cond":COND|null,"sub":P,"commit":ID|null,"
   rationale":"..."}
      same, appended at the end of the program (before commit)
   {"kind":"replace","where":ID,"node":P,"rationale":"..."}     replace an op in place (
   keep the id)
   {"kind":"set_commit","ref":ID,"rationale":"..."}              change which artifact is
   committed
   {"kind":"set_instruction","where":ID|"*","instruction":"...","rationale":"..."}
      set the INSTRUCTION SLOT (<= 800 chars, appended to the op's prompt) of one LLM-
   calling op, of every
      LLM op inside a par group, or ("*") of every answer-producing solve/program/revise
   op: a persona,
      precise answer-format rules (what goes inside the box / how lists, units, forms are
    written), a
      checklist of pitfalls, a solving strategy. Vote/evidence aggregates, select and
   exec checks have no
      prompt. Any op node may also carry "instruction" when created inside a patch.
RULE: a new solution/aggregate artifact must be able to reach the answer -- list it in "
   commit" (highest
priority first; earlier targets remain fallbacks), read it from a later op, or use it in
   a condition.
Otherwise the patch is rejected statically as dead computation.
ACCEPTANCE: a patch is kept only if it fixes clearly more problems than it breaks (one-
   sided sign test on
fixed vs broken, p <= 0.2) within budget -- so target failure modes that recur across
   many problems, and
gate new computation with conditions so that already-correct problems are left untouched.
```

Blocks filled into the user message

```
{NODE COVERAGE block}
NODE COVERAGE on the search set ({n} problems; a patch must change outcomes on many
   problems to pass the sign test):
  {node} [{op}]: ran on {k}/{n}, accuracy there {acc}%
  if {cond}: taken {k}/{n} (acc when taken {acc}%), skipped {k'} (acc when skipped {acc
  '}%)
  committed {ref}: {k}/{n} (acc {acc}%)

{FAILURE TAXONOMY block}
FAILURE TAXONOMY of the current program ({n_fail} failures / {n} problems):
  empty final answer (extraction/format failure): {count}  e.g. final='...' gold='...';
   ...
  selection failure (a candidate had the gold answer but was not committed): {count}  e.g
   . ...
  generation failure (no candidate had the gold answer; compare final vs gold for FORMAT
   mismatches -- a wrong form of a right value is fixed by an answer-format instruction, a
    wrong value needs new computation): {count}  e.g. ...

{failure / success digest}, one per problem
PROBLEM: {problem; if longer than 2000 chars: first 500 ...[m chars omitted]... last
   1500}
GOLD: {gold}   FINAL: {committed answer or (empty)}   CORRECT|WRONG   calls={calls}
  {node} [{op}] reads={reads} -> answer={answer or (none)} ok={ok} conf={conf}  ev: {
   evidence, 120 chars}
  if {cond} -> taken|skipped
  {node}: skipped ({reason})
  commit {ref}

{history line}, one per patch tried
r{round}p{j}: net={+n} (fixed {n_fix}, broken {n_brk}, p={p}, changed outcome/cost on {k
   }/{n} problems), calls {c}, ACCEPTED|rejected :: {rationale}
r{round}p{j}: INVALID ({validator error}) :: {patch JSON}
r{round}p{j}: RUNTIME ERROR ({error}) :: {patch JSON}

{CODE NOTE} (LiveCodeBench only)
TASK TYPE: code generation -- the final answer is a complete Python program. On this task
    run_check with mode "exec" executes the candidate program on the problem's PUBLIC
   examples and reports the failing cases (costs no LLM call); `program` behaves like
   solve; aggregate/agree/n_distinct compare programs by their outputs on the public
   examples.
```

Repair message (after a static rejection)

```
{GRAMMAR: verbatim, Listing "Grammar given to the synthesizer"}

CURRENT PROGRAM (size {size} nodes, hard limit 24; budget {B} calls/question):
{program, pretty-printed}

JSON:
{program, JSON}

YOUR PATCH:
{patch, JSON}

It was REJECTED before evaluation: {validator error}

Return ONE corrected patch that keeps the same idea and fixes exactly this error (e.g.
   drop or merge nodes to stay
within the size limit, replace an existing op instead of inserting, define an artifact
   before it is read, commit or
read every new artifact). Output ONLY the JSON object.
```

E.4 A Complete Synthesizer Response

The complete response below comes from the first round of the MuSR search whose selected program appears in Table 1. The current program was the SC-3 seed (three CoT solvers at temperature 0.7 and a vote; 68.8% search accuracy, 3.0 calls). Table 5 gives the gate's verdict on each of the five patches. Four passed. The search chose the most accurate patch at the lowest cost: patch 0, a zero-call instruction slot. No patch in rounds 2–10 passed the gate, and validation selected the same program. Patch 4 reaches the same search accuracy at twice the calls; patch 1 changed no problem's outcome.

Patch	Kind	Fixed	Broken	p	Search acc.	Calls	Verdict
0	`set_instruction`	40	20	0.007	77.5	3.00	accepted, selected
1	`insert_after`	0	0	1.000	68.8	4.09	rejected
2	`insert_after`	16	3	0.002	74.5	4.19	accepted
3	`insert_after`	13	6	0.084	71.9	5.14	accepted
4	`insert_after`	32	12	0.002	77.5	6.00	accepted

Table 5: Gate verdicts on the MuSR round-1 patches (search split, α=0.2). Fixed and broken count problems whose outcome changed relative to the SC-3 program.

Synthesizer response, MuSR round 1 (verbatim JSON, re-indented)

```
[
 {
  "kind": "set_instruction",
  "where": "*",
  "instruction": "Track the story mechanically before answering. For object-location
   questions, distinguish the object's true final location from the queried person's
   belief: record each move, who observed it, and where that person would search first;
   unseen moves do not update belief. For allocation questions, score every person against
    every task using explicit strengths and weaknesses, then check each complete option
   and assign everyone exactly once. For culprit questions, compare motive, means,
   opportunity, access, and contradictions for every suspect rather than following the
   most vivid accusation. Re-read the exact question stem, verify the chosen option
   against the story, and conclude only with Answer: <letter>.",
  "rationale": "A benchmark-wide state-tracking and option-checking instruction cheaply
   targets the recurring false-belief, allocation, and salience errors visible in
   unanimous failures without adding calls."
 },
 {
  "kind": "insert_after",
  "where": "agg",
  "diagnostic": {
   "type": "op",
   "id": "v1",
```

```
    "op": "verify",
    "reads": [
     "agg.answer"
    ],
    "k": 1,
    "instruction": "Independently reconstruct only the facts needed by the question,
    paying special attention to what the queried person observed, the final chronology, and
     whether the selected option satisfies every role or suspect constraint; reject the
    candidate if any concrete fact conflicts."
   },
   "cond": {
    "failed": "v1"
   },
   "sub": {
    "type": "op",
    "id": "r1",
    "op": "revise",
    "reads": [
     "agg",
     "v1.evidence"
    ],
    "instruction": "Correct the answer using the verifier's concrete conflict, rebuild the
     relevant timeline or option comparison from the story, and output Answer: <letter>."
   },
   "commit": [
    "r1",
    "agg"
   ],
   "rationale": "An isolated verification runs on the high-coverage majority answer and
    triggers candidate-changing revision only when it finds a concrete inconsistency, using
     at most two additional calls."
  },
  {
   "kind": "insert_after",
   "where": "agg",
   "diagnostic": {
    "type": "op",
    "id": "challenger1",
    "op": "solve",
    "reads": [
     "problem"
    ],
    "style": "cot",
    "temperature": 0.0,
    "instruction": "Solve independently from scratch using a structured table: for
    location questions track true location, each person's observations, and each person's
    resulting belief after every event; for allocations compare every option role by role;
    for culprit questions compare every suspect on motive, means, opportunity, and
    contradictions. Do not follow narrative emphasis without checking the exact question."
   },
   "cond": {
    "disagree": [
     "agg",
     "challenger1"
    ]
   },
   "sub": {
    "type": "op",
    "id": "crit1",
    "op": "solve",
    "reads": [
     "s1",
     "s2",
     "s3",
     "challenger1"
    ],
    "style": "critique",
    "temperature": 0.0,
    "instruction": "Adjudicate the disagreement by checking each proposed answer directly
    against the event chronology and all option constraints; do not count duplicated
    reasoning as independent evidence, and return Answer: <letter>."
   },
   "commit": [
    "crit1",
    "agg"
   ],
```

```
  "rationale": "A differently conditioned independent solution can expose correlated
   unanimous errors, while the critique call is gated on an observable disagreement and
   the original vote remains the fallback."
 },
 {
  "kind": "insert_after",
  "where": "agg",
  "diagnostic": {
   "type": "seq",
   "body": [
    {
     "type": "op",
     "id": "d1",
     "op": "decompose",
     "reads": [
      "problem"
     ],
     "instruction": "Decompose into the minimal factual questions needed to decide the
   answer: queried perspective and observations, chronological object states, or each
   person's evidence for each task or suspect criterion."
    },
    {
     "type": "op",
     "id": "ds1",
     "op": "solve",
     "reads": [
      "d1"
     ],
     "style": "subproblems",
     "temperature": 0.0,
     "instruction": "Solve each factual sub-question from explicit story evidence,
   combine them only after resolving chronology and perspective, test the result against
   every offered option, and output Answer: <letter>."
    }
   ]
  },
  "cond": {
   "disagree": [
    "agg",
    "ds1"
   ]
  },
  "sub": {
   "type": "op",
   "id": "j1",
   "op": "aggregate",
   "reads": [
    "agg",
    "ds1"
   ],
   "rule": "llm",
   "instruction": "Choose between the original vote and decomposed solution by checking
   their factual claims against the story; prefer an explicit chronology or complete
   option comparison over unsupported consensus, and output Answer: <letter>."
  },
  "commit": [
   "j1",
   "agg"
  ],
  "rationale": "Decomposition creates a structurally different candidate on every problem
    and spends the sixth call on adjudication only when that candidate challenges the
   original majority."
 },
 {
  "kind": "insert_after",
  "where": "agg",
  "diagnostic": {
   "type": "par",
   "id": "alt2",
   "body": [
    {
     "type": "op",
     "id": "epistemic1",
     "op": "solve",
     "reads": [
      "problem"
     ],
     "style": "cot",
```

```
      "temperature": 0.0,
      "instruction": "Build an explicit chronological state ledger. For each relevant
    event record the object's actual state, who could observe it, and each person's belief;
     answer a 'would look' question from that person's belief rather than reality. For
    other question types, build the analogous person-by-role or suspect-by-evidence matrix.
     Conclude with Answer: <letter>."
     },
     {
      "type": "op",
      "id": "eliminate1",
      "op": "solve",
      "reads": [
       "problem"
      ],
      "style": "cot",
      "temperature": 0.7,
      "instruction": "Evaluate every answer option independently and eliminate it only
    with a quoted or precisely paraphrased story fact. Check unseen moves and stale beliefs
    , unique assignment coverage, and all suspects rather than anchoring on vivid details.
    Conclude with Answer: <letter>."
     }
    ]
   },
   "cond": true,
   "sub": {
    "type": "op",
    "id": "judge2",
    "op": "aggregate",
    "reads": [
     "agg",
     "epistemic1",
     "eliminate1"
    ],
    "rule": "llm",
    "instruction": "Adjudicate the three candidates from the underlying story evidence,
    discount shared conclusions that repeat the same unsupported inference, verify
    chronology or every option constraint, and output exactly Answer: <letter>."
   },
   "commit": [
    "judge2",
    "agg"
   ],
   "rationale": "Using all three remaining calls for two complementary candidate-
    generating analyses and evidence-based adjudication targets the large agreeing-but-
    wrong population while retaining the original vote as a fallback."
  }
 ]
```


F Reproduction Protocol and Fidelity Records

Protocol. Every method uses `gpt-5.6-luna` through a commercial API gateway, with up to 16,384 output tokens. ReProt's operation-specific caps are given in Appendix E.2. Temperatures follow each method's paper (single-agent methods 0, topologies and debate 1.0). All methods share an adapter for question text, answer extraction and scoring. They receive byte-identical prompts (multiple-choice stems end with the instruction `Conclude with "Answer: <letter>".`) and use the same scoring rules. Multiple-choice scoring uses letter match. For MATH-500 and OlympiadBench, the mathematics scorer normalises LaTeX in the Hendrycks style: brace-less fractions, `lhs=` prefixes, `\text` units, thousands separators, degree, percent and currency signs, base subscripts, letter answers, `\left`/`\right`, and matrices. It preserves the distinction between ordered tuples and unordered lists, as well as interval bracket types. It then applies an antlr-free symbolic/numeric equivalence check (LaTeX to SymPy, with random-point evaluation for expressions). TheoremQA uses 1% relative tolerance, boolean or list match. LiveCodeBench uses execution pass@1 over public and private tests. The mathematics scorer replaced an earlier one that rejected equivalent answers with different surface forms. Every baseline on these mathematics benchmarks was re-scored from saved predictions with unchanged answer extraction. Empty predictions count as wrong.

We evaluate on MATH-500 (500), LiveCodeBench v6 (175), GPQA-Diamond (198), SuperGPQA-hard (1,000, seed-888 sample) and MMLU-Pro (500). OlympiadBench uses 400 of 581 items and TheoremQA 500 of 747, after holding out seed-42 calibration samples of 181 and 247 items. MuSR

uses 400 test items from a partition grouped by story (Appendix G). Every method, including the baselines, is scored on the same fixed test items. MuSR search, validation and test stories are disjoint. We use seeds 42, 1, 2; for learned-structure methods, each seed applies to training and evaluation. After infinite loops stalled runs, we added a 10-second sandbox timeout to official executors that ran generated code without one (the G-Designer family). This affects wall-clock only.

Caveats. The gateway's content filter blocks 5.4–5.8% of MuSR items for every method and seed. GPQA-Diamond has no training split. Its calibration pool for learned methods and for REPROT consists of SuperGPQA items disjoint from the SuperGPQA evaluation sample. TheoremQA and OlympiadBench use held-out calibration pools for REPROT, whereas the learned-structure baselines keep their stand-in training pools; their scores on these two reduced test sets are recomputed from saved predictions. For MuSR, all methods are evaluated on the fixed, story-disjoint test partition (Appendix G). For LiveCodeBench, methods whose official code assumes HumanEval-style docstrings draw internal tests from the problem's public test cases; hidden tests are used only for the final score.

Method	Source	Protocol as run / principal deviations
Vanilla	reimpl.	Single call with the shared stem; no few-shot examples.
CoT	reimpl.	Two-stage zero-shot CoT (reason, then extract), per Kojima et al. (2022).
Self-Consistency	reimpl.	n=5 at τ=0.7, majority vote over extracted answers (execution agreement on code).
PHP	reimpl.	Zero-shot base; hint accumulates prior answers; stop on two consecutive agreements; cap 4 rounds.
Chain/Star/Tree/Complete/Random	reimpl.	5 agents + aggregator, τ=1.0; the complete graph is two synchronous rounds.
LLM-Debate	official prompts	3 agents, 2 rounds, majority vote; prompts verbatim from the released scripts, with the mathematics template made format-agnostic for non-numeric answers.
LLM-Blender	official repo	5 candidates from the single backbone ranked by the released PairRM; single-model self-ensemble rather than the original multi-model pool.
AutoGen	`pyautogen 0.2.35`	Assistant + user proxy, $\leq$ 3 turns; code execution enabled on LiveCodeBench.
DyLAN	official repo	Official `LLMLP` executed unmodified: LLM ranker, agent selection, importance scores and early stopping active.
GPTSwarm	official repo	Official swarm with REINFORCE edge optimisation on the training pool; node optimisation off as in the released configuration.
G-Designer	official repo	Official training loop on the training pool; loss decreases and encoder parameters move.
AgentPrune	official repo	Official one-shot spatial–temporal pruning trained on the training pool.
CARD	official repo	Official conditional generator with checkpointing; single backbone instead of the heterogeneous pool.

Table 6: Reproduction fidelity. "reimpl." denotes a reimplementation from the paper in this study's harness. "official repo" denotes a pinned clone executed with its own training or inference loop. The code release includes the source, commit, protocol parameters and every deviation.

G CALIBRATION POOLS AND FINAL EVALUATION

Search and validation pools are disjoint from the evaluated items. MuSR stories are also disjoint across all three pools. The multiple-choice pools are seed-shuffled. The mathematics and code pools use in-distribution material disjoint from the test sets.

- **MMLU-Pro**: validation items after the 500 evaluated ones in the official order, seed-shuffled; 700/300 search/validation items.
- **SuperGPQA**: hard-split rows outside the evaluated 1,000, seed-shuffled; 700/300.
- **GPQA-Diamond** (no training split): SuperGPQA rows with exactly four options, excluding the evaluated SuperGPQA items; 300/131.
- **MATH-500**: MATH-train problems of difficulty level 5 in one seed-42 shuffle, a slice disjoint from those reserved for the other mathematics pools; 350/150.
- **LiveCodeBench**: 228 LiveCodeBench problems from earlier releases (contests dated August 2023 to December 2024, disjoint from the January–April 2025 evaluation window),

with the statement examples as public tests and the hidden tests capped at 30 per problem; split $65/35$ into $148/80$.

- **TheoremQA and OlympiadBench** (no training split): seed-42 samples of 247 and 181 evaluation items are held out for calibration. OlympiadBench's pool also includes its 93 multi-answer rows outside the evaluation set. The pools are split $65/35$ into $160/87$ and $178/96$ search/validation items, leaving 500 and 400 test items shared by all methods.
- **MuSR**: questions are grouped by their underlying story before assigning groups to search, validation or test. Each story belongs to exactly one partition. The test set contains 400 questions and is shared by all methods.

Protocol selection and final evaluation. The final search configuration, selection rule and evaluation procedure are fixed before test evaluation. Patch proposals and acceptance use $\mathcal{D}_{\mathrm{s}}$. The final program is selected from the candidate archive using pooled search–validation accuracy, with ties broken by cost (Eq. 19). The selected program is then frozen and evaluated on the held-out test set with seeds 42, 1 and 2. Test results are used only for final evaluation and analysis, not to select configurations, search runs or reported programs. The main results, paired comparisons, controls and figures use this final evaluation.

MuSR: story-disjoint evaluation. All questions sharing a story, including those about different entities or outcomes, are assigned to the same partition. The search, validation and test partitions share no stories. Protocol search and selection use only their assigned partitions. The selected protocol is frozen before the final test evaluation.

MuSR: input digests and learned protocol. Each failure example shown to the synthesizer is a digest of the item and its trace. For long items, the digest retains the first 500 and the last 1,500 characters to include the question and answer options at the end. The final MuSR protocol is SC-3 with one learned instruction (Appendix I); it uses 3.0 calls and achieves 79.1 ± 0.6 test accuracy (seeds 42/1/2: 78.8, 79.8, 78.7). The MuSR cases in Appendix J.2 trace this protocol. It has no conditional branch, so the branch analysis in Appendix H.3 covers the three other benchmarks with the most frequent branching.

H Additional Results and Analyses

The figures in Section 5 and this appendix use the reported runs' saved per-item records and search logs. Accuracies are measured on test sets unless marked as search-split.

Per-benchmark fronts. Figure 5 shows the frontier for each of the eight benchmarks. Among the seventeen reproduced baselines drawn there (ADAS and AFlow are not drawn), no method dominates REPROT on any benchmark in accuracy and mean calls. REPROT has the highest reported mean on seven benchmarks. On MATH-500, its 95.9 is 0.2 below the best mean, with fewer calls than the methods at 96.1. Its largest margins over the best of these seventeen are on TheoremQA (83.5 against 77.7 for PHP, at 1.0 call) and MuSR (79.1 against 64.2 for LLM-Debate, at 3.0 calls against 6.0). The other leads are 1.1 on OlympiadBench, 1.2 on LiveCodeBench, 0.2 on GPQA, 0.4 on SuperGPQA and 0.5 on MMLU-Pro. These mean rankings alone do not establish statistical significance.

Head-to-head. Figure 4 counts, for each baseline, the benchmarks on which REPROT is significantly better, not significantly different, or significantly worse. We use an exact two-sided sign test on the discordant (seed, item) pairs pooled over the three seeds, with a tie when $p \geq 0.05$. Against the nine top-tier baselines REPROT wins on three to six benchmarks and has no significant loss. TheoremQA and MuSR are wins in every comparison. Against AFlow, the record is 4–4–0: wins on OlympiadBench, GPQA, SuperGPQA and MMLU-Pro, and ties elsewhere. Against Vanilla, AutoGen and LLM-Blender it wins all eight benchmarks. Removing TheoremQA and MuSR still separates REPROT from all nine tier members in pooled tests (against G-Designer: 277 versus 183 discordant pairs, p=1.36×10^{-5}) and from AFlow (308 versus 174, p=1.08×10^{-9}).

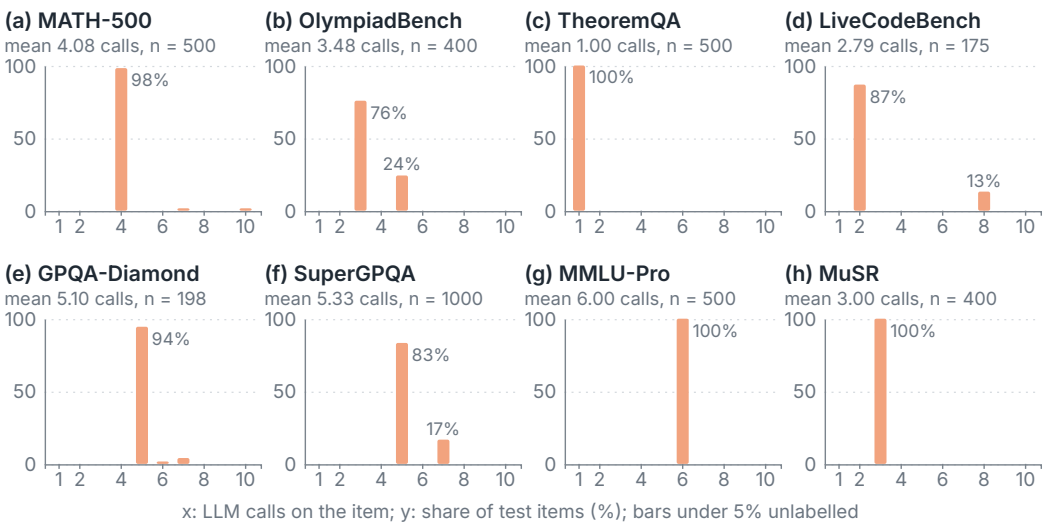

Figure 7: Distribution of LLM calls per test item for the learned program (seed 42). Bars under 5% are unlabelled.

Search dynamics. Figure 6 traces each seed-42 search. The root is the best seed with at most 3.0 calls on the search split. The gate accepted 1–4 patches per benchmark and rejected the rest (40–49 candidates). Search-split gains overstate test gains. OlympiadBench rises from 64.0 to 84.8 on the search split, mostly through the round-1 answer-format instruction (its search pool contains the 93 multi-answer rows absent from the test set). The corresponding test change is 85.8 $\rightarrow$ 89.0. On MMLU-Pro, the patched program reached 86.3 at 3.3 calls on the search split. The unpatched Debate seed (86.0, 6.0 calls) won on pooled search and validation accuracy and became $P^{\star}$. The calls strip shows that the budget binds only on MMLU-Pro (6.0). The other programs stay below it (TheoremQA at 1.0, MuSR at 3.0).

Adaptive cost. Figure 7 shows the learned programs' calls per item. Four programs are adaptive. OlympiadBench uses 3 calls on 76% of items and 5 on the rest. LiveCodeBench uses 2 calls when the first two solutions agree (87%) and 8 otherwise. SuperGPQA uses 5 or 7 calls, and MATH-500 uses 4 on 98% of items. TheoremQA (1), MuSR (3) and MMLU-Pro (6) use a fixed number of calls, and GPQA is nearly fixed at 5. Meeting the mean-call budget depends mainly on the cheaper branch, without relying on a cap on the expensive branch.

Fix/break against the root. Figure 8 plots Table 9. Fixes exceed breaks on all eight benchmarks, and five comparisons reach $p<0.05$: OlympiadBench (18 versus 5, $p=0.011$), TheoremQA (66 versus 7, $p<0.001$), LiveCodeBench (12 versus 3, $p=0.035$), SuperGPQA (18 versus 6, $p=0.023$) and MuSR (106 versus 31, $p<0.001$). TheoremQA and MuSR have the largest gains. All 66 TheoremQA fixes and 85 of the 106 MuSR fixes address generation failures. On TheoremQA, 52 of the 66 fixes (hatched) are items where the root already had the correct value in a rejected answer form (Appendix J.1).

Seed spread. Figure 9 shows the per-seed averages of REPROT and the five strongest reproduced baselines. REPROT ranges from 83.66 to 84.66. Its lowest seed is above every run of the seventeen reproduced baselines (highest: G-Designer's seed 2, 81.01) and AFlow (range 82.15–83.04). The seed spread of REPROT (1.00) is comparable to G-Designer's (1.60) and AgentPrune's (1.03). The complete graph and LLM-Debate vary less.

Token cost. Call counts measure LLM invocations. Figure 10 accounts for their length by plotting accuracy against mean prompt-plus-completion tokens per question from runner logs. Averaged over the eight benchmarks, REPROT uses 6.0k tokens per question, against 8.3k for G-Designer and 18.3k for the complete graph, and has the lowest token count among the reproduced baselines above 80%. AFlow, not drawn, uses fewer (5.1k), and ADAS uses more (15.7k), measured identically

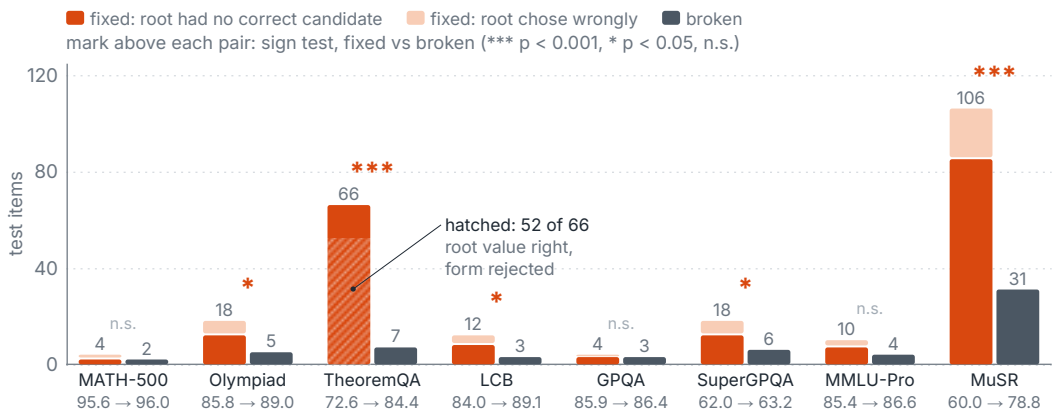

Figure 8: Items fixed and broken by the learned program relative to its root P_0 (seed 42), with fixes split by generation and selection failure of P_0. Marks indicate the exact two-sided sign test on fixed vs. broken items. Values below each name give $P_0 \rightarrow$ REPROT test accuracy. Hatching denotes TheoremQA fixes where the root value was already correct.

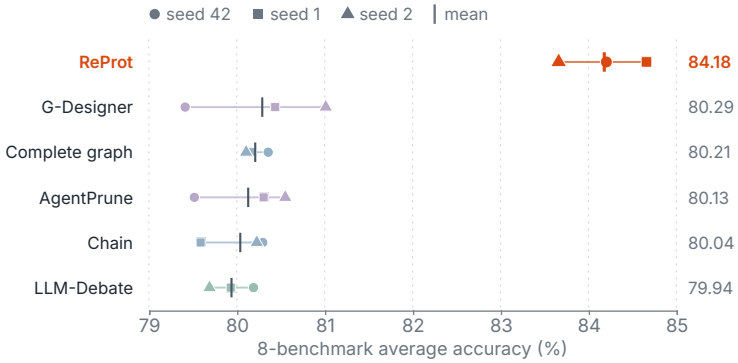

Figure 9: Per-seed eight-benchmark averages (symbols) and three-seed means (ticks) for REPROT and the five strongest reproduced baselines (AFlow's seeds: 82.15–83.04).

from their result files. No plotted baseline dominates it in accuracy and mean tokens on any benchmark, although cheaper methods remain available. On OlympiadBench, TheoremQA and MuSR, the baselines ran the full evaluation set and REPROT ran only the test items. Baseline token means therefore cover a superset of the scored items. AgentPrune is absent because its runner did not log tokens.

H.1 ORIGINAL ADAS AND AFLOW

We ran the official ADAS (Hu et al., 2025) (Meta Agent Search, 30 generations, bootstrap-median fitness, its seven initial agents) and AFlow (Zhang et al., 2025c) (MCTS over operator workflows, up to 20 rounds, five validation repeats) using their released code and default hyper-parameters. We changed only the models (executor `gpt-5.6-luna`, meta/optimizer `gpt-5.6-sol`), the search and validation pools (the same as REPROT's) and the test protocol (the test sets of Section 5.1, seeds 42, 1, 2, cache off). Table 1 reports mean executor calls per problem for all methods. Table 7 gives the executor cost of each search. AFlow's near-zero s.d. on GPQA and SuperGPQA comes from its original temperature-0 single-call workflow there. Its predictions still differ across seeds.

H.2 CONTROLS IN FULL

Section 5.4 reports five controls.

The seed pool. The controls (Table 2) distinguish the contributions of search, language, seeds and prompts. The seeds compete during selection. Seven of eight benchmarks returned a patched program, and MMLU-Pro returned the unpatched Debate seed (pooled 85.7 versus 85.5 for the

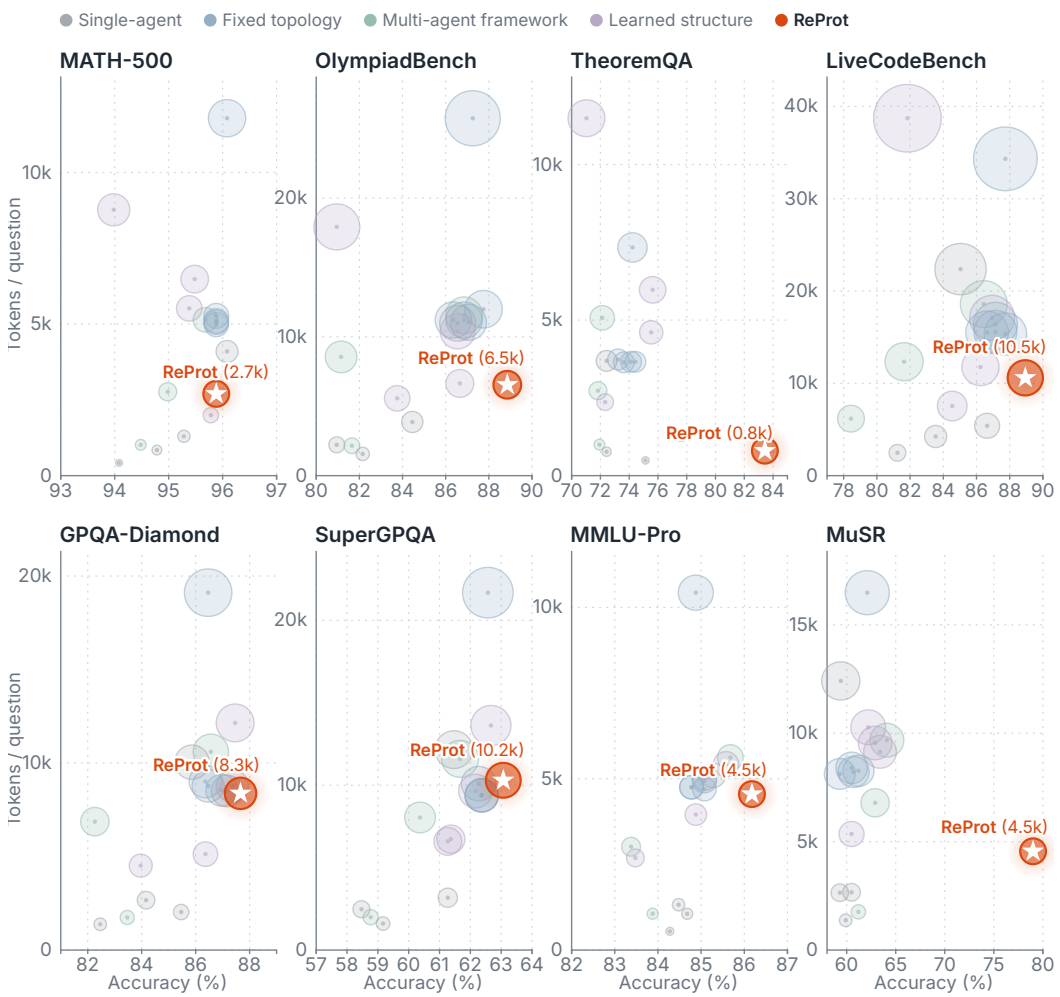

Figure 10: Accuracy vs. mean tokens per question (prompt + completion, three seeds) for each benchmark. Bubble area $\propto$ tokens (REPROT's disc has a minimum size to keep its marker visible). AgentPrune is absent because its runner did not log tokens. ADAS and AFlow are omitted, with averages reported in the text. On OlympiadBench, TheoremQA and MuSR, baseline tokens are averaged over the full evaluation set and REPROT's over the test items only.

accepted program). The gate accepted 1–4 of 50 candidates per benchmark. Against the root P_0 at seed 42, the returned protocol gains on all eight test sets: 18.8 points on MuSR, 11.8 on TheoremQA, 5.1 on LiveCodeBench, 3.2 on OlympiadBench, 1.2 each on SuperGPQA and MMLU-Pro, 0.5 on GPQA and 0.4 on MATH-500. Fixes exceed breaks at $p<0.05$ in five comparisons (Table 9). MMLU-Pro's gain comes from selecting a different, unpatched seed. We did not run per-question routing or an oracle over the pool.

Free-code search at the same synthesizer and budget. The ADAS-style control (Hu et al., 2025) (earlier configuration†) replaces the language with a free Python function `solve(problem, api)`. The same synthesizer rewrites it in full each round from the same seeds, under the same gate, budget and test protocol (Appendix I). On OlympiadBench, it reaches 88.6 ± 0.5 at 2.7 calls, 0.3 below REPROT and within the seed variation. It accepts one rewrite (54 fixed, 7 broken) whose operative content is a strict answer-format instruction and a solver persona. On MMLU-Pro, no candidate passed the gate, and the validation-selected program scores 84.4 ± 0.5 at 2.3 calls. A prompt-level change can outweigh any structural change. Both free code and REPROT's instruction slot (OlympiadBench, TheoremQA) find such changes.

	MATH-500	Olympiad	TheoremQA	LCB	GPQA	SuperGPQA	MMLU-Pro	MuSR
executor calls during search (thousands)								
ADAS	44.5	24.3	16.7	17.3	37.9	74.1	50.8	22.0
AFlow	30.9	43.3	25.7	19.6	24.8	36.7	93.4	22.9
REPROT	27.7	15.6	21.9	5.7	17.1	62.6	67.6	21.4
executor tokens during search (millions, in+out)								
ADAS	85.3	47.6	19.4	53.8	66.6	167.5	56.6	51.3
AFlow	38.7	94.6	18.2	60.6	17.4	55.8	80.1	30.6
REPROT	29.3	25.7	12.5	14.0	15.6	106.8	47.5	31.1
meta / optimizer calls								
ADAS	75	64	83	91	87	88	72	74
AFlow	13	20	20	20	16	10	20	17
REPROT	10	10	10	10	10	14	10	11

Table 7: Search cost of the original ADAS and AFlow and the reported REPROT searches. Counts include executor cache misses and exclude REPROT's cache hits across patches.

Instruction slot. Searching without the slot (*structure-only*†) averages 79.03 against 84.18 and returns the Debate seed on three benchmarks. The largest differences are MuSR ($59.2 \rightarrow 79.1$, where the searches also differ in the digest fix), TheoremQA ($72.4 \rightarrow 83.5$, one instructed CoT call) and OlympiadBench ($86.5 \rightarrow 88.9$). The searches also differ in rounds and pools, so this comparison bounds the slot's effect but does not isolate it.

Same sub-protocol, different trigger. Holding operations and visibility fixed, we replace every program conditional, including conditions inherited from the seed, by $\top$ (*first branch*: each If takes its first branch, loops run to their bound) or by $\mathrm{ndistinct}(I) \geq 2$ over the candidates so far (*disagreement*). We then re-run the test set at seed 42. The first-branch override spends 10.0 calls against 4.1 on MATH-500 (95.0 versus 96.0) and 9.0 against 5.1 on GPQA (83.8 versus 86.4). On LiveCodeBench, *disagreement* spends 7.2 calls against 2.8 and scores 83.4 versus 89.1. The first-branch override collapses the cascade to its two-call vote and scores 79.4, a 9.7-point drop. Trigger choice changes cost by up to $2.6\times$ and can also affect accuracy substantially. These are single-seed comparisons. Re-runs of unchanged programs span 81.6–84.4 on TheoremQA.

Same sub-protocol, different visibility. At fixed operations and conditions, *isolated* turns every full-text read into an answer-only read, and *open* does the reverse. The largest differences from the learned run are on LiveCodeBench: 85.1 for isolated and 84.6 for open against 89.1 for the learned program, gaps of 4.0 and 4.5 points. On MMLU-Pro, isolated scores 84.0 against 86.6; on MATH-500 it gains 0.4. On GPQA, open gains 0.5, while on SuperGPQA isolated and open score 62.0 and 62.4 against 63.2. The table marks variants that are identical to the learned program, for which differences reflect repeated execution. These controls provide single-seed evidence. The MuSR program has no full-text read, so neither variant changes it.

H.3 WHERE THE CALLS GO

Figure 11 examines the three programs that branch most frequently: OlympiadBench, LiveCodeBench and SuperGPQA. MATH-500 and GPQA also have conditional execution (Figure 7). Figure 11 groups each benchmark's test items, pooled over the three seeds, by the executed branch. It scores REPROT and four baselines (complete graph, G-Designer, LLM-Debate, SC) on exactly those items. Items routed to the more expensive branch have lower accuracy for every method, by 11–36 points. The benefit on that branch differs by benchmark. On OlympiadBench, REPROT scores 73.3 against 69.8–75.3 for the four baselines. On LiveCodeBench it scores 79.7 against 55.8–67.5, and on SuperGPQA 49.7 against 41.2–44.2. On the cheaper branch, REPROT scores 94.0 on OlympiadBench at 3 calls against 91.2–92.0 at 5–11 calls. On LiveCodeBench the agreement branch costs 2 calls and scores 90.6, within the baselines' range of 88.3–91.7. MuSR is absent because its final protocol has no conditional branch.

On SuperGPQA, the three extra solvers run when the first two *agree*. The within-trace analysis of this program records 16 agreed wrong answers corrected and 4 agreed right answers changed to wrong answers, pooled over seeds. This analysis compares intermediate and final answers. Figure 11 compares complete methods on each branch.

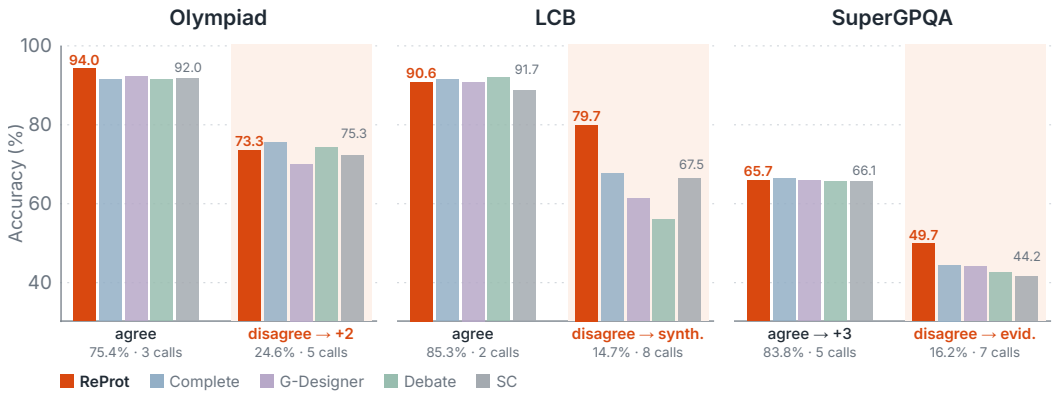

Figure 11: Accuracy by executed branch (test items, three seeds pooled). Values below each group give the share of items and REPROT's mean calls on them. Shading marks the higher-call branch. The grey number gives the best of the four baselines on the same items. The more expensive branch contains harder items for every method. REPROT leads the four comparators on that branch for LiveCodeBench and SuperGPQA.

I LEARNED PROTOCOLS

Table 8 gives each root, how many of the 50 proposed patches were valid and accepted, and each accepted patch on the path to the selected program $P^\star$, with its $(n_{\text{fix}}, n_{\text{brk}}, p, \hat{c})$ on the search set. It also reports the search, validation and pooled accuracy of $P^\star$.

Seed search accuracies follow this order: CoT / SC-3 / PHP / Debate / Star / cascade, then PoT / exec-fix / PoT-check where run: GPQA 60.3/58.7/61.3/62.3/60.7/61.0; SuperGPQA 60.4/59.4/61.6/61.7/60.6/62.0; MMLU-Pro 84.7/85.0/85.3/86.0/84.1/84.1; MuSR 66.7/68.8/67.1/65.4/64.1/65.4; MATH-500 90.6/92.9/93.4/93.4/92.6/92.6, 85.1/92.9/92.3; OlympiadBench 56.2/64.0/59.6/62.9/61.2/62.9, 56.7/60.7/62.9 (the pool includes the multi-answer rows); TheoremQA 78.1/78.1/76.9/75.0/75.0/76.9, 58.8/77.5/61.3; LiveCodeBench 85.1/87.8/86.5/89.9/87.2/88.5, exec-fix 87.8 and SC-5 88.5 (SC-5 was a seed on LiveCodeBench only).

The six validation candidates have these search/validation accuracies: GPQA $P^\star$ 66.7/62.6, the two round-1 accepted programs 65.3/62.6 and 63.0/64.9, Debate 62.3/64.1, PHP 61.3/62.6, cascade 61.0/63.4; SuperGPQA $P^\star$ 62.4/67.0, cascade 62.0/66.3, Debate 61.7/66.0, PHP 61.6/65.0, Star 60.6/64.3, CoT 60.4/63.3; MMLU-Pro the accepted program 86.3/83.7, Debate ($P^\star$) 86.0/85.0, PHP 85.3/83.3, SC-3 85.0/84.7, CoT 84.7/83.3, cascade 84.1/85.7; MuSR $P^\star$ 77.5/75.2, the three other round-1 accepted programs 77.5/70.4 (at 6.0 calls), 74.5/64.8, 71.9/63.2, SC-3 68.8/65.6, PHP 67.1/65.6; MATH-500 $P^\star$ 94.3/96.7, PHP 93.4/94.7, Debate 93.4/96.7, exec-fix 92.9/96.0, SC-3 92.9/96.0, cascade 92.6/97.3; OlympiadBench $P^\star$ 84.8/81.2, the round-1 instruction program 82.6/82.3, and Debate, SC-3, cascade and PoT-check at 61.2–64.0 / 64.6–68.8; TheoremQA $P^\star$ 81.3/78.2, CoT 78.1/71.3, SC-3 78.1/72.4, exec-fix 77.5/72.4, PHP 76.9/71.3, cascade 76.9/73.6; LiveCodeBench $P^\star$ 91.2/86.2, Debate 89.9/83.8, cascade 88.5/83.8, SC-5 88.5/86.2, exec-fix 87.8/85.0, SC-3 87.8/86.2.

Selection. Seven of eight searches returned an accepted program. On MMLU-Pro, the one accepted program leads on search (86.3) but not on validation (83.7). The Debate seed has higher pooled accuracy (85.7 against 85.5), so $P^\star$ is that unpatched seed. At seed 42, it scores 1.2 above the PHP root on test with nearly three times as many calls (Section 5.4). On SuperGPQA, the accepted program leads on search and validation and scores 1.2 above the cascade root on test. The released archives contain all instruction texts; the listings below give those discussed in the main text.

Benchmark	Root P_0 (acc. @ calls)	valid/inv./acc.	Accepted patches ($n_{\text{fix}}, n_{\text{brk}}, p, \hat{c}$)	$P^\star$: search / val. / pooled @ calls
GPQA	PHP 61.3@2.1	50/0/3	r1 `wrap_root`: three independent instructed solves and a vote (25, 13, .036, 5.1); r10 `insert_after`: if all three differ, decompose and solve the sub-questions (5, 1, .11, 5.1)	66.7 / 62.6 / 65.4 @5.1
SuperGPQA	cascade 62.0@2.9	48/2/1	r6 `insert_after st1`: if `c1` and `c2` agree, three more instructed solves and a five-way vote (3, 0, .125, 5.3)	62.4 / 67.0 / 63.8 @5.3
MMLU-Pro	PHP 85.3@2.0	50/0/1	r5 `insert_after h1`: an instructed independent solve; if it disagrees with `h1`, two more instructed solves and a vote (9, 2, .033, 3.3); not selected	Debate seed: 86.0 / 85.0 / 85.7 @6.0
MuSR	SC-3 68.8@3.0	50/0/4	r1 `set_instruction` on all three solvers: track the story mechanically, per subtask (40, 20, .007, 3.0)	77.5 / 75.2 / 76.7 @3.0
MATH-500	PHP 93.4@2.1	50/0/1	r3 `insert_after h1` in the hint loop: instructed CoT and program in parallel, voted if the program ran, they agree and they differ from `h1` (3, 0, .125, 4.3)	94.3 / 96.7 / 95.0 @4.1
OlympiadBench	SC-3 64.0@3.0	50/0/2	r1 `set_instruction` on every solver: answer-format rule (38, 5, $<$.001, 3.0); r3 `insert_after agg`: if the three disagree, two more instructed solves and a vote (5, 1, .11, 3.6)	84.8 / 81.2 / 83.6 @3.6
TheoremQA	CoT 78.1@1.0	49/1/1	r1 `set_instruction` on the single solve (9, 4, .13, 1.0)	81.3 / 78.2 / 80.2 @1.0
LiveCodeBench	cascade 88.5@2.8	41/9/1	r2 `insert_after v2` in the disagreement branch: an instructed critique reading all five programs and the verifiers' evidence, committed first (4, 0, .0625, 3.0)	91.2 / 86.2 / 89.5 @3.0

Table 8: Search summary per benchmark (10 rounds of 5 patches). "valid/inv./acc." gives the numbers of statically valid, invalid and accepted candidates; rN denotes the round. GPQA accepted one further patch and MuSR three further round-1 patches off the path to $P^\star$. The selected program is the seed or accepted program with the highest pooled search-and-validation accuracy (Eq. 19), with ties broken by cost.

TheoremQA. The CoT seed is the root. One accepted patch applies `set_instruction` to its single solve (n_{fix}=9, n_{brk}=4, p=0.13, $\hat{c}$=1.0). The structure is unchanged, with one call per problem:

```
seq {
  s1 = solve(reads=[problem]) {"style":"cot", "instruction":I}
  commit s1
}
```

The instruction I is:

> Re-read the question stem and compute the requested quantity rather than a nearby quantity; enumerate all cases and verify the result by substitution, dimensional analysis, or a second calculation when possible. Obey the answer type exactly: for a requested single number, output a plain decimal numeral with no fraction, symbolic constants, units, percent sign, or surrounding math notation (convert 1/e, pi-expressions, and rational values to decimals); for integer questions output an integer, and for Boolean questions output exactly True or False. Respect requested rounding and end with exactly `Final answer: <answer>`.

OlympiadBench. The root is SC-3. The round-1 patch sets one answer-format instruction F on all three solvers (38, 5, p<0.001, 3.0). The round-3 patch adds two instructed solves when the three disagree (5, 1, 0.11, 3.56):

```
seq {
  par sc {
    s1,s2,s3 = solve(reads=[problem])
      {"style":"cot","temperature":0.7,"instruction":F}
  }
  agg = aggregate(reads=[sc.answers]) {"rule":"vote"}

  if {"disagree":["s1","s2","s3"]} {
    par dx {
      dextra1 = solve(reads=[problem])
```

```
        {"style":"cot","temperature":0.0,"instruction":X1}
      dextra2 = solve(reads=[problem])
        {"style":"cot","temperature":1.0,"instruction":X2}
    }
    dextra_vote = aggregate(reads=[sc.answers, dx.answers])
      {"rule":"vote"}
  }

  commit dextra_vote > agg
}
```

The instructions are F: "Return only the mathematical contents expected inside `\boxed{}`, with no box, prose, variable assignment, set-builder notation, or surrounding delimiters unless intrinsically required. For multiple scalar answers, output a comma-separated list; for ordered pairs/triples, list each tuple explicitly rather than saying 'or,' 'is a permutation of,' or using shorthand. Include every case, preserve a clear canonical order, and never emit stray LaTeX delimiters or an empty answer. Before finalizing, reread exactly what quantity and answer form the question requests." X_1: "Solve independently from first principles, explicitly checking all cases and the exact quantity requested; return only the complete final mathematical contents expected inside the box." X_2: "Use a substantially different approach, actively search for counterexamples and omitted cases, and return only the complete requested answer without shorthand or ellipses."

GPQA. The root is PHP. The round-1 `wrap_root` adds three independent instructed solves and a vote (25, 13, 0.036, 5.08). The round-10 `insert_after` decomposes the problem when all three differ (5, 1, 0.11, 5.13):

```
seq {
  s1 = solve(reads=[problem]) {"style":"cot"}

  repeat(max=3) until {"agree":["s1","h1"]} {
    h1 = solve(reads=[problem, s1.answer]) {"style":"hint"}
  }

  par ip {
    ind1 = solve(reads=[problem])
      {"style":"cot","temperature":0.7,"instruction":A1}
    ind2 = solve(reads=[problem])
      {"style":"cot","temperature":0.7,"instruction":A2}
    ind3 = solve(reads=[problem])
      {"style":"direct","temperature":1.0,"instruction":A3}
  }
  ind_vote = aggregate(reads=[ip.answers]) {"rule":"vote"}

  if {"n_distinct":["ind1","ind2","ind3"], "ge":3} {
    td3 = decompose(reads=[problem]) {"instruction":D}
    ts3 = solve(reads=[td3])
      {"style":"subproblems","temperature":0.0,"instruction":S}
  }

  commit ts3 > ind_vote > h1 > s1
}
```

The instructions are A_1: "Solve independently from first principles, compare every option, and finish with exactly Answer: <letter>." A_2: "Independently derive the answer without assuming another solver's conclusion; check negations and option wording." A_3: "Choose an option only after eliminating the alternatives, then output exactly Answer: <letter>." D: "Decompose the multiple-choice problem into the exact rule or fact being tested, any required derivation, and an explicit comparison of every option." S: "Solve each sub-question independently, combine them without relying on the prior vote, and output exactly Answer: <letter>."

MuSR. The root is SC-3. In the final search, the first accepted patch sets one instruction on all three solvers (40, 20, .007, 3.0). No patch is accepted in rounds 2–10, so $P^\star$ is SC-3 with that instruction:

```
seq {
  par sc {
    s1,s2,s3 = solve(reads=[problem])
      {"style":"cot","temperature":0.7,"instruction":L}
  }
  agg = aggregate(reads=[sc.answers]) {"rule":"vote"}
  commit agg
}
```

The instruction is L: "Track the story mechanically before answering. For object-location questions, distinguish the object's true final location from the queried person's belief: record each move, who observed it, and where that person would search first; unseen moves do not update belief. For allocation questions, score every person against every task using explicit strengths and weaknesses, then check each complete option and assign everyone exactly once. For culprit questions, compare motive, means, opportunity, access, and contradictions for every suspect rather than following the most vivid accusation. Re-read the exact question stem, verify the chosen option against the story, and conclude only with Answer: <letter>." The MuSR cases in Fig. 12 trace this final protocol.

LiveCodeBench. The root is the cascade seed. One accepted patch adds an instructed critique in the disagreement branch. It reads both groups' programs and the verifiers' evidence and is committed first (4, 0, 0.0625, 3.01). The structure matches the structure-only search, now with an instruction:

```
seq {
  par st1 {
    c1 = solve(reads=[problem]) {"style":"cot"}
    c2 = solve(reads=[problem])
      {"style":"cot","temperature":0.0}
  }

  if {"agree":["c1","c2"]} {
    agg1 = aggregate(reads=[st1.answers]) {"rule":"vote"}
  }
  else {
    par st2 {
      d1,d2,d3 = solve(reads=[problem])
        {"style":"cot","temperature":1.0}
    }

    v1 = verify(reads=[problem, c1.answer]) {"k":1}
    v2 = verify(reads=[problem, c2.answer]) {"k":1}

    dis_synth = solve(
      reads=[st1.answers, st2.answers, v1.evidence, v2.evidence]
    ) {"style":"critique","temperature":0.0,"instruction":L}

    agg1 = aggregate(reads=[st1.answers, st2.answers])
      {"rule":"evidence"}
  }

  commit dis_synth > agg1
}
```

The instruction is L: "Produce the final complete Python program. Reconcile the five proposed algorithms rather than voting by surface similarity; use the verifier objections, trace invariants and boundary cases, check complexity against the full constraints, and mentally execute every public example. Correct the algorithm or implementation whenever needed."

Fix/break counts. Table 9 reports the counts used in Section 5.5 for all eight benchmarks, with the exact two-sided sign test on fixed versus broken items.

Free-code control. The control ran in the earlier search configuration (R=8, K=4; the earlier OlympiadBench and MMLU-Pro pools) and was not repeated for the final one. Final selection takes the top six candidates by search accuracy, including rejected candidates, then ranks them by validation accuracy. REPROT admits only seeds and accepted programs and ranks them by pooled accuracy. The OlympiadBench control searched a 400-item pool of MATH level-5 and OlympiadBench multi-answer items in rounds 0–4 and the 117-item held-out slice from round 5, so its search

trajectory is not comparable to Table 8. Its root was the cascade seed written as code (77.0 on the first pool at 2.7 calls). The accepted rewrite of round 3 reached 88.8 at 2.3 calls. The final program of round 4 adds a keyword trigger ("sin^", "arithmetic sequence", "log_", ...) that runs two extra exact-computation solves. This patch was never accepted on its own (11 fixed, 7 broken, p=0.24); it was chosen on validation (61/64). On MMLU-Pro, the root scored 85.7 at 2.4 calls, and no candidate of 32 passed the gate. The final program is the gate-rejected candidate of round 1 (86.0 on the search set at 2.3 calls). It runs two parallel CoT solves and a "quantitative" keyword trigger whose program-verification branch is dead code (an `await` bug), leaving a judge call. The control ran with a 600-second per-problem timeout and its own cache.

Cost. Executor use during search is reported as uncached calls / cache hits (input / output tokens): GPQA 17,137 / 70,609 (4.36M / 11.24M); SuperGPQA 62,636 / 105,335 (34.87M / 71.92M); MMLU-Pro 67,575 / 80,802 (23.19M / 24.35M); MuSR 21,447 / 37,063 (23.45M / 7.69M); MATH-500 27,713 / 61,076 (9.01M / 20.30M); OlympiadBench 15,612 / 27,281 (5.01M / 20.72M); TheoremQA 21,932 / 8,361 (3.91M / 8.64M); LiveCodeBench 5,735 / 21,036 (5.58M / 8.40M). The totals are 239,787 uncached calls, 411,563 hits and 109.4M / 173.2M tokens, with 10 synthesizer calls per benchmark plus 4 repair calls on SuperGPQA and 1 on MuSR. Wall-clock time through round 10 was 1.8–6.7 hours per benchmark, with the searches of a round sharing one gateway. The counts cover only the reported run of each search. Free-code control costs (earlier configuration) were: OlympiadBench 24,758 uncached executor calls, 9,968 hits, 5.96M / 30.9M tokens, 47 candidate evaluations, 2.9 hours; MMLU-Pro 21,560 / 12,383, 6.26M / 10.1M, 44 evaluations, 1.5 hours. The synthesizer made 8 calls with 84k / 49k and 69k / 46k tokens.

Benchmark	n	both correct	sel. fixed	sel. kept	gen. fixed	gen. kept	broken	P_0	REPROT	p
MATH-500	500	476	2	0	2	18	2	95.6	96.0	.69
OlympiadBench	400	338	6	8	12	31	5	85.8	89.0	.011
TheoremQA	500	356	0	0	66	71	7	72.6	84.4	<.001
LiveCodeBench	175	144	4	3	8	13	3	84.0	89.1	.035
GPQA	198	167	1	5	3	19	3	85.9	86.4	1.0
SuperGPQA	1000	614	6	80	12	282	6	62.0	63.2	.023
MMLU-Pro	500	423	3	15	7	48	4	85.4	86.6	.18
MuSR	400	209	21	8	85	46	31	60.0	78.8	<.001

Table 9: Fix/break counts on the test sets at seed 42. The columns report items both P_0 and REPROT solve; P_0's selection failures (a candidate in its trace held the answer) and generation failures (none did), each split into fixed and kept by REPROT; items P_0 solved and REPROT broke; the two accuracies; and the exact two-sided sign test on fixed versus broken items.

J CASE STUDIES

J.1 THEOREMQA: FIXES AND BREAKS

We examine REPROT's 66 fixes and 7 breaks against the CoT root at seed 42 on TheoremQA. The scorer takes the first number in the extracted answer and accepts it within $\max(10^{-3}, 1\%)$ of the gold value; a fraction such as `9/32` is read as 9, and a symbolic answer such as `\frac{\pi}{3}` as 3. We call a failure *form-only* if the predicted expression, evaluated numerically, is within that tolerance of the gold answer.

Most fixes correct the answer form. Of the 66 fixes, 52 are form-only: the root's answer had the right value in a rejected form. The learned instruction I (Appendix I) asks for a plain decimal, with fractions and π converted, so the scorer accepts the answer. Examples include item 45, root `9/32`, learned `0.28125` (gold 0.28125); item 42, root $\frac{\pi}{3}$, learned 1.0472 (gold 1.0472); and item 138, root $8\sqrt{10}$, learned 25.298 (gold 25.3). The other 14 fixes change the value, as in item 232, root 10.00, learned 10.76 (gold 10.75), and item 589, root 15120, learned 12600. Of the 7 breaks, two are list answers for which the learned program returned a different list. Another one (item 47) is a longer decimal, `0.3333333333` against gold 0.33, which falls just outside the 1% tolerance.

Baselines have the same form-only failures. Form-only failures occur in almost every baseline at seed 42, accounting for 43 (G-Designer) to 68 (DyLAN) of their 115–146 TheoremQA failures,

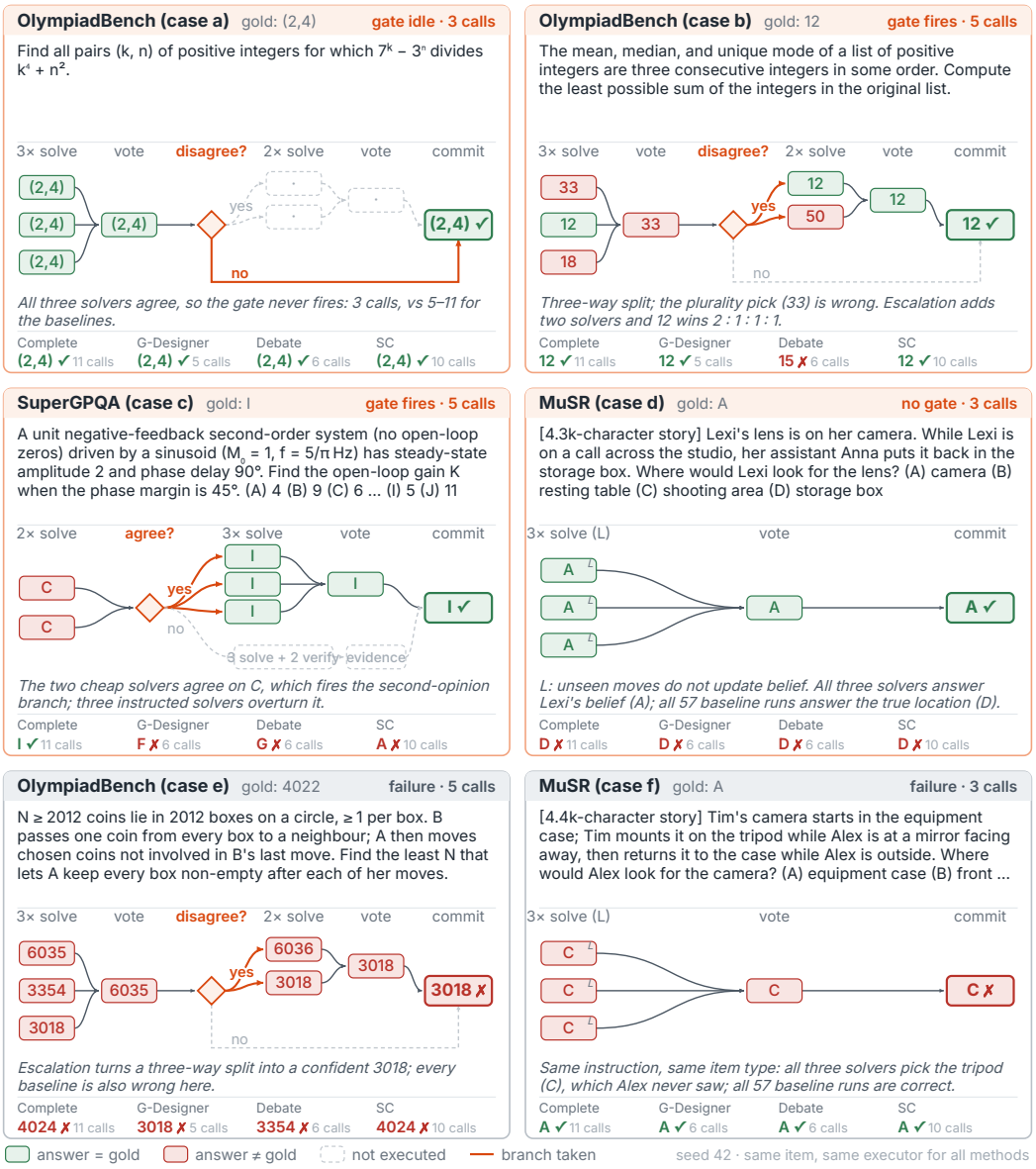

Figure 12: Executed traces on six test items (seed 42). Each box shows an abbreviated problem (the MuSR items span 4.3–4.4k characters) and the program REPROT learned for that benchmark. The executed path is solid, unexecuted nodes are dashed, and each node's answer is coloured by whether it matches the gold answer. It also gives the answers and calls of four baselines (complete graph, G-Designer, LLM-Debate, SC) on the same item. Cases (e) and (f) are failures. The MuSR boxes (d), (f) trace the reported program (SC-3 with instruction L, App. I); the text gives the selection rule.

and 51 for the complete graph. Only PHP (6) and LLM-Debate (12) largely avoid them; REPROT has none. With lenient scoring, the complete graph would reach $(500 - 128 + 51)/500 = 84.6\%$ at seed 42, against REPROT's 84.4%. We do not rescore any method. The TheoremQA margin largely measures compliance with the scorer's answer format rather than stronger reasoning.

J.2 EXECUTED TRACES

Figure 12 shows six test items at seed 42, with the program learned for each benchmark, its executed path, every node's answer, and the answers of four baselines. We chose (a)–(c) and (e) to illustrate

	(a) Olympiad 13 gold `(2,4)`	(b) Olympiad 347 gold `12`	(c) SuperGPQA 19 gold `I`	(d) MuSR 361 gold `A`	(e) Olympiad 93 gold `4022`	(f) MuSR 446 gold `A`
Vanilla	`(k,n)=(2,4)` 1c 3/3	`12` 1c 2/3	`A` 1c 1/3	`D` 1c 0/3	`4024` 1c 0/3	`A` 1c 3/3
CoT	`(k,n)=(2,4) or (...` 2c 2/3	`27` 2c 1/3	`G` 2c 0/3	`D` 2c 0/3	`3018` 2c 0/3	`A` 2c 3/3
SC	`(k,n)=(2,4)` 10c 3/3	`12` 10c 3/3	`A` 10c 1/3	`D` 10c 0/3	`4024` 10c 0/3	`A` 10c 3/3
PHP	`{(k,n)}={(2,4)}` 3c 1/3	`12` 4c 3/3	`F` 2c 0/3	`D` 2c 0/3	`2683` 2c 0/3	`A` 2c 3/3
Chain	`{(2,4)}` 6c 2/3	`12` 6c 2/3	`C` 6c 0/3	`D` 6c 0/3	`4024` 6c 0/3	`A` 6c 3/3
Star	`(k,n)=(2,4)` 6c 3/3	`12` 6c 3/3	`B` 6c 0/3	`D` 6c 0/3	`3018` 6c 0/3	`A` 6c 3/3
Tree	`(k,n)=(2,4)` 6c 3/3	`12` 6c 3/3	`I` 6c 2/3	`D` 6c 0/3	`4024` 6c 0/3	`A` 6c 3/3
Random	`(k,n)=(2,4)` 6c 3/3	`12` 6c 3/3	`A` 6c 0/3	`D` 6c 0/3	`6036` 6c 0/3	`A` 6c 3/3
Complete	`(k,n)=(2,4)` 11c 2/3	`12` 11c 3/3	`I` 11c 2/3	`D` 11c 0/3	`4024` 11c 0/3	`A` 11c 3/3
LLM-Debate	`(k,n)=(2,4)` 6c 3/3	`15` 6c 2/3	`G` 6c 2/3	`D` 6c 0/3	`3354` 6c 0/3	`A` 6c 3/3
LLM-Blender	`(k,n)=(2,4)` 5c 2/3	`12` 5c 2/3	`I` 5c 1/3	`D` 5c 0/3	`3018` 5c 0/3	`A` 5c 3/3
AutoGen	`(k,n)=(2,4)` 1c 3/3	`21` 1c 0/3	`G` 1c 0/3	`D` 1c 0/3	`4024` 1c 0/3	`A` 1c 3/3
G-Designer	`(k,n)=(2,4)` 5c 3/3	`12` 5c 3/3	`F` 6c 0/3	`D` 6c 0/3	`3018` 5c 0/3	`A` 6c 3/3
AgentPrune	`(2,4)` 5c 1/3	`12` 5c 2/3	`F` 6c 0/3	`D` 6c 0/3	`4024` 5c 0/3	`A` 6c 3/3
CARD	`{(2,4)}` 5c 1/3	`12` 5c 2/3	`A` 6c 0/3	`D` 6c 0/3	`4024` 5c 0/3	`A` 6c 3/3
GPTSwarm	`(k,n)=(2,4)` 3c 3/3	`33` 3c 1/3	`A` 6c 0/3	`D` 3c 0/3	`4024` 3c 0/3	`A` 3c 3/3
DyLAN	`(k,n)=(2,4)` 4c 2/3	`12` 4c 1/3	`C` 12c 2/3	`D` 5c 0/3	`list` 11c 0/3	`A` 5c 3/3
ADAS	`(k,n)=(2,4)` 6c 3/3	`12` 6c 2/3	`I` 9c 1/3	`D` 10c 0/3	`4024` 6c 0/3	`A` 10c 3/3
AFlow	`(2,4)` 6c 3/3	`20` 6c 2/3	`A` 1c 0/3	`D` 2c 0/3	`3018` 6c 0/3	`A` 2c 3/3
REPROT	`(2,4)` 3c 3/3	`12` 5c 2/3	`I` 5c 2/3	`A` 3c 3/3	`3018` 5c 0/3	`C` 3c 0/3
baselines correct (seed 42 / all seeds)	15/19 (46/57)	14/19 (40/57)	4/19 (12/57)	0/19 (0/57)	0/19 (0/57)	19/19 (57/57)

Table 10: Results for every method in Table 1 on the six items in Figure 12. Each cell gives the extracted answer at seed 42 (shaded if correct), its calls (c), and how many of the three seeds yield a correct answer. "`list`" means that DyLAN returned its internal agent-ranking list instead of an answer. The REPROT row uses the reported programs shown in the figure (MuSR: SC-3 with instruction L, Appendix I).

how each branch works and fails. We selected the MuSR items from all 400 test items using this rule for the reported program. Case (d) has the fewest correct baseline runs (0 of 57) among items REPROT answers correctly at all three seeds, with ties broken by index. Case (f) has the most correct baseline runs (57 of 57) among items REPROT answers incorrectly at all three seeds. Neither is typical according to Figure 11; most items follow the path in (a). Table 10 extends the figure to every baseline in Table 1 and all three seeds. For each item, the text below gives the full problem, executed nodes' instructions, every node's answer and vote tally, and results at seeds 1 and 2.

What is recorded. For every test item, the evaluator stores each executed node's extracted answer, every vote tally and branch decision, and the call count. The accounts use these records (`data/cases/` in the release). Test evaluation disables the response cache, so the solvers' free-text reasoning is not stored. A check's verdict is stored only as a ∼200-character excerpt. We quote them verbatim and do not paraphrase unrecorded reasoning. Tied votes go to the answer that appears first in read order (s_1, s_2, s_3, then the extra solvers). Baseline answers use the same scorer as Table 1.

(a) OlympiadBench item 13: the gate stays idle. *"Find all pairs (k, n) of positive integers for which $7^k - 3^n$ divides $k^4 + n^2$."* The gold answer is $(2, 4)$. The program in Appendix I runs three chain-of-thought solvers at temperature 0.7 with the answer-form instruction F. At seed 42, all three return $(2, 4)$ and the vote is $\{(2, 4)\colon 3\}$. The `disagree` condition is false, so the program commits after 3 calls. On OlympiadBench, 75% of items take this path (Figure 11). Seed 2 gives the same result. At seed 1, s_1 returns $(2, 4), (8, 8)$, so the solvers disagree and the gate fires. Both extra solvers return $(2, 4)$, which the vote commits after 5 calls (the two format it differently; the vote treats both as $(2, 4)$). The gate spends two calls on an already correct vote. 15 of the 19 baselines are also correct at seed 42, using 1–11 calls. Chain and CARD return the set $\{(2, 4)\}$ and PHP returns $\{(k, n)\} = \{(2, 4)\}$, all of which the scorer rejects. CoT adds the non-solution $(4, 8)$.

(b) OlympiadBench item 347: escalation on a split. *"The mean, median, and unique mode of a list of positive integers are three consecutive integers in some order. Compute the least possible sum of the integers in the original list."* The gold answer is 12. At seed 42, the three solvers return s_1=33, s_2=12, s_3=18. The vote $\{33\colon 1, 12\colon 1, 18\colon 1\}$ has no majority and would commit the wrong answer, 33. The gate fires and runs two extra solvers: `dextra1` at temperature 0 with X_1 ("solve independently from first principles, explicitly checking all cases") returns 12, and `dextra2` at temperature 1 with X_2 ("use a substantially different approach, actively search for counterexamples") returns 50. The five-way vote $\{12\colon 2, 33\colon 1, 18\colon 1, 50\colon 1\}$ commits 12 (5 calls). At seed 2, the solvers

return $12, 12, 20$. Their lack of unanimity triggers the gate; the extra solvers return 15 and 12, and 12 wins 3:1:1. At seed 1, the five answers are $15, 44, 20, 33, 21$. No sample is 12, every answer has one vote, and the tie goes to s_1=15. Escalation can recover the answer only when the extra samples contain it. 14 of 19 baselines are correct at seed 42. The incorrect answers are LLM-Debate (15), AutoGen (21), GPTSwarm (33), CoT (27) and AFlow (20).

(c) SuperGPQA item 19: a second opinion on agreement. *"Given a unit negative feedback second-order system (without open-loop zeros), when the input sinusoidal signal has an amplitude $M_0 = 1$ and the frequency is adjusted to $f = \frac{5}{\pi}$ Hz, the steady-state response has an amplitude of 2 and a phase delay of $90°$. Determine the open-loop gain K of the system when the phase margin is $y = 45°$."* Options (A) 4, (B) 9, (C) 6, (D) 3, (E) 8, (F) 10, (G) 7, (H) 2, (I) 5, (J) 11; gold (I). The learned program runs two chain-of-thought solvers, c_1 at the default temperature and c_2 at temperature 0. If they agree, it runs three more solvers with different instructions: e_1 (temperature 1) "Solve independently from first principles, recompute all quantities carefully, compare the result against every option"; e_2 (direct answer, temperature 0.7) "Use an independent concise derivation; reread the stem for the requested quantity, check units and powers of ten"; e_3 (temperature 0.7) "Test the supplied options against the governing definition or equation rather than trusting an intuitive result". A five-way vote then decides. If they disagree, the program runs three solvers at temperature 1, verifies c_1 and c_2, and aggregates by evidence. This branch was accepted in round 6 on weak evidence (3 fixes, 0 breaks, p=0.125; Table 8). At seed 42, c_1=c_2=C, while e_1, e_2, e_3 all return I. The vote $\{\text{C:2}, \text{I:3}\}$ commits I (5 calls), overturning an agreement that a cascade would have committed. At seed 2, the same branch fails: c_1=c_2=A, the extra solvers return G, I, G, and the tie $\{\text{A:2}, \text{G:2}, \text{I:1}\}$ goes to A. At seed 1, c_1=A and c_2=I disagree. The three extra solvers return G, B, I, and the evidence aggregate commits I (7 calls). The item is difficult for every method: 4 of 19 baselines are correct at seed 42 (Tree, Complete, LLM-Blender, ADAS), and 12 of 57 runs are correct across seeds. A is the most common answer.

(d) MuSR item 361: tracking who saw the move. The task is an object-placement story (4,302-character story plus question). To summarize, Lexi, a photographer, mounts her lens on the camera for a shoot. Afterwards, while Lexi is on a phone call at the far end of the studio, her assistant Anna removes the lens and returns it to the storage box. Later, Jake, the model, moves the camera itself to a table while Lexi is in her office. *"Which location is the most likely place Lexi would look to find the lens given the story? (A) camera (B) resting table (C) shooting area (D) storage box."* The gold answer is (A): Lexi last saw the lens on the camera and did not see Anna move it. The reported program runs three chain-of-thought solvers, each with instruction L, which includes "distinguish the object's true final location from the queried person's belief: record each move, who observed it, and where that person would search first; unseen moves do not update belief", and votes. At seed 42, all three return A and the vote is $\{\text{A:3}\}$ (3 calls). At seed 1, the vote is again 3:0; at seed 2, it is $\{\text{A:2}, \text{D:1}\}$. Every one of the 57 baseline runs (19 methods, three seeds) answers D, the lens's true location, including the four in the figure at 6–11 calls. This is one of 84 items on which REPROT is correct at all three seeds and fewer than half of the baseline runs are. The converse holds for 16 items: REPROT is wrong at all seeds and more than half of the baseline runs are correct. Case (f) is the most extreme.

(e) OlympiadBench item 93: escalation into a wrong consensus. *"Players A and B play a game with $N \geq 2012$ coins and 2012 boxes arranged around a circle. Initially A distributes the coins among the boxes so that there is at least 1 coin in each box. Then the two of them make moves in the order $B, A, B, A, \ldots$ by the following rules: On every move of his B passes 1 coin from every box to an adjacent box. On every move of hers A chooses several coins that were not involved in B's previous move and are in different boxes. She passes every chosen coin to an adjacent box. Player A's goal is to ensure at least 1 coin in each box after every move of hers, regardless of how B plays and how many moves are made. Find the least N that enables her to succeed."* The gold answer is 4022. At seed 42, the solvers split three ways: s_1=6035, s_2=3354, s_3=3018. The gate fires, `dextra1` returns 6036 and `dextra2` returns 3018, and the vote $\{\text{3018:2}\}$ commits 3018. At seeds 1 and 2, the program commits 4024. At seed 2, `dextra1` returned the gold answer 4022, the only one of the 15 REPROT solves on this item to do so (no baseline's final answer is 4022 at any seed). It lost the vote to 4024 (2:2:1, tie to s_1). No baseline is correct at any seed; the most common

answers are 4024 (30 of 57) and 3018 (12). The gate spent its calls where the solvers were unsure, as intended. Here, the two repeated wrong answers outvote the sole correct sample.

(f) MuSR item 446: the same instruction, misapplied. The task (4,418 characters) is an object-placement story of the same kind. Tim's camera starts in the equipment case. Tim mounts it on the tripod while Alex, the model, practises poses at a mirror that faces away from Tim. After the shoot, while Alex is outside on the patio with a journalist, Tim returns the camera to the equipment case. *"Which location is the most likely place Alex would look to find the high-end camera given the story? (A) equipment case (B) front table (C) tripod."* The gold answer is (A): Alex saw neither move, so Alex believes the camera is at its starting location. At seed 42, all three instructed solvers return C, the tripod, where the camera was while Alex was not watching. The vote is {C:3} (3 calls). At seed 1, the vote is {C:2, A:1}; at seed 2, it is 3:0 for C. All 57 baseline runs answer A. Because the solvers' reasoning is not recorded, we cannot identify the error's source. The record shows that all three instructed solvers share the error, which the vote, the program's only aggregation, cannot correct.

Interpreting the six traces. Branch decisions are visible and inexpensive to audit. Every call in Figure 12 corresponds to a node, and the gate fires on disagreement (b, e) or, on SuperGPQA, on agreement (c). Where extra calls help, they add samples containing the correct answer (b and c at seed 42). The observed vote fails when the correct answer is absent (b at seed 1) or outvoted (e at seed 2). The same branch fails on that item at another seed (b, c), so a single trace does not establish that a branch helps; Figure 11 gives the aggregate results. On MuSR, the learned change is one instruction, with no change to the branches. Cases (d) and (f) show its differing effects at 3 calls. In (d), all three instructed solvers answer with the queried person's belief, which no baseline run does. In (f), they err together where every baseline run is correct. Across the test set, the first case is more common (84 items against 16).