ReaComp: Compiling LLM Reasoning into Symbolic Solvers for Efficient Program Synthesis

Anonymous authors
Paper under double-blind review

Abstract

LLMs can solve program synthesis tasks but remain inefficient and unreliable on hard instances requiring large combinatorial search. Given a small set of reasoning traces without ground truth labels, we use coding agents to compile them into reusable symbolic program synthesizers over constrained DSLs. The resulting solvers require no LLM calls at test time and are strong standalone systems: symbolic solver ensembles reach 91.3% accuracy on PBEBench-Lite and 84.7% on PBEBench-Hard, outperforming LLMs with test-time scaling for the latter by +16.3 percentage points at zero LLM inference cost. They also complement LLM search, improving PBEBench-Hard accuracy from 68.4% to 85.8% while reducing reported token usage by 78%, and raising SLR-Bench hard-tier accuracy from 34.4% to 58.0% in a neuro-symbolic hybrid setting. Compared to directly using coding agents as per-instance solvers, induced solvers are substantially more Pareto-efficient, amortizing a small one-time construction cost over many zero-token executions. Finally, most solvers transfer zero-shot to a real historical linguistics task, predicting sound changes in natural language data, reaching 80.1% accuracy under ensembling and recovering some plausible linguistic rules. Together, these results show that, in domains with exact verification, reasoning traces can be compiled into reusable symbolic solvers that solve many tasks directly and complement LLM inference on hard cases. We release code and data for reproducibility.

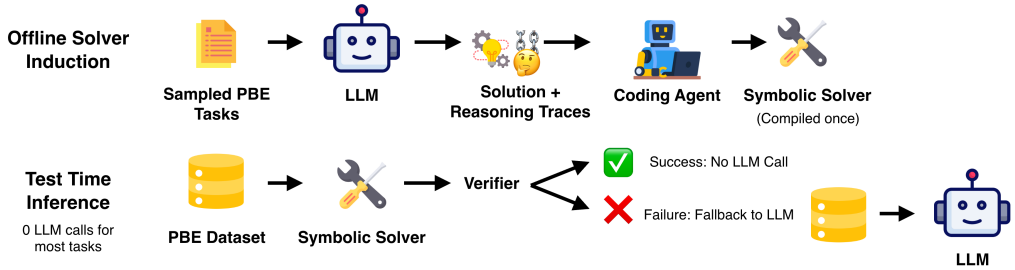


Figure 1: ReaComp: **offline solver induction** compiles LLM reasoning traces into a reusable symbolic solver; **test-time inference** applies the solver directly (zero LLM cost) and falls back to LLM search only for unresolved cases.

1 Introduction

The Programming by Example (PBE) paradigm (Gulwani, 2011) provides a controlled setting for studying structured reasoning and compositional generalization, with applications spanning code transformation, data wrangling, text processing, and historical linguistics. Recent work shows that large language models (LLMs), when combined with scaling strategies such as Best-of-K sampling and iterative refinement, can achieve strong performance on PBE tasks (Li & Ellis, 2024; Naik et al., 2025b). In particular, prior work like PBEBench

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.

(Naik et al., 2025b) demonstrates that sufficiently powerful reasoning models, together with these scaling strategies, can achieve high accuracy on PBE tasks without explicit symbolic structure.

However, this performance comes at a substantial computational cost. Harder instances require disproportionately more attempts and tokens (Figure 3), and the reasoning traces show that even strong models like `gpt-oss-120b` loop within a single attempt and explore redundantly across attempts in iterative refinement settings, reflecting failures of compositional generalization and inefficient search. LLM-based search performs well on short, simple programs but degrades significantly on longer, compositional ones, which we hypothesize is because LLMs lack an explicit mechanism for structuring search over candidate programs, leading to wasted computation on difficult tasks.

A natural approach is to augment LLMs with tools or libraries of reusable abstractions induced from experience (Wang et al., 2024b; Stengel-Eskin et al., 2024; Yue et al., 2025), but compute-matched evaluations show that their gains come mostly from extra inference-time compute rather than effective tool reuse (Sesterhenn et al., 2025). We observe the same for TroVE in a matched harness: its toolbox fills with near-duplicate task-specific functions, and reusing them lowers accuracy (§4).

In this work, we propose ReaComp: instead of improving inference-time reasoning, we *compile* it into a reusable symbolic structure. Our approach consists of two stages: **offline solver induction** and **test-time inference**. In the offline phase, we induce *symbolic solvers* directly from LLM reasoning traces by using coding agents to distill recurring strategies and failure modes into standalone program synthesizers over a constrained DSL. These solvers require no LLM calls at test time and can solve a large fraction of tasks directly, often matching or exceeding LLM search on harder instances. At test time, the solver provides a cheap, structured first pass, and the LLM is invoked only as a fallback on unresolved cases (Zhang & Park, 2025).

We evaluate ReaComp on recent, challenging benchmarks, PBEBench (Naik et al., 2025b) and SLR-Bench (Helff et al., 2025), and show that induced symbolic solvers achieve strong standalone performance while significantly improving inference efficiency. We target verifier-backed domains, where a DSL can enumerate over hypotheses. Symbolic solver ensembles reach 91.3% accuracy on PBEBench-Lite and 84.7% on PBEBench-Hard, outperforming a test-time scaling LLM baseline on Hard by +16.3 percentage points (pp) at zero LLM inference cost. On SLR-Bench, symbolic solvers match frontier LLMs on the hardest tier and, when combined with direct feedback, improve hard-tier accuracy from 34.4% to 58.0%. Hybrid inference reduces token usage by up to 78%. By amortizing a small one-time construction cost over many zero-token executions, induced solvers Pareto-dominate approaches that directly apply the same coding agent to each task individually. Finally, most solvers transfer zero-shot to forward reconstruction in historical linguistics, predicting how sounds change across related languages, reaching 80.1% accuracy under ensembling and recovering plausible sound change rules.

Contributions. We propose ReaComp, contributing: (1) a method for inducing reusable symbolic solvers from LLM reasoning traces via coding agents; (2) evidence that induced solvers are strong standalone systems, outperforming LLM test-time scaling on hard instances at zero per-task LLM inference cost, the coding agent that built the solver, and a matched-harness library-learning baseline (TroVE) by wide margins; (3) a hybrid neuro-symbolic pipeline that Pareto-dominates per-task agentic effort, reducing token usage by up to 78% while achieving state-of-the-art results; (4) a demonstration that most induced solvers generalize zero-shot to novel distributions, with a case study on forward reconstruction in historical linguistics.

2 Method

Problem formulation. Let $\mathcal{X}$ and $\mathcal{Y}$ denote the input and output spaces. A PBE task is a finite set of examples $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$. The goal is to infer a program $p \in \mathcal{P}$ (from a DSL $\mathcal{P}$) such that $p(x_i) = y_i$ for all examples. We use a task-level reward $R(p; \mathcal{D}) \in [0, 1]$,

with $R = 1$ indicating a fully correct solution. We propose ReaComp, a two-stage approach that (1) compiles LLM reasoning traces into symbolic solvers and (2) uses these solvers in a hybrid inference pipeline with LLM-based search.

2.1 LLM Search Procedures

An LLM defines a conditional distribution $p \sim \pi_\theta(\cdot \mid \mathcal{D})$. We consider two standard test-time scaling strategies, both selecting $p^* = \arg\max_{p \in \mathcal{C}} R(p; \mathcal{D})$:

Best-of-K (BoK). Sample K programs independently and select the highest-scoring candidate: $\mathcal{C} = \{p_k\}_{k=1}^K$.
Direct Feedback (DF). Generate a trajectory $\{p^{(t)}\}_{t=1}^T$ via iterative refinement conditioned on verifier feedback: $\mathcal{C} = \{p^{(t)}\}_{t=1}^T$. Early stopping is applied for both when $R = 1$.

Both define candidate sets $\mathcal{C}$ used in our hybrid inference.

2.2 Offline Solver Induction

Solver induction is a one-time offline procedure that compiles reasoning traces into a standalone symbolic program synthesizer. Given training tasks $\{\mathcal{D}_j\}_{j=1}^m$ and corresponding traces $\{\tau_j\}_{j=1}^m$, we induce a solver $S_\phi : \mathcal{D} \mapsto \mathcal{P}$ by approximately maximizing $\sum_j R(S_\phi(\mathcal{D}_j); \mathcal{D}_j)$ via a coding agent that distills reusable structure and failure modes from traces.

Trace dataset. We construct a dataset $\mathcal{T} = \{(\mathcal{D}_j, \tau_j)\}_{j=1}^m$ of reasoning traces generated by an LLM (e.g., `gpt-oss-120b`), balanced across task difficulty and success/failure. Each trace contains intermediate reasoning steps, candidate programs, and outcomes. The induction tasks are a small subset of the benchmark tasks; no ground-truth programs are accessed during induction. The coding agent observes only the input-output examples available to any solver at test time. Induction is therefore label-free: the induction instances are unlabeled benchmark tasks that supply reasoning traces (including failed attempts). The trace LLM (`gpt-oss-120b`, fixed across all runs and ablations) is distinct from the coding agent that writes the solver.

Coding-agent-based induction. We use a coding agent $\mathcal{A}_\psi$ to iteratively construct a solver implementation. The agent operates over a sandboxed container containing the trace dataset and the verifier code (read-only access to avoid hacking the verifier). Access to the verifier allows the agent to test candidate solvers and iteratively debug or refine them. At each step, the agent proposes edits to the workspace, executes code, and refines the solver based on performance over $\mathcal{T}$. The final output is a standalone Python solver file `SOLVER.py` and a corresponding algorithm description file (`SOLVER_ALGORITHM.md`). The building prompt (Appendix D.2) fixes the solver's interface (function signature, DSL, read-only verifier access, behavioral requirements) but not its search algorithm. The solver emerges from a multi-step agentic loop (49–102 turns per QO run; Table 14), and identical inputs yield qualitatively different algorithms across runs (Appendix F.2).

We experiment with two coding agents for solver induction: (i) **Claude Code (CC)** using `claude-sonnet-4-6` in an unconstrained interactive session, and (ii) **Qwen + OpenHands (QO)** using `Qwen3.6-35B-A3B` with a bounded trajectory (max 500 steps per run) to model a resource-constrained setting. For both agents, we construct a trace dataset of approximately 100 examples per benchmark (PBEBench and SLR-Bench), sampled to balance task difficulty and outcome (success vs. failure), enabling the agent to learn both effective strategies and common failure modes. We additionally perform ablations with QO on PBEBench to study the impact of reasoning traces (CoT vs. no-CoT), trace dataset size (100, 48, 12), and solver algorithm variance across three independent induction trajectories with the same trace dataset.

2.3 Efficient Hybrid Inference

At test time, the solver proposes a set of candidates $\mathcal{C}_S = S_\phi(\mathcal{D})$ and selects the best $p_S = \arg\max_{p \in \mathcal{C}_S} R(p; \mathcal{D})$; if $R(p_S; \mathcal{D}) = 1$ it is accepted at zero LLM cost. Otherwise we fall back to LLM search (BoK or DF) and return $\arg\max_{p \in \mathcal{C}_L} R(p; \mathcal{D})$, where $\mathcal{C}_L$ is the LLM candidate set. The fallback is independent: the LLM does not see or refine the solver's near-miss candidates. We also report a solver-seeded variant that warm-starts the fallback from the near-miss (Tables 1 and 3; Appendix F.8). This amortizes solver construction over many zero-token executions: S_ϕ handles the majority of tasks directly while LLM search is reserved for harder residual cases.

Building prompts, solver interfaces, and agent configurations are provided in Appendix D.2. Algorithms 1 and 2 (Appendix D.1) formalize both steps.

3 Experimental Setup

We evaluate whether LLM reasoning traces can be compiled into reusable symbolic solvers that improve accuracy, efficiency, and performance on hard compositional program induction tasks. Experiments span two domains: programming-by-example (PBE) string transformation and scalable logical reasoning (SLR).

Benchmarks and metrics. We evaluate on PBEBench Naik et al. (2025b) (PBEBench-Lite and PBEBench-Hard) and SLR-Bench Helff et al. (2025). PBEBench requires synthesizing ordered cascades of string rewrite programs from input-output examples. PBEBench-Lite contains 1,008 tasks with cascade lengths 2–5 (approximately 252 tasks per level) and a maximum program budget of 5. PBEBench-Hard contains 1,216 tasks with cascade lengths 2–20 (64 tasks per level) and a maximum program budget of 20. SLR-Bench contains 1,000 inductive logic programming tasks across curriculum levels 1–20, grouped into four tiers (basic, easy, medium, hard). Each task requires inducing a Prolog rule from background facts and labeled examples. PBEBench programs are evaluated by exact execution and SLR-Bench rules by the official `partial_score` reward. Evaluation is fully deterministic and verifier-based: a task is solved only when the reward is 1.0.

DSLs. *PBEBench:* a program is an ordered cascade $p = (r_1, \ldots, r_k)$ of rewrites $r_i = \texttt{replace}(A_i, B_i)$ with $|A_i| \in [1, 3]$, $|B_i| \in [0, 3]$, applied in order to the input string. *SLR-Bench:* a program is a Prolog rule `eastbound(T) :- Body.`, where `Body` is built from literals over the background predicates (e.g., `has_car/2`, `car_color/2`).

We report accuracy (perfect solution found), mean reward, token usage and cost (token accounting discussed in §E.1). We additionally report a complexity gap Δ between predicted and ground-truth programs. For PBEBench, we follow the complexity measure defined in the original paper, but compute it *only over correctly solved tasks*: incorrect outputs are often degenerate and would bias the mean downward if included. For SLR-Bench, Δ is defined analogously using rule complexity (number of top-level body literals excluding `has_car/2`), and is also reported only on correct solutions. We further report edit similarity on PBEBench and tier-wise accuracy on SLR-Bench.

LLM baselines. We instantiate the LLM search procedures described in §2 using `gpt-oss-120b` served via vLLM. BoK (Best-of-L chosen by verifier) and DF (Direct Feedback by verifier) together cover the two dominant paradigms for test-time LLM scaling. We evaluate BoK with $K = 32$ and DF with up to 32 refinement steps and early stopping; temperature is fixed at 1.0 for both. Token budgets are 32,768 tokens per attempt on PBEBench-Lite and SLR-Bench, and 16,384 on PBEBench-Hard. On PBEBench-Hard, we evaluate BoK only, as DF becomes prohibitively expensive and prone to lock-in for long cascades. We additionally include a coding-agent baseline (`Qwen3.6-35B-A3B` via OpenHands, 100-step budget per task) that directly solves each task in isolation, providing a comparison point that isolates the contribution of solver induction over per-task agentic effort. We also include a library-learning baseline, TroVE (Wang et al., 2024b), in the same `Qwen3.6-35B-A3B` + OpenHands + verifier harness, compute-matched to the coding-agent

baseline (CoT, 16K tokens per call), on PBEBench-Lite and SLR-Bench (a lower-budget default configuration is in Appendix F.7).

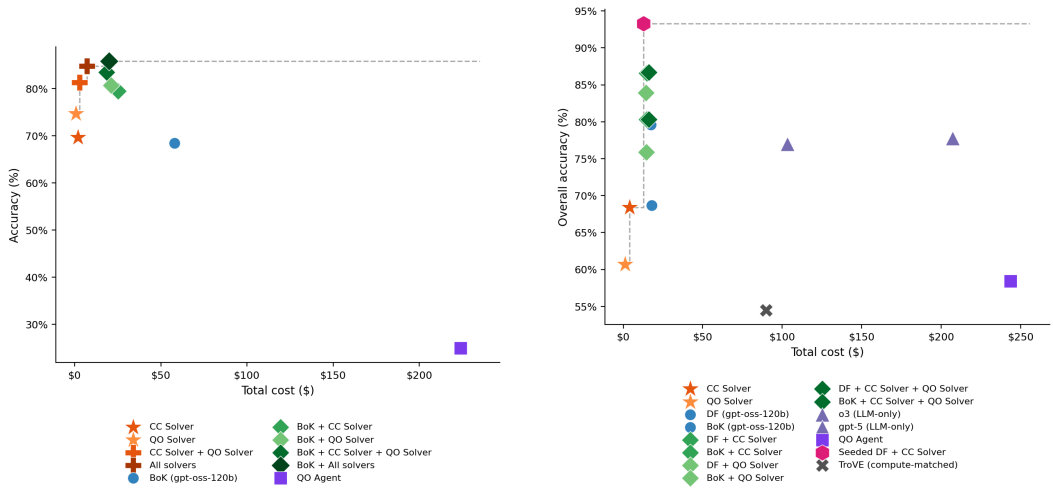
Symbolic solver induction and hybrid inference. We induce solvers offline as described in §2, using **CC** (`claude-sonnet-4-6`) and **QO** (`Qwen3.6-35B-A3B`). Because coding-agent stochasticity yields distinct implementations, we report individual solvers and unions across runs: CC+QO denotes the union of the CC and QO solvers, while **"All Symbolic"** denotes all 6 QO solvers from the PBEBench ablations plus the CC solver. We omit **"All Symbolic"** for SLR-Bench, where only one QO and one CC solver are induced. All solvers are standalone, require no LLM calls at inference time, and are effectively deterministic (<1% per-task flips on PBEBench-Hard). Each solver call runs in its own child process under a per-task time limit (60 s for PBEBench; 3600 s for SLR-Bench, where most time is spent in SWI-Prolog verifier calls of ≈300 ms each); tasks that exceed it count as failures. We evaluate the hybrid procedure from §2, applying symbolic solvers before LLM search.

Table 1: **PBEBench-Lite results.** We compare reported single-attempt baselines, direct coding agent solving, test-time scaling, coding agent-induced symbolic solvers, and efficiency-oriented hybrids. Δ denotes predicted minus ground-truth cascade complexity on correct solutions only. Edit similarity is averaged over parseable answers. ‡Only 77% of TroVE answers parse, versus ≥98.8% for all other systems.

System	Acc%	Mean Reward	Edit Sim	Δ	Tokens (M)	Inf. Cost ($)	Constr. Cost ($)
LLM-only methods							
gpt-oss-120b (single)†	62.5	—	69.9	—	—	—	—
GPT-5 (single)†	72.4	—	76.5	—	—	—	—
Qwen3.6-35B-A3B (OpenHands)	87.2	0.9544	94.9	+1.47	395.3	85.38	—
DF-32 (gpt-oss-120b)	92.4	0.9796	97.3	+2.11	111.1	16.74	—
BoK (gpt-oss-120b)	93.8	0.9808	**97.8**	+2.19	68.0	12.20	—
Library learning (same Qwen3.6-35B-A3B + OpenHands harness)							
TroVE (compute-matched)	53.2	0.6323	85.3‡	+0.94	71.1	64.60	—
Symbolic solvers							
CC	80.4	0.9438	93.7	+3.00	**0**	**0**	2.00
QO	65.7	0.9022	87.4	+3.01	**0**	**0**	0.85
CC + QO (union)	84.6	0.9607	94.9	+2.16	**0**	**0**	2.85
All Symbolic	91.3	0.9772	96.6	+1.88	**0**	**0**	7.11
Hybrid (symbolic + LLM fallback)							
DF + QO	92.9	0.9810	97.5	+2.89	90.2	13.50	0.85
DF + CC	93.1	0.9815	97.6	+3.00	80.5	11.97	2.00
DF + All Symbolic	93.2	0.9817	97.6	+2.18	78.7	11.69	7.11
BoK + QO	93.8	0.9808	97.8	+2.94	50.7	9.12	0.85
BoK + CC	93.9	0.9810	97.8	+2.94	45.6	8.22	2.00
BoK + All Symbolic	93.9	0.9810	97.8	+2.19	43.5	7.83	7.11
Hybrid with solver-seeded fallback (DF warm-started from the solver's near-miss; Appendix F.8)							
Seeded DF + CC	**94.5**	**0.9875**	**98.3**	+3.00	41.2	6.45	2.00

Table 2: **PBEBench-Hard results.** We compare direct coding-agent solving, LLM test-time search, coding-agent-induced symbolic solvers, and efficiency-oriented hybrids on long-horizon cascade synthesis. Δ denotes predicted minus ground-truth cascade complexity on correct solutions only.

System	Acc%	Mean Reward	Edit Sim	Δ	Tokens (M)	Inf. Cost ($)	Constr. Cost ($)
LLM-only methods							
BoK	68.4	0.9428	89.9	+5.14	332.1	57.83	—
Qwen3.6-35B-A3B (OpenHands)	24.9	0.8411	73.2	+1.36	1046.8	∼224	—
Symbolic solvers							
CC Solver	69.7	0.9873	97.2	+8.06	**0**	**0**	2.00
QO Solver	74.7	0.9836	96.8	+5.26	**0**	**0**	0.85
CC + QO	81.2	0.9905	98.3	+5.35	**0**	**0**	2.85
All Symbolic	84.7	**0.9920**	**98.6**	+4.56	**0**	**0**	7.11
Hybrid (symbolic + LLM fallback)							
BoK + CC	79.4	0.9508	91.4	+7.82	130.0	23.08	2.00
BoK + QO	80.7	0.9496	91.3	+5.48	114.5	20.41	0.85
BoK + CC + QO	83.5	0.9531	91.9	+5.48	87.6	15.63	2.85
BoK + All Symbolic	**85.8**	0.9570	92.7	+4.64	71.6	12.78	7.11



(a) PBEBench-Hard. Frontier: QO → CC+QO → All solvers → BoK+All solvers.

(b) SLR-Bench. Frontier: QO → CC → Seeded DF+CC.

Figure 2: **Cost–performance Pareto frontiers.** Symbolic solvers dominate the low-cost regime, while hybrid methods achieve the best trade-offs and absolute performance (PBEBench-Lite frontier in Appendix F.1, Figure 4).

Table 3: **SLR-Bench results.** We compare reported frontier-model baselines Helff et al. (2025), direct coding-agent solving, LLM test-time search, coding-agent-induced symbolic solvers, and efficiency-oriented hybrids on inductive Prolog rule synthesis. Tier columns report accuracy by curriculum tier (Basic / Easy / Medium / Hard), each with 250 tasks. Δ denotes predicted minus ground-truth rule complexity on correct solutions only.

System	Acc%	Basic	Easy	Medium	Hard	Tokens (M)	Inf. Cost ($)	Constr. Cost ($)	Δ
LLM-only methods									
o3†	77.8	99	93	74	45	4.30	207.24	—	—
gpt-5†	77.0	100	90	72	46	16.40	103.13	—	—
BoK	68.7	100	100	57.6	17.2	225.3	17.88	—	-0.611
DF-32	79.6	100	99.6	84.4	34.4	224.2	17.43	—	-0.834
Qwen3.6-35B-A3B (OpenHands)	58.4	100	86.0	32.8	14.8	1232.6	243.41	—	-0.354
Library learning (same Qwen3.6-35B-A3B + OpenHands harness)									
TroVE (compute-matched)	54.5	100	76.4	29.6	12.0	210.6	89.83	—	-0.499
Symbolic solvers									
CC Solver	68.4	100	78.4	48.4	46.8	**0**	**0**	4.01	-0.756
QO Solver	60.7	100	71.2	34.4	37.2	**0**	**0**	1.28	-1.087
Hybrid (symbolic + LLM fallback)									
BoK + QO	75.9	100	100	62.8	40.8	162.8	13.21	1.28	-1.063
BoK + CC	80.3	100	100	68.4	52.8	132.4	10.80	4.01	-0.796
BoK + CC + QO	80.3	100	100	68.4	52.8	132.4	10.80	5.29	-0.795
DF + QO	83.9	100	99.6	86.4	49.6	166.1	13.01	1.28	-1.123
DF + CC	86.6	100	99.6	88.8	58.0	138.8	10.90	4.01	-0.924
DF + CC + QO	86.7	100	99.6	88.8	58.4	138.8	10.90	5.29	-0.924
Hybrid with solver-seeded fallback (DF warm-started from the solver's near-miss; Appendix F.8)									
Seeded DF + CC	**93.3**	100	100	**96.8**	**76.4**	100.2	8.75	4.01	-1.023

4 Results

For symbolic solvers, we report one-time solver construction cost (no per-task LLM usage). For hybrid methods, reported tokens reflect *LLM fallback inference only* (tasks solved by the symbolic solver incur zero tokens); we report per-task inference cost and the *one-time solver construction cost* in separate columns (total cost is their sum). † denotes reported results from prior work.

PBEBench-Lite. Results are shown in Table 1 (full details in Appendix F.1). The QO solver shown corresponds to "100 examples + CoT, run 2" out of the 6 ablations in Appendix F.2. LLM baselines achieve strong performance with test-time scaling (BoK: 93.8%, DF: 92.4%), but incur substantial token usage. A direct coding-agent baseline (OpenHands)

achieves 87.2% accuracy, outperforming individual symbolic solvers but falling below LLM scaling methods at substantially higher cost. Induced symbolic solvers are competitive as standalone systems, with the CC solver reaching 80.4% accuracy at zero inference cost, outperforming single-attempt LLM baselines. Combining multiple solvers yields 91.3% accuracy without any LLM usage, within 2.5pp of BoK. Hybrid inference substantially reduces LLM token usage: BoK + All Symbolic lowers total tokens from 68.0M to 43.5M (-36%) while maintaining peak accuracy among independent-fallback hybrids (93.9%). Among independent-fallback hybrids, BoK + QO achieves Pareto efficiency, matching peak BoK accuracy while reducing both token usage and total cost ($9.12 inference + $0.85 construction = $9.97 vs. $12.20). BoK + CC yields a marginal additional accuracy gain of 0.1pp at comparable cost ($8.22 inference + $2.00 construction = $10.22), as the one-time solver construction overhead ($2.00 CC, $0.85 QO) is negligible relative to per-task inference savings. Warm-starting the DF fallback from the CC solver's near-miss (Seeded DF + CC; Algorithm 3) raises accuracy to 94.5% while using about half the fallback tokens of plain DF + CC (41.2M vs. 80.5M), and Pareto-dominates BoK + QO on cost and accuracy ($6.45 inference + $2.00 construction = $8.45). Across all methods, predicted cascade complexity remains close to ground truth ($\Delta \approx +2$), indicating that accuracy gains are not attributable to trivial or degenerate programs.

PBEBench-Hard. Results are shown in Table 2 (full details in Appendix F.3). On long-horizon cascade synthesis (lengths 2–20), the performance regime shifts substantially relative to Lite: the direct coding-agent baseline (OpenHands) achieves only 24.9% accuracy at high token cost ($224), while induced symbolic solvers surpass LLM BoK scaling (68.4%) at zero per-task LLM cost. QO solver ("100 examples + CoT, run 2") reaches 74.7% and CC solver reaches 69.7%. The All Symbolic union (6 QO solvers and CC solver) further improves to 84.7%, exceeding BoK by 16.3pp with no LLM inference. Hybrid inference provides the largest absolute gains on this split: BoK + All Symbolic reaches 85.8% while reducing reported token cost by 78% relative to BoK alone (71.6M vs. 332.1M tokens; $12.78 vs. $57.83 inference cost, plus $7.11 one-time construction), as the solver eliminates LLM cost on the 84.7% of tasks it handles correctly. Predicted cascade complexity substantially overshoots ground truth ($\Delta \approx +5$–$+8$) on Hard, reflecting the difficulty of recovering compact solutions at long cascade lengths; the hybrid partially mitigates this by selecting simpler BoK candidates for tasks where the solver fails.

SLR-Bench. Results are shown in Table 3 (full details in Appendix F.6). Performance across tiers is highly non-uniform: BoK achieves 100% on Basic and Easy but collapses to 17.2% on Hard, while DF is more balanced (34.4% Hard) at comparable token cost. The direct coding-agent baseline consumes the most tokens by far while achieving the lowest overall accuracy and the weakest Hard-tier performance of any method we evaluate. Induced symbolic solvers exhibit the complementary profile: the CC Solver reaches 46.8% on Hard, matching the best reported frontier-model results (o3: 45%, GPT-5: 46%) at zero per-task LLM cost. Hybrid inference leverages this complementarity effectively: DF + CC Solver lifts Hard accuracy to 58.0%, surpassing all prior reported baselines on this tier. The best independent-fallback hybrid, DF + CC + QO, achieves 86.7% overall accuracy, almost +9pp above the best SLR-Bench leaderboard result (o3: 77.8%), at 138.8M tokens ($10.90 inference + $5.29 one-time construction = $16.19), compared to $207.24 for o3. Warm-starting the DF fallback from the CC solver's near-miss (Seeded DF + CC) raises overall accuracy to 93.3% and Hard-tier accuracy to 76.4%, with fewer fallback tokens than plain DF + CC (100.2M vs. 138.8M) and a lower total cost ($8.75 inference + $4.01 construction = $12.76). In contrast to PBEBench, all systems predict *simpler* rules than ground truth ($\Delta < 0$), reflecting the abundance of shorter equivalent Prolog forms; symbolic solvers undershoot most aggressively, consistent with their ascending-complexity search strategy.

Library-learning baseline (TroVE). Even compute-matched, TroVE (53.2% Lite, 54.5% SLR) trails the plain coding agent in the same harness and the best ReaComp hybrid by 40.7pp and 32.2pp (12.0% vs. 46.8% for the CC solver on SLR Hard). Its toolbox memorizes rather than abstracts: importing a toolbox function solves 0.5% of SLR-Bench tasks versus 92% when writing a fresh solution, consistent with Sesterhenn et al. (2025); we do not claim library learning fails in general (Appendix F.7).

List Functions. To check whether the approach carries over to other PBE domains, we also run ReaComp on the 250 tasks of BIG-bench List Functions (Rule, 2020), where each task maps integer lists to integer lists and is scored on 8 held-out examples. The union of the induced solvers solves 51.6% of tasks without any LLM calls. Running this union before BoK improves accuracy from 87.6% (BoK alone) to 90.4% and reduces LLM token usage by 36.2% (Appendix F.9).

Solver induction ablations. We study how the trace dataset composition affects induced solver quality (§F.2). Removing reasoning traces (no-CoT) significantly degrades performance, especially on harder instances (e.g. PBEBench-Hard accuracy drops from 74.7% to 24.8%), indicating that final programs alone are insufficient supervision for learning robust solvers. In contrast, reducing the number of CoT examples has a smaller and less consistent effect: a 48-example CoT solver achieves performance comparable to some 100-example runs, while 12 examples is clearly insufficient. Notably, performance differences across solvers induced on the same trace dataset (100-examples + CoT) can be large (e.g., 53.4%–79.2% on Lite, 51.8%–74.7% on Hard), suggesting that solver quality is highly sensitive to the algorithmic strategy discovered during induction rather than trace dataset size alone.

Real-world case study: forward reconstruction. We evaluate induced symbolic solvers on a real-world forward reconstruction task (data construction details in Appendix F.4) from historical linguistics, consisting of 3,077 proto-daughter language pairs with large variation in number of examples (ranging from 1 to 857) and an unseen IPA alphabet. Without any retraining, both CC and Qwen solvers achieve strong zero-shot performance ($\approx$ 70% accuracy), demonstrating robustness to distribution shift beyond synthetic PBEBench tasks. Increasing the maximum cascade length from 20 to 100 yields only modest gains (e.g., 69.0%→70.2% for CC), suggesting that performance is not primarily limited by search depth. However, one independently induced solver (the unique-op permutation strategy in §F.2) achieves only 4% accuracy under distribution shift, illustrating that algorithmic brittleness, not data shift alone, limits generalization. Combining solvers via union substantially improves robustness, reaching up to 80.1% accuracy and demonstrating strong complementarity. Additional analysis, including compression experiments and qualitative examples, is provided in Appendix F.4.

5 Discussion

Efficiency under saturated performance. On PBEBench-Lite, accuracy is already near saturation for LLM-based methods, leaving limited room for improvement. In this regime, the primary benefit of solver induction is improved efficiency rather than accuracy. Hybrid inference substantially reduces LLM token usage, indicating that many tasks can be resolved through reusable symbolic structure instead of repeated LLM search. The token reductions are proportional to solver coverage and independent of per-token pricing, suggesting the efficiency gains scale to any deployment context. Beyond tokens, solver inference completes in under 5 minutes for all 1000+ tasks (pure Python, no LLM calls), compared to $\sim$2 days for DF or BoK at 8 parallel workers, a wall-clock gap of over 500$\times$ (Appendix C). Overall, these results indicate that even lightweight solvers suffice in saturated regimes, while more powerful solvers become critical on harder tasks.

Symbolic solvers as primary baselines on Hard tasks. On PBEBench-Hard, the performance landscape inverts: LLM test-time scaling becomes the weaker baseline while symbolic solvers emerge as the primary competitive systems. This inversion reflects the fundamental mismatch between independent LLM sampling and long-horizon cascade tasks: at cascade lengths 14–17, BoK collapses to below 47% while symbolic solvers maintain 55–70% accuracy through structured search. Notably, hybrid mean reward on Hard (e.g., BoK+CC: 0.9508) is lower than the standalone CC Solver (0.9873): when the solver fails, the LLM fallback is invoked independently and its partial-credit answers on hard tasks score lower than the solver's near-misses. Warm-starting the fallback from the solver's near-miss instead recovers much of this gap (e.g., +21.2pp on the SLR-Bench tasks the solver fails) while using fewer fallback tokens (Tables 1 and 3; Appendix F.8). The same pattern holds on SLR-Bench, where the Hard tier exposes a sharp divide between LLM and symbolic

approaches. Notably, the direct coding-agent baseline, despite consuming the most tokens of any system, achieves the lowest hard-tier performance, illustrating that per-task agentic effort does not substitute for the structured inductive search that symbolic solvers perform. Hybrid methods combine both strengths: DF handles the Basic and Easy tiers near-perfectly while the symbolic solver provides hard-tier robustness that LLM search alone cannot reach, yielding the best overall and hard-tier results across all benchmarks evaluated.

Solver induction ablations confirm that reasoning traces are critical, and that variance across induction runs matters more than dataset size, suggesting solver induction is a search problem over algorithmic space rather than a data-scaling problem.

Generalization to real forward reconstruction. Most induced solvers transfer zero-shot to real IPA data (Appendix F.4), but compression analysis reveals a gap between unconstrained accuracy (≈70–80%) and compact-rule performance (≈30%), suggesting current reward functions favor per-example patching over discovering generalizable structure. Although the solvers do recover some plausible sound laws, this tendency is a formidable limitation for linguistic applications.

6 Related Work

Neurosymbolic Reasoning with LLMs. A broad line of work improves LLM reasoning by coupling neural generation with symbolic execution, external tools, or explicit verification (Gao et al., 2023; Yao et al., 2022; Wang et al., 2024a); we discuss this work in Appendix G. This line of work, however, largely treats execution as an online verifier, tool-use substrate, or feedback signal for individual LLM-generated trajectories. In contrast, ReaComp uses execution traces offline to induce a reusable symbolic solver, which is then deployed at test time before falling back to LLM search only on residual cases. Vision-Language Programs (Wüst et al., 2026) combine VLM functions with task-specific induced primitives during program synthesis, and ActivationReasoning (Helff et al., 2026) applies logical reasoning over SAE-derived activation concepts of LLMs; both couple neural perception with symbolic structure at inference time, whereas ReaComp compiles the reasoning itself into a standalone solver.

Program Induction with LLMs. Most prior work on program induction from examples uses neural models or LLMs as per-task generators or search heuristics. What remains less explored is whether the search behavior of an LLM itself can be compiled into a standalone solver that amortizes inference across tasks. ReaComp addresses this gap by using a coding agent to induce symbolic solvers from LLM reasoning traces, then combining the induced solver with BoK or DF search in a hybrid test-time pipeline. We also discuss prior work on program induction from examples and work related to library learning in Appendix G, along with RL and latent-reasoning formulations (Lee et al., 2024; Baek et al., 2026), ARC and DSL-completeness, and refinement methods.

7 Conclusion and Future Work

We show that, in domains with a enumerable DSL and a fast exact verifier, LLM reasoning traces can be compiled into reusable symbolic solvers that are strong standalone systems, match or exceed frontier-model performance on hard tasks at zero per-task inference cost, and substantially reduce token usage as hybrids, Pareto-dominating direct per-task agentic effort. The induced solvers are complementary to LLM search: they excel precisely where LLMs fail (long-horizon cascades, hard Prolog tiers) while LLMs cover the remaining tasks, and the best hybrids set new state-of-the-art results on both PBEBench and SLR-Bench. Most solvers also generalize to an unseen IPA alphabet and real linguistic data without any source code/algorithmic changes, though compact-rule performance (≈30%) lags behind unconstrained accuracy (≈70%), suggesting solvers favor verifier-satisfying solutions over generalizable structure. Future work could explore more extensive search over solver induction runs to optimize for robustness to domain shift, computational efficiency, or program complexity (the latter being directly relevant for linguistic interpretability); iterative solver improvement via targeted coding-agent debugging sessions; and automatic selection or mixture of induced solvers based on task features.

AI use statement

We used a generative AI coding agent (Claude Code) to help write the code for experiments and to carry out quantitative analysis of the results. No generative AI tools were used to formulate research questions, ideas, or hypotheses, which are original contributions of the authors. All datasets are used from prior work and cited appropriately. AI is also not used for any theoretical claims, proofs or translations, as these are not applicable to our work. We have reviewed all AI-assisted work: AI-assisted code was reviewed and tested by the authors, and all reported numbers were checked against the experiment logs. We take responsibility for the final content of this work, including text, claims, or artifacts (like code) produced with the aid of generative AI. We have also submitted all relevant artifacts, including traces, solvers, and datasets for scrutiny/review.

Reproducibility statement

We describe both steps of ReaComp in Section 2 and formalize them in Appendix D.1; the full solver-building prompts are in Appendix D.2. Models, sampling settings, token budgets, and baselines are specified in Section 3 and Appendix E, including how tokens and costs are counted (Appendix E.1). Hardware, serving setup, pricing, and wall-clock times are listed in Appendix C, and the statistical tests are described in Appendix F.5. The supplementary material contains our code, all induced solvers, model predictions, and running instructions, along with the PBEBench and SLR-Bench test sets and the forward reconstruction dataset used in our case study.

References

Junyeob Baek, Mingyu Jo, Minsu Kim, Mengye Ren, Yoshua Bengio, and Sungjin Ahn. Generative recursive reasoning. *arXiv preprint arXiv:2605.19376*, 2026.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Shraddha Barke, Emmanuel Anaya Gonzalez, Saketh Ram Kasibatla, Taylor Berg-Kirkpatrick, and Nadia Polikarpova. Hysynth: Context-free llm approximation for guiding program synthesis. *Advances in Neural Information Processing Systems*, 37:15612–15645, 2024.

Matthew Bowers, Theo X Olausson, Lionel Wong, Gabriel Grand, Joshua B Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proceedings of the ACM on Programming Languages*, 7(POPL):1182–1213, 2023.

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.

Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. Arc-agi-2: A new challenge for frontier ai reasoning systems. *arXiv preprint arXiv:2505.11831*, 2025.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pp. 990–998. PMLR, 2017.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pp. 835–850, 2021.

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

Subha Ghoshal and Ali Al-Bustami. When do tools and planning help large language models think? a cost-and latency-aware benchmark. In *SoutheastCon 2026*, pp. 01–08. IEEE, 2026.

Gabriel Grand, Lionel Wong, Maddy Bowers, Theo X Olausson, Muxin Liu, Joshua B Tenenbaum, and Jacob Andreas. Lilo: Learning interpretable libraries by compressing and documenting code. *arXiv preprint arXiv:2310.19791*, 2023.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 317–330. ACM, 2011. doi: 10.1145/1926385.1926423.

Lukas Helff, Ahmad Omar, Felix Friedrich, Antonia Wüst, Hikaru Shindo, Rupert Mitchell, Tim Woydt, Patrick Schramowski, Wolfgang Stammer, and Kristian Kersting. Slr: Automated synthesis for scalable logical reasoning. *arXiv preprint arXiv:2506.15787*, 2025.

Lukas Helff, Ruben Härle, Wolfgang Stammer, Felix Friedrich, Manuel Brack, Antonia Wüst, Hikaru Shindo, Patrick Schramowski, and Kristian Kersting. Activationreasoning: Logical reasoning in latent activation spaces. In *International Conference on Learning Representations*, volume 2026, pp. 57457–57480, 2026.

Anubha Kabra, Sanketh Rangreji, Yash Mathur, Aman Madaan, Emmy Liu, and Graham Neubig. Program-aided reasoners (better) know what they know. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 2262–2278, 2024.

Hosung Lee, Sejin Kim, Seungpil Lee, Sanha Hwang, Jihwan Lee, Byung-Jun Lee, and Sundong Kim. Arcle: The abstraction and reasoning corpus learning environment for reinforcement learning. *arXiv preprint arXiv:2407.20806*, 2024.

Chunyang Li, Weiqi Wang, Tianshi Zheng, and Yangqiu Song. Patterns over principles: The fragility of inductive reasoning in llms under noisy observations. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 19608–19626, 2025a.

Wen-Ding Li and Kevin Ellis. Is programming by example solved by llms? *Advances in Neural Information Processing Systems*, 37:44761–44790, 2024.

Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer Dunn, Hao Tang, Wei-Long Zheng, Yewen Pu, et al. Combining induction and transduction for abstract reasoning. In *International Conference on Learning Representations*, volume 2025, pp. 20981–21024, 2025b.

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.

Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, Jun 1947. ISSN 1860-0980. doi: 10.1007/BF02295996. URL `https://doi.org/10.1007/BF02295996`.

Atharva Naik, Darsh Agrawal, Hong Sng, Clayton Marr, Kexun Zhang, Nathaniel R Robinson, Kalvin Chang, Rebecca Byrnes, Aravind Mysore, Carolyn Rose, et al. Programming by examples meets historical linguistics: A large language model based approach to sound law induction. *arXiv preprint arXiv:2501.16524*, 2025a.

Atharva Naik, Yash Mathur, Darsh Agrawal, Manav Kapadnis, Yuwei An, Clayton Marr, Carolyn Rose, David Mortensen, et al. Pbebench: A multi-step programming by examples reasoning benchmark inspired by historical linguistics. *arXiv preprint arXiv:2505.23126*, 2025b.

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. Bustle: Bottom-up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381*, 2020.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. *arXiv preprint arXiv:2308.03188*, 2023.

Linlu Qiu, Liwei Jiang, Ximing Lu, Melanie Sclar, Valentina Pyatkin, Chandra Bhagavatula, Bailin Wang, Yoon Kim, Yejin Choi, Nouha Dziri, et al. Phenomenal yet puzzling: Testing inductive reasoning capabilities of language models with hypothesis refinement. In *International Conference on Learning Representations*, volume 2024, pp. 1452–1481, 2024.

Joshua S Rule, Steven T Piantadosi, Andrew Cropper, Kevin Ellis, Maxwell Nye, and Joshua B Tenenbaum. Symbolic metaprogram search improves learning efficiency and explains rule learning in humans. *Nature Communications*, 15(1):6847, 2024.

Joshua Stewart Rule. *The child as hacker: building more human-like models of learning*. PhD thesis, Massachusetts Institute of Technology, 2020.

Tobias Sesterhenn, Ian Berlot-Attwell, Janis Zenkner, and Christian Bartelt. A compute-matched re-evaluation of trove on math. *arXiv preprint arXiv:2507.22069*, 2025.

Yunfan Shao, Linyang Li, Yichuan Ma, Peiji Li, Demin Song, Qinyuan Cheng, Shimin Li, Xiaonan Li, Pengyu Wang, Qipeng Guo, et al. Case2code: Scalable synthetic data for code generation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 11056–11069, 2025.

Shivam Singhal, Eran Malach, Tomaso Poggio, and Tomer Galanti. Llm-erm: Sample-efficient program learning via llm-guided search. *arXiv preprint arXiv:2510.14331*, 2025.

Elias Stengel-Eskin, Archiki Prasad, and Mohit Bansal. Regal: Refactoring programs to discover generalizable abstractions. *arXiv preprint arXiv:2401.16467*, 2024.

Gust Verbruggen, Ashish Tiwari, Mukul Singh, Vu Le, and Sumit Gulwani. Execution-guided within-prompt search for programming-by-example. In *The Thirteenth International Conference on Learning Representations*, 2025.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024a.

Zhiruo Wang, Daniel Fried, and Graham Neubig. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks. *arXiv preprint arXiv:2401.12869*, 2024b.

Antonia Wüst, Wolfgang Stammer, Hikaru Shindo, Lukas Helff, Devendra Singh Dhami, and Kristian Kersting. Synthesizing visual concepts as vision-language programs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 17346–17356, 2026.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. Rewoo: Decoupling reasoning from observations for efficient augmented language models. *arXiv preprint arXiv:2305.18323*, 2023.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Murong Yue, Zhiwei Liu, Liangwei Yang, Jianguo Zhang, Zuxin Liu, Haolin Chen, Ziyu Yao, Silvio Savarese, Caiming Xiong, Shelby Heinecke, et al. Toollibgen: Scalable automatic tool creation and aggregation for llm reasoning. *arXiv preprint arXiv:2510.07768*, 2025.

Shuning Zhang and Yongjoo Park. Pbe meets llm: When few examples aren't few-shot enough. *arXiv preprint arXiv:2507.05403*, 2025.

Yufeng Zhao, Junnan Liu, Hongwei Liu, Dongsheng Zhu, Yuan Shen, Songyang Zhang, and Kai Chen. Dissecting tool-integrated reasoning: An empirical study and analysis. *arXiv preprint arXiv:2508.15754*, 2025.

A Broader Impact

This work studies the induction of symbolic solvers from LLM reasoning traces and their combination with LLM test-time search for program synthesis tasks. The contributions are primarily methodological: we introduce a framework for coding-agent-induced solver induction and demonstrate its application on two benchmarks in program-by-example synthesis and inductive logic programming.

Positive impacts. Reducing the per-task LLM cost of program synthesis, through reusable symbolic solvers that replace repeated LLM inference, lowers the computational and financial barrier to deploying capable synthesis systems, potentially improving accessibility. Improved program synthesis tools have broad beneficial applications in data transformation, programming assistance, and scientific discovery (e.g., in formal linguistics, as demonstrated in our case study).

Potential negative impacts. We do not foresee direct negative societal impacts from this work. The string-replacement and Prolog-rule DSLs studied here are highly constrained and domain-specific, with no straightforward path to harmful applications such as disinformation generation, surveillance, or privacy violations. More generally, improvements in program synthesis efficiency could in principle accelerate the development of automation tools with broader downstream effects; however, this path is indirect and not specific to our method. The coding agents used in solver induction (LLMs instructed to write code) could in principle produce solvers that behave incorrectly or unpredictably, but our evaluation framework includes a verifier that explicitly checks correctness on held-out examples, limiting the risk of undetected failure.

Fairness and privacy. The benchmarks used in this work are synthetic or drawn from published linguistics datasets and contain no personal data. No demographic information is collected or modelled; no deployment decisions affecting individuals are made. We therefore identify no fairness or privacy concerns specific to this work.

B Limitations

Scope of evaluation. Our experiments are conducted on two program synthesis domains: string-replacement cascades (PBEBench) and binary train-classification rules (SLR-Bench). Both domains use highly structured, verifiable DSLs. It is not clear whether solver induction generalizes to domains with more complex DSLs, open-ended output spaces, or without a fast and exact verifier. The forward reconstruction case study (Appendix F.4) provides initial evidence of zero-shot transfer, but that domain shares structural properties with PBEBench. We therefore restrict our claims to verifier-backed PBE settings with a constrained DSL. A pilot on List Functions (Appendix F.9) is a third such domain, but there most of the hybrid's accuracy comes from the strong fallback model and the solvers' contribution is mainly efficiency. Benchmarks without a shared DSL or exact program verifier (e.g., ARC-AGI-2 beyond its DSL-expressible subset, ACRE) are outside the setting we study.

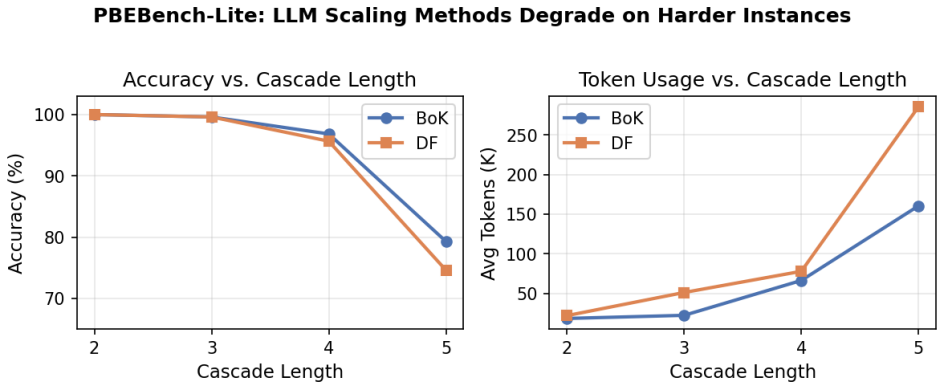


Figure 3: **LLM scaling methods degrade on harder PBEBench-Lite instances.** Accuracy (left) and average token usage (right) of Best-of-K (BoK) and Direct Feedback (DF) scaling strategies applied to gpt-oss-120b, broken down by cascade length on PBEBench-Lite. Both methods solve cascade-length-2 tasks perfectly but collapse at length 5: BoK accuracy drops to 79.3% and DF to 74.6%, while token usage explodes, DF consumes an average of 285K tokens per task at length 5 (13× more than at length 2), and BoK consumes 160K tokens (9× more). This shows that harder instances trigger more LLM attempts, compounding both cost and error.

Run-to-run variance in solver induction. Solver quality varies substantially across independent induction runs on identical demonstrations (PBEBench-Lite accuracy 53–79% for three identical 100-example CoT runs with Qwen3.6-35B-A3B). This variance reflects qualitatively different algorithms discovered by the coding agent across trajectories and is not reducible by fixing a random seed, as the agent's exploration is governed by its own sampling. Reported results for the QO Solver use the best single run (selected by performance on the induction demo set, not the test set); ensemble results over multiple runs are provided but depend on the number of induction attempts. The CC Solver was induced in a single session per benchmark; no multi-run variance estimate is available for it. This variance could confound the CoT and dataset-size ablations in Appendix F.2, since single ablation runs are compared against multiple CoT runs. However, we note that the performance reductions under the most extreme ablations, removing CoT entirely (Hard accuracy 24.8% vs. 51.8–74.7% for CoT runs), exceeds the observed run-to-run algorithmic variance (up to 22.9pp on Hard), providing evidence that these ablation effects reflect some genuine differences rather than sampling noise alone.

Dependence on trace quality and coding agent capability. Solver quality depends on the quality of the reasoning traces it is induced from: the traces supply the search strategy, so traces without chain-of-thought (final programs only) yield much weaker solvers, while reducing the number of CoT examples from 100 to 48 has little effect. Solver quality is also sensitive to the capability of the coding agent used for induction: our ablations show that removing chain-of-thought supervision dramatically degrades performance (e.g., Hard accuracy drops from 74.7% to 24.8% for the no-CoT QO Solver), and that 12 demonstration examples are insufficient for reliable induction. Weaker coding agents or smaller trace budgets may not successfully induce useful solvers, making the approach contingent on access to sufficiently capable code-generation models.

Single construction pass. Each solver is induced in a single offline pass and is not updated with feedback from the tasks it fails on at test time. An iterative variant (re-inducing from LLM-verified solutions on the solver's failures) may improve hard-tail coverage but is not evaluated here.

Complexity bias. Induced solvers can produce programs of higher complexity than ground truth (positive Δ throughout on PBEBench), particularly on PBEBench-Hard.

While accuracy is unaffected, over-complex programs are less interpretable and may generalize poorly to unseen distributions. The compression analysis in the forward reconstruction case study (Appendix F.4) reveals a large gap between unconstrained accuracy (≈70–80%) and compact-rule performance (≈30%), suggesting that the verifier-based reward function favours solutions that fit the given examples rather than those that capture generalizable structure.

Token savings depend on solver coverage. The efficiency gains from hybrid inference are proportional to the fraction of tasks the symbolic solver solves perfectly. Solvers with lower coverage provide smaller token reductions. On PBEBench-Lite, where the CC Solver covers 80.4% of tasks perfectly, token savings are substantial (∼33% for BoK + CC); on harder tasks or weaker solvers, savings are smaller. Practitioners should therefore evaluate solver coverage on their target distribution before expecting comparable efficiency gains.

Single-model LLM baselines. All BoK and DF baselines use a single model (gpt-oss-120b). Results may not generalize to other reasoning models, and the relative advantage of symbolic solvers may vary with model capability. In particular, as the frontier model's performance on these benchmarks approaches saturation, the accuracy benefit of symbolic solvers diminishes, and efficiency becomes the primary motivation.

C Computational Environment

This section documents the hardware, software, and per-experiment compute budgets required to reproduce all results in the paper.

C.1 Hardware

Table 4 summarises the hardware used for each experimental role.

Table 4: Hardware configuration by experimental role.

Role	Hardware	Notes
Open-source LLM inference (vLLM)	2 × NVIDIA A100 80 GB	Internal cluster; serves `gpt-oss-120b` and `Qwen3.6-35B-A3B`
Claude Code solver induction	Anthropic API (`claude-sonnet-4-6`)	Interactive CLI session; no local GPU required
SLR-Bench verifier	CPU	SWI-Prolog subprocess (∼300 ms per rule)
Eval / analysis scripts	CPU	Python (stdlib, Pandas, Matplotlib)

All self-hosted inference was executed on an internal university cluster. **Reported dollar costs are reference estimates computed using public API pricing** (DeepInfra for `gpt-oss-120b`; AtlasCloud for `Qwen3.6-35B-A3B`) **for comparison purposes only**; actual experiments were run on the cluster at no direct monetary cost.

C.2 Software

Table 5 lists the software versions used across all experiments.

Table 5: Software versions.

Component	Version
Python	3.11
vLLM	0.8.x
openhands-sdk	1.16.1
Apptainer image	`python:3.11-slim` + numpy / scipy / sympy / evaluate / datasets
SWI-Prolog	9.x
Anthropic Python SDK	Latest (`claude-sonnet-4-6`)

C.3 Per-Experiment Compute

Reference pricing used for cost estimates:

- **gpt-oss-120b**: \$0.039 / M input tokens, \$0.190 / M output tokens (reasoning billed as output), DeepInfra public pricing
- **Qwen3.6-35B-A3B**: \$0.1612 / M input tokens, \$0.9653 / M output tokens, Atlas-Cloud public pricing

Token counts are exact, taken from provider usage fields logged in every output JSONL. "M tok" denotes millions of tokens (input + output combined).

PBEBench-Lite (1008 tasks, `max_programs` = 5) Token and cost breakdown is shown in Table 6.

Table 6: Per-experiment token and reference cost breakdown for PBEBench-Lite.

Experiment	Model	Tasks	M tok	Ref. cost	Notes
DF-32	gpt-oss-120b	1008	111.1	\$16.74	32 attempts max, single-turn per attempt
BoK-32	gpt-oss-120b	1008	68.0	\$12.20	32 independent samples
QO Agent / DirectSolve	Qwen3.6-35B-A3B	1008	395.3	\$85.38	Complete; avg 11.1 steps/task (max 100)
CC Solver (build)	claude-sonnet-4-6 (CC CLI)	1 session	—	\$2.00	One-time; ∼30 min, exact cost from tracked run
QO Solver run 1 (build)	Qwen3.6-35B-A3B	1 session	4.46	\$0.79	76 turns, 201 min
QO Solver run 2 (build)	Qwen3.6-35B-A3B	1 session	4.82	\$0.85	72 turns, 34 min
QO Solver run 3 (build)	Qwen3.6-35B-A3B	1 session	6.24	\$1.10	80 turns, 44 min
QO Solver (100ex, no CoT, build)	Qwen3.6-35B-A3B	1 session	7.81	\$1.34	102 turns, 44 min (ablation)
QO Solver (48ex + CoT, build)	Qwen3.6-35B-A3B	1 session	4.18	\$0.74	82 turns, 26 min (ablation)
QO Solver (12ex + CoT, build)	Qwen3.6-35B-A3B	1 session	1.65	\$0.30	49 turns, 15 min (ablation)

Solver inference incurs zero per-task LLM cost (pure Python). Ensemble outputs are computed deterministically from individual outputs; no additional LLM calls are required.

PBEBench-Hard (1216 tasks, `max_programs` = 20) Token and cost breakdown is shown in Table 7.

Table 7: Per-experiment token and reference cost breakdown for PBEBench-Hard.

Experiment	Model	Tasks	M tok	Ref. cost	Notes
BoK-32	gpt-oss-120b	1216	332.1	\$57.83	Complete
QO Agent / DirectSolve	Qwen3.6-35B-A3B	1216	1046.8	\$224	Complete; avg 861K tok/task; avg 18.9 steps/task
CC Solver (inference only)	—	1216	0	\$2.00 build	Same solver as Lite; build ∼30 min
QO Solver run 2 (inference only)	—	1216	0	\$0.85 build	Same solver as Lite run 2

SLR-Bench (1000 tasks) Token and cost breakdown is shown in Table 8.

Table 8: Per-experiment token and reference cost breakdown for SLR-Bench.

Experiment	Model	Tasks	M tok	Ref. cost	Notes
DF-32	gpt-oss-120b	1000	224.2	\$17.43	Complete
BoK-32	gpt-oss-120b	1000	225.3	\$17.88	Complete
QO Agent / DirectSolve	Qwen3.6-35B-A3B	1000	1232.6	\$243.41	Complete; avg 1,233K tok/task; avg 14.8 steps/task
CC Solver (build)	claude-sonnet-4-6 (CC CLI)	1 session	—	\$4.01	One-time; ∼30 min, exact cost from tracked run
QO Solver run 1 (build)	Qwen3.6-35B-A3B	1 session	2.93	\$0.51	68 turns, 25 min
QO Solver run 2 (build)	Qwen3.6-35B-A3B	1 session	7.49	\$1.28	84 turns, 47 min

C.4 Total Compute

Table 9 summarises total reference costs across all reported experiments.

All costs are reference figures based on public API pricing. Actual compute was executed on an internal cluster (2×A100 80 GB) at no direct monetary cost.

C.5 Execution Time

Table 10 reports approximate wall-clock times for each experimental phase.

Table 9: Total reference compute costs across all reported experiments.

Category	**Ref. cost**
gpt-oss-120b inference (Lite DF + BoK, Hard BoK, SLR DF + BoK)	∼$122
Qwen3.6-35B-A3B, QO Agent / DirectSolve (Lite + Hard + SLR)	∼$552
Qwen3.6-35B-A3B, QO Solver induction (6 PBE + 2 SLR runs)	∼$7
claude-sonnet-4-6, CC Solver induction (1 PBE + 1 SLR session)	∼$6 ($2.00 PBE + $4.01 SLR)
Total (reported experiments)	**∼$687**

Table 10: Approximate wall-clock times by experimental phase.

Experiment	**Wall-clock time**
QO Solver build (one run)	15–201 min (median ∼40 min)
CC Solver build (one session)	∼30 min (PBE and SLR)
DF / BoK inference (∼1000 tasks, 8 workers)	∼2 days per benchmark
QO Agent / DirectSolve (1000 tasks, 8 workers)	24–72 hours (hard-tier tasks ∼3.5M tok/task)
Symbolic solver inference (all tasks)	<5 min (pure Python, no LLM calls)
SLR-Bench verifier	∼300 ms per rule; ∼5 min for 1000 tasks

C.6 Parallelism

DF, BoK, and QO Agent baselines are executed with 8 parallel workers via Python `ThreadPoolExecutor` (one task per thread). Solver induction (QO and CC) runs as a single sequential agent session per induction run; multiple runs were launched independently in parallel. Ensemble aggregation is single-threaded and deterministic, requiring no additional LLM calls.

D Method Details

D.1 Algorithms

We formalize both steps of Section 2 in Algorithms 1 and 2. w_t denotes the evolving code sandbox, a_t is a coding-agent action (e.g., edit, execution, or file write), and $\mathcal{A}_\psi$ is the agent policy over actions. ExtractSolver parses the workspace into an executable solver S_{ϕ_t} when a valid implementation (e.g., `SOLVER.py`) is present. q_t measures solver quality on the trace dataset via the verifier, effectively capturing the fit of the solver over the training trace dataset.

In practice, t^* corresponds to the agent's final committed solver state at termination; the agent self-evaluates on $\mathcal{T}$ inline during the run to guide refinement rather than through post-hoc checkpoint selection across all steps.

Algorithm 1 Offline Solver Induction via Coding Agent

Require: trace LLM π_θ, verifier R, tasks $\{\mathcal{D}_j\}_{j=1}^m$, coding agent $\mathcal{A}_\psi$, step budget N

$\tau_j \leftarrow \textsc{Trace}(\pi_\theta, R, \mathcal{D}_j)$ for $j = 1, \ldots, m$
$\mathcal{T} \leftarrow \{(\mathcal{D}_j, \tau_j)\}_{j=1}^m$
Initialize workspace w_0 (empty codebase)
for $t = 0, \ldots, N-1$ **do**
    $a_t \sim \mathcal{A}_\psi(\cdot \mid w_t, \mathcal{T}, R)$
    $w_{t+1} \leftarrow \textsc{Apply}(w_t, a_t)$
    $S_{\phi_t} \leftarrow \textsc{ExtractSolver}(w_{t+1})$
    if S_{ϕ_t} is valid **then**
        $q_t \leftarrow \sum_{j=1}^m R(S_{\phi_t}(\mathcal{D}_j); \mathcal{D}_j)$
    end if
    if $\textsc{Done}(a_t)$ **then**
        break
    end if
end for
$t^* \leftarrow \arg\max_t q_t$
return solver $S_\phi \leftarrow S_{\phi_{t^*}}$

Algorithm 2 Hybrid Test-Time Inference

Require: task $\mathcal{D}$, solver S_ϕ, verifier R, LLM policy π_θ, mode M

$\mathcal{C}_S \leftarrow S_\phi(\mathcal{D})$ ▷ may return multiple candidates
$p_S \leftarrow \arg\max_{p \in \mathcal{C}_S} R(p; \mathcal{D})$
if $R(p_S; \mathcal{D}) = 1$ **then**
    return p_S
end if
$\mathcal{C}_L \leftarrow \textsc{LLMSearch}(\pi_\theta, R, \mathcal{D}, M)$
return $\arg\max_{p \in \mathcal{C}_L} R(p; \mathcal{D})$

D.2 Solver Building Prompts

PBEBench Solver Prompt

You are an expert in symbolic program induction.

Write a single Python file implementing a solver for a given Programming by Example (PBE) task.

Task

Write a Python-based symbolic program synthesizer that infers a transformation program from a set of (input_string, output_string) pairs.

You will be shown examples of an LLM solving similar tasks in @DEMOS_PBEBENCH.json, including reasoning traces from both successful and unsuccessful attempts across easy and hard cases. Use these to understand the task structure and take inspiration from the problem-solving strategies, especially in cases where the LLM struggles.

Output:
- a Python solver file @SOLVER.py
- a markdown file @SOLVER_ALGORITHM.md explaining the algorithm
The solver should use the verifier defined in @rewards/pbebench.py to evaluate candidate programs. If no correct program is found, it should return the top-K highest scoring programs, where K is a parameter taken by the solver.
Requirements

* Output exactly one Python file and one markdown file.
* Use only the Python standard library.
* No external data, APIs, or dataset-specific assumptions.
* The solver must generalize across tasks in @DEMOS_PBEBENCH.json
Interface
Implement:
def solve_pbe(examples):
"""
examples: list of (input_string, output_string)
returns: dict with at least:
- "success": bool
- "program": structured representation of the inferred transformation
"""
Behavior
The solver should:
* infer a program consistent with the examples and compatible with the verifier
* use (not reimplement) the verifier to score candidate programs
* prefer simple, compositional rules with low description complexity
* follow the domain-specific language (DSL) defined in @DEMOS_PBEBENCH.json
* search over candidate transformations and select ones that match all examples
* if no fully consistent program is found, return the top-K highest scoring programs
* return structured programs or hypotheses that could be useful for downstream refinement if partially incorrect

SLR-Bench Solver Prompt (1/2)

You are an expert in symbolic program induction.

Write a single Python file implementing a solver for a given SLR-Bench task.

Task

Write a Python-based symbolic program synthesizer that infers a Prolog rule from a set of (background_facts, direction _label) pairs, where direction_label is either "eastbound" or "westbound".

You will be shown examples of an LLM solving similar tasks in @DEMOS_SLRBENCH.json, including reasoning traces from both successful and unsuccessful attempts across easy and hard cases. Use these to understand the task structure and take inspiration from the problem-solving strategies, especially in cases where the LLM struggles.

Output:
- a Python solver file @SOLVER_SLR.py
- a markdown file @SOLVER_SLR_ALGORITHM.md explaining the algorithm

The solver should use the verifier defined in @rewards/slr_bench.py to evaluate candidate rules. If no correct rule is found, it should return the top-K highest scoring rules, where K is a parameter taken by the solver.

Requirements

* Output exactly one Python file and one markdown file.
* Use only the Python standard library.
* No external data, APIs, or dataset-specific assumptions.
* The solver must generalize across tasks in @DEMOS_SLRBENCH.json

Interface

Implement:

```
def solve_slr(examples):
"""
examples: list of (facts_string, direction_label)
facts_string — space-separated Prolog ground facts for one train
direction_label — "eastbound" or "westbound"
returns: dict with at least:
- "success": bool
- "program": Prolog rule string of the form "eastbound(T) :- Body."
"""
```

SLR-Bench Solver Prompt (2/2)

Domain-Specific Language

The rule must be a Prolog clause of the form ‘eastbound(T) :- Body.‘ where Body is a conjunction of literals drawn from these predicates:

- ‘has_car(Train, Car)‘ — Car is part of Train
- ‘car_num(Car, CarNumber)‘ — position of Car (positive integer)
- ‘car_color(Car, Color)‘ — Color ∈ red, blue, green, yellow, white
- ‘car_len(Car, Length)‘ — Length ∈ short, long
- ‘has_wall(Car, WallType)‘ — WallType ∈ full, railing

Prefer rules with the fewest body literals (use ‘rule_complexity()‘ from ‘rewards/slr_bench.py‘ to measure this).

Performance

Each call to the verifier (‘judge.compute‘) invokes SWI-Prolog as a subprocess and costs 300ms. Hard tasks can have hundreds of thousands of candidate rules. Minimize the number of verifier calls — use the examples to prune the candidate space in Python before invoking the verifier.

Behavior

The solver should:
* infer a rule consistent with the examples and compatible with the verifier
* use (not reimplement) the verifier to score candidate rules
* prefer simple rules with the fewest body literals
* follow the domain-specific language defined above and in @DEMOS_SLRBENCH.json
* search over candidate rules and select ones that correctly classify all examples
* if no fully consistent rule is found, return the top-K highest scoring rules
* return structured rules or hypotheses that could be useful for downstream refinement if partially incorrect

E More Experimental Details

E.1 Token accounting.

Token costs for `gpt-oss-120b` are computed using open-router pricing ($0.039/M input, $0.19/M output; reasoning tokens billed as output). Solver construction costs for QO are computed from OpenHands trajectories using native tokenization for `Qwen3.6-35B-A3B`. For `Qwen3.6-35B-A3B` as well, we use OpenRouter pricing for the AtlasCloud provider (OpenRouter lists multiple providers for this model) ($0.1612/M input, $0.9653/M output; reasoning tokens billed as output). Symbolic solvers incur zero per-task inference cost; construction costs are small ($\sim$ $0.3–$1.3 per QO solver; $2.00 and $4.01 for the CC solvers on PBEBench and SLR-Bench) and amortize after a handful of tasks. For hybrid methods, reported token usage includes only fallback LLM calls, reflecting effective cost under reuse.

F More Results

F.1 Additional Details: PBEBench-Lite

Search space. Each PBEBench program is a `replace(A,B)` operation where the predicate A has length 1–3 and the replacement B has length 0–3. For an alphabet of size V, this gives

$$(V + V^2 + V^3)(1 + V + V^2 + V^3)$$

possible programs. A cascade of length k therefore induces a search space exponential in k.

Table 11: PBEBench search-space sizes. Each row reports the number of possible `replace(A,B)` programs and the resulting worst-case cascade search space.

Setting	V	Programs	Max cascade	Search space
Lite (per-task avg.)	13	5.66×10^{6}	5	$\approx 5.8\times10^{33}$
Lite (global vocab.)	17	2.72×10^{7}	5	$\approx 1.5\times10^{37}$
Hard (global vocab.)	52	2.06×10^{10}	20	$\approx 1.8\times10^{206}$

Even the Lite search space is far beyond exhaustive enumeration: at 10^9 program evaluations per second and $\approx 10^7$ seconds per year, the per-task-average Lite setting would require roughly 10^{17} years; the Hard setting is cosmologically larger ($\approx 10^{190}$ years). The induced symbolic solvers avoid this enumeration by extracting candidate predicates directly from input-output differences, pruning replacements to a small candidate set, and applying greedy or beam search over cascades. Thus, they evaluate hundreds to thousands of candidates per task rather than enumerating the full combinatorial space.

Breakdown by cascade length. For PBEBench-Lite even for the symbolic solvers the performance degrades sharply at cascade length of 5, indicating difficulty in handling ordering interactions under tight program budgets. We also show comparisons of the accuracy (full correctness), mean reward obtained (partial correctness) and complexity of the generated programs for the CC and QO solvers, compared to the scaling methods (DF and BoK) and the QO direct coding agent baseline in Figures 5, 6 and 7. The figures show that due to the simpler nature of the PBEBench-Lite dataset it is nearly saturated by the scaling strategies and they tend to dominate even for harder cases. However the symbolic solvers still lead to token efficiency and the performance gap is smaller when considering partial correctness via the mean reward. Additionally directly solving the problems with a coding agent performs worse compared to simple LLM scaling but is better than the solvers. For complexity the solvers tend to induce more complex programs than LLM scaling but the coding agent due to its multi-turn refinement strategy ends up achieving the lowest complexity/simplest programs.

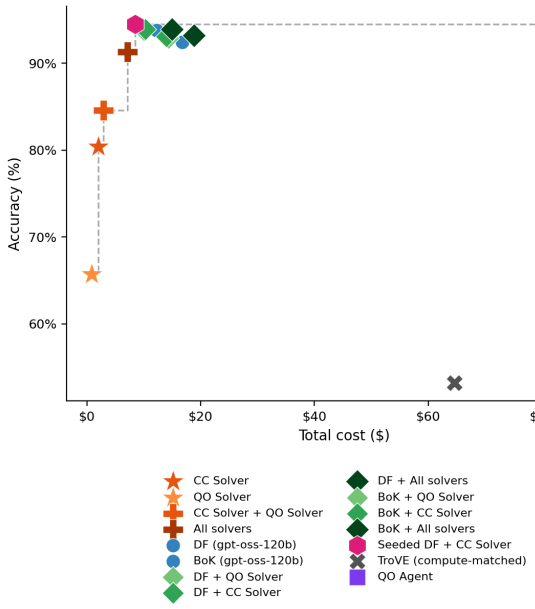


Figure 4: **PBEBench-Lite cost–accuracy Pareto frontier** (total cost = inference + one-time construction). Frontier: QO → CC → CC+QO → All solvers → Seeded DF+CC. Symbolic solvers dominate the low-cost regime; hybrids cluster at 93–95% for $8–19.

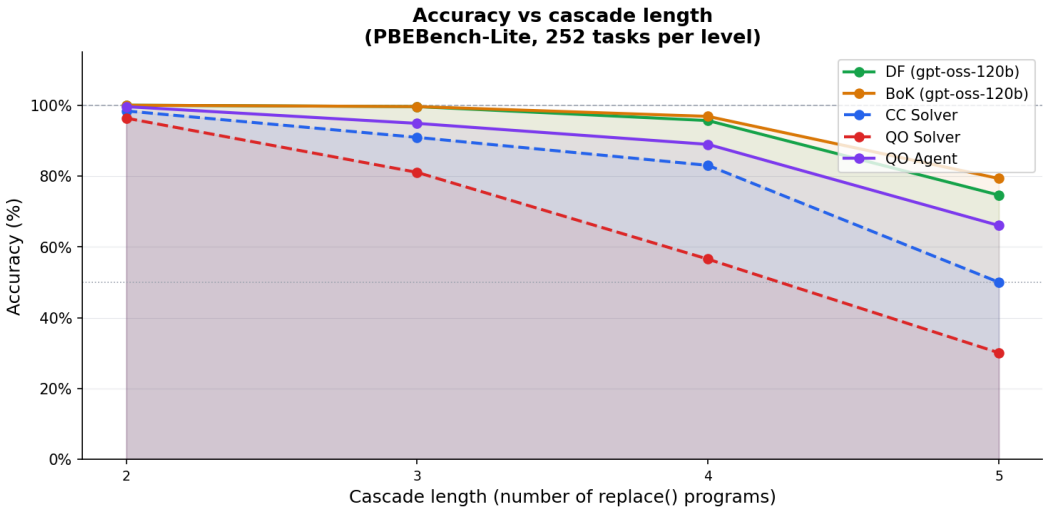


Figure 5: **PBEBench-Lite Accuracy vs Cascade Length:** LLM scaling strategies dominate, even on harder instances, even though the overall data is relatively simpler compared to PBEBench-Hard.

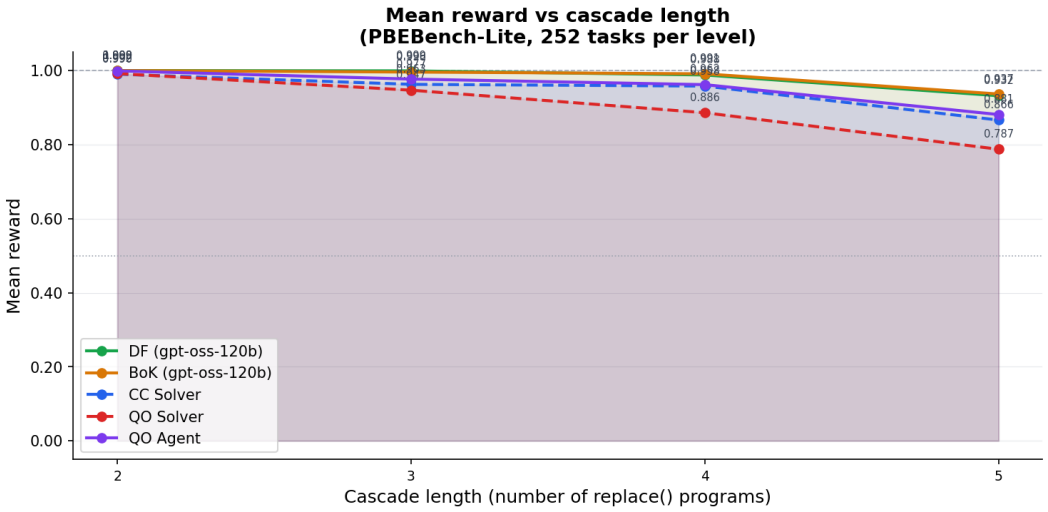


Figure 6: **PBEBench-Lite mean reward vs. cascade length.** The partial-credit gap between solvers and LLM scaling methods is smaller than the full-accuracy gap, confirming that solver failures are predominantly near-misses.

F.2 Solver Construction Ablations and Analysis

Setup. We analyze how the composition of the demos file (number of examples and presence of reasoning traces) affects solver quality on PBEBench. We induce six solvers via the Qwen3.6-35B-A3B + OpenHands agent, varying either the demos file or nothing at all while keeping all other settings fixed (same prompt, verifier, and seed-42 balanced sampling across success/failure × easy/hard quadrants). Token costs are measured using the native Qwen tokenizer.

Performance. Table 13 denotes the performance of the induced solver for various trace dataset and solver inductions settings.

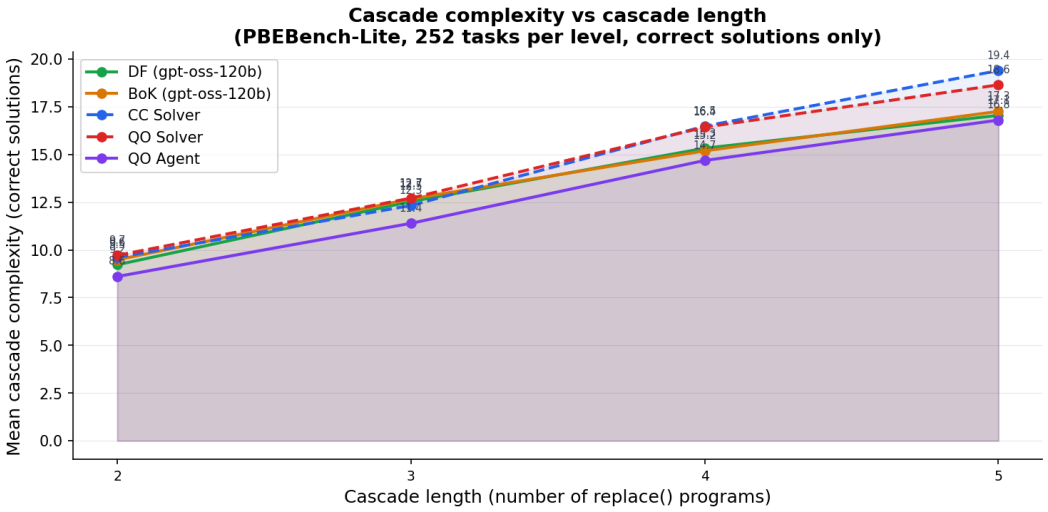


Figure 7: **PBEBench-Lite cascade complexity vs. cascade length.** Symbolic solvers overshoot ground-truth complexity ($\Delta \approx +3$); the direct coding agent (QO Agent) achieves the lowest complexity through multi-turn refinement.

Table 12: CC and QO solver accuracy and mean reward by cascade length on PBEBench-Lite.

CL	N	Acc% (CC)	Reward (CC)	Acc% (QO)	Reward (QO)
2	246	98.4	0.990	96.3	0.992
3	253	90.9	0.963	81.0	0.947
4	253	83.0	0.958	56.5	0.886
5	256	50.0	0.866	30.1	0.788

Solver construction token cost. The token costs for each solver estimated using the OpenHands trajectories with the native tokenizer and OpenRouter pricing for the Atlas-Cloud provider (§E.1) are shown in Table 14 for each PBEBench specific solver. The cost tends to be in a small range of \$0.3-1.34 showing the affordability of symbolic solver induction. These are QO solvers; the CC solvers cost \$2.00 (PBEBench) and \$4.01 (SLR-Bench).

Qualitative trajectory analysis. Each OpenHands run produces a different solver algorithm despite identical inputs, indicating exploration over algorithmic space.

100 examples + CoT, run 1 (Greedy + residual fixing). Extracts edit regions and generates candidates via substitution, split edits, and context extension. Greedy selection maximizes fixes while minimizing regressions, followed by residual repair passes. Captures multi-step edits through split candidates but often produces overly complex cascades.

100 examples + CoT, run 3 (Unique-op permutations). Separates forced vs optional operations and enumerates permutations over forced ones, followed by greedy and combinatorial search over optional candidates. Explicitly searches 2-step sequences for hard pairs. Highly effective on Lite but degrades on Hard due to fewer forced constraints.

100 examples + CoT, run 2 (Safety-first greedy + lookahead). Applies a hard constraint to avoid modifying unchanged examples, combined with 2-step lookahead for interaction effects. Produces a tighter search space and strong overall performance.

100 examples, no CoT (Beam search). Defaults to a generic beam search with heuristic scoring. Lacks structural insights (e.g., edit regions or ordering), leading to poor scaling on longer cascades.

Table 13: Solver performance across trace dataset configurations. The Lite Acc and Hard Acc columns denote the accuracy on PBEBench-Lite and PBEBench-Hard while the Lite ES and Hard ES columns denote the edit similarity scores, with Algorithm summarizes the approach of the solver. Demo accuracy is evaluated on the PBEBench trace dataset instances the solver was fitted on.

Demos	Run	Lite Acc.	Lite Edit Sim.	Hard Acc.	Hard Edit Sim.	Demo Acc.	Demo Reward	Algorithm
100 examples + CoT	1	53.4	78.6	58.9	94.5	71.4	0.9875	greedy + multi-pass residual fixing
100 examples + CoT	2	65.7	87.4	74.7	96.8	87.9	0.9930	safety-first greedy + 2-step lookahead
100 examples + CoT	3	**79.2**	**96.7**	51.8	90.0	75.8	0.8774	unique-op permutations + greedy + 2-op sequences
100 examples, no CoT	1	42.1	67.6	24.8	82.9	48.4	0.9633	beam search + heuristic scoring
48 examples + CoT	1	55.7	78.6	**76.2**	96.5	93.5	0.9930	multi-start greedy + permutation reorder
12 examples + CoT	1	47.7	80.2	50.4	94.3	58.3	0.9917	adaptive beam search

Table 14: Solver construction token costs at AtlasCloud pricing for Qwen3.6-35B-A3B. Each turn accumulates the full context as input; retry bloat is excluded.

Demos	Run	Turns	Input tokens	Output tokens	Total tokens	Cost ($)
100 examples + CoT	1	76	4,379,412	82,982	4,462,394	0.79
100 examples + CoT	2	72	4,725,111	93,839	4,818,950	0.85
100 examples + CoT	3	80	6,122,331	114,529	6,236,860	1.10
100 examples, no CoT	1	102	7,721,064	93,617	7,814,681	1.34
48 examples + CoT	1	82	4,103,766	76,919	4,180,685	0.74
12 examples + CoT	1	49	1,607,936	43,796	1,651,732	0.30
SLR, run 1	68	–	2,888,522	42,666	2,931,188	0.51
SLR, run 2	84	–	7,404,083	90,826	7,494,909	1.28

48 examples + CoT (Multi-start + reorder). Separates construction and ordering, using permutation search to explicitly handle interaction effects. Achieves the best Hard performance among Qwen runs.

12 examples + CoT (Adaptive beam). Uses adaptive beam search with diversity constraints but lacks sufficient signal to discover structural insights, resulting in weaker performance.

Claude Code PBEBench solver. The Claude Code solver uses a two-phase safe/unrestricted beam search with candidate extraction from `difflib.SequenceMatcher`. Phase 1 enforces a safety constraint by restricting candidates to patterns that do not occur in already-correct inputs, while Phase 2 relaxes this constraint for rare cases where a correct replacement must touch such a pattern. Candidates are generated dynamically from intermediate states, so later beam depths can discover feeding and bleeding interactions induced by earlier programs. Context-extended patterns improve specificity and reduce unintended replacements. It explicitly models intermediate states, enabling discovery of ordering interactions. This yields the strongest single-solver performance (80.4% Lite, 69.7% Hard).

Qwen SLR run 1: layered hypothesis generation with early exit. This solver parses predicates and argument domains, classifies predicates as car-level or train-level, and generates candidate body literals accordingly. It searches in ascending complexity layers, starting with one literal, then two, then three. Once a perfect rule is found within a layer, it exits early, thereby preferring the simplest correct rule. Candidates are purely conjunctive and are ranked by score and complexity.

Qwen SLR run 2: in-Python filter + budget-limited verification. This solver moves most candidate filtering into Python to avoid expensive SWI-Prolog calls. It generates direct separating properties, arithmetic predicates, negation-as-failure rules, and universal negation rules, using both positive and negative examples. If no single-property rule passes, it escalates to two- and three-property conjunctions, but only sends a small top-K set to the verifier. Despite the more elaborate design, performance is weaker: the verifier budget is too tight for medium and hard tasks, and the Python approximation may reject rules the official verifier would accept.

Claude Code SLR solver. The Claude Code SLR solver performs ascending-complexity search with a local Python evaluator. It parses facts into a normalized train-agnostic car model, re-indexing cars by `car_num`, and dynamically discovers predicates. It then enumerates candidate rules by increasing complexity, including single-car, two-property, and two-car rule templates. The local evaluator emulates Prolog existential semantics and optionally re-scores top candidates with the official SWI-Prolog verifier. Its key strengths are train-id invariance, predicate discovery, and complexity-controlled search. It achieves 68.4% overall accuracy, with 100% on Basic, 78.4% on Easy, and approximately 48% on Medium/Hard; its high mean score indicates that failures are often near-misses rather than complete failures.

Summary pattern. CoT traces enable the agent to learn structural insights about the DSL (edit regions, feeding interactions, program ordering). Without CoT it defaults to brute-force beam search. With CoT but too few examples (12), it produces a competent but under-informed search. The best solvers share a common trait: an explicit mechanism for handling program-order interactions (permutation reorder, 2-step lookahead, or split-candidate pairs).

Key findings. First, CoT reasoning traces are critical for solver induction. Removing CoT from the 100-example demos drops performance from 53.4% to 42.1% on Lite and from 58.9% to 24.8% on Hard. Second, run-to-run variance is large: the three 100-example CoT Qwen runs span 53.4%–79.2% on Lite and 51.8%–74.7% on Hard, with each run inventing a qualitatively different algorithm. Third, 12 examples with CoT is insufficient, suggesting a lower bound on the amount of demonstration diversity needed. Fourth, the no-CoT solver takes more turns than the corresponding CoT run, suggesting that reasoning traces reduce the need for unguided probing. Finally, solver construction is cheap relative to inference, with single-run induction costs between $0.30 and $1.34.

Table 15: Symbolic solver ensembles.

Ensemble	Lite Acc	Hard Acc	Tokens
CC only	80.4	69.7	0
Best Qwen	79.2	51.8	0
3 Qwen (CoT)	85.5	78.9	0
All Qwen	86.2	82.0	0
All Symbolic	**91.3**	**84.7**	0

Solver ensembles. Ensembling recovers variance across runs and yields the strongest overall performance, achieving 91.3% on Lite and 84.7% on Hard without any LLM inference cost.

F.3 Additional Details: PBEBench-Hard

Search space. PBEBench-Hard uses the global alphabet ($V = 52$: uppercase letters, lowercase letters, and digits) with cascades of length 2–20. The per-program search space grows to 2.06×10^{10} candidates; a length-20 cascade therefore has $\approx 1.8 \times 10^{206}$ possible program sequences. At 10^9 evaluations per second this would require $\approx 10^{190}$ years, far beyond any enumeration budget. The induced symbolic solvers reduce this by extracting candidate predicates directly from input–output diffs, restricting replacement candidates to a small typed set, and applying beam or greedy search so that only hundreds to thousands of candidates are evaluated per task.

Breakdown by cascade length. Table 16 reports per-cascade-length accuracy and mean reward for both the CC and QO solvers. Both solvers are strong at short cascades (CL 2–8, $\geq$84%) and degrade gradually through CL 9–17, then collapse at CL 18–20 where even finding a partial near-miss requires a very long search horizon. Despite near-zero accuracy at CL 20, mean reward remains above 0.92 at every level, confirming that failures are

Table 16: CC and QO solver accuracy and mean reward by cascade length on PBEBench-Hard (64 tasks per level).

CL	N	Acc% (CC)	Reward (CC)	Acc% (QO)	Reward (QO)
2	64	93.8	0.999	100.0	1.000
3	64	96.9	0.999	95.3	0.999
4	64	92.2	0.998	95.3	0.998
5	64	93.8	0.999	96.9	0.998
6	64	84.4	0.997	89.1	0.995
7	64	84.4	0.996	92.2	0.994
8	64	92.2	0.998	89.1	0.995
9	64	79.7	0.994	92.2	0.998
10	64	81.2	0.991	87.5	0.990
11	64	82.8	0.996	87.5	0.996
12	64	71.9	0.993	85.9	0.990
13	64	73.4	0.991	76.6	0.984
14	64	59.4	0.983	67.2	0.980
15	64	65.6	0.988	70.3	0.983
16	64	67.2	0.986	68.8	0.983
17	64	54.7	0.984	71.9	0.978
18	64	35.9	0.972	32.8	0.963
19	64	12.5	0.958	12.5	0.943
20	64	1.6	0.935	7.8	0.925

dominated by near-misses rather than complete failures. Figures 8, 9, and 10 show the full accuracy, mean reward, and complexity profiles across cascade lengths, comparing CC and QO solvers with BoK and the QO direct coding-agent baseline. In contrast to PBEBench-Lite, LLM test-time scaling (BoK) collapses at long cascades while the symbolic solvers maintain substantially higher accuracy through CL 17, illustrating the complementarity that makes hybrid inference most effective on Hard.

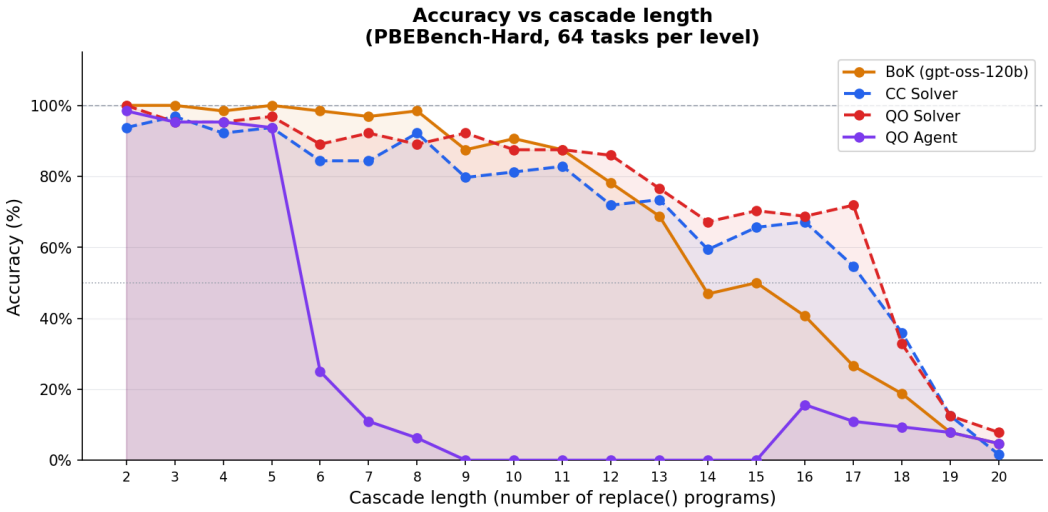


Figure 8: **PBEBench-Hard accuracy vs. cascade length.** Symbolic solvers (CC, QO) maintain substantially higher accuracy than BoK at long cascade lengths (CL 14+), where independent LLM sampling collapses.

Breakdown by BFCC interaction type. Table 17 reports CC solver accuracy across the 16 BFCC (bleeding, feeding, counterbleeding, counterfeeding) interaction categories. Tasks with no BFCC interactions are easiest (92.7%); adding feeding interactions consis-

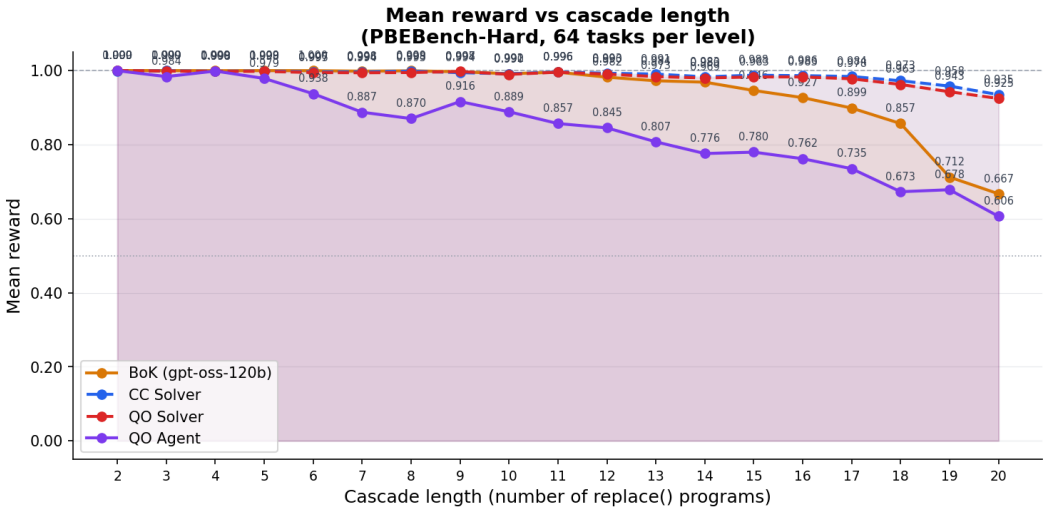


Figure 9: **PBEBench-Hard mean reward vs. cascade length.** Even at CL 20 where accuracy approaches zero, mean reward remains above 0.92 for both solvers, indicating near-misses rather than complete failures.

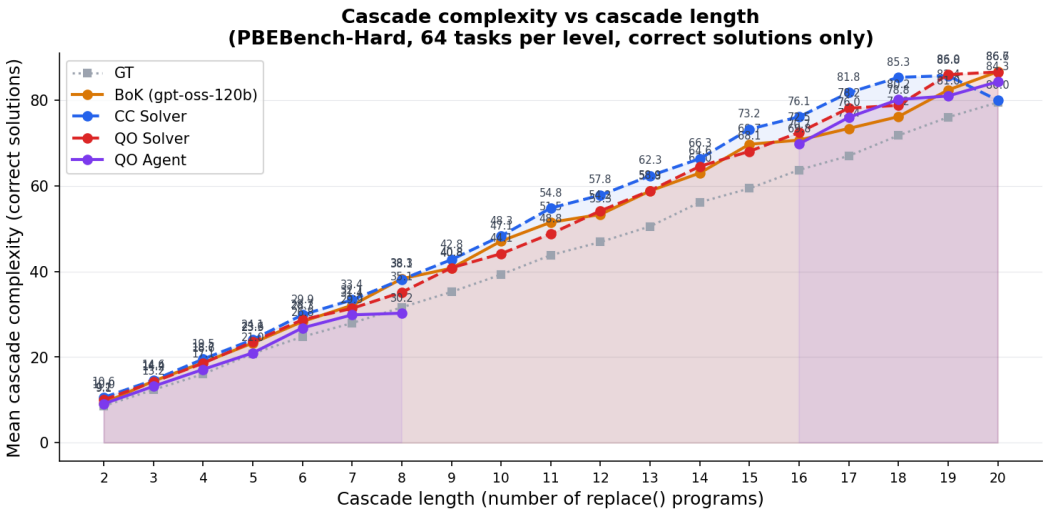


Figure 10: **PBEBench-Hard cascade complexity vs. cascade length.** Symbolic solvers overshoot ground-truth complexity substantially ($\Delta \approx +5$–$+8$) due to valid-but-verbose inductive solutions at long cascades; BoK complexity tracks GT more closely.

tently reduces accuracy more than bleeding alone. The hardest category is all-four-BFCC (40.2%), which constitutes 22% of the dataset; dense mutual ordering interactions are the primary bottleneck for beam-search-based solvers.

Solver failure analysis. The dominant failure mode on both PBEBench-Lite and PBEBench-Hard is *near-misses*: the solver finds a cascade that correctly transforms all but 1–3 input-output pairs, reflected in high mean rewards (0.944 on Lite, 0.987 on Hard) despite imperfect accuracy. Three systematic causes account for most failures: (1) tasks where the correct program can only be discovered through a long intermediate chain (cascade length ≥ 18), where the beam horizon is exhausted before the necessary ordering is found; (2) all-four-BFCC tasks that require holding multiple mutually interacting ordering constraints simultaneously within the beam; (3) tasks where the pattern that fixes changed

Table 17: CC solver accuracy by BFCC interaction type on PBEBench-Hard.

BFCC relationships	N	Acc%
None	55	92.7
Bleeding	47	91.5
Bleeding, Counterfeeding	56	91.1
Counterfeeding	64	90.6
Counterbleeding	45	84.4
Counterfeeding, Counterbleeding	60	81.7
Bleeding, Counterbleeding	65	81.5
Feeding	64	75.0
Feeding, Bleeding	59	78.0
Feeding, Bleeding, Counterfeeding	72	73.6
Feeding, Counterfeeding, Counterbleeding	74	73.0
Bleeding, Counterfeeding, Counterbleeding	77	71.4
Feeding, Counterfeeding	76	65.8
Feeding, Counterbleeding	58	67.2
Feeding, Bleeding, Counterbleeding	78	66.7
Feeding, Bleeding, Counterfeeding, Counterbleeding	266	40.2

Table 18: Inference-time determinism of CC and QO solvers on PBEBench-Hard (two independent runs, identical settings).

System	Run 1 Acc%	Run 2 Acc%	Agreement	Flips
CC Solver	69.65	69.74	1211/1216 (99.6%)	5
QO Solver	58.88	57.89	1180/1216 (97.0%)	36

pairs also appears as a substring in unchanged pairs, the safe-phase beam discards it and the unrestricted phase does not recover in time.

Solver determinism. Both solvers were re-run on PBEBench-Hard under identical conditions to measure inference-time determinism. Table 18 reports the results. Note that the QO Solver row evaluates Run 1 (58.9% Hard accuracy); the primary QO Solver result reported in the main paper uses Run 2 (74.7%), which was selected based on best performance across induction runs. Both runs are equivalent for measuring inference-time determinism, as we are measuring whether a fixed solver produces consistent outputs across independent evaluations. All discrepancies between runs occur at the partial-credit boundary ($0.96 \leftrightarrow 1.0$ or $0.98 \leftrightarrow 1.0$); no task flips between clearly solved and clearly failed. Both solvers are effectively deterministic: the CC solver agrees on 99.6% of tasks across runs and the QO solver on 97.0%. Run-to-run variance in the paper therefore refers to independent *induction* attempts (different solver algorithms), not to inference variance of a fixed solver.

F.4 Additional Details: Real-World Forward Reconstruction

We evaluate induced symbolic solvers on a real-world forward reconstruction dataset consisting of 3,077 language-pair tasks from Austronesian comparative linguistics[1]. Each task requires inducing an ordered cascade of sound change rules from sparse input-output examples (median 4 examples). Compared to PBEBench, this setting introduces (i) an unseen IPA alphabet, (ii) variable and unknown cascade lengths, and (iii) no ground-truth programs.

Both solvers achieve strong zero-shot performance ($\sim$70%) without retraining. Increasing cascade length yields modest gains, after which performance saturates.

[1] https://zenodo.org/records/7741197

Table 19: Real-FR accuracy at different maximum cascade lengths (k). Rows marked – were only evaluated at $k = 100$. Run variance across three identical 100-example CoT induction runs (runs 1–3) is reported to illustrate algorithmic sensitivity.

System	$k = 20$	$k = 50$	$k = 100$
Individual solvers			
CC Solver	69.0	70.1	**70.2**
Qwen Solver (run 2, 100ex+CoT)	67.5	69.5	**70.5**
Qwen Solver (run 1, 100ex+CoT)	—	—	53.2
Qwen Solver (run 3, 100ex+CoT)	—	—	4.1†
Qwen Solver (48ex+CoT)	—	—	61.4
Qwen Solver (12ex+CoT)	—	—	29.4
Qwen Solver (no-CoT)	—	—	44.9
Ensemble (union, $k = 100$)			
CC + Qwen run 2	—	—	76.1
CC + Qwen runs 1+2	—	—	76.5
All Qwen (5 solvers)	—	—	78.9
All solvers (excl. run 3)	—	—	**80.1**

† Run 3 uses a "unique-op forcing" strategy that fails under real IPA distributions.

Table 20: Compression sweep for QO solver (run 2) on Real-FR. `max_programs` = $\max(2, \lceil n_{\text{examples}}/\text{ratio} \rceil)$.

Setting	Acc%	Avg programs
Unconstrained ($k = 100$)	69.9	26.35
ratio = 1.0	43.8	10.47
ratio = 2.0	31.9	6.49
ratio = 3.0	30.5	4.82
ratio = 5.0	**29.6**	**3.62**

Effect of cascade length. Increasing the maximum number of programs (cascade length) from 20 to 100 yields modest gains (e.g., CC: 69.0%→70.2%, Qwen: 67.5%→70.5%), after which performance plateaus. This suggests that while real sound-change systems are deeper than PBEBench cascades, the primary bottleneck is candidate generation rather than search depth.

Solver variance and ensembling. Across independently induced solvers, performance ranges from 4.1% to 70.5%, reflecting substantial algorithmic variability. For example, three Qwen runs with identical training data achieve accuracies ranging from 4.1% to 70.5%. Some solver designs (e.g., unique-op forcing) fail under distribution shift, indicating sensitivity to assumptions about the training regime. Combining solvers via union significantly improves performance, reaching up to 80.1% accuracy, indicating that different solvers capture complementary strategies.

Compression analysis. To evaluate whether solvers discover compact rules or rely on memorization, we constrain cascade length relative to the number of examples. Under strong compression (programs limited to $\leq n_{\text{examples}}/5$), accuracy drops to ≈30%, revealing a subset of tasks that admit compact, shared rules. The remaining ≈40 percentage points (from 30% to 70%) correspond to cases where solvers construct longer cascades that effectively patch individual examples rather than discovering general rules. Accuracy plateaus at 30% under strong compression, indicating that only a subset of tasks admit compact, shared rules with current solvers. The remaining gains arise from longer cascades that effectively patch individual examples.

Sparse examples as the primary bottleneck. Real-FR tasks have a median of 4 examples per task (minimum 1, maximum 857), far sparser than PBEBench's guaranteed 5+ examples. With 1–2 examples, multiple distinct cascades are consistent with the evidence, and any short program that happens to work scores 1.0 by coincidence. Single-example tasks remain unsolvable in a principled sense regardless of compression ratio; the per-task accuracy gains from compression sweeps occur almost entirely on tasks with ≥3 examples.

Qualitative analysis. Under strong compression (ratio = 5, programs ≤ $\lceil n_{\text{ex}}/5 \rceil$), induced rules on multi-example tasks are often linguistically plausible sound laws, for example:

- `replace('#b', '#f')` (Proto-Austronesian→Amis): word-initial /b/→/f/ lenition, well-attested.
- `replace(' #', 'h#')` + `replace('wi', 'i')` (→Bahasa Indonesia): word-final glottal→/h/ and /wi/→/i/ glide deletion, both documented Indonesian changes.
- `replace('# ', '#h')` + `replace('aw#', 'au#')` (→Banjarese): word-initial / /→/h/ and diphthong shift, classic Banjarese.
- `replace(' ', '')` (→Bidayuh): glottal stop deletion, extremely common cross-linguistically.
- `replace('da', 'ra')` + `replace('u ', 'o ')` (→Atayal Squliq): /d/→/r/ and vowel lowering before glottal, both real Atayal changes.

In contrast, single-example tasks admit coincidental word-level substitutions regardless of compression ratio, and are noise by construction.

DSL expressivity vs. linguistic validity. The `replace(A,B)` DSL is strictly more expressive than real sound laws. A genuine sound law maps one phoneme (or phoneme class in an environment) to another: /b/→/f/, / /→∅, /u/→/o/ before glottal. The DSL permits arbitrary string substitutions with no phonological structure, e.g. `replace('m','ja')` would require a consonant to simultaneously become a vowel sequence, unknown in natural language. The reward function (training accuracy) cannot distinguish these cases; both score 1.0. This has two consequences: (i) false positives on sparse tasks (≤2 examples), where almost any short program achieves 1.0 by coincidence; and (ii) complexity inflation on rich tasks, where the unconstrained solver uses 26 programs on average, far more than any real sound-change history.

Reward function limitations and future directions. The current training-accuracy reward is appropriate for PBEBench (compact ground truth exists) but misaligned for Real-FR. Four directions would improve evaluation: (1) *held-out evaluation*, even a 3/1 example split would filter coincidental single-word patches; (2) *complexity penalty*, e.g. reward = pass_rate$-\lambda \cdot n_{\text{programs}}/n_{\text{changed_examples}}$; (3) *phonological validity filter*, reject substitutions whose pattern spans more than one IPA segment; (4) *minimum example threshold*, exclude or down-weight tasks with fewer than 3–5 examples, which are noise-dominated regardless of method.

F.5 Statistical Significance Testing

Methodology. We test all pairwise comparisons between systems on PBEBench-Lite, PBEBench-Hard, and SLR-Bench using two complementary tests. For binary per-task correctness (solved/unsolved), we use the **exact two-sided McNemar test** (McNemar, 1947), which operates on discordant pairs, tasks where one system succeeds and the other fails, and is the standard choice for paired binary outcomes. For continuous per-task mean reward, we use the **two-sided Wilcoxon signed-rank test**, a non-parametric paired test that makes no normality assumption and is appropriate for reward distributions that are heavy-tailed near 1.0. Both tests are conducted at $\alpha = 0.05$; we report exact p-values with $^{*}p<0.05$, $^{**}p<0.01$, $^{***}p<0.001$. No multiple-comparison correction is applied; these tests are exploratory and intended to distinguish genuine performance gaps from sampling noise.

The same tests are applied to SLR-Bench; results are in Table 25.

PBEBench-Lite results (Tables 21–23). All comparisons between qualitatively distinct system categories (individual solvers vs. LLM scaling vs. hybrids) are strongly significant ($p<0.001$ by both tests), confirming that the accuracy differences are not due to chance. Notable exceptions reveal the saturation regime: BoK vs. BoK+CC (ΔAcc = +0.10pp, $p = 1.0$) and BoK vs. BoK+All Sym (ΔAcc = +0.10pp, $p = 1.0$) are not significant by either test, consistent with the ceiling effect. The BoK vs. DF comparison (ΔAcc = −1.49pp) reaches conventional significance by McNemar ($p = 0.020$) but not by Wilcoxon ($p = 0.595$), indicating that while BoK has a slight edge in binary solve rate, the two systems are essentially equivalent on partial-credit reward. DF hybrids (DF+CC, DF+All Sym) show small but statistically significant improvements over DF alone ($p<0.01$), confirming that even marginal accuracy gains from solver augmentation are real on this dataset. The accuracy difference between All Symbolic and BoK-based hybrids is significant ($p<0.001$), but the corresponding Wilcoxon p-values are not ($p > 0.15$), showing that hybrids close the accuracy gap through binary correctness on a handful of tasks without substantially changing the reward distribution.

PBEBench-Hard results (Table 24). All comparisons are significant by both tests ($p<0.001$) with one exception: BoK vs. CC Solver (ΔAcc = +1.23pp) is not significant by McNemar ($p = 0.379$), confirming that these two systems are statistically tied on accuracy despite the CC Solver's substantially higher mean reward (ΔRew = +0.0445, $p<0.001$). This reflects the hard regime: BoK and CC Solver solve largely complementary subsets of tasks, so the overall pass rates are similar while per-task rewards diverge sharply. Every hybrid improvement over its constituent components is significant ($p<0.001$), including the small BoK+All Sym vs. BoK+CC+QO difference (+2.30pp), validating that ensemble gains across the full solver suite are genuine. Notably, BoK+CC vs. BoK+QO (ΔAcc = +1.23pp, $p = 0.124$) is not significant, consistent with the near-equivalent performance of the two individual solvers on this split.

Table 21: PBEBench-Lite pairwise significance tests (part 1/3): LLM baselines as System A. ΔAcc = B−A accuracy (pp); ΔRew = B−A mean reward. p(Mc): exact two-sided McNemar test. p(Wi): two-sided Wilcoxon signed-rank test. $^{*}p<0.05$, $^{**}p<0.01$, $^{***}p<0.001$.

System A	System B	ΔAcc	p(Mc)	ΔRew	p(Wi)
BoK	DF	−1.49	0.020^{*}	−0.0012	0.595
BoK	QO Agent	−6.65	$< 0.001^{***}$	−0.0264	$< 0.001^{***}$
BoK	CC Solver	−13.49	$< 0.001^{***}$	−0.0369	$< 0.001^{***}$
BoK	QO Solver	−28.17	$< 0.001^{***}$	−0.0786	$< 0.001^{***}$
BoK	All Symbolic	−2.58	0.001^{**}	−0.0036	0.303
BoK	BoK + QO	+0.00	1.000	+0.0000	1.000
BoK	BoK + CC	+0.10	1.000	+0.0002	0.317
BoK	BoK + All Sym	+0.10	1.000	+0.0002	0.317
BoK	DF + QO	−0.99	0.110	+0.0002	0.976
BoK	DF + CC	−0.79	0.200	+0.0008	0.729
BoK	DF + All Sym	−0.69	0.265	+0.0010	0.661
DF	QO Agent	−5.16	$< 0.001^{***}$	−0.0252	$< 0.001^{***}$
DF	CC Solver	−12.00	$< 0.001^{***}$	−0.0357	$< 0.001^{***}$
DF	QO Solver	−26.69	$< 0.001^{***}$	−0.0774	$< 0.001^{***}$
DF	All Symbolic	−1.09	0.235	−0.0024	0.580
DF	BoK + QO	+1.49	0.020^{*}	+0.0012	0.595
DF	BoK + CC	+1.59	0.011^{*}	+0.0014	0.540
DF	BoK + All Sym	+1.59	0.011^{*}	+0.0014	0.540
DF	DF + QO	+0.50	0.062	+0.0014	0.034^{*}
DF	DF + CC	+0.69	0.016^{*}	+0.0020	0.014^{*}
DF	DF + All Sym	+0.79	0.008^{**}	+0.0022	0.008^{**}

SLR-Bench results (Table 25). Nearly all comparisons are significant by both tests ($p<0.001$), with a few interpretable exceptions. BoK vs. CC Solver (ΔAcc = −0.30pp, $p = 0.896$) is not significant by McNemar, confirming these two systems are statistically tied in binary accuracy despite CC Solver having substantially higher mean reward (ΔRew

Table 22: PBEBench-Lite pairwise significance tests (part 2/3): coding-agent baseline and individual symbolic solvers as System A. Column headers as in part 1.

System A	**System B**	**ΔAcc**	p(Mc)	**ΔRew**	p(Wi)
QO Agent	CC Solver	−6.85	$< 0.001^{***}$	−0.0105	0.064
QO Agent	QO Solver	−21.53	$< 0.001^{***}$	−0.0522	$< 0.001^{***}$
QO Agent	All Symbolic	+4.07	$< 0.001^{***}$	+0.0228	$< 0.001^{***}$
QO Agent	BoK + QO	+6.65	$< 0.001^{***}$	+0.0264	$< 0.001^{***}$
QO Agent	BoK + CC	+6.75	$< 0.001^{***}$	+0.0266	$< 0.001^{***}$
QO Agent	BoK + All Sym	+6.75	$< 0.001^{***}$	+0.0266	$< 0.001^{***}$
QO Agent	DF + QO	+5.65	$< 0.001^{***}$	+0.0266	$< 0.001^{***}$
QO Agent	DF + CC	+5.85	$< 0.001^{***}$	+0.0272	$< 0.001^{***}$
QO Agent	DF + All Sym	+5.95	$< 0.001^{***}$	+0.0274	$< 0.001^{***}$
CC Solver	QO Solver	−14.68	$< 0.001^{***}$	−0.0417	$< 0.001^{***}$
CC Solver	All Symbolic	+10.91	$< 0.001^{***}$	+0.0333	$< 0.001^{***}$
CC Solver	BoK + QO	+13.49	$< 0.001^{***}$	+0.0369	$< 0.001^{***}$
CC Solver	BoK + CC	+13.59	$< 0.001^{***}$	+0.0371	$< 0.001^{***}$
CC Solver	BoK + All Sym	+13.59	$< 0.001^{***}$	+0.0371	$< 0.001^{***}$
CC Solver	DF + QO	+12.50	$< 0.001^{***}$	+0.0371	$< 0.001^{***}$
CC Solver	DF + CC	+12.70	$< 0.001^{***}$	+0.0377	$< 0.001^{***}$
CC Solver	DF + All Sym	+12.80	$< 0.001^{***}$	+0.0379	$< 0.001^{***}$
QO Solver	All Symbolic	+25.60	$< 0.001^{***}$	+0.0750	$< 0.001^{***}$
QO Solver	BoK + QO	+28.17	$< 0.001^{***}$	+0.0786	$< 0.001^{***}$
QO Solver	BoK + CC	+28.27	$< 0.001^{***}$	+0.0788	$< 0.001^{***}$
QO Solver	BoK + All Sym	+28.27	$< 0.001^{***}$	+0.0788	$< 0.001^{***}$
QO Solver	DF + QO	+27.18	$< 0.001^{***}$	+0.0788	$< 0.001^{***}$
QO Solver	DF + CC	+27.38	$< 0.001^{***}$	+0.0794	$< 0.001^{***}$
QO Solver	DF + All Sym	+27.48	$< 0.001^{***}$	+0.0796	$< 0.001^{***}$

Table 23: PBEBench-Lite pairwise significance tests (part 3/3): All Symbolic and hybrid-vs-hybrid comparisons. Column headers as in part 1.

System A	**System B**	**ΔAcc**	p(Mc)	**ΔRew**	p(Wi)
All Symbolic	BoK + QO	+2.58	0.001^{**}	+0.0036	0.303
All Symbolic	BoK + CC	+2.68	$< 0.001^{***}$	+0.0038	0.271
All Symbolic	BoK + All Sym	+2.68	$< 0.001^{***}$	+0.0038	0.271
All Symbolic	DF + QO	+1.59	0.064	+0.0038	0.289
All Symbolic	DF + CC	+1.79	0.033^{*}	+0.0044	0.179
All Symbolic	DF + All Sym	+1.88	0.023^{*}	+0.0046	0.155
BoK + QO	BoK + CC	+0.10	1.000	+0.0002	0.317
BoK + QO	BoK + All Sym	+0.10	1.000	+0.0002	0.317
BoK + QO	DF + QO	−0.99	0.110	+0.0002	0.976
BoK + QO	DF + CC	−0.79	0.200	+0.0008	0.729
BoK + QO	DF + All Sym	−0.69	0.265	+0.0010	0.661
BoK + CC	BoK + All Sym	+0.00	1.000	+0.0000	1.000
BoK + CC	DF + QO	−1.09	0.071	+0.0000	0.959
BoK + CC	DF + CC	−0.89	0.136	+0.0006	0.792
BoK + CC	DF + All Sym	−0.79	0.185	+0.0008	0.721
BoK + All Sym	DF + QO	−1.09	0.071	+0.0000	0.959
BoK + All Sym	DF + CC	−0.89	0.136	+0.0006	0.792
BoK + All Sym	DF + All Sym	−0.79	0.185	+0.0008	0.721
DF + QO	DF + CC	+0.20	0.625	+0.0006	0.257
DF + QO	DF + All Sym	+0.30	0.250	+0.0008	0.102
DF + CC	DF + All Sym	+0.10	1.000	+0.0002	0.317

= +0.027, p<0.001), consistent with the finding that CC Solver produces fewer complete failures but near-identical solve rates. DF vs. BoK+CC (ΔAcc = +0.70pp, p = 0.621)

Table 24: Pairwise significance tests on PBEBench-Hard. ΔAcc = B−A accuracy difference (pp). ΔRew = B−A mean reward difference. p(McNemar) uses the exact two-sided McNemar test on per-task binary correctness. p(Wilcoxon) uses the two-sided Wilcoxon signed-rank test on per-task rewards. $^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$.

System A	**System B**	**ΔAcc%**	p(McNemar)	**ΔRew**	p(Wilcoxon)
BoK	CC Solver	+1.23	0.379	+0.0445	$< 0.001^{***}$
BoK	QO Solver	+6.25	$< 0.001^{***}$	+0.0408	$< 0.001^{***}$
BoK	All Symbolic	+16.28	$< 0.001^{***}$	+0.0493	$< 0.001^{***}$
BoK	BoK + CC	+11.02	$< 0.001^{***}$	+0.0080	$< 0.001^{***}$
BoK	BoK + QO	+12.25	$< 0.001^{***}$	+0.0069	$< 0.001^{***}$
BoK	BoK + CC + QO	+15.05	$< 0.001^{***}$	+0.0103	$< 0.001^{***}$
BoK	BoK + All Sym	+17.35	$< 0.001^{***}$	+0.0142	$< 0.001^{***}$
CC Solver	QO Solver	+5.02	$< 0.001^{***}$	-0.0037	$< 0.001^{***}$
CC Solver	All Symbolic	+15.05	$< 0.001^{***}$	+0.0048	$< 0.001^{***}$
CC Solver	BoK + CC	+9.79	$< 0.001^{***}$	-0.0365	$< 0.001^{***}$
CC Solver	BoK + QO	+11.02	$< 0.001^{***}$	-0.0376	$< 0.001^{***}$
CC Solver	BoK + CC + QO	+13.82	$< 0.001^{***}$	-0.0342	$< 0.001^{***}$
CC Solver	BoK + All Sym	+16.12	$< 0.001^{***}$	-0.0303	$< 0.001^{***}$
QO Solver	All Symbolic	+10.03	$< 0.001^{***}$	+0.0085	$< 0.001^{***}$
QO Solver	BoK + CC	+4.77	$< 0.001^{***}$	-0.0328	$< 0.001^{***}$
QO Solver	BoK + QO	+6.00	$< 0.001^{***}$	-0.0339	$< 0.001^{***}$
QO Solver	BoK + CC + QO	+8.80	$< 0.001^{***}$	-0.0305	$< 0.001^{***}$
QO Solver	BoK + All Sym	+11.10	$< 0.001^{***}$	-0.0266	$< 0.001^{***}$
All Symbolic	BoK + CC	-5.26	$< 0.001^{***}$	-0.0412	$< 0.001^{***}$
All Symbolic	BoK + QO	-4.03	$< 0.001^{***}$	-0.0424	$< 0.001^{***}$
All Symbolic	BoK + CC + QO	-1.23	0.028^{*}	-0.0390	$< 0.001^{***}$
All Symbolic	BoK + All Sym	+1.07	$< 0.001^{***}$	-0.0350	$< 0.001^{***}$
BoK + CC	BoK + QO	+1.23	0.124	-0.0012	0.292
BoK + CC	BoK + CC + QO	+4.03	$< 0.001^{***}$	+0.0022	$< 0.001^{***}$
BoK + CC	BoK + All Sym	+6.33	$< 0.001^{***}$	+0.0062	$< 0.001^{***}$
BoK + QO	BoK + CC + QO	+2.80	$< 0.001^{***}$	+0.0034	$< 0.001^{***}$
BoK + QO	BoK + All Sym	+5.10	$< 0.001^{***}$	+0.0074	$< 0.001^{***}$
BoK + CC + QO	BoK + All Sym	+2.30	$< 0.001^{***}$	+0.0040	$< 0.001^{***}$

is also not significant, indicating that simply adding the CC Solver to BoK matches DF's accuracy without a statistically distinguishable difference. In contrast, every DF-based hybrid provides a strongly significant improvement over DF alone (p<0.001), confirming that symbolic solvers add genuine complementary coverage beyond what DF's sequential search achieves. The BoK+QO vs. DF+QO comparison (ΔAcc = +7.80pp, p<0.001) is significant by McNemar but not by Wilcoxon ($p = 0.359$), reflecting that DF+QO's accuracy gain comes from resolving tasks at the binary solve boundary without changing the reward distribution on partial-credit tasks.

F.6 Additional Details: SLR-Bench

Search space. SLR-Bench requires inducing a Prolog rule `eastbound(T) :- has_car(T,C), lit`$_1$`, …` with up to 4 body literals drawn from L ground literals per task. The candidate count is $\binom{L}{1} + \binom{L}{2} + \binom{L}{3} + \binom{L}{4}$, where L grows with curriculum level. Each candidate requires a SWI-Prolog subprocess call (50–200 ms), making brute force infeasible at levels 15–20. The induced solvers address this by (1) extracting only task-relevant literals, (2) searching layer-by-layer and stopping at the first layer with a perfect rule, and (3) pruning by syntactic features before evaluation.

Per-tier and per-level accuracy. Figure 11a shows pass rate by curriculum level for all systems. LLM methods (BoK, DF) are strong through levels 1–10 but degrade sharply from level 11 onward, with BoK collapsing to near-zero at levels 16–20. In contrast, the CC

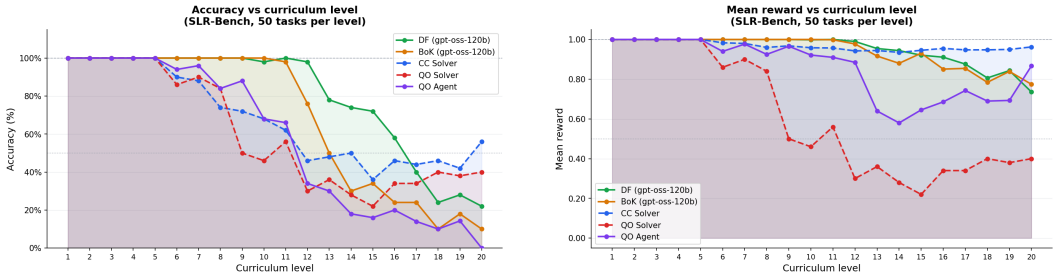
Table 25: Pairwise significance tests on SLR-Bench. ΔAcc = B−A accuracy difference (pp). ΔRew = B−A mean reward difference. p(McNemar) uses the exact two-sided McNemar test on per-task binary correctness. p(Wilcoxon) uses the two-sided Wilcoxon signed-rank test on per-task rewards. $^{*}p<0.05$, $^{**}p<0.01$, $^{***}p<0.001$.

System A	**System B**	**ΔAcc%**	p(McNemar)	**ΔRew**	p(Wilcoxon)
BoK	DF	+10.90	$< 0.001^{***}$	+0.0086	$< 0.001^{***}$
BoK	CC Solver	−0.30	0.896	+0.0265	$< 0.001^{***}$
BoK	QO Solver	−8.00	$< 0.001^{***}$	−0.3334	$< 0.001^{***}$
BoK	BoK + QO	+7.20	$< 0.001^{***}$	+0.0141	$< 0.001^{***}$
BoK	BoK + CC	+11.60	$< 0.001^{***}$	+0.0215	$< 0.001^{***}$
BoK	BoK + CC + QO	+11.60	$< 0.001^{***}$	+0.0215	$< 0.001^{***}$
BoK	DF + QO	+15.00	$< 0.001^{***}$	−0.0054	0.005^{**}
BoK	DF + CC	+17.80	$< 0.001^{***}$	+0.0316	$< 0.001^{***}$
BoK	DF + CC + QO	+17.90	$< 0.001^{***}$	+0.0317	$< 0.001^{***}$
DF	CC Solver	−11.20	$< 0.001^{***}$	+0.0179	0.003^{**}
DF	QO Solver	−18.90	$< 0.001^{***}$	−0.3420	$< 0.001^{***}$
DF	BoK + QO	−3.70	0.002^{**}	+0.0054	0.564
DF	BoK + CC	+0.70	0.621	+0.0129	0.006^{**}
DF	BoK + CC + QO	+0.70	0.621	+0.0129	0.006^{**}
DF	DF + QO	+4.10	$< 0.001^{***}$	−0.0140	0.005^{**}
DF	DF + CC	+6.90	$< 0.001^{***}$	+0.0230	$< 0.001^{***}$
DF	DF + CC + QO	+7.00	$< 0.001^{***}$	+0.0230	$< 0.001^{***}$
CC Solver	QO Solver	−7.70	$< 0.001^{***}$	−0.3599	$< 0.001^{***}$
CC Solver	BoK + QO	+7.50	$< 0.001^{***}$	−0.0124	$< 0.001^{***}$
CC Solver	BoK + CC	+11.90	$< 0.001^{***}$	−0.0050	0.131
CC Solver	BoK + CC + QO	+11.90	$< 0.001^{***}$	−0.0050	0.131
CC Solver	DF + QO	+15.30	$< 0.001^{***}$	−0.0319	0.123
CC Solver	DF + CC	+18.10	$< 0.001^{***}$	+0.0051	$< 0.001^{***}$
CC Solver	DF + CC + QO	+18.20	$< 0.001^{***}$	+0.0052	$< 0.001^{***}$
QO Solver	BoK + QO	+15.20	$< 0.001^{***}$	+0.3474	$< 0.001^{***}$
QO Solver	BoK + CC	+19.60	$< 0.001^{***}$	+0.3549	$< 0.001^{***}$
QO Solver	BoK + CC + QO	+19.60	$< 0.001^{***}$	+0.3549	$< 0.001^{***}$
QO Solver	DF + QO	+23.00	$< 0.001^{***}$	+0.3280	$< 0.001^{***}$
QO Solver	DF + CC	+25.80	$< 0.001^{***}$	+0.3650	$< 0.001^{***}$
QO Solver	DF + CC + QO	+25.90	$< 0.001^{***}$	+0.3650	$< 0.001^{***}$
BoK + QO	BoK + CC	+4.40	$< 0.001^{***}$	+0.0074	$< 0.001^{***}$
BoK + QO	BoK + CC + QO	+4.40	$< 0.001^{***}$	+0.0074	$< 0.001^{***}$
BoK + QO	DF + QO	+7.80	$< 0.001^{***}$	−0.0194	0.359
BoK + QO	DF + CC	+10.60	$< 0.001^{***}$	+0.0176	$< 0.001^{***}$
BoK + QO	DF + CC + QO	+10.70	$< 0.001^{***}$	+0.0176	$< 0.001^{***}$
BoK + CC	BoK + CC + QO	+0.00	1.000	+0.0000	1.000
BoK + CC	DF + QO	+3.40	0.001^{**}	−0.0269	0.008^{**}
BoK + CC	DF + CC	+6.20	$< 0.001^{***}$	+0.0101	$< 0.001^{***}$
BoK + CC	DF + CC + QO	+6.30	$< 0.001^{***}$	+0.0102	$< 0.001^{***}$
BoK + CC + QO	DF + QO	+3.40	0.001^{**}	−0.0269	0.008^{**}
BoK + CC + QO	DF + CC	+6.20	$< 0.001^{***}$	+0.0101	$< 0.001^{***}$
BoK + CC + QO	DF + CC + QO	+6.30	$< 0.001^{***}$	+0.0102	$< 0.001^{***}$
DF + QO	DF + CC	+2.80	$< 0.001^{***}$	+0.0370	$< 0.001^{***}$
DF + QO	DF + CC + QO	+2.90	$< 0.001^{***}$	+0.0370	$< 0.001^{***}$
DF + CC	DF + CC + QO	+0.10	1.000	+0.0000	0.317

Solver maintains 30–50% accuracy at levels 16–20 and achieves the highest Hard-tier pass rate of any single system.

Table 26: SLR-Bench search space by curriculum level.

Curriculum level	Ground literals L	Candidates (≤4-body)	@50 ms/eval	@200 ms/eval
1	5	30	<1 s	<1 s
5	11	561	<1 s	<1 s
10	19	5,035	∼4 min	∼17 min
15	33	46,937	∼39 min	∼2.6 hr
20	50	251,175	∼3.5 hr	∼14 hr



(a) Pass rate by curriculum level.

(b) Mean reward by curriculum level.

Figure 11: **SLR-Bench: per-level breakdown.** LLM methods dominate at low curriculum levels; symbolic solvers sustain accuracy at Hard levels (16–20) where LLMs collapse.

Symbolic solver breakdown by rule complexity. Complexity groupings follow the binning used in the original SLR-Bench paper (Helff et al., 2025). Table 27 shows that CC Solver accuracy degrades gracefully with rule complexity, maintaining mean reward above 0.94 at all complexity levels. In contrast, QO Solver's mean reward collapses at complexity 2+ (0.32–0.51), reflecting a larger fraction of complete failures rather than near-misses.

Table 27: SLR-Bench symbolic solver accuracy by ground-truth rule complexity.

Complexity	N	Acc% (CC)	Reward (CC)	Acc% (QO)	Reward (QO)
1	50	100.0%	1.0000	100.0%	1.0000
1–2	350	93.1%	0.9890	94.3%	0.9429
2–3	150	67.3%	0.9612	50.7%	0.5067
3–4	100	47.0%	0.9435	33.0%	0.3300
4–5	250	44.4%	0.9462	31.6%	0.3160
5	100	49.0%	0.9563	39.0%	0.3900

Complexity prediction. Figure 12 shows predicted vs. ground-truth rule complexity for all systems. In contrast to PBEBench where all systems overshoot ground-truth complexity ($\Delta > 0$), on SLR-Bench all systems predict *simpler* rules than ground truth ($\Delta < 0$), reflecting the abundance of shorter equivalent Prolog forms. Symbolic solvers undershoot most aggressively ($\Delta \approx -0.76$ to -1.09) consistent with their ascending-complexity search that stops at the first correct rule. LLM-based methods undershoot less ($\Delta \approx -0.61$ to -0.83), as they occasionally produce more complex correct rules.

F.7 Library-Learning Baseline: TroVE

We implement TroVE (Wang et al., 2024b) inside the same Qwen3.6-35B-A3B + OpenHands + verifier harness used for our QO solver and the direct coding-agent baseline, so any difference reflects the induction method rather than the model or infrastructure. The implementation keeps TroVE's three modes (IMPORT a toolbox function, CREATE a new one, SKIP the toolbox and solve directly), a growing importable toolbox, usage-based trimming, and candidate selection with $K = 3$ candidates per task. Prompts state the exact

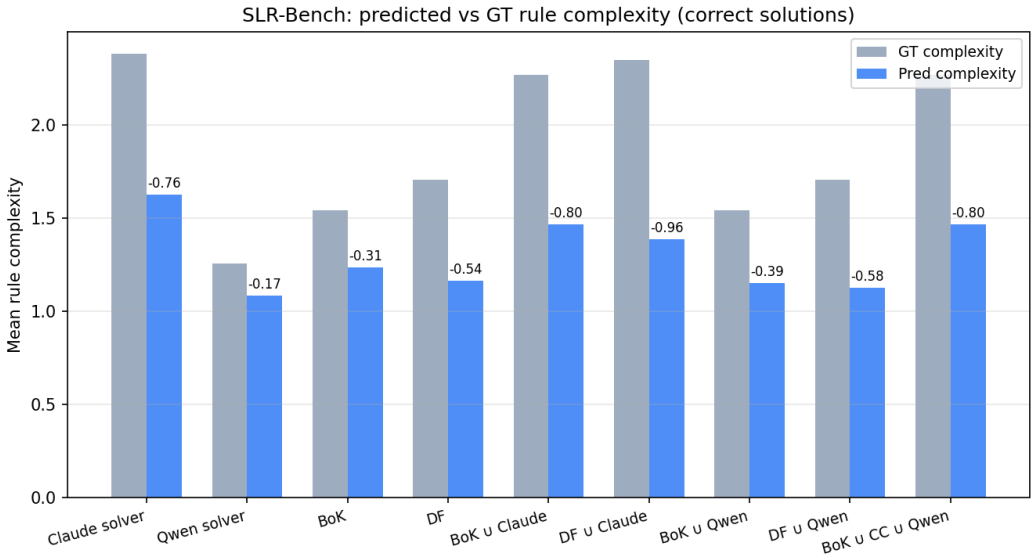


Figure 12: **SLR-Bench: predicted vs. ground-truth rule complexity.** All systems predict simpler rules than ground truth; symbolic solvers undershoot most aggressively due to ascending-complexity search.

DSL and output format each verifier parses. We make two deviations, neither of which favors ReaComp: candidates are selected by verifier reward rather than self-consistency (an exact verifier is available), and generations are sandboxed with a per-execution memory limit (some induced "general" toolbox functions are brute-force searches that otherwise exhaust memory). The default configuration disables thinking and uses 4K tokens per call; the compute-matched configuration enables thinking and uses 16K tokens per call, the same per-call budget as the coding-agent baseline. We did not match *total* tokens: TroVE lacks the agent's multi-step tool-use loop, so it uses fewer tokens per task by construction. We run PBEBench-Lite and SLR-Bench only. Tables 1 and 3 report the compute-matched configuration.

Results. TroVE reaches 47.9% (default) and 53.2% (compute-matched) on PBEBench-Lite, and 44.1% and 54.5% on SLR-Bench (tiers 100 / 66.8 / 9.6 / 0.0 and 100 / 76.4 / 29.6 / 12.0). Compute-matched, it trails the plain coding agent by 34.0pp on Lite and 3.9pp on SLR, and the best ReaComp hybrid by 40.7pp and 32.2pp.

The toolbox memorizes rather than abstracts. Inspecting the induced toolboxes shows near-duplicate, task-specific functions: on PBEBench-Lite, some functions hardcode a single task's answer, and on SLR-Bench, CREATE calls repeatedly reuse one function name, so the "library" is a single slot overwritten with a hardcoded rule string from the most recent task. Reuse hurts: importing a toolbox function solves 2 of 393 SLR-Bench tasks (0.5%), versus 538 of 584 (92%) when the model writes a fresh solution (SKIP), with the same pattern on Lite (IMPORT 24% vs. SKIP 67%). This matches the compute-matched re-evaluation of Sesterhenn et al. (2025), who find TroVE's gains shrink to about 1%. By contrast, ReaComp compiles one general algorithm (e.g., ascending-complexity search over the predicate space on SLR-Bench) that re-derives each task's answer from its own examples. We evaluate TroVE with a single coding agent only; a stronger agent or a richer toolbox listing might abstract better, so we do not claim that library learning fails in general.

F.8 Case Study: Solver-Seeded Fallback

In our hybrids the LLM fallback is invoked independently of the solver (Algorithm 2), even though the solver returns its top-scored near-miss programs. As a case study, we instead warm-start Direct Feedback from the CC solver's near-miss program, so the LLM refines

it rather than searching from scratch, on the tasks the solver fails (gpt-oss-120b, $K = 32$). This is a form of refinement of partially correct solutions (Madaan et al., 2023; Chen et al., 2024; Pan et al., 2023), not part of our core contribution; we report it as a separate row group in Tables 1 and 3. We did not run it on PBEBench-Hard.

Algorithm 3 gives the procedure; it differs from plain DF only in that the feedback history starts from the solver's near-miss instead of being empty.

Algorithm 3 Hybrid Inference with Solver-Seeded Direct Feedback

Require: task $\mathcal{D}$, solver S_ϕ, verifier R, LLM policy π_θ, attempt budget K

$\mathcal{C}_S \leftarrow S_\phi(\mathcal{D})$
$p_S \leftarrow \arg\max_{p \in \mathcal{C}_S} R(p; \mathcal{D})$
if $R(p_S; \mathcal{D}) = 1$ **then**
  return p_S
end if
$H \leftarrow [(p_S, \textsc{Feedback}_R(p_S; \mathcal{D}))]$ ▷ plain DF: $H \leftarrow [\,]$
$p^* \leftarrow p_S$
for $k = 1, \ldots, K$ **do**
  $p_k \sim \pi_\theta(\cdot \mid \mathcal{D}, H)$
  if $R(p_k; \mathcal{D}) > R(p^*; \mathcal{D})$ **then**
    $p^* \leftarrow p_k$
  end if
  if $R(p_k; \mathcal{D}) = 1$ **then**
    break
  end if
  $H \leftarrow H \cup [(p_k, \textsc{Feedback}_R(p_k; \mathcal{D}))]$
end for
return p^*

Solver-failed subset	Plain DF	Seeded DF	Gain
SLR-Bench ($N = 316$)	57.6	78.8	+21.2
SLR-Bench, Hard tier ($n = 133$)	21.1	55.6	+34.6
PBEBench-Lite ($N = 198$)	64.6	72.2	+7.6
PBEBench-Lite, cascade length 5 ($n = 128$)	52.3	62.5	+10.2

Table 28: **Solver-seeded vs. plain DF fallback** (accuracy %, on tasks the CC solver fails).

Seeding helps on both benchmarks, and the gain grows with difficulty, where the solver's partial progress is most informative. On the full sets, seeding lifts DF + CC from 86.6% to 93.3% on SLR-Bench (Hard tier 58.0% to 76.4%) and from 93.1% to 94.5% on PBEBench-Lite. On the solver-failed subsets, the seeded fallback uses 100.2M tokens ($8.75) on SLR-Bench versus 138.8M ($10.90) for plain DF, and 41.2M ($6.45) on PBEBench-Lite versus 80.5M ($11.97), since it solves more tasks early (11.7 of 32 attempts on average).

F.9 Pilot: BIG-bench List Functions

To test the recipe on a third verifier-backed PBE domain, we ran ReaComp end-to-end on List Functions (Rule, 2020): 250 integer-list-to-integer-list induction tasks, each with 32 example pairs split into 24 shown and 8 held-out. A task counts as solved only if the induced program reproduces every held-out pair exactly; the natural-language description of the target function is withheld from both the trace model and the solver. Induction is label-free, as in the main experiments. We use two demonstration sets: a pilot set of 10 tasks (best gpt-oss-120b Best-of-K trace per task) and a balanced set of 25 tasks × 4 traces (balanced across success and failure and across input-output edit distance). Traces come from gpt-oss-120b (BoK, $K = 8$), and solvers are induced by Qwen3.6-35B-A3B via OpenHands.

Results. Single solvers are weak and high-variance (16.0–35.2% held-out), but the union of all 9 solvers reaches 51.6% at zero LLM inference cost. Using this ensemble as a zero-cost first pass with BoK fallback on the remaining tasks raises accuracy from 87.6% (BoK alone) to 90.4% while cutting LLM tokens by 36.2%. The induced solvers are verifier-guided searches over list primitives (slice, reverse, sort, filter, deduplicate, arithmetic, indexing, interleave, cumulative operations) rather than lookup tables.

For reference (same 250 functions, differing protocols): direct GPT-4o reaches 39.9% and DeepSeek-R1 with sample-steered rule refinement about 83% (Li et al., 2025a); the human mean is about 52% (Rule et al., 2024), under an online-learning protocol that is not directly comparable; hypothesis refinement (Qiu et al., 2024) is lower. Most of the hybrid's accuracy comes from the strong fallback model (BoK alone already exceeds prior results we found), so here the solvers mainly contribute efficiency rather than a large accuracy gain. Extending to MiniSCAN and Mini-ARC is left for future work.

G Additional Related Work

We discuss additional bodies of literature relevant to our work.

Neurosymbolic Reasoning with LLMs. A broad line of work improves LLM reasoning by coupling neural generation with symbolic execution, external tools, or explicit verification. PAL (Gao et al., 2023) delegates intermediate computations to executable programs, which can reduce arithmetic and logical errors. Follow-up work by Kabra et al. (2024) shows that program-aided reasoners can also be better calibrated than text-only chain-of-thought prompting. ReAct (Yao et al., 2022) extends this paradigm to interleaved reasoning and acting during tool use. ReWOO (Xu et al., 2023) further separates reasoning from observation-heavy execution to reduce tool-calling overhead. CodeAct (Wang et al., 2024a) treats executable Python code as a unified action space for LLM agents, allowing actions to be executed and revised through an interpreter over multiple turns. More recent analyses such as TIR (Zhao et al., 2025) study when tool-integrated reasoning helps in practice and when the added orchestration primarily increases latency or compute, and cost-aware evaluations make this trade-off even more explicit (Ghoshal & Al-Bustami, 2026). LLM-PV (Singhal et al., 2025) uses an LLM as a proposal prior over programs and validates candidate programs by execution on held-out examples. Our Best-of-K (BoK) baseline is closest in setup to this proposal-and-selection view: both sample candidate programs from an LLM and use execution-based signals to select among them.

Program Induction with LLMs. Work on program induction from examples predates LLMs. DeepCoder (Balog et al., 2016) learns to guide search over a DSL from input-output examples, while RobustFill (Devlin et al., 2017) studies neural program induction for string transformations under noisy I/O. BUSTLE (Odena et al., 2020) further shows that learned models can guide bottom-up search over programs from input-output examples by prioritizing promising compositions of intermediate values during synthesis. In the LLM era, Li & Ellis (2024) show that off-the-shelf models remain weak PBE solvers but can become competitive after fine-tuning, especially in-distribution. HYSYNTH (Barke et al., 2024) proposes a hybrid approach in which LLM completions are distilled into a task-specific, context-free surrogate model that then guides symbolic program synthesis. Verbruggen et al. (2025) propose execution-guided within-prompt search for PBE, where the model iteratively samples, combines, and annotates candidate code with execution results to carry out search within a single prompt. In historical linguistics Naik et al. (2025a) cast sound law induction as PBE and explore synthetic training data for this setting. PBEBench (Naik et al., 2025b) and SLR-Bench (Helff et al., 2025) provide executable, procedurally generated reasoning benchmarks that make compositional induction and scaling behavior more measurable. In a complementary application setting, PBE Meets LLM (Zhang & Park, 2025) studies tabular transformations with a hybrid solver-first pipeline, while Case2Code (Shao et al., 2025) scales induction from program behavior into a synthetic data source for code models.

Library Learning with LLMs. Our method is also related to library learning and abstraction discovery. DreamCoder (Ellis et al., 2021) is the canonical precursor, jointly learning reusable symbolic abstractions and a neural search policy through wake-sleep library learning. Stitch (Bowers et al., 2023) revisits library learning from a symbolic perspective, introducing corpus-guided top-down synthesis for extracting reusable abstractions from a program corpus much more efficiently than prior deductive approaches. LILO (Grand et al., 2023) builds on this line by combining LLM-guided synthesis with Stitch-style symbolic compression and automatic natural-language documentation of learned abstractions. More recent LLM-based work revisits this idea through tool and abstraction discovery. ReGAL (Stengel-Eskin et al., 2024) refactors programs into more general reusable abstractions. TroVE (Wang et al., 2024b) induces verifiable toolboxes for programmatic tasks. ToolLibGen (Yue et al., 2025) extracts tools from reasoning traces and aggregates them into larger libraries for reuse. Voyager (Wang et al., 2023) offers a related skill-library perspective in embodied agents, where executable skills are accumulated and retrieved over long horizons. At the same time, a compute-matched re-evaluation of TroVE (Sesterhenn et al., 2025) suggests that some reported gains may shrink once inference-time compute is carefully controlled.In our matched-harness comparison (Appendix F.7), TroVE's toolbox likewise fills with memorized task-specific functions whose reuse lowers accuracy.

RL and latent-reasoning formulations. ARCLE (Lee et al., 2024) casts ARC as an MDP and learns sequences of grid-edit actions with RL, and reports that RL over the raw action space is hard. ReaComp keeps structured search but moves it offline: it searches over whole programs scored by a verifier and compiles the result into a standalone solver with zero per-task cost. Where no verifier exists, the RL formulation has the advantage. Latent recursive reasoning such as GRAM (Baek et al., 2026) amortizes reasoning into latent trajectories and needs no DSL, a strength on open-world tasks. ReaComp makes the opposite trade, externalizing reasoning as an explicit program: it gives up differentiability but yields an artifact that is inspectable, reusable, and nearly free at test time. We see the two as complementary.

ARC and DSL-completeness. ReaComp targets domains where a compact shared DSL covers the task family and a fast exact verifier is available. DSL-based program search was pivotal in early ARC: the 2020 Kaggle winner used a hand-built grid DSL with brute-force search, and Hodel's arc-dsl[2] provides a DSL for the ARC-AGI-1 training set. The frontier has since moved beyond any fixed DSL: Li et al. (2025b) keep a primitive library but still allow arbitrary Python code, and ARC-AGI-2 was constructed to remove brute-forceable tasks (Chollet et al., 2025), so a fixed DSL covers only part of it. We therefore see ARC-AGI-1 and the DSL-expressible part of ARC-AGI-2 as in scope in principle but search-hard, since the obstacle is tractable search depth rather than expressivity; a fixed DSL as the backbone of a broader arbitrary-program synthesizer is a promising extension. ARC-AGI-3 is an interactive, agentic benchmark rather than a static input-output task and falls outside the PBE setting studied here.

Other inductive-reasoning benchmarks. Among the benchmarks surveyed by Qiu et al. (2024), List Functions (Rule, 2020) and MiniSCAN fit our setting (compact primitives and exact-match verifiers), and we report a List Functions pilot in Appendix F.9. ACRE is a mismatch: a visual abductive-causal benchmark without an exact program verifier. Qiu et al. (2024) also split LLM hypothesis proposal from symbolic rule application; this is convergent with our approach, and ReaComp automates the same division of labor by compiling the reasoning into an executable solver.

Refining partially correct solutions. The solver-seeded fallback in Appendix F.8 belongs to the literature on refining partially wrong outputs with feedback (Madaan et al., 2023; Chen et al., 2024; Pan et al., 2023); we include it as an ablation, not as a contribution.

[2] `https://github.com/michaelhodel/arc-dsl`