SOLVITA: ENHANCING LARGE LANGUAGE MODELS FOR COMPETITIVE PROGRAMMING VIA AGENTIC EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) still struggle with the rigorous reasoning demands of hard competitive programming. While recent multi-agent frameworks attempt to bridge this reliability gap, they remain fundamentally stateless: they rely on static retrieval and discard the valuable experience gained from solving previous problems and debugging their solutions. To address this, we present Solvita, an agentic evolution framework that enables continuous learning without requiring weight updates to the underlying LLM. Solvita coordinates the Planner, Solver, Oracle, and Hacker in a closed loop. Trainable knowledge networks use reinforcement learning from task feedback to improve the retrieval of prior experience. Solvita achieves the highest pass@3 in 14 of 15 combinations of models and benchmarks across CodeContests, APPS, and AetherCode. In paired comparisons, it performs comparably to commercial agents on their native backbones and achieves higher pass@3 in all 24 comparisons on other backbones. In controls using GPT-5.4 with matched budgets, learned Solver routing outperforms BM25 retrieval and LLM skill selection by 3.9 to 7.5 percentage points. On Codeforces, Solvita reaches a maximum rating estimate of 3394 using CodeElo, with gains of up to 526 points over matched bare backbones.

1 INTRODUCTION

Algorithmic problem solving requires translating a natural language specification into a correct, efficient program. This demands precise formalization, deliberate strategy selection, and rigorous verification. Competitive programming benchmarks capture these demands in a controlled setting and have become a standard testbed for evaluating structured reasoning in large language models (LLMs) (Li et al., 2022; Hendrycks et al., 2021; Shi et al., 2024; Jain et al., 2025).

A common baseline for LLM coding asks the model to generate a solution directly in one response, without explicitly separating understanding, planning, coding, and verification (Chen et al., 2021). Multistep pipelines such as AlphaCodium (Ridnik et al., 2024) and MapCoder (Islam et al., 2024) add staged reasoning and iterative refinement, but confine their state to the current problem rather than updating persistent memory from outcomes across tasks. Likewise, the original RAG formulation retrieves from a fixed external index rather than learning reusable strategies online from prior attempts (Lewis et al., 2020). By contrast, human programmers accumulate reusable strategies and failure patterns across problems.

To bridge this gap, we introduce **Solvita**[1], a multi-agent framework that learns from experience across competitive programming tasks. Solvita replaces static pipelines with a closed loop comprising the Planner, Solver, Oracle, and Hacker. The Planner produces a canonical problem representation, while the Solver, Oracle, and Hacker each maintain a trainable knowledge network; the LLM backbone remains frozen. As the system solves problems and generates tests, it uses reinforcement learning to update the routing weights of these networks based on test outcomes and adversarial feedback.

[1]The name *Solvita* combines *solve* with the Latin *vita* ("life"; cf. Italian *vita*, English *vital/vitality*), evoking a solver that brings life to problem solving.

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.

Key mechanisms. ❶ **Structured skill retrieval.** A query, method, and skill (QMS) graph connects prior problems, solution decompositions, and reusable code templates; contrastive entries pair correct and incorrect solutions. ❷ **Learning retrieval from paired outcomes.** Contrastive REINFORCE learns to sample skill candidate groups using the difference in the fraction of tests passed between runs with and without QMS augmentation on the same training problem. ❸ **Adaptive testing and adversarial feedback.** The Oracle certifies generated tests, while the Hacker searches for inputs that expose failures. Contextual bandits learn which strategies to retrieve, and failures discovered during training enrich all three memories. ❹ **Targeted program repair.** The Solver edits faulty code through search-and-replace patches, accepting a patch only if previously passing tests still pass.

Contributions. Our work makes three contributions:

- We introduce **Solvita**, a **multi-agent framework** that integrates problem abstraction, program synthesis and repair, certified testing, and adversarial validation. Its closed loop shares failure feedback across agents and retains reusable experience across tasks.
- We develop **trainable knowledge networks** for the Solver, Oracle, and Hacker, organizing solution methods, certification strategies, and adversarial attacks. The Solver's **QMS graph** connects problem descriptions to reusable code skills through solution decompositions, capturing both successful approaches and their failure modes.
- We combine **memory growth with reinforcement learning** while keeping the **LLM backbone frozen**. Paired Solver outcomes, certification results, and adversarial failures provide distinct learning signals for the three networks.

2 DATA

We build a source pool for cold-start initialization and online learning in three steps: collecting competitive programming problems from multiple sources, normalizing them into a common JSON record format, and applying four stages of filtering for completeness, tag balance, redundancy, and difficulty. Section 4.4 varies only the initialization subset, including a condition initialized with an empty graph while retaining online learning.

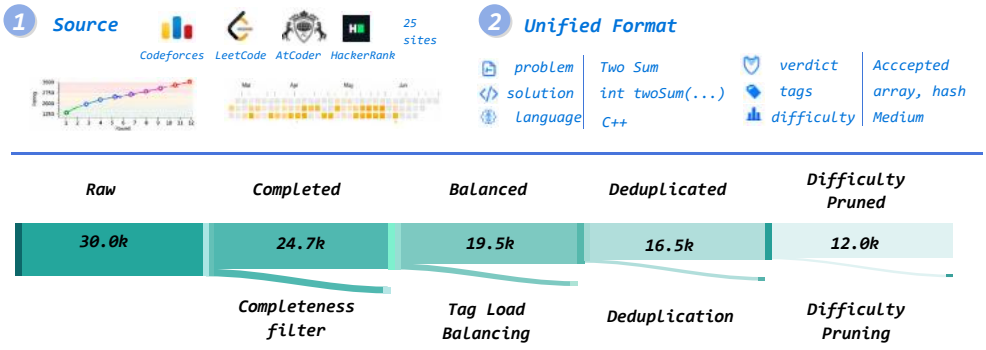


Figure 1: Data pipeline overview. Candidate problems are normalized and filtered for completeness, tag balance, benchmark isolation, redundancy, and difficulty. The resulting pool of 12.0k problems is split without overlap into 6,699 problems for cold-start initialization and 5,318 for online training.

2.1 COLLECTION

We build the corpus only from sources eligible for these benchmarks. Candidate problems come from the training splits of APPS (Hendrycks et al., 2021) and CodeContests (Li et al., 2022), and from the training split of the CodeContests+ `1x` configuration (Wang et al., 2025). Held-out evaluation problems are removed before we construct any downstream knowledge source. After filtering, the 12,017 problems are partitioned without overlap: 5,318 training problems retain statements, tests, metadata, editorials, and verdict traces for online learning, while 6,699 cold-start problems pair reference problems with editorial solutions and seed the knowledge networks.

2.2 Schema Unification

To reconcile the heterogeneous formats used across platforms, we normalize all collected artifacts into a unified JSON schema. Each record exposes a fixed set of canonical fields covering the problem statement, typed variable declarations and constraint bounds, sample and hidden tests stored as I/O pairs, editorial text, submission source with judge verdict and execution time, and algorithmic tags drawn from a controlled vocabulary. Interactive protocols, special judges, and conventions for including multiple test cases in one input are mapped to standardized flags. Each record also includes a canonical problem identifier used for matching problems across sources and ensuring benchmark isolation. The unified schema is the single input format consumed by all downstream stages.

2.3 Filtering

Starting from 30,018 problems, the unified corpus is refined through four sequential filters. Tag load balancing precedes benchmark isolation and deduplication to reduce the cost of overlap auditing, and difficulty pruning is applied last so that the floor is set using the distribution after deduplication. ❶ **Completeness.** A problem is retained only if it carries both public and private test sets, algorithmic tags, a difficulty signal (e.g., rating, tier, or division), and a parseable I/O specification with explicit constraint bounds; this stage leaves 24,712 problems. ❷ **Tag load balancing.** A handful of tags would otherwise dominate the corpus and bias training toward their solution patterns, so we cap each tag at $C_{\max}$ problems and subsample problems from overrepresented tags, keeping the number of problems per tag within a constant factor of the smallest tag count and yielding 19,486 problems. ❸ **Benchmark isolation and deduplication.** We remove benchmark overlap and within-corpus duplicates, leaving 16,503 problems. ❹ **Difficulty pruning.** Trivially easy problems are removed because the base LLM solves them without experience augmentation: survivors are ranked by platform difficulty and any problem below the per tag floor $d_{\min}^{(\tau)}$ is dropped, producing the final source pool of 12,017 problems. Appendix A lists the difficulty mapping for each platform, the value of δ, the difficulty floor $d_{\min}^{(\tau)}$ for each tag, the cap $C_{\max}$, and the initial and retained problem counts for each platform.

3 Solvita

3.1 Framework Overview

Solvita coordinates four agents (Figure 2): the **Planner** canonicalizes the problem; the **Solver** generates code and repairs it via search-and-replace patches; the **Oracle** builds a certified internal test suite; and the **Hacker** generates adversarial tests. The Solver, Oracle, and Hacker each maintain a persistent, trainable knowledge network and share failure feedback across the loop.

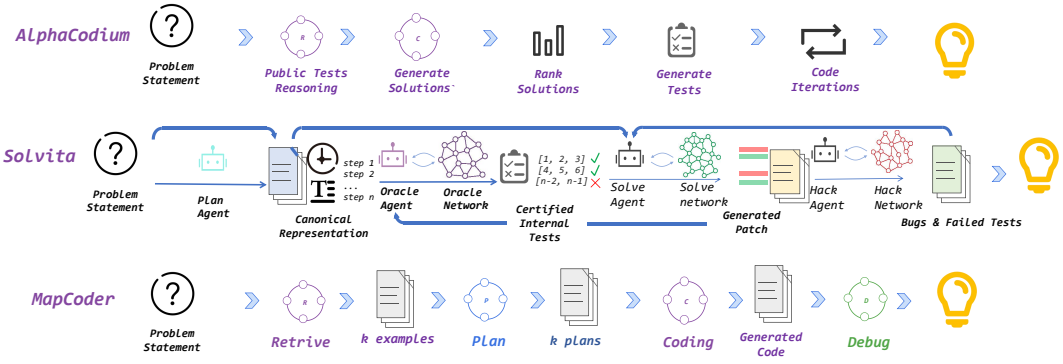


Figure 2: The Solvita architecture and its comparison with existing agent frameworks. The Planner produces a canonical problem representation; the Solver, Oracle, and Hacker form a cycle of solving, certification, and adversarial testing with refinement through patches and three trainable knowledge networks that share failure feedback.

Within a shared taxonomy, Oracle and Hacker use contextual bandits (Li et al., 2010) to retrieve strategies and learn from rewards for certification and bug exposure, respectively (Appendix E). The Solver learns QMS skill sampling from paired outcomes and receives advice retrieved by a bandit during deployment. Bandit features and defaults are in Appendix B.

3.2 PLANNER

The Planner abstracts the raw problem into a canonical specification of its objective, variables, and constraints, together with algorithmic tags for downstream processing. The `abstract_problem` prompt and JSON output schema are provided in Appendix F.

3.3 SOLVER

The Solver uses the canonical problem representation and retrieved skills to generate a C++ program and validates it against public tests and tests generated by the Oracle. On subsequent iterations, it applies *patch-based repair*: rather than regenerating the full solution, it emits search-and-replace edit blocks targeting only the diagnosed fault. A patch is accepted only if all regression tests (previously passing) still pass, preserving prior correctness while focusing effort on the unresolved case. The full prompt set (initial generation, patch decision, SEARCH/REPLACE patch, regeneration, and failure analysis) is given in Appendix F. The Solver's knowledge network has three layers: query, method, and skill (QMS). Appendix H documents its storage layout, featurizer, and mechanism for propagating events.

The Solver is backed by a heterogeneous directed graph $\mathcal{G} = (\mathcal{V}_Q \cup \mathcal{V}_M \cup \mathcal{V}_S,\ \mathcal{E}_{QM} \cup \mathcal{E}_{MS})$ with three layers (Figure 3). **Q nodes** store the description and metadata of previously encountered problems. **M nodes** decompose solutions into DAGs of function blocks; *contrastive* M nodes pair a correct and incorrect solution sharing the same approach to isolate failure points, while *analysis* M nodes summarize accepted solutions as standalone trajectories. **S nodes** store annotated algorithmic skills with C++ templates, linked to M nodes via identifiers of function blocks (deterministic match or embedding fallback). Given a new problem q_{new}, the top-k similar Q nodes are retrieved and expanded into an activated subgraph; each reachable skill s receives a selection score aggregated over all two-hop paths,

$$\phi(s \mid q_{\text{new}}) \;=\; \sum_{\substack{q_i,\, m_j:\\ q_i \to m_j \to s \,\in\, \mathcal{G}}} \mathrm{Sim}\big(q_{\text{new}},\, q_i\big) \;\cdot\; w_{\text{qm}}^{(i,j)} \;\cdot\; w_{\text{ms}}^{(j,s)}, \tag{1}$$

where w_{qm} and w_{ms} are learned edge weights. From a candidate pool C, QMS samples a group $A = (s_1, \ldots, s_K)$ sequentially without replacement using softmax scores, renormalized after each draw. The frozen LLM selects skills only from A and produces a subproblem plan; the selected skills and associated Q/M evidence form the augmentation. Retrieval and sampling budgets are in Appendix G.

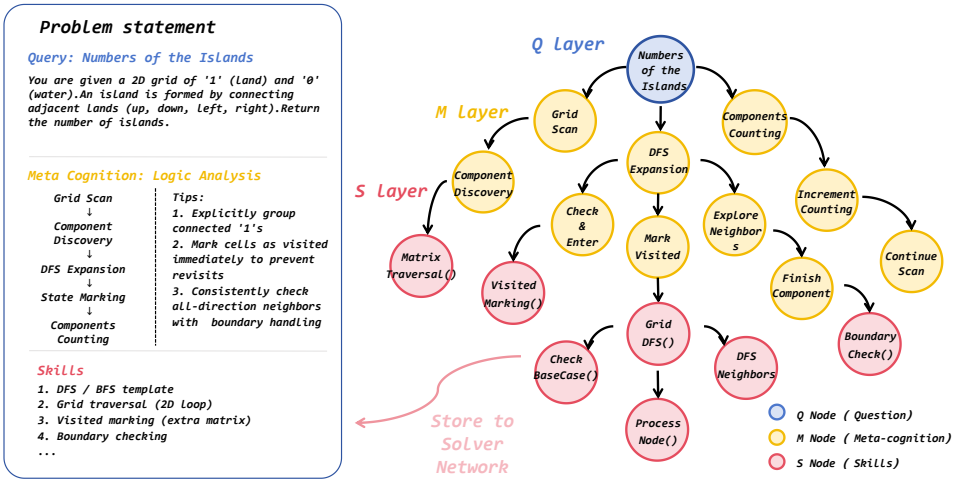


Figure 3: The three-layer Solver knowledge network. Q nodes (top) store problem descriptions and metadata; M nodes (middle) hold contrastive or analysis solution decompositions as DAGs of function blocks; S nodes (bottom) store annotated skills with code templates. Solid arrows denote deterministic links via identifiers of function blocks; dashed arrows denote fallback based on semantic similarity. Edge thickness reflects learned weight magnitude.

Training via contrastive REINFORCE. With the LLM frozen, QMS training pairs Solver runs with and without QMS augmentation under identical tests, budgets, and stopping rules. Both retain LLM planning and disable the advice interface. One candidate group is sampled per training problem and retained throughout the run. At Solver termination, R is the fraction of tests passed, yielding $\Delta R = R_{\text{with}} - R_{\text{without}}$; subsequent Oracle certification, Hacker search, and downstream patch-drift decisions are excluded. The sampled group A is the policy action. Writing $A_{<t} = \{s_1, \ldots, s_{t-1}\}$, its likelihood and update are

$$P_{\mathbf{w}}(A \mid q_{\text{new}}) = \prod_{t=1}^{K} \frac{\exp\big(\phi(s_t \mid q_{\text{new}})/T\big)}{\sum_{u \in C \setminus A_{<t}} \exp\big(\phi(u \mid q_{\text{new}})/T\big)}, \qquad \nabla_{\mathbf{w}} \mathcal{L} = -\Delta R \, \nabla_{\mathbf{w}} \log P_{\mathbf{w}}(A \mid q_{\text{new}}), \qquad \mathbf{w} \leftarrow \mathbf{w} - \beta \nabla_{\mathbf{w}} \mathcal{L}. \tag{2}$$

The gradient passes through ϕ to QM/MS weights, with MS groups renormalized after each update (Appendix B). The graph also grows: when both runs succeed, no node is added; when both fail, a contrastive M node pairs the incorrect output with the closest correct corpus solution; when outcomes differ, their outputs are directly paired. Section 4.3 separates graph growth from weight learning, and Appendix I reports diagnostics of paired rewards.

3.4 ORACLE

A shared preparation pipeline obtains a validator, reference program, and optional custom checker, using official artifacts only when available in training and generating them otherwise. The Oracle then builds internally certified supervision in four stages: (1) generate and repair a conventional testlib generator; (2) check the reference program on public samples; (3) generate N_{target} inputs and retain an input only when it passes the validator, the reference program returns a nonempty output, and the available judging procedure passes, yielding $\rho = N_{\text{cert}}/N_{\text{target}}$; and (4) apply the following gate to the input and output sets I and O, together with evidence C_{ma} from a custom checker for problems with multiple valid answers. Public samples are excluded from N_{cert}.

$$A(x, f) \;=\; \mathbb{1}\big[\, |I| > 0 \,\wedge\, |O| > 0 \,\wedge\, \rho \geq \tau \;\wedge\; \big(\text{route}(x) \neq \text{multi_answer} \,\vee\, C_{\text{ma}} \neq \emptyset\big)\big]. \tag{3}$$

Here τ is the acceptance threshold. Passing this gate is internal evidence, not a proof of correctness (Appendix C; Figure 5a). Rejected artifacts trigger a retry with another family for the reference program from $\mathcal{F}$ (e.g., top-down DP or brute force). The bandit policy of Section 3.1 selects families from problem structure and historical success.

Training. For each training problem x and selected family f, one mutually exclusive reward branch is selected from the pipeline state, certification ratio ρ, and, after full certification, the cross-check result. Appendix C gives the complete ordered rule. The bandit updates the selected family on the active features $\Phi(x)$, favoring families that yield certified tests for similar problems.

3.5 HACKER

The Hacker analyzes each candidate in a sandbox, produces a structured vulnerability report, and routes attacks among $\mathcal{U} = \{\text{semantic}, \text{stress}, \text{antihash}\}$: corner cases, stress tests at the input limits, and lattice-based hash collisions. In the joint pipeline, its distinct adversarial generators use tests and judging artifacts retained after the Oracle's checks. If a route fails, the Hacker tries fallback routes until it breaks the candidate or exhausts the budget. Valid failing inputs join the regression suite and return to the Solver for repair or regeneration; during training, their patterns also update the role-specific memories.

Training. For a chosen route u, let g_{valid} be the fraction of generated inputs accepted by the validator, g_{break} the fraction of valid inputs that expose a failure, g_{sev} the mean severity of exposed failures, and c the compile-failure count. The reward for the selected route in each round is

$$r_{\text{hack}}(x, u) \;=\; \text{clip}_{[-1,+1]}\Big(w_v \, g_{\text{valid}} \;+\; w_b \, g_{\text{break}} \;+\; w_s \, g_{\text{sev}} \;-\; \kappa(c) \Big). \tag{4}$$

Break rate receives the largest weight; Appendix D gives the weights, severity scale, compile penalty, and correction for degenerate rounds. The shared bandit updates the selected route on $\Phi(x)$; a successful break also adds a contrastive Solver entry and records the missed input pattern in the Oracle network. Prompts for the Oracle and Hacker and budgets for attack rounds are listed in Appendices F and G, respectively.

4 EXPERIMENTS

We evaluate Solvita on three benchmarks for competitive programming: CodeContests (Li et al., 2022) (CC, 165 problems), APPS (Hendrycks et al., 2021) (competition test split, 1,000 problems), and AetherCode (Wang et al., 2026) (AC, 400-problem 2024 open split), as well as recent Codeforces rounds. The main comparison uses five frontier backbones (Table 1), with the same model powering every agent within a run; diagnostic and component ablations use representative backbone subsets, matched within each table. Main comparisons and ablations report pass@3, the fraction of problems solved in any of three complete runs; Codeforces uses CodeElo rating estimates (Appendix G.6). We compare against commercial coding agents, open source frameworks for code generation, and a stateless single-pass generator. All results use benchmark-isolated checkpoints with memory writes disabled during evaluation. Each Solvita run executes three skill-plan branches and submits the one passing the most internal tests; token costs include all branches. Decoding settings, pipeline budgets, and baseline protocols are in Appendix G; prompts are in Appendix F.

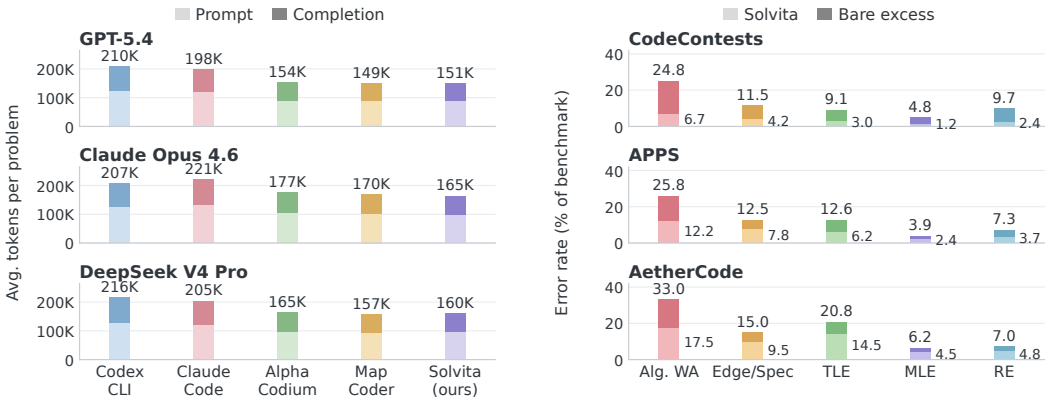
4.1 COMPARISON WITH AGENTS FOR CODE GENERATION

Solvita attains the highest pass@3 in 14 of the 15 comparisons of models and benchmarks in Table 1, with the only exception on AetherCode under Claude Opus 4.6. Each commercial agent performs best with its original backbone, with lower scores when paired with other models. Solvita outperforms the open source frameworks in every comparison, with the largest relative differences on AetherCode. Against commercial agents on their native backbones, differences lie within paired 95% bootstrap intervals of zero; Solvita leads in all 24 other comparisons with Codex CLI and Claude Code, 17 of them significant after Benjamini–Hochberg correction (Appendix I).

Table 1: Main results (pass@3, %) on CodeContests (CC), APPS, and AetherCode (AC). Each backbone is a column group; **bold** marks the best per column.

	GPT-5.4			Claude Opus 4.6			Qwen3.6-Plus			DeepSeek V4 Pro			Grok 4			Overall
Method	CC	APPS	AC	CC	APPS	AC	CC	APPS	AC	CC	APPS	AC	CC	APPS	AC	Avg
Single-pass	40.00	37.90	18.00	44.85	40.10	22.75	33.94	30.80	9.50	47.27	42.50	24.00	38.18	34.60	15.50	31.99
Codex CLI	81.82	67.10	48.50	70.30	59.80	44.25	60.00	47.90	22.50	79.39	66.30	45.00	73.33	57.20	31.50	56.99
Claude Code	70.91	60.40	42.75	80.00	69.00	**54.25**	58.79	46.30	21.50	76.97	64.50	43.75	72.12	55.50	29.75	56.43
AlphaCodium	60.61	52.40	33.00	64.24	56.20	36.50	53.33	42.30	14.50	70.91	53.40	33.00	60.61	46.10	21.00	46.54
MapCoder	57.58	54.30	30.50	60.00	55.10	38.25	50.91	41.00	16.00	66.67	54.80	35.00	62.42	44.20	19.50	45.75
Solvita (ours)	**82.42**	**67.70**	**49.25**	**80.61**	**69.30**	53.75	**69.70**	**55.10**	**26.00**	**89.09**	**68.10**	**51.50**	**78.18**	**58.50**	**33.50**	**62.18**

Figure 4 shows that Solvita's token cost per problem and run is similar to that of open source agent frameworks and substantially lower than that of commercial CLI agents. With GPT-5.4, Solvita reduces errors in all five categories across all three benchmarks; the dark segments show the bare single-pass model's excess error rate. Appendix I provides counting conventions and category definitions.



(a) Average token consumption.

(b) Residual error categories.

Figure 4: Token consumption across coding frameworks and residual error rates with GPT-5.4.

4.2 Ablation of Components and Knowledge Networks

In Table 2, *Single-pass* denotes monolithic generation, while *without training* uses the multi-agent scaffold with empty, untrained knowledge networks. Each network is added independently and evaluated after 1.5k, 3k, and all 5,318 training problems; *Full system* uses all three networks after the same 5,318 problems. The scaffold substantially outperforms single-pass generation. Each network adds further gains, with the Solver contributing the most; the full system scores highest in every evaluated cell.

Table 2: Component ablation on CC, APPS, and AetherCode (pass@3, %).

	GPT-5.4			**Claude Opus 4.6**			**Qwen3.6-Plus**		
Configuration	CC	APPS	AC	CC	APPS	AC	CC	APPS	AC
Single-pass	40.00	37.90	18.00	44.85	40.10	22.75	33.94	30.80	9.50
without training	67.88	54.50	35.00	66.06	55.10	39.50	55.76	42.00	12.00
+ Solver network @1.5k	70.30	57.40	37.75	67.88	58.60	42.75	59.39	44.90	15.75
+ Solver network @3k	73.33	60.20	41.50	72.12	61.30	45.75	61.21	47.50	18.00
+ Solver network @5,318	76.36	62.10	43.25	73.94	63.40	48.50	64.85	49.70	20.75
+ Hacker network @1.5k	69.70	55.90	36.25	67.27	56.80	41.50	57.58	43.40	13.50
+ Hacker network @3k	71.52	57.10	37.25	69.09	58.00	42.50	58.18	44.70	14.25
+ Hacker network @5,318	72.12	58.20	38.75	71.52	58.90	43.00	60.00	45.90	15.50
+ Oracle network @1.5k	70.30	56.90	38.00	68.48	57.90	42.25	58.18	44.40	14.50
+ Oracle network @3k	72.73	58.90	40.75	71.52	60.00	44.75	60.00	46.50	16.75
+ Oracle network @5,318	75.15	60.70	42.00	72.73	61.80	46.75	62.42	48.20	18.50
Full system	**82.42**	**67.70**	**49.25**	**80.61**	**69.30**	**53.75**	**69.70**	**55.10**	**26.00**

4.3 Controlled Analysis of Solver Memory

We use GPT-5.4 to separate graph coverage from learned routing and compare QMS with simpler retrieval. Following the setting in Table 2 that trains only the Solver network, we hold the multi-agent scaffold, Oracle and Hacker memories, and pipeline budgets fixed. Graph controls use checkpoints up to 4.5k training problems, and retrieval controls are evaluated at 4.5k.

Table 3: Graph growth and routing controls with GPT-5.4 (pass@3, %). Macro-averages span CC, APPS, and AC; final columns report 4.5k.

	Macro-average			At 4.5k		
Control	1.5k	3.0k	4.5k	CC	APPS	AC
Graph growth only	54.7	55.6	56.0	71.52	57.70	38.75
Weight learning only	54.9	56.5	56.9	72.73	58.60	39.50
Complete QMS	**55.2**	**58.3**	**60.4**	**75.76**	**61.80**	**43.50**

Table 4: Solver retrieval controls with GPT-5.4 (pass@3, %) under matched pipeline and retrieval budgets.

Memory	CC	APPS	AC
None	67.88	54.50	35.00
BM25	69.09	55.80	36.00
LLM pick	71.52	57.90	38.50
QMS trained	**75.76**	**61.80**	**43.50**

Graph coverage versus learned routing. We compare three settings: graph growth alone, which expands Q, M, and S nodes while keeping uniform weights fixed; weight learning alone, which trains weights on a fixed matched graph with identical reachable skills; and complete QMS, which enables both. Full control specifications are in Appendix I. Complete QMS outperforms both controls at every checkpoint (Table 3). Thus, neither mechanism alone accounts for the full memory benefit.

Retrieval alternatives under matched budgets. Table 4 compares no Solver memory, BM25 (Robertson & Zaragoza, 2009), LLM skill selection, and trained QMS under the same scaffold, corpus, retrieval slots, and budgets for LLM calls and tokens. Because Oracle and Hacker memories are empty in this setting, None coincides with *without training* in Table 2. Both retrieval baselines improve over no memory, but trained QMS outperforms both across all three benchmarks. Together with the control with graph growth alone above, these results support learned routing beyond simple retrieval.

4.4 COLD-START CORPUS SCALING

We vary subsets stratified by tag from the 6,699 problems for cold-start initialization, while keeping the disjoint online training stream of 5,318 problems and the evaluation protocol fixed. This experiment evaluates full Solvita with GPT-5.4: the 0% condition starts all three networks empty but retains all online updates, whereas 100% uses every cold-start problem and matches Table 1. It is therefore distinct from the None baseline in Table 4, which disables persistent Solver memory; all evaluations use frozen checkpoints with memory writes disabled.

Table 5 shows that learning remains effective without curated cold-start data, while a larger initialization corpus improves accuracy under a fixed budget on all three benchmarks. Gains diminish near the full corpus. This is consistent with improved sample efficiency from initialization, although performance measured at a fixed budget cannot distinguish faster learning from a higher asymptotic ceiling. Nor does corpus scaling by itself rule out solution reuse; the audit of benchmark isolation in Section 2.3 and Appendix A addresses overlap separately.

Table 5: Cold-start scaling for full Solvita with GPT-5.4 (pass@3, %). Seed subsets are stratified by tag; all conditions use the same disjoint online training stream of 5,318 problems.

Initialization	Seed problems	CC	APPS	AC
0%	0	71.52	58.00	38.50
25%	1,674	78.18	63.40	44.50
50%	3,349	80.61	66.00	47.25
100%	6,699	**82.42**	**67.70**	**49.25**

4.5 PATCH REPAIR VERSUS FULL REGENERATION

Table 6 compares adaptive patch repair with always-regenerate under matched iteration budgets ($N_{\max} = 8$), retrieval, and decoding. In patch mode, `patch_decision` may still route a systemic flaw to full regeneration; the baseline regenerates at every iteration. *Solve* is pass@3 (%), and *Iters* is mean executed Solver iterations per problem. *TokSv* is approximate Solver completion-token savings (%) relative to regeneration: $100(1 - \sum_i T_i^{\text{patch}} / \sum_i T_i^{\text{regen}})$. Here T_i includes actual completion tokens from the initial draft and subsequent Solver iterations on problem i, excluding prompt tokens and Oracle/Hacker calls; both modes allow early exit, and sums cover all evaluation problems. A dash marks the regeneration baseline. Adaptive patching improves pass@3 in every setting, with fewer iterations and approximately 73–79% fewer Solver completion tokens. Appendix I details the repair protocol, default settings, and conversion from rounded token summaries.

Table 6: Patch repair versus full regeneration.

		CodeContests (165)			**APPS (1,000)**			**AetherCode (400)**		
Backbone	Strategy	Solve	Iters	TokSv	Solve	Iters	TokSv	Solve	Iters	TokSv
GPT-5.4	Full regeneration	75.76	5.18	—	62.10	4.41	—	41.25	5.32	—
	Patch repair	**82.42**	**3.74**	**73.0**	**67.70**	**3.16**	**77.6**	**49.25**	**3.95**	**75.7**
Claude Opus 4.6	Full regeneration	73.94	5.27	—	64.30	4.48	—	45.50	5.41	—
	Patch repair	**80.61**	**3.86**	**74.3**	**69.30**	**3.24**	**78.9**	**53.75**	**4.03**	**77.2**

4.6 ORACLE AND HACKER DIAGNOSTIC CONFIGURATIONS

All configurations retain preparation of a checker, validator, and reference program. Oracle-only enables conventional tests and certification, Hacker-only enables adversarial diagnosis, and Both passes artifacts checked by the Oracle to the Hacker. Each configuration prepares artifacts independently under matched budgets (one shared set within Both) and is evaluated on the same candidates, so the results compare complete configurations rather than decisions on identical artifacts. Both achieves the highest detection rate and stronger-test share, while Oracle filtering is consistent with its higher preservation of official acceptance than Hacker-only. Full details are in Appendix I.

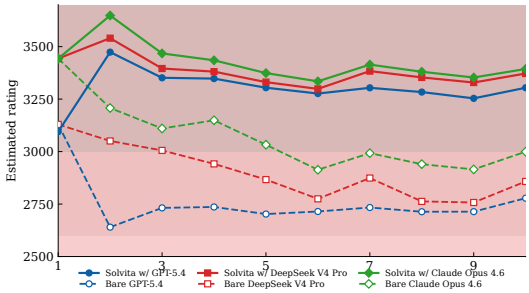
4.7 CODEFORCES COMPETITION EVALUATION

To complement the offline benchmarks, we replay ten Codeforces rounds covering 79 distinct problems, evaluating Solvita in a pass@1 setting with one submission per problem. Using the

CodeElo estimator (Quan et al., 2025), Solvita with Claude Opus 4.6, DeepSeek V4 Pro, and GPT-5.4 obtains mean rating estimates of 3394 ± 103, 3372 ± 94, and 3304 ± 94, respectively (mean $\pm$ SE across rounds). The corresponding gains over matched bare backbones are 394, 514, and 526 points (Figure 5b). Round selection, scoring, and aggregation details are provided in Appendix G.6.

Backbone	Config.	Det'd	Pres'd	Str.
Claude Opus 4.6	Oracle-only	82.4	**96.8**	8.5
	Hacker-only	87.6	92.9	11.4
	Both	**92.8**	96.0	**19.6**
GPT-5.4	Oracle-only	79.8	**96.1**	7.4
	Hacker-only	85.2	91.8	10.2
	Both	**90.6**	95.4	**17.9**
DeepSeek V4 Pro	Oracle-only	76.9	**95.2**	6.3
	Hacker-only	82.7	90.5	8.8
	Both	**88.1**	94.1	**15.6**

(a) Oracle and Hacker diagnostic configurations.



(b) Running means of rating estimates computed using CodeElo across ten chronological rounds (vertical axis: 2500–3700).

Figure 5: Diagnostics for the Oracle and Hacker, and evaluation on Codeforces. **(a)** Results across Claude Opus 4.6, GPT-5.4, and DeepSeek V4 Pro; Det'd: officially rejected candidates detected; Pres'd: preservation of official acceptance; Str.: share of disagreement problems confirmed to reflect stronger tests (counts in Table 19). **(b)** Solvita (solid) and bare-backbone (dashed) CodeElo trajectories.

5 Related Work

Code generation and self-improving agents. LLM code generation ranges from direct synthesis and the generation and filtering of large numbers of candidates (Chen et al., 2021; Li et al., 2022) to structured inference and iterative testing (Ridnik et al., 2024; Islam et al., 2024; Huang et al., 2023; Le et al., 2024; Zelikman et al., 2023). Self-improvement methods refine outputs (Madaan et al., 2023), search reasoning trajectories (Yao et al., 2023; Besta et al., 2024; Zhou et al., 2024), or train model weights using answer supervision (Zelikman et al., 2022) or RL (Le et al., 2022; Guo et al., 2025); Tao et al. (2024) survey this broader landscape.

Memory and adversarial validation. Persistent memory includes skill libraries (Wang et al., 2024), experience records (Zhao et al., 2024; Shinn et al., 2023; Qian et al., 2024), and managed or retrieved context (Packer et al., 2023; Lewis et al., 2020), as surveyed by Zhang et al. (2025). Test generation spans search guided by coverage (Lemieux et al., 2023), fuzzing with LLMs (Deng et al., 2023), generated test suites (Chen et al., 2023; Liu et al., 2023; Schäfer et al., 2024), and test construction and hacking for competitive programming (Wang et al., 2025; Shi et al., 2026). Solvita combines these directions through three role-specific knowledge networks whose contents and retrieval weights are updated from feedback while the LLM remains frozen.

6 Conclusion

We introduced Solvita, a framework that learns from experience in competitive programming while keeping LLMs frozen. Its closed loop integrates problem abstraction, program synthesis and patch-based repair, certified testing, and adversarial validation. Three role-specific knowledge networks organize reusable solving, certification, and attack strategies, with the Solver's QMS graph connecting problem structure to implementation knowledge. Learning from feedback updates retrieval weights, while shared failures enrich these memories. Benchmark comparisons and controlled ablations support improved solution quality and complementary contributions from graph growth, learned routing, and adversarial validation; patch-based repair also reduces token use relative to full regeneration. Together, these results support structured memory and feedback from program execution as a practical way to improve agentic programming without updating model weights. Limitations and future directions are discussed in Appendices J and K respectively.

AI USE STATEMENT

In this work, generative AI tools are part of the experimental object of study: frontier LLMs serve as frozen backbones inside Solvita, and all reported benchmark results are obtained from these controlled experiments. Separately, we used generative AI tools to edit manuscript wording for readability, check bibliographic metadata, and format LaTeX. We have not used generative AI tools to generate synthetic evaluation labels, formulate mathematical claims, or write proofs, and those remaining required-disclosure tasks are not applicable. We reviewed AI-assisted wording and bibliography entries against the cited sources and take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies agents for competitive programming. The Hacker generates adversarial test inputs against contest submissions in order to expose incorrect code; it is not a cybersecurity attack system. Experiments use public problems from programming contests and do not involve human subjects or private user data.

REPRODUCIBILITY STATEMENT

We document the source-pool filters and benchmark isolation in Section 2 and Appendix A; controlled comparisons of Solver memory in Section 4.3; corpus scaling with subsets stratified by tag in Section 4.4; diagnostics of paired Solver rewards in Appendix I; agent prompts in Appendix F; backbone, decoding, budget, ensemble, and baseline settings with per-method information access in Appendix G; persistent state during training and evaluation in Appendix H.5; the protocol for estimating Codeforces ratings in Appendix G.6; and storage of knowledge networks and bandit scoring in Appendix B and Appendix H. We will include anonymized source code with the supplementary material.

REFERENCES

Anthropic. System card: Claude Opus 4.6. System card, Anthropic, feb 2026. URL https://www-cdn.anthropic.com/6a5fa276ac68b9aeb0c8b6af5fa36326e0e166dd/Claude%20Opus%204.6%20System%20Card.pdf.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):17682–17690, 2024. doi: 10.1609/aaai.v38i16.29720. URL https://doi.org/10.1609/aaai.v38i16.29720.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ktrw68Cmu9c.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, jul 2021. doi: 10.48550/arXiv.2107.03374. URL https://arxiv.org/abs/2107.03374.

DeepSeek-AI. DeepSeek-V4-Pro GA release. DeepSeek API Documentation, aug 2026. URL https://api-docs.deepseek.com/news/news260813/.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 423–435. Association for Computing Machinery, 2023. doi: 10.1145/3597926.3598067. URL https://doi.org/10.1145/3597926.3598067.

Daya Guo et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, sep 2025. doi: 10.1038/s41586-025-09422-z. URL https://www.nature.com/articles/s41586-025-09422-z.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023. doi: 10.48550/arXiv.2312.13010. URL https://arxiv.org/abs/2312.13010.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. MapCoder: Multi-agent code generation for competitive problem solving. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4912–4944, Bangkok, Thailand, aug 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.269. URL https://aclanthology.org/2024.acl-long.269/.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida I. Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

kenkoooo and contributors. AtCoder Problems. GitHub repository, 2015. URL https://github.com/kenkoooo/AtCoderProblems. Accessed September 26, 2026.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 21314–21328. Curran Associates, Inc., 2022. doi: 10.52202/068431-1549. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/8636419dea1aa9fbd25fc4248e702da4-Paper-Conference.pdf.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. CodeChain: Towards modular code generation through chain of self-revisions with representative sub-modules. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=vYhglxSj8j.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering*, pp. 919–931, Melbourne, Australia, may 2023. IEEE. ISBN 978-1-6654-5701-9. doi: 10.1109/ICSE48619.2023.00085. URL https://ieeexplore.ieee.org/document/10172800.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In

H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 9459–9474. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html.

Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th International Conference on World Wide Web*, pp. 661–670. ACM, 2010. doi: 10.1145/1772690.1772758. URL https://doi.org/10.1145/1772690.1772758.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, dec 2022. doi: 10.1126/science.abq1158. URL https://doi.org/10.1126/science.abq1158.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36. Curran Associates, Inc., 2023. doi: 10.52202/075280-0943. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf.

OpenAI. GPT-5.4 Thinking System Card. System card, OpenAI, mar 2026. URL https://openai.com/index/gpt-5-4-thinking-system-card/.

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

Chen Qian, Yufan Dang, Jiahao Li, Wei Liu, Zihao Xie, YiFei Wang, Weize Chen, Cheng Yang, Xin Cong, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. Experiential co-learning of software-developing agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5628–5640, Bangkok, Thailand, aug 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.305. URL https://aclanthology.org/2024.acl-long.305/.

Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, Zekun Wang, Jian Yang, Zeyu Cui, Yang Fan, Yichang Zhang, Binyuan Hui, and Junyang Lin. CodeElo: Benchmarking competition-level code generation of LLMs with human-comparable Elo ratings. *arXiv preprint arXiv:2501.01257*, 2025. doi: 10.48550/arXiv.2501.01257. URL https://arxiv.org/abs/2501.01257.

Qwen Team. Qwen3.6-Plus: Towards real world agents. Qwen Official Blog, apr 2026. URL https://qwen.ai/blog?id=qwen3.6.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with AlphaCodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024. doi: 10.48550/arXiv.2401.08500. URL https://arxiv.org/abs/2401.08500.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 4(1–2):1–174, 2009. doi: 10.1561/1500000019. URL https://doi.org/10.1561/1500000019.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2024. doi: 10.1109/TSE.2023.3334955. URL https://doi.org/10.1109/TSE.2023.3334955.

Jingwei Shi, Xinxiang Yin, Jing Huang, Shengyu Tao, and Jinman Zhao. CodeHacker: Automated test case generation for detecting vulnerabilities in competitive programming solutions. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2352–2382, San Diego, California, United States, jul 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.108. URL https://aclanthology.org/2026.acl-long.108/.

Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. Can language models solve olympiad programming? In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=kGa4fMtP9l.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Zhengwei Tao, Ting-En Lin, Xiancai Chen, Hangyu Li, Yuchuan Wu, Yongbin Li, Zhi Jin, Fei Huang, Dacheng Tao, and Jingren Zhou. A survey on self-evolution of large language models. *arXiv preprint arXiv:2404.14387*, 2024. doi: 10.48550/arXiv.2404.14387. URL https://arxiv.org/abs/2404.14387.

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. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=ehfRiF0R3a.

Zihan Wang, Siyao Liu, Yang Sun, Ming Ding, and Hongyan Li. CodeContests+: High-quality test case generation for competitive programming. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 5576–5600, Suzhou, China, nov 2025. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.299. URL https://aclanthology.org/2025.findings-emnlp.299/.

Zihan Wang, Jiaze Chen, Zhicheng Liu, Haojie Pan, Markus Mak, Yidi Du, Geonsik Moon, Aaron Tua, Kunshuo Peng, Jiayi Lu, Boqian Zou, Chenyang Ran, Guang Tian, Shoutai Zhu, Yeheng Duan, Zhenghui Kang, Zhenxing Lin, Shangshu Li, Qiang Luo, Qingshen Long, Zhiyong Chen, Yihan Xiao, Yurong Wu, Daoguang Zan, Mingxuan Wang, and Ming Ding. AetherCode: Evaluating LLMs' ability to win in premier programming competitions. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=lSM6MtjQcM.

xAI. Grok 4 Model Card. Model card, xAI, aug 2025. URL https://data.x.ai/2025-08-20-grok-4-model-card.pdf.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate problem solving with large language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 11809–11822. Curran Associates, Inc., 2023. doi: 10.52202/075280-0517. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/271db9922b8d1f4dd7aaef84ed5ac703-Paper-Conference.pdf.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping reasoning With reasoning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 15476–15488. Curran Associates, Inc., 2022. doi: 10.52202/068431-1126. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/639a9a172c044fbb64175b5fad42e9a5-Abstract-Conference.html.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D. Goodman, and Nick Haber. Parsel: Algorithmic reasoning with language models by composing decompositions. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 31466–31523. Curran Associates, Inc., 2023. doi: 10.52202/075280-1367. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/6445dd88ebb9a6a3afa0b126ad87fe41-Paper-Conference.pdf.

zerotrac. LeetCode problem rating project. GitHub repository, 2021. URL https://github.com/zerotrac/leetcode_problem_rating. Accessed September 26, 2026.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6), nov 2025. doi: 10.1145/3748302. URL https://dl.acm.org/doi/10.1145/3748302.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, mar 2024. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160. PMLR, 2024. URL https://proceedings.mlr.press/v235/zhou24r.html.

A Data Pipeline Configuration

This appendix lists the exact configuration of every step of the filtering pipeline in Section 2.3. It also specifies the benchmark-isolation procedure applied in step (3).

Problem difficulty ratings. We use the problem ratings listed in Table 7 directly for the difficulty signal in step (1) and the per tag floors in step (4). LeetCode problems without a rating use the Easy, Medium, or Hard fallback intervals.

Table 7: Problem difficulty ratings and fallback intervals used for data filtering.

Platform	Difficulty source	Rating / interval
Codeforces	Official problem rating	Original rating
AtCoder	AtCoder Problems (kenkoooo and contributors, 2015)	Estimated rating
LeetCode	LeetCode Problem Rating (zerotrac, 2021)	Estimated rating
LeetCode fallback	Easy	800–1500
	Medium	1200–1900
	Hard	1700+

Notes. Per tag difficulty cutoffs are listed in Table 10.

Table 8: Per-tag problem counts before and after tag load balancing (step 2). The fraction kept is computed as count after balancing divided by count before balancing.

Tag	Before Balancing	After Balancing	Fraction Kept
implementation	3819	2300	60.2%
math	3200	2300	71.9%
greedy	2840	2300	81.0%
dp	2625	2300	87.6%
data_structures	2329	2118	90.9%
constructive_algorithms	1759	1594	90.6%
brute_force	1429	1302	91.1%
graphs	1416	1359	96.0%
sortings	1409	1290	91.6%
binary_search	1403	1336	95.2%
dfs_and_similar	1357	1168	86.1%
trees	1225	1155	94.3%
strings	1154	1007	87.3%
number_theory	994	943	94.9%
combinatorics	804	788	98.0%

Notes. (1) Counts after balancing are computed after applying the per-tag cap $C_{\max} = 2300$. (2) Fractions are computed per tag as (after balancing) / (before balancing). (3) Statistics are computed over all tag occurrences; a problem carrying multiple tags contributes to each of its tags. The table lists the 15 most frequent tags in the corpus.

Tag load balancing. Step (2) caps each tag at $C_{\max} = 2300$ surviving problems. Tags above the cap are subsampled uniformly at random within their difficulty distribution so that the per tag count after balancing stays within a constant factor of the smallest surviving tag.

Benchmark isolation and canonical identifiers.

Each normalized record carries a canonical identifier derived from the source judge and its stable problem ID or slug. These identifiers are used to match equivalent problems across sources and to exclude held-out evaluation splits from the source pool. Overlap with the APPS competition test split and the full CodeContests test split is removed by exact identifier match. Because AetherCode does not consistently expose compatible platform identifiers, we additionally run an embedding-based audit against the full AetherCode evaluation set. This isolation pass is applied before constructing the skill subset or any persistent knowledge store. Step (3) removes 2,983 problems in total: 2,611 by exact identifier match and the remaining 372 by the AetherCode semantic audit together with within-corpus semantic deduplication. Table 9 summarizes the candidate and held-out splits.

Table 9: Candidate sources and held-out evaluation splits used in benchmark isolation.

Dataset family	Split / set	Role in Solvita
APPS	train	candidate source for Solvita-Train
CodeContests	train	candidate source for Solvita-Train
CodeContests+	train (`1x`)	candidate source for Solvita-Train
APPS	competition test (1,000)	held out; removed by canonical-ID match
CodeContests	full test split (165)	held out; removed by canonical-ID match
AetherCode	v1 2024 open set (400)	held out; audited by embedding overlap

Notes. (1) Only the first three rows are eligible to seed the cold-start corpus. (2) The editorial-derived skill subset and all persistent knowledge stores are materialized only after this isolation pass. (3) AetherCode does not consistently expose platform identifiers compatible with the other sources, hence the separate embedding-based audit.

Overlap audit and deduplication threshold.

Step (3) embeds normalized statements with `text-embedding-3-large`. The cosine threshold is $\delta = 0.93$, chosen on a manually labeled validation set of 500 candidate pairs. The same threshold is used for the AetherCode overlap audit and for within-corpus semantic deduplication.

Retained tags and per tag difficulty floors. After steps (1)–(2) we retain $T = 107$ algorithmic tags. Each tag τ has its own difficulty floor $d_{\min}^{(\tau)}$ used in step (4), set so that the easiest surviving instance of τ is still non-trivial for the base LLM.

Table 10: Per-tag statistics and difficulty floors.

Tag	Retained after Step (3)	$d_{\min}^{(\tau)}$	Retained after Step (4)
implementation	1817	1400	1537
math	1886	1600	1423
greedy	1925	1600	1222
dp	1891	1600	1470
data_structures	1783	1600	1215
constructive_algorithms	1322	1600	1086
brute_force	946	1400	734
graphs	974	1600	601
sortings	905	1400	629
binary_search	1003	1600	818
dfs_and_similar	957	1600	773
trees	806	1600	571
strings	745	1300	503
number_theory	684	1400	435
combinatorics	526	1400	319

Notes.(1) Statistics are computed over all tag occurrences; a problem carrying multiple tags contributes to each of its tags. The table lists the 15 most frequent tags in the corpus.

Table 11: Per-platform problem counts at each filtering stage. The total row sums to the source-pool figures reported in Section 2.3.

Platform	Raw Collected	After Completeness	After Balance	After Deduplication	After Difficulty Pruning
Codeforces	15862	15573	11634	8841	7194
AtCoder	4187	3984	3471	3316	2111
Aizu	2645	2524	2276	2242	1280
Others	7324	2631	2105	2104	1432
Total	30,018	24,712	19,486	16,503	12,017

Notes. (1) The 12,017-problem total is subsequently split without overlap into 6,699 cold-start and 5,318 online-training problems. (2) The final column reflects the source pool after applying the per-tag difficulty floors $d_{\min}^{(\tau)}$ (Table 10); a problem may be counted against its primary platform even if it was retained under different tags post-balancing.

Per platform raw and surviving counts.

B Retrieval Policy Details

Strategy selection and runtime advice. The bandit based on state features is used for Oracle/Hacker strategy selection and for retrieving advice during deployment. The learning rate is $\alpha = 0.01$, rewards lie in $r \in [-1, 1]$, and a tag-overlap bonus of $+0.05$ per matching tag provides a prior. Feature keys encode the agent's current finite-state-machine position (e.g., `FSM:SOLVE_DRAFT`),

any failure type from the previous iteration (`FAIL:TIMEOUT`), and problem-level tags (`TAG:dp`). Parameters are persisted in JSON with atomic file-locking writes. Items whose running average reward drops below -0.3 after 20 or more uses are automatically deprecated.

QMS update. The group A records candidates in sampling order, and its log likelihood sums the conditional log probabilities in Eq. 2. During each update, gradients flow through the sampling likelihood to QM/MS weights while the observed return and LLM prompt are held fixed. After each update, weights are clipped below at 10^{-6} and M–S weights are renormalized within each M group (Appendix G).

C Oracle Certification and Reward Details

A problem instance is $x = (d, c, p, \kappa)$ where d is the description, c the constraints, p the public samples, and κ the structural context. The Oracle produces a supervision artifact $y = (F, f^*, T, V, A, m)$ containing the candidate family set, selected family, certified tests, verifier provenance, acceptance indicator, and metadata. Family selection uses the bandit policy of Section 3.1,

$$f^*(x) \;=\; \arg\max_{f \in F(x)} \; s_f(x), \qquad s_f(x) \;=\; b_f \;+\; \sum_{k \in \Phi(x)} W_{k,\,f}, \tag{5}$$

where $\Phi(x)$ is the set of active feature keys, and the acceptance gate follows Eq. 3.

The count N_{cert} includes only retained newly generated tests, not public samples. A counted test has an input accepted by the validator and a nonempty output produced successfully by a reference program that first passes the public samples; the output must also pass the available judging procedure. During offline training, judging first uses the dataset-provided `correct_solution`, followed by a custom checker, the reference program, and exact output matching. The dataset solution is unavailable during evaluation and deployment, where the same fallback order begins with the custom checker. A dataset solution or official checker supplies evidence independent of the generated reference program. Otherwise, checks based on generated reference programs and checkers establish only internal consistency and may preserve shared errors. A generated checker for multiple-answer problems is also checked to accept a public correct output and reject malformed output. Consequently, $\rho = N_{\text{cert}}/N_{\text{target}}$ is the fraction passing this pipeline, not an estimate of the tests' true correctness probability.

Offline Oracle training uses the following ordered, first-match rule:

$$r_{\text{oracle}} \;=\; \begin{cases} -0.6, & \text{state not ready or test list empty,} \\ -1.0, & N_{\text{cert}} = 0 \text{ and a crash is recorded,} \\ -0.7, & N_{\text{cert}} = 0 \text{ with no recorded crash,} \\ \text{round}(0.9\rho, 2), & 0 < \rho < 1, \\ r_{\text{verify}}, & \rho \geq 1. \end{cases} \tag{6}$$

For single-answer problems, the full-certification cross-check executes the dataset-provided `correct_solution`, when available, and assigns

$$r_{\text{verify}} = \begin{cases} +1.0, & \text{all generated outputs agree,} \\ -0.2, & \text{only a subset agrees,} \\ -0.5, & \text{none agree,} \\ 0, & \text{no dataset solution can be executed.} \end{cases} \tag{7}$$

For multiple-answer problems, $r_{\text{verify}} = +1.0$ when verification passes and -0.2 otherwise. These branches are mutually exclusive and describe offline Oracle training rather than online solving.

D Hacker Reward: Degenerate-Round Details

Let V be all sandbox verdicts, V_{valid} the valid-input subset, $V_{\text{break}} \subseteq V_{\text{valid}}$ those exposing a failure, and c the compilation failure count. Thus $g_{\text{valid}} = |V_{\text{valid}}|/|V|$ and $g_{\text{break}} = |V_{\text{break}}|/\max(|V_{\text{valid}}|, 1)$. The

default per-round reward is the clipped composition of Eq. 4; on degenerate rounds (`Gen_Failed` with no valid verdict) the explicit correction $r = -0.6 - \min(0.3,\ 0.1\,c)$ replaces the default so that repeated generator failures still produce a usable gradient. The judge follows the same priority order as the Oracle, with the dataset `correct_solution` available only during offline training.

Component weights. The Hacker reward of Eq. 4 is a weighted combination of the valid-input rate g_{valid}, the break rate g_{break}, and the average severity g_{sev}, minus a compile-failure penalty $\kappa(c)$. The default weights and penalty are tabulated in Table 12. The bulk of the budget sits on w_b so that actually breaking the candidate dominates the gradient, while w_v keeps the router from collapsing to high-severity but invalid-by-validator inputs.

Quantity	Symbol	Value
Valid-input weight	w_v	0.20
Break-rate weight	w_b	0.55
Severity weight	w_s	0.25
Compile penalty (per failure, capped at 0.3)	$\kappa(c)$	$\min(0.3,\ 0.1\,c)$
Reward clip range	—	$[-1.0,\ +1.0]$

Table 12: Hacker reward component weights and compile-failure penalty schedule.

Per-verdict severity. The severity table $\omega(\cdot)$ maps each per-input verdict to a scalar in $[0, 1]$ that feeds the round-level $g_{\text{sev}} = \frac{1}{|V_{\text{break}}|} \sum_{v \in V_{\text{break}}} \omega(v)$. Table 13 lists the values used across all reported runs; they encode the rough cost-to-fix ordering used by competitive-programming judges (a wrong-answer is cheap to triage, a sandbox crash is expensive).

Verdict	$\omega(\cdot)$
Wrong Answer (WA)	0.50
Time Limit Exceeded (TLE)	0.65
Memory Limit Exceeded (MLE)	0.75
Runtime Error (RE)	0.85
Sandbox crash / fatal signal	1.00

Table 13: Per-verdict severity weights used by the Hacker reward. Values were calibrated once on a 200-problem dev split and held fixed across all reported runs.

Degenerate-round sanity. When $|V_{\text{valid}}| = 0$ the round contains no informative break signal, but the generator may still have learned (or unlearned) something — e.g. producing inputs the validator rejects en masse. The fixed -0.6 baseline plus the linear compile penalty ensures the bandit signal stays bounded and nonzero so the route weights still update; without this correction, repeated `Gen_Failed` rounds would silently zero out the gradient and freeze the router on its current arm.

E Oracle and Hacker Strategy Taxonomy

Figure 6 instantiates the role-aligned separation described in Section 3.1. Both networks organize experience under six shared algorithm categories—Math, Graph/Tree, String, DP/Bitmask, Geometry, and Validation—but store different strategy units. Oracle memory contains routes for constructing reliable supervision, including exact enumeration, reference solvers, and specification-conformant checkers. Hacker memory instead contains attacks targeting implementation failures, including worst-case structures, state explosion, numerical boundaries, and invalid-case validation. The shared categories provide a common problem representation, while the distinct stored units let each learned router optimize for certification or bug exposure.

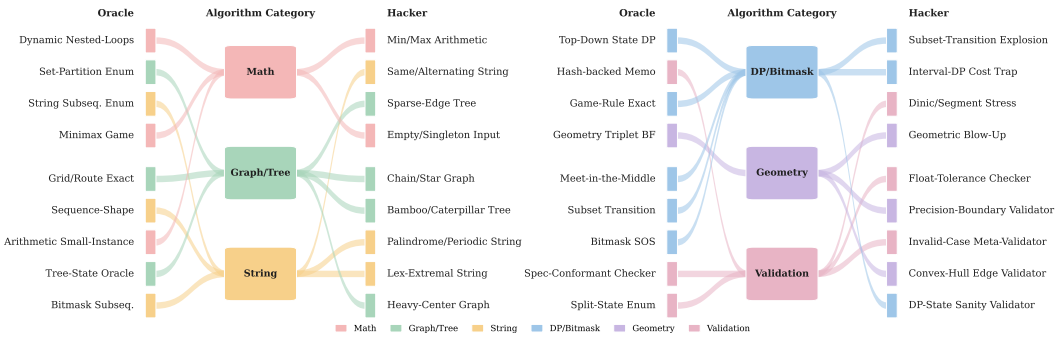


Figure 6: Representative Oracle certification strategies and Hacker attack strategies organized around shared algorithm categories.

F Prompt Details

This appendix collects the actual prompt templates used by every agent in our experiments. All prompts are stored in a single YAML file (`config/prompt_template.yaml`) and rendered through a placeholder substitution layer (`<KEY>` tokens are filled at call time). For brevity we show the role-defining portions; cross-cutting boilerplate (output schemas, fast-I/O reminders, JSON-escaping warnings) is preserved verbatim where it materially affects behavior. The same prompts are used across all five backbones in our experiments.

Conventions. Unless noted otherwise, every JSON-output prompt below requires *strict* JSON: no Markdown fences, no commentary outside the object, and embedded newlines/tabs/backslashes escaped (`\n`, `\t`, `\\`). C++ outputs follow a single template: `#include`-headers only (no `<bits/stdc++.h>`), C++17, fast I/O, with every major data-structure allocation justified against the input bounds; we explicitly highlight these only where the prompt diverges from this default. The TLE budget rule used throughout is the standard 10^8 ops/sec heuristic: `iterations / 10^8` $\leq$ `time_limit_seconds`.

F.1 Planner

The Planner reduces the raw problem statement to a canonical form and a tag set. Both the system message and the user message are short and JSON-bound so that downstream agents can parse the output deterministically.

Planner — abstract_problem (system)

```
You extract a precise canonical formulation and algorithmic tags for a competitive
programming task.
Respond with a single JSON object only.  No markdown fences.
```

Planner — abstract_problem (user)

```
Original problem:
<PROBLEM_DESC>

Known problem types (hints):  <PROBLEM_TYPES>

Constraints JSON:
<CONSTRAINTS_JSON>

Primary tags (level-1, coarse topics - prefer at least one when it fits):
<TAG_WHITELIST_LEVEL1>

Fine-grained tags (level-2 - optional; add when they precisely match the task):
<TAG_WHITELIST_LEVEL2>

Allowed tags reference (union of level-1 and level-2 vocabularies):
<TAG_WHITELIST>
```

```
Return JSON with this exact shape:
{
  "canonical_problem": {
    "objective": "string",
    "inputs": {},
    "outputs": {},
    "constraints": {"normalized": {}, "derived": []},
    "required_properties": [],
    "edge_cases": []
  },
  "algorithmic_tags_level1": ["primary tags from the level-1 list only"],
  "algorithmic_tags_level2": ["optional fine-grained tags from the level-2 list only"],
  "abstract_confidence": 0.0,
  "abstract_trace": {"rationale": "why this abstraction fits", "warnings": []}
}
Rules:
- canonical_problem must be self-contained and precise.
- algorithmic_tags_level1 are the main coarse topic labels for this task; use only
entries from the level-1 list above.
- algorithmic_tags_level2 are optional finer-grained labels; use only entries from the
level-2 list above, and do not restate the same idea already covered by level-1.
- If a tag could fit both levels, put it in level-1 only and omit it from level-2.
- abstract_confidence in [0,1] reflects how well the statement constraints are satisfied.
- abstract_trace must briefly explain tagging and any ambiguity.
```

F.2 SOLVER: SKILL SELECTION

After the canonical form is available, QMS samples five candidates from a pool of the top 20 skills. The frozen LLM receives their identifiers, titles, descriptions, and ϕ scores, alongside the problem summary and activated Q/M context, and returns selected skill identifiers and a valid subproblem DAG for the current task. The strict JSON contract (no fences, exact id copying, no commentary) is enforced because the identifiers are used for direct skill lookup.

Solver — solver_skill_selection (system)

```
You output exactly one JSON object and nothing else.
Required keys: (1) "selected_skill_ids" (array of strings) -- copy ids exactly from the
candidate list;
(2) "subproblem_dag" (object) with "nodes" (array of {id, description}) and "edges"
(array of {from_id, to_id}).
The DAG must contain at least one node and be acyclic: edges mean dependency (from_id
before to_id).
Use abstract algorithmic descriptions for nodes (not story fluff).
If no candidate skill applies, use [] for selected_skill_ids.
Always produce a meaningful subproblem_dag for the current problem.
No markdown fences, no commentary, no text before or after the JSON.
```

Solver — solver_skill_selection (user)

```
You are selecting algorithmic skills (reference snippets) for a competitive programming
task, and you must
also output a sub-problem DAG that decomposes the current problem into abstract
sub-goals.
Use the problem summary, the activated graph context (similar Q / M evidence), and the
candidate list
(φ ranks retrieval relevance).
Choose between <MIN_SELECT> and <MAX_SELECT> (inclusive) skills that genuinely help solve
the current task.
If no candidate is relevant, return [] for selected_skill_ids. Otherwise prefer at least
<MIN_SELECT> skills when that many good fits exist in the list.

## Problem summary

<PROBLEM_SUMMARY>

## Activated graph context (top-Q similarity and linked M-nodes)
Structured evidence from the skill graph: similar problems (Q) and their logic pipelines
(M).
Align skill choices with this retrieval context.

<ACTIVATED_GRAPH_CONTEXT>
```

```
## Candidate skills (id, title, description, ϕ)

<CANDIDATE_SKILLS_BLOCK>

## Valid skill ids (copy exactly, one per line)
Your JSON strings must match one of these lines verbatim (including hyphens):

<VALID_SKILL_IDS_BLOCK>

## Output format (strict)
Return only one JSON object and nothing else -- no markdown fences, no commentary, no
reasoning.
Required keys:
  - "selected_skill_ids": string[] -- ids copied verbatim from the candidate list.
  - "subproblem_dag": { "nodes": [...], "edges": [...] }
    - nodes: each item is { "id": string (e.g. P1), "description": string } --
abstract algorithmic sub-goal, not story text.
    - edges: each item is { "from_id": string, "to_id": string } -- dependency
(from_id must be solved before to_id). The graph must be a DAG (no cycles).
Always return a meaningful subproblem_dag with at least one node, including when
selected_skill_ids is [].
```

Solver — solver_skill_selection (user, minimal retry)

```
Output only one JSON object. No markdown fences, no reasoning text.
Include "selected_skill_ids" and "subproblem_dag" { "nodes": [...], "edges": [...] }
(DAG, acyclic).
Always generate a meaningful DAG for the current problem with at least one node.
Choose <MIN_SELECT>-<MAX_SELECT> ids from the list below (or [] if none apply).

## Task (short)

<PROBLEM_SUMMARY_SHORT>

## Candidates (copy ids exactly from the bottom list)

<CANDIDATE_LINES_MINIMAL>

## Valid skill ids (verbatim, one per line)

<VALID_SKILL_IDS_BLOCK>

Example:
{
  "selected_skill_ids":["xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"],
  "subproblem_dag":{
    "nodes":[{"id":"P1","description":"Parse input and build data structure"}],
    "edges":[]
  }
}
```

F.3 SOLVER: CODE GENERATION AND PATCH REPAIR

The Solver alternates between full-program generation (initial draft) and search-and-replace patch repair (subsequent iterations, up to $N_{\max} = 8$). A separate `patch_decision` prompt routes between the two modes based on whether the failure looks localized or systemic. The generation prompt embeds an explicit *resource audit* requirement so that the model rejects sketches that would be infeasible at the stated constraint bounds.

Solver — generate_code.initial

```
You are solving a competitive programming problem.  Apply the following three skills in
order before writing any code.

## Problem Context

Problem:
<PROBLEM_DESC>

<ABSTRACT_TAGS_LEVEL2_BLOCK>

Proposed algorithm:
<ALGORITHM>

Implementation steps:
<STEPS>

<CONSTRAINTS_BLOCK>

<PUBLIC_BLOCK>

<GEN_BLOCK>

<SOLVER_GRAPH_BLOCK>

<SELF_VALIDATION_BLOCK>

<MEMORY_ADVICE>

--

## Skill 1:  Brainstorming -- Design Before Implementation

<HARD-GATE>
Do NOT write any C++ code until you have completed the design steps below.
This applies regardless of how simple the problem appears.
</HARD-GATE>

Produce a concise algorithm design (target:  under 400 words total) covering:

1.  Core approach:  In 1-3 sentences, name the EXACT technique and data structures.  Be
specific:  not "tree DP" but "rerooting DP storing subtree expected values"; not "greedy"
but "leftist-heap merge tracking incremental cost per frontier".  If you cannot name the
specific data structure and key operation in one sentence, you have not converged -- try
a different angle before proceeding.
2.  Complexity + TLE check:  State time O(?)  and space O(?).  Estimate operations
concretely.  Apply the 10^8 simple ops/sec rule for C++:  if `operations / 10^8 >
time_limit_seconds`, the approach is TLE -- redesign.  Do not rationalize that constants
will save you.
3.  Correctness invariant:  The one-sentence key insight that makes this approach
correct.

If the proposed algorithm in "Implementation steps" above would lead to an unsafe or
hard-to-implement solution, you are free and encouraged to choose a better approach
instead.

--

## Skill 2:  Test-Driven Development -- Verify Algorithm Before Coding (Red Phase)

Iron Law:  NO CODE WITHOUT A VERIFIED ALGORITHM.

Trace your chosen algorithm on each public sample input:
- Show key intermediate values and the final output your algorithm produces.
- Confirm it matches the expected output.

If any sample trace fails:  STOP. Redesign the algorithm.  Do not write code for a
failing approach.

Only proceed to implementation after all samples pass the mental trace.

--

## Skill 3:  Writing Plans -- Implementation Plan Before Coding

Before writing any code, decompose the implementation into bite-sized steps.  For this
problem, list:
```

1. **Data structures to define** (e.g., struct Node, priority_queue type, etc.)
2. **Main function steps** in order (e.g., "read input -> build trie -> run DP -> output")
3. **Any tricky sub-routine** that needs careful implementation (e.g., "merge two heaps", "BFS with state compression")

This plan is a 5-10 line outline, not prose. Write it down before touching code.

--

Skill 4: Implementation (Green Phase) + Verification Before Completion

Now implement following your plan. Requirements:
- Use standard C++ (C++17)
- Do NOT use non-standard headers like #include <bits/stdc++.h>
- Include all necessary standard headers explicitly
- Implement fast I/O
- Every major data structure allocation must be defensible given the maximum input bounds -- state its size explicitly in a comment if non-obvious
- Prefer streaming, rolling-state, or sparse representations when full dense tables would exceed memory
- If the "Implementation steps" sketch above would lead to an unsafe implementation, adapt -- do not follow it literally

After writing the code, before outputting it, verify:
- **Time (strict):** Count the dominant loop's iterations for maximum input. Use the **10^8 ops/sec rule**: `iterations / 10^8 <= time_limit_seconds`. If you wrote O(N * Q) and N=Q=10^5, that is 10^10 ops = 100s -- TLE. Redesign if this check fails. Do NOT assume constants will save you.
- **Memory:** For each major allocation, is the total within the memory limit?
- **Edge cases:** Does the code handle n=1 and n=max?

If any check fails: fix the code first.

--

Output your full response in this exact structure:

Design
[Algorithm design, sample traces, and implementation plan from Skills 1-3 above]

Solution
```cpp
[Complete C++17 solution]
```

Solver — generate_code.think

You are analyzing a competitive programming problem. Your task is ONLY to determine the best algorithm -- do NOT write any C++ code in this response.

Discipline (non-negotiable)

TDD-style rule for competitive programming: Before declaring an algorithm correct, you must demonstrate it on actual computation, not mental simulation. The Iron Law:

> NO `VERDICT: PROCEED` WITHOUT A `<run_cpp>` OR `<run_python>` BLOCK THAT EXECUTED AND PRODUCED THE EXPECTED OUTPUT IN THIS TURN.

"Mental traces" are forbidden as the sole evidence. If your reasoning concludes "and clearly the answer is X", you must run code that prints X.

Brainstorming rule (Codex `brainstorming` skill): For any non-trivial problem, list 2-3 candidate algorithms with their complexity classes BEFORE picking one. State the trade-off (time vs. space vs. implementation difficulty) and pick the one that strictly fits the constraints. If only one approach fits, say so explicitly.

Verification rule (Codex `verification-before-completion` skill): Evidence before claims. If you haven't run a verification command in this message, you cannot claim it passes. "It should work" is not evidence -- `stdout matches expected on tests T1, T2, T3` is evidence.

Anti-pattern: confidently wrong math. A formula that "looks right" can hide an algebraic step that breaks under specific inputs. The cure is brute force comparison on small random inputs, not more squinting at the formula. Two-parameter parameterizations of n-parameter structures are a classic trap; if your derivation reduced parameter count, write a brute force enumerator and confirm the count agrees on n=2, n=3, n=4 before trusting your reduction.

Anti-pattern: TLE that "should fit". A complexity that "should run in 0.5s" often runs in 8s due to constant factors (cache misses, branch mispredictions, integer division). When in doubt, compile a draft via `<run_cpp>` on a worst-case-shaped input near the constraint bound. If it doesn't finish in 5s in the sandbox, it won't finish on the judge.

--

Problem Context

Problem:
<PROBLEM_DESC>

<ABSTRACT_TAGS_LEVEL2_BLOCK>

Proposed algorithm hint (may be incomplete or wrong -- use your own judgment):
<ALGORITHM>

<CONSTRAINTS_BLOCK>

<PUBLIC_BLOCK>

<ORACLE_STATUS_BLOCK>

<HARD_GATE_BLOCK>

<MEMORY_ADVICE>

--

Your Task: Algorithm Convergence (NO CODE)

You can verify your reasoning by embedding tool blocks in your reply. The harness executes them and feeds the output back to you:

- `<run_python>...</run_python>` -- Python sandbox (`sys`, `math`, `itertools`, `collections`, `bisect`, `heapq`, `random`); 5s CPU. Use it to write a tiny brute-force solver and check small cases.
- `<run_cpp>...</run_cpp>` -- Real C++ compilation + execution (full standard library; the same `g++ -O2 -std=c++17` compiler the judge uses; 5s CPU). To pipe stdin, wrap it as:

  ```
  <run_cpp>
  INPUT_BEGIN
  3
  1 2 3
  INPUT_END
  #include <bits/stdc++.h>
  using namespace std;
  int main(){ ... }
  </run_cpp>
  ```

  The C++ tool is the BEST way to detect TLE early: pick a worst-case-shaped input near the constraint bound and run a candidate implementation. If it doesn't finish in 5s in the sandbox, it won't finish on the judge.

Strongly recommended workflow: write a Python brute force (small inputs, exponential is OK), write a candidate C++ implementation, run BOTH on the same small random inputs in tool blocks, and only declare PROCEED when their outputs agree on at least 5 random tests AND your C++ runs within budget on a worst-case-shaped input.

Think through the problem and converge on the best algorithm. Produce:

1. **Algorithm** (1-3 sentences, maximally specific): Name the exact data structure and core operation -- e.g., "leftist-heap merge tracking incremental cost per trie frontier node", not just "greedy" or "DP". If you are not certain, name the best candidate and explain why it fits.

2. **Complexity + TLE check:** State O(time) and O(space). Apply the 10^8 ops/sec rule: `ops / 10^8 <= time_limit_seconds`. If this check fails, redesign before continuing.

3. **Correctness invariant:** One sentence stating why this approach produces the correct answer.

4. **Sample trace:** Trace your algorithm on the first public sample. Show key intermediate values and confirm the output matches.

5. **Implementation sketch** (3-5 bullets): The key data structures to define and the

```
main function's control flow -- specific enough that no further design decisions are
needed when writing code.

6.  Verdict -- end your response with EXACTLY ONE of these two lines, on its own line:
  - `VERDICT: PROCEED` -- if your algorithm clearly satisfies the time/memory limits and
your derivation is sound.
  - `VERDICT: REDESIGN_NEEDED -- <one-sentence reason>` -- if your complexity analysis
exceeds the time limit, your formula has a step you cannot verify, or you discovered an
inconsistency.  Be honest:  choosing REDESIGN_NEEDED is preferred over forcing a flawed
solution forward.

Stop here.  Do NOT output any C++ code.  The next step will ask for the implementation.
```

Solver — generate_code.code_only

```
Based on the algorithm design you provided above, implement the complete C++ solution.

## Problem Context (for reference)

Problem:
<PROBLEM_DESC>

<CONSTRAINTS_BLOCK>

<PUBLIC_BLOCK>

<GEN_BLOCK>

<SELF_VALIDATION_BLOCK>

<MEMORY_ADVICE>

## Implementation Requirements

Write the complete C++ implementation.  Requirements:
- Use standard C++ (C++17)
- Do NOT use non-standard headers like #include <bits/stdc++.h>
- Include all necessary standard headers explicitly
- Implement fast I/O
- Every major data structure allocation must be defensible given the maximum input bounds
-- state its size in a comment if non-obvious
- Prefer streaming, rolling-state, or sparse representations when full dense tables would
exceed memory
- If your algorithm sketch above would lead to an unsafe implementation, adapt -- follow
your own analysis, not the sketch literally

After writing the code, verify:
- Time (strict):  Count dominant-loop iterations at max input.  `iterations / 10^8 <=
time_limit_seconds`.  If not, redesign.
- Memory:  Each major allocation within memory limit?
- Edge cases:  n=1 and n=max handled?

Output your full response in this exact structure:

### Verification
[Time/memory/edge-case checks]

### Solution
"'cpp
[Complete C++17 solution]
"'
```

Solver — generate_code.patch_decision

```
You are choosing the repair mode for a failing C++ solution.

Problem:
<PROBLEM_DESC>

<ABSTRACT_TAGS_LEVEL2_BLOCK>

Algorithm:
<ALGORITHM>

Implementation steps:
```

```
<STEPS>

## Current Code:
"'cpp
<PREV_CODE>
"'

## Objective Failure Evidence:
<FAILURES_BLOCK>

<AGGREGATE_FAILURES_BLOCK>

<DIAGNOSTIC_BLOCK>

## Auxiliary References (use only as secondary hints):
<FEEDBACK_TEXT>

<FIXES_BLOCK>

<MEMORY_ADVICE>

## Your Task
Based on the objective evidence above:
1.  First judge whether the failures indicate a localized bug or a systemic/global flaw.
2.  Then identify the most likely error type.
3.  Choose the better repair mode:
  - `patch` if the overall approach is still sound and the issue is localized.
  - `full_regen` if the overall approach is likely wrong or patching would keep drifting
further.

Rules:
- Base the decision primarily on the objective evidence and current code.
- Treat the auxiliary references as secondary hints only.
- Do not guess beyond the provided evidence.
- Return ONLY valid JSON with this schema:
  {"mode":"patch|full_regen","confidence":"low|medium|high","reason":"one short sentence"}
```

Solver — generate_code.python_oracle

```
Write a Python brute-force oracle and a small-input generator.  We will:
1.  Run your brute force on each public sample input and verify it produces the expected
sample output.  If it doesn't, your understanding of the problem is wrong (or your brute
force is buggy) and you'll be asked to revise.
2.  Use your input generator + brute force as a reference to cross-validate the C++
solution we will write next on N random small inputs.

Both scripts must:
- Be syntactically valid Python.
- Use ONLY: `sys`, `math`, `itertools`, `collections`, `bisect`, `heapq`, `random` (no
other imports).
- Read input from stdin (`sys.stdin.read()`) and write to stdout (`print` or
`sys.stdout.write`).
- Run within 5 seconds CPU per invocation.

The brute force should be maximally simple -- exponential time is fine because we only
run it on small inputs.  Resist the temptation to optimize; correctness for small N
matters far more than speed.

The input generator must print exactly ONE valid input matching the problem's format,
with small parameters (e.g.  n <= 6, values <= 10) so the brute force finishes quickly.
Successive calls should produce different inputs.

Problem:
<PROBLEM_DESC>

<CONSTRAINTS_BLOCK>

<PUBLIC_BLOCK>

<FEEDBACK_BLOCK>

Output ONLY valid JSON with this schema (no markdown fence, no extra prose):
  {
    "brute_force":  "<full Python script>",
    "input_generator":  "<full Python script>"
  }
```

Solver — generate_code.patch (SEARCH/REPLACE format)

```
You are debugging a C++ solution that is FAILING tests.  Your task is to generate
SEARCH/REPLACE edits to fix the bugs.

Problem:
<PROBLEM_DESC>

<ABSTRACT_TAGS_LEVEL2_BLOCK>

Algorithm:
<ALGORITHM>

Implementation steps:
<STEPS>

## Current Code (BUGGY):
"'cpp
<PREV_CODE>
"'

## Test Failures:
<FAILURES_BLOCK>

<AGGREGATE_FAILURES_BLOCK>

<FEEDBACK_TEXT>

<FIXES_BLOCK>

<MEMORY_ADVICE>

## Your Task:
Apply systematic debugging before proposing any fix.

## Skill:  Systematic Debugging -- Root Cause Before Fix

Iron Law:  NO FIXES WITHOUT ROOT CAUSE INVESTIGATION FIRST.

Phase 1 -- Root cause investigation:
1.  Read the failure evidence carefully.  What does the wrong output tell you about where
the code diverges?
2.  Trace the failing test case through the code step by step.  Track key variables at
each step.
3.  Identify the exact line or logic block where the value first goes wrong.
4.  Name the root cause category:  overflow / off-by-one / wrong formula / missing edge
case / wrong data structure / TLE / MLE / other.

Phase 2 -- Fix hypothesis:
- State one specific fix hypothesis that addresses the root cause.
- If the root cause is a global algorithmic flaw (not a local bug), this should be
'full_regen' territory -- say so rather than patching.
- Re-check time and space complexity after the proposed fix.

Only after completing both phases, generate the SEARCH/REPLACE edits.

Every *SEARCH/REPLACE* edit must use this EXACT format:
«««< SEARCH
<exact contiguous code snippet from the current code>
=======
<replacement code with the fix>
»»»> REPLACE

CRITICAL RULES:
1.  The SEARCH block must match the current code EXACTLY (including whitespace,
indentation)
2.  The SEARCH block must appear EXACTLY ONCE in the code
3.  You can have multiple SEARCH/REPLACE blocks to fix multiple issues
4.  Preserve proper indentation in the REPLACE block
5.  Make minimal, surgical changes - only fix what's broken
6.  Re-check BOTH time and space complexity before proposing edits
7.  Replace unsafe data structures if the current implementation appears to allocate
memory proportional to a dangerous product of input dimensions
8.  Do not preserve an existing approach just because it matches the plan if it is not
implementable within the stated limits

Example:
«««< SEARCH
  for (int i = 1; i <= n; i++) {
```

```
    sum += arr[i];
  }
=======
  for (int i = 0; i < n; i++) {
    sum += arr[i];
  }
»»»> REPLACE

Generate the SEARCH/REPLACE edits now:
```

Solver — generate_code.regenerate

```
You are repairing a failing C++ solution.  Apply the same design-first principles before
rewriting.

Problem:
<PROBLEM_DESC>

<ABSTRACT_TAGS_LEVEL2_BLOCK>

Proposed algorithm:
<ALGORITHM>

Implementation steps:
<STEPS>

## Previous Code (FAILING):
“‘cpp
<PREV_CODE>
“‘

<CONSTRAINTS_BLOCK>

<PUBLIC_BLOCK>

<GEN_BLOCK>

## Test Failures:
<FAILURES_BLOCK>

<FEEDBACK_TEXT>

<FIXES_BLOCK>

<MEMORY_ADVICE>

--

## Skill 1:  Diagnose Before Rewriting

<HARD-GATE>
Do NOT rewrite code until you have diagnosed WHY the previous attempt failed and decided
on a better approach.
</HARD-GATE>

1.  Root cause:  What specifically caused the failures?  (wrong algorithm, wrong formula,
wrong data structure, overflow, memory, TLE?)
2.  New approach:  If the algorithm itself is wrong, state the new correct approach with
the same specificity required in initial generation.  If only implementation was wrong,
state what changes are needed.
3.  Complexity recheck:  Verify time O(?)  and space O(?)  for the new approach fit the
constraints.

--

## Skill 2:  Sample Verification (Red Phase)

Before rewriting, trace the new approach on the failing sample cases to confirm it would
produce the correct output.

--

## Skill 3:  Rewrite (Green Phase)

Requirements:
- Return ONLY the complete C++17 source code in a fenced block (no SEARCH/REPLACE).
- Keep valid parts of previous code when possible, but rewrite structure if the approach
```

```
changed.
- Fix the listed failures and avoid regressions on passing cases.
- Every major allocation must be sized within bounds; state sizes in comments if
non-obvious.

Output your full response in this structure:

### Diagnosis and New Approach
[Root cause analysis and new algorithm design from Skills 1 and 2]

### Solution
"'cpp
[Complete C++17 solution]
"'
```

Solver — generate_code.canonical_problem_block

```
Objective: <OBJECTIVE>
Inputs: <INPUTS_JSON>
Outputs: <OUTPUTS_JSON>
Constraints: <CONSTRAINTS_JSON>
Required Properties: <REQUIRED_PROPERTIES>
```

Solver — generate_code.self_validation_header

```
Self-validation failed: <FAIL_COUNT> issues in <TOTAL_RUN>/<TOTAL_VERIFY> cases tested:
```

Solver — generate_code.self_validation_footer

```
Please fix these issues.
```

F.4 SOLVER: FAILURE ANALYSIS

When tests fail, the Solver invokes a structured failure-analysis prompt that forces the LLM to step-trace the simplest failing case before proposing fixes. The categorical `error_pattern` feeds back into the bandit's failure-type feature key (`FAIL:TIMEOUT`, `FAIL:OFF_BY_ONE`, etc., see Appendix B).

Solver — analyze_feedback.compilation

```
The following C++ code has compilation errors:

Code:
"'cpp
<CODE>
"'

Compilation Errors:
<ERROR_TEXT>

Provide:
1. Root cause of the errors
2. Specific fixes needed
3. Corrected code snippets

Be concise and actionable.
```

Solver — analyze_feedback.test_failure

```
You are a competitive programming debugging expert. Analyze the following failures and
provide CONCRETE fixes.

## Problem Description
<PROBLEM_DESC>

## Selected Approach
Algorithm: <ALGORITHM>
Steps:
```

```
<STEPS_TEXT>

## Current Status
Iteration: <ITERATION>
Pass Rate: <PASS_RATE>
Total Failed: <FAILED_COUNT>
Error Pattern: <ERROR_PATTERN>

## Current Code
```cpp
<CODE>
```

Representative Failures (most important cases to fix)
<FAILURES_TEXT>
<DIAGNOSTIC_SECTION>
Your Task
1. Pick the SIMPLEST failure case above. Trace the code execution step-by-step with
that input. Track key variables.
2. Identify WHERE and WHY the code produces wrong output.
3. Determine the root cause category: overflow, off-by-one, wrong formula, missing edge
case, TLE, etc.
4. Provide SPECIFIC code-level fixes (not vague suggestions).

Return ONLY valid JSON (no markdown, no explanation outside JSON):
{
  "analysis": "<detailed step-by-step trace showing where the bug is>",
  "root_cause": "<one-line root cause>",
  "error_pattern": "<category: overflow/off-by-one/
    wrong-formula/missing-edge-case/tle/other>",
  "suggested_fixes": [
    "<specific fix 1, e.g. 'Change line X: use long long instead of int'>",
    "<specific fix 2, e.g. 'Add special case handling when n==1'>"
  ]
}
```

Solver — analyze_feedback.hack_failure

```
You are a debugging expert. The solution passed all basic tests but FAILED adversarial
hack tests.

Problem:
<PROBLEM_DESC>

Current Algorithm:
<ALGORITHM>

Implementation Steps:
<STEPS_JSON>

Code:
```cpp
<CODE>
```

HACK FAILURES (The code logic is likely correct for simple cases but fails edge cases):
<HACK_FAILURES_TEXT>

Task:
1. Analyze why the code fails these specific hack cases.
2. Identify the root cause (e.g. overflow, edge case, logic hole).
3. Provide a fixed C++ solution.

Return ONLY JSON:
{
  "analysis": "<analysis of hack failures>",
  "suggested_fixes": ["<fix 1>", "<fix 2>"]
}
```

F.5 ORACLE: CERTIFIED TEST GENERATION

The Oracle prompt set has four sub-prompts (generator, validator, checker, solver) corresponding to the four artifacts that compose a certified test suite (Section 3.4). The generator and validator both use `testlib` as a strict scaffolding contract; the checker is asked to *independently verify* the candidate

output rather than compare against a reference, with two distinct skeletons depending on whether the problem has a unique answer or admits multiple valid answers.

Oracle — generate_tests.generator

```
You are a generator agent.  Write a C++17 program that outputs exactly one valid test
case to stdout.

Hard requirements:
- Use testlib:  #include "testlib.h"
- Do NOT use non-standard headers like #include <bits/stdc++.h>
- Call registerGen(argc, argv, 1)
- Use rnd.next(...)  for randomness (no std::random, no srand/rand)
- Do not parse or set a random seed inside the program
- The harness will run your generator many times with different seeds and expects
multiple distinct valid outputs
- Do NOT hardcode one fixed test case or ignore rnd.next(...)
- Different seeds should usually lead to meaningfully different inputs, not the same
bytes every time
- Prefer generating from multiple structural families instead of a single canned example
- Do not print any extra text
- Prefer semantically targeted edge cases over generic large random data.
- When exact certification is important, favor modest-size cases that still expose tricky
logic.

Minimal skeleton (illustrative only):
#include "testlib.h"
int main(int argc, char* argv[]) {
  registerGen(argc, argv, 1);
  return 0;
}

The program must produce ONE valid input instance that satisfies all constraints.

Problem Description:
<PROBLEM_DESC>

Constraints:
<CONSTRAINTS>

Public Tests:
<PUBLIC_TESTS>
<FEEDBACK_BLOCK>
<ADVICE_BLOCK>
Return ONLY a valid JSON object.  No other text, no markdown, no code fences.
CRITICAL JSON RULE: All newlines inside string values MUST be escaped as \n.  Tabs as \t.
Backslashes as \\.  Do NOT include literal newline characters inside any JSON string
value -- this will break JSON parsing.
Schema:
{"generator_cpp":  "#include ...\\nint main(){...}\\n"}
```

Oracle — generate_tests.validator

```
You are a validator agent.  Write a C++17 program that reads one test case from stdin and
validates it.

Hard requirements:
- Use testlib:  #include "testlib.h"
- Do NOT use non-standard headers like #include <bits/stdc++.h>
- Call registerValidation(argc, argv)
- Validate the INPUT ONLY. Do NOT read any answer token, candidate output, or expected
output.
- Stop immediately after consuming the valid input; then call inf.readEof()
- Use only these testlib input APIs unless strictly necessary:  inf.readInt,
inf.readLong,
  inf.readToken, inf.readSpace, inf.readEoln, inf.readEof, ensuref
- Do NOT use unsupported helpers such as peekChar or speculative stream-inspection APIs
- Use ensuref(...)  to report specific constraint violations
- Include #include <unordered_set> if using unordered_set
- Exit 0 on success, non-zero on failure
- Do not print anything on success

Minimal skeleton (illustrative only):
#include "testlib.h"
#include <unordered_set>
int main(int argc, char* argv[]) {
  registerValidation(argc, argv);
```

```
  int n = inf.readInt();
  inf.readSpace();
  int m = inf.readInt();
  inf.readEoln();
  ensuref(n >= 1, "n must be >= 1");
  std::string s = inf.readToken();
  inf.readEof();
  return 0;
}

Problem Description:
<PROBLEM_DESC>

Constraints:
<CONSTRAINTS>

Public Tests:
<PUBLIC_TESTS>
<FEEDBACK_BLOCK>
Return ONLY a valid JSON object.  No other text, no markdown, no code fences.
CRITICAL JSON RULE: All newlines inside string values MUST be escaped as \n.  Tabs as \t.
Backslashes as \\.  Do NOT include literal newline characters inside any JSON string
value -- this will break JSON parsing.
Schema:
{"validator_cpp":  "#include ...\\nint main(){...}\\n"}
```

Oracle — generate_tests.checker

```
You are a checker/verifier agent.  Write a C++17 program that independently verifies
whether a candidate output is correct for a given input -- WITHOUT needing any reference
answer.

CRITICAL DESIGN PRINCIPLE:
Verifying a solution is far easier than computing it.  Your checker must independently
determine correctness by re-deriving the answer or checking invariants, NOT by comparing
against a reference.  DO NOT read from the ans stream at all.

Approach (choose one based on the problem):
A) For problems with a unique answer:  Compute the correct answer yourself using a simple
  brute-force algorithm (even O(n^3) or O(n^4) is fine -- checkers run on small data),
  then compare against the candidate output.
B) For problems with multiple valid answers:  Read the candidate output and verify it
  satisfies all problem constraints (e.g., is it a valid permutation?  Does the graph
  satisfy the required property?  Is the value optimal?).

Hard requirements:
- Use testlib:  #include "testlib.h"
- Do NOT use non-standard headers like #include <bits/stdc++.h>
- Call registerTestlibCmd(argc, argv)
- Read input via inf (the test input)
- Read candidate output via ouf (the output to verify)
- DO NOT read from ans -- your checker must work without any reference answer
- Use quitf(_ok, "...") if the candidate output is correct
- Use quitf(_wa, "...") with a specific error message if incorrect
- Implement your own verification logic inside the checker
- Do NOT intentionally reject large valid inputs just because a tiny brute-force checker
would be slow
- Never introduce empty objects unless the problem statement explicitly defines them

Minimal skeleton (approach A -- unique answer):
#include "testlib.h"
#include <vector>
int main(int argc, char* argv[]) {
  registerTestlibCmd(argc, argv);
  int n = inf.readInt();
  int expected = brute_force_solve(n);
  int got = ouf.readInt();
  if (got != expected) quitf(_wa, "expected %d, got %d", expected, got);
  quitf(_ok, "ok");
}

Minimal skeleton (approach B -- multiple valid answers):
#include "testlib.h"
#include <vector>
int main(int argc, char* argv[]) {
  registerTestlibCmd(argc, argv);
  int n = inf.readInt();
```

```
  int answer = ouf.readInt();
  if (!is_valid(answer)) quitf(_wa, "invalid answer");
  quitf(_ok, "ok");
}

Problem Description:
<PROBLEM_DESC>

Constraints:
<CONSTRAINTS>

Public Tests:
<PUBLIC_TESTS>
<CHECKER_ADVICE_BLOCK>
<FEEDBACK_BLOCK>
Return ONLY a valid JSON object.  No other text, no markdown, no code fences.
CRITICAL JSON RULE: All newlines inside string values MUST be escaped as \n.  Tabs as \t.
Backslashes as \\.  Do NOT include literal newline characters inside any JSON string
value -- this will break JSON parsing.
Schema:
{"checker_cpp":  "#include ...\\nint main(){...}\\n"}
```

Oracle — generate_tests.solver

```
You are an independent reference-solution author.  Write a COMPLETE, COMPILABLE C++17
program that solves the following problem.

CRITICAL REQUIREMENTS:
1.  Your code MUST be a complete standalone program with #include, main(), cin/cout.
2.  Read input from stdin, write output to stdout, matching the exact I/O format shown in
the public tests.
3.  Correctness is the first priority.  Prefer a semantically exact small-scale reference
solver over a brittle formula.
4.  The certification inputs are allowed to be modest; do NOT sacrifice correctness for
aggressive large-constraint optimization.
5.  The program MUST compile with:  g++ -std=c++17 -O2
6.  Keep the solver independent and trustworthy as a reference oracle.

Attempt guidance:
<STAGE_GUIDANCE>
<SOLVER_ADVICE_BLOCK>

Problem Description:
<PROBLEM_DESC>

Constraints:
<CONSTRAINTS>

Public Tests (your program MUST produce the exact expected output for these):
<PUBLIC_TESTS_BLOCK>

Algorithmic Strategy Reference (use for inspiration, do NOT copy verbatim):
<TEMPLATES_JSON>

<FEEDBACK_BLOCK>
Return ONLY a JSON object.  No markdown, no explanation.
Schema:
{
  "selected_family_id":  "<family_id from the strategy reference>",
  "template_name":  "<name of the strategy you are using>",
  "solver_cpp":  "<complete C++17 source code>"
}
```

Oracle — generate_tests.solver_stage (attempts 1 / 2 / 3+)

```
[Attempt 1] Write an independent reference solution.  Prefer a straightforward
polynomial-time algorithm, direct simulation, or simple derivation.  Avoid factorial
or exponential search unless the certification inputs are provably tiny.  Prefer simple
reference logic over contest-specific tricks.

[Attempt 2] The previous solver had correctness, runtime, or execution issues.  Keep
correctness first, but the solver must run within the certification limits.  Do not use
factorial or exponential search.  Upgrade to a more scalable but still simple reference
algorithm if needed.
```

```
[Attempt 3+] Write a robust reference solution that still remains easy to trust.  It
must run within the certification limits and address the observed failure mode.  Do not
use factorial or exponential search.  If necessary, add sparse debug prints to stderr
using std::cerr « "TRACE: " « vars « std::endl;.  Never print inside tight loops unless
sampled.
```

The Oracle also issues stage-conditioned guidance (attempt 1, attempt 2, attempt 3+) that biases the LLM toward progressively more scalable but still simple algorithms after each failed certification round; judge scores and failure penalties appear in Appendix C.

F.6 HACKER: CODE ANALYST AND ADVERSARIAL GENERATORS

The Hacker is implemented as a two-stage cascade. First, a *Code Analyst* controller inspects the candidate code, optionally calls sandboxed tools (`run_python`, `run_cpp`) to verify hypotheses, and emits a vulnerability report. Second, the report routes to one of three specialized generator prompts (*semantic*, *stress*, *anti_hash*). Each generator has its own *checklist* and *patch* prompts for SEARCH/REPLACE repair when its output is rejected by the validator.

Hacker — code_analyst.system

```
You are the Code Analyst controller for an adversarial hacker workflow.
Return ONLY valid JSON.
Do not include prose before or after the JSON.
If you need a tool, output a tool-call JSON object only.
If you are ready to conclude, output a final-report JSON object only.
```

Hacker — code_analyst.main

```
You are the Code Analyst, the strategy planner for an adversarial Hacker System.
Your goal is to find bugs, logic flaws, or vulnerabilities (WA, TLE, RE, MLE) in the
provided C++ target code.

PROBLEM DESCRIPTION:
<PROBLEM_DESC>

CONSTRAINTS:
<CONSTRAINTS_JSON>

TARGET SOLUTION CODE (May contain bugs):
"'cpp
<TARGET_CODE>
"'
<ADVICE_SECTION>

AVAILABLE TOOLS:
You can verify your hypothesis by writing short probe codes.  Do not guess blindly if you
can test it!
Tool 1:  'run_python'
Use this to perform precise mathematical calculations (e.g., combinations, large numbers)
to check constraints and overflows.
Inputs:  {"tool":  "run_python", "parameters":  {"script_code":  "..."}}

Tool 2:  'run_cpp'
Use this to compile and execute small C++ snippets to test specific runtime behaviors or
replicate target logic.
Inputs:  {"tool":  "run_cpp", "parameters":  {"cpp_code":  "..."}}

RESPONSE FORMAT:
You must return a valid JSON object.  You have two choices:
Choice A: Call a tool to gather information.
{
  "tool":  "run_python",
  "parameters":  {"script_code":  "import math\nprint(math.comb(100, 50))"}
}

Choice B: Submit the Final Vulnerability Report (if you are confident).
{
  "bug_class":  "overflow|hash_collision|index_oob|tle|logic_branch|unknown",
  "confidence":  "high|medium|low",
  "evidence":  ["e.g.  math.comb(100,50) overflows 2^63"],
  "suggested_route":  "anti_hash|semantic|stress",
```

```
  "input_hypothesis": ["large_n", "degenerate_tree", "collision_string"]
}
Note on routes: `anti_hash` if polynomial hash is used. `semantic` for most logical
bugs. `stress` ONLY if entirely clueless.

HISTORY OF YOUR ACTIONS & RESULTS:
<HISTORY_TEXT>

Analyze the code, call tools if needed to verify, and output valid JSON.
```

Hacker — code_analyst.json_repair

```
Your previous reply was not valid JSON for the Code Analyst protocol.

Rewrite the same intent as ONE valid JSON object.
Allowed outputs:
1. A tool call:
{
  "tool": "run_python|run_cpp",
  "parameters": {...}
}

2. A final report:
{
  "bug_class": "overflow|hash_collision|index_oob|tle|logic_branch|unknown",
  "confidence": "high|medium|low",
  "evidence": ["..."],
  "suggested_route": "anti_hash|semantic|stress",
  "input_hypothesis": ["..."]
}

Previous reply:
<PREVIOUS_RESPONSE>

PROBLEM DESCRIPTION:
<PROBLEM_DESC>

CONSTRAINTS:
<CONSTRAINTS_JSON>

TARGET SOLUTION CODE:
"'cpp
<TARGET_CODE>
"'
<ADVICE_SECTION>

Return valid JSON only. Do not add any explanation, markdown, or commentary.
```

Hacker — code_analyst.force_tool

```
Your current vulnerability report is too weak to submit as a final answer.

You must call exactly one tool before you can submit a final report.
Do not submit a final report yet.
Return ONLY a tool call JSON object in one of these forms:
{
  "tool": "run_python",
  "parameters": {"script_code": "..."}
}
or
{
  "tool": "run_cpp",
  "parameters": {"cpp_code": "..."}
}

PROBLEM DESCRIPTION:
<PROBLEM_DESC>

CONSTRAINTS:
<CONSTRAINTS_JSON>

TARGET SOLUTION CODE:
"'cpp
<TARGET_CODE>
"'
```

```
<ADVICE_SECTION>
PREVIOUS WEAK REPORT:
<WEAK_REPORT_JSON>

HISTORY OF ACTIONS:
<HISTORY_TEXT>

Call one tool that will increase confidence in the bug class or input hypothesis.
```

Hacker — semantic.generator

```
You are the Semantic Generator, a specialized C++ coder for an adversarial Hacker System.
Your job is to write a standalone C++ program that generates a single, highly-targeted
test case designed to trigger the specific vulnerability described by the Code Analyst.

PROBLEM DESCRIPTION:
<PROBLEM_DESC>
<ADVICE_SECTION>
CONSTRAINTS (The output of your C++ generator MUST satisfy ALL of these):
<CONSTRAINTS_TEXT>
<PREVIOUS_ISSUES_SECTION>
<PREVIOUS_INPUT_SECTION>

VULNERABILITY REPORT (from Code Analyst):
<REPORT_JSON>

INSTRUCTIONS FOR C++ GENERATOR:
1.  Write a complete, compilable C++17 program (`int main() {...}`).
2.  The program must print EXACTLY ONE valid test case to standard output (`std::cout`).
3.  VALIDITY-FIRST: the generated input MUST satisfy all format and validator constraints
before you try to make it adversarial.
4.  Focus on producing the input data structures matching the `input_hypothesis` in the
report.
5.  If the previous attempt was rejected, fix those exact issues before changing anything
else.
6.  DO NOT use uninitialized variables or undefined behavior in your generator.
7.  If you need randomness, you MAY use `<random>` (`std::mt19937`), but since this is
the Semantic Generator, deterministic construction of the edge case is preferred when
possible.

CRITICAL FORMATTING RULES:
1.  Return ONLY the C++ code.
2.  DO NOT wrap the code in markdown blocks (e.g., "`cpp ...  "`).
3.  The very first line should be `#include <...>` or similar valid C++.

Write the C++ generator code now:
```

Hacker — semantic.checklist

```
You are repairing a C++ Semantic Generator after a failed attack-input attempt.
Analyze the failure and produce a compact JSON checklist before patching the code.

PROBLEM DESCRIPTION:
<PROBLEM_DESC>

INPUT VALIDITY CONSTRAINTS:
<CONSTRAINTS_TEXT>
<ADVICE_SECTION>
LATEST FAILURE TYPE: <FAILURE_KIND>
LATEST FAILURE REASON:
<FAILURE_REASON>
<ISSUES_SECTION>
<INPUT_SECTION>
VULNERABILITY REPORT:
<REPORT_JSON>

PREVIOUS GENERATOR CODE:
<LAST_GENERATOR_CODE>

CHECKLIST RULES:
1.  VALIDITY BEFORE ATTACK: first restore validator-accepted input, then preserve
adversarial intent.
2.  `must_fix` must focus on the concrete compile/runtime/validator issue from the latest
failure.
3.  `do_not_regress` must list already-required legality properties that must remain
```

```
true.
4.  `attack_goal` must keep the attack aligned with the analyst report instead of
collapsing into trivial valid input.
5.  Be failure-type aware:
  - compile_failed -> prioritize syntax/API/build repairs
  - validator_rejected -> prioritize legality/format repairs
  - runtime_error -> prioritize execution safety
  - empty_output -> prioritize guaranteed emission of one valid case

Return ONLY valid JSON with exactly this schema:
{
  "must_fix":  ["..."],
  "do_not_regress":  ["..."],
  "attack_goal":  ["..."]
}
```

Hacker — semantic.patch

```
You are applying a minimal patch to a C++ Semantic Generator.
You must patch the existing generator with minimal SEARCH/REPLACE edits instead of
rewriting it from scratch.

PROBLEM DESCRIPTION:
<PROBLEM_DESC>

INPUT VALIDITY CONSTRAINTS:
<CONSTRAINTS_TEXT>
<ADVICE_SECTION>
LATEST FAILURE TYPE: <FAILURE_KIND>
LATEST FAILURE REASON:
<FAILURE_REASON>
<INPUT_SECTION>
VULNERABILITY REPORT:
<REPORT_JSON>

REPAIR CHECKLIST:
<CHECKLIST_JSON>

CURRENT GENERATOR CODE:
<LAST_GENERATOR_CODE>

PATCH RULES:
1.  VALIDITY-FIRST: fix the latest failure before any attack refinement.
2.  Preserve already-correct code whenever possible.
3.  Keep the attack goal from the checklist after legality is restored.
4.  Output only SEARCH/REPLACE blocks.
5.  Each SEARCH block must match the current generator code exactly once.
6.  Make the patch minimal and surgical.  Do NOT replace the whole file.

Required patch format:
<PATCH_FORMAT>
```

Hacker — stress.generator

```
You are the Stress Test Generator, a specialized C++ coder for an adversarial Hacker
System.
Your job is to write a standalone C++ program (`int main()`) that acts as a
high-throughput Fuzzer.

PROBLEM DESCRIPTION:
<PROBLEM_DESC>

CONSTRAINTS (The output of your C++ fuzzer MUST strictly satisfy these boundaries):
<CONSTRAINTS_TEXT>

INSTRUCTIONS FOR C++ FUZZER:
1.  Write a complete, compilable C++17 program.
2.  VALIDITY-FIRST: the generated input MUST pass the problem validator and explicitly
enforce all structural constraints in code.
3.  The program must print EXACTLY ONE valid test case to standard output, but this test
case should be as LARGE and COMPLEX as the constraints allow.
4.  You MUST use `<random>` and `std::mt19937_64` initialized with a random device or
fixed seed.
5.  Scale up the generation loop to approach the maximum `N`, `M`, or `K` allowed.
6.  Emphasize boundary values (e.g.  generating values alternating between min and max
```

```
allowed).
7.  Optimize the generator for speed using '\n' instead of 'std::endl' and fast I/O
  ('std::ios_base::sync_with_stdio(false);').

CRITICAL FORMATTING RULES:
1.  Return ONLY the C++ code.
2.  DO NOT wrap the code in markdown blocks (e.g., "'cpp ...  "').
3.  The very first line should be '#include <...>' or similar valid C++.

Write the C++ Stress Test Generator code now:
```

Hacker — stress.checklist

```
You are repairing a C++ Stress Test Generator after a failed attempt.
Analyze the failure and produce a compact JSON checklist before patching the code.

PROBLEM DESCRIPTION:
<PROBLEM_DESC>

INPUT VALIDITY CONSTRAINTS:
<CONSTRAINTS_TEXT>

LATEST FAILURE TYPE: <FAILURE_KIND>
LATEST FAILURE REASON:
<FAILURE_REASON>
<ISSUES_SECTION>
<INPUT_SECTION>
PREVIOUS GENERATOR CODE:
<LAST_GENERATOR_CODE>

CHECKLIST RULES:
1.  VALIDITY BEFORE ATTACK: first restore validator-accepted input, then keep the case
large and boundary-heavy.
2.  'must_fix' must target the concrete compile/runtime/validator failure from the latest
attempt.
3.  'do_not_regress' must preserve constraints already satisfied by the generator.
4.  'attack_goal' must keep the output large, high-throughput, and stress-oriented after
repairs.
5.  Be failure-type aware:
  - compile_failed -> prioritize syntax/API/build repairs
  - validator_rejected -> prioritize legality/format repairs
  - runtime_error -> prioritize execution safety
  - empty_output -> prioritize guaranteed emission of one valid case

Return ONLY valid JSON with exactly this schema:
{
  "must_fix":  ["..."],
  "do_not_regress":  ["..."],
  "attack_goal":  ["..."]
}
```

Hacker — stress.patch

```
You are applying a minimal patch to a C++ Stress Test Generator.
You must patch the existing generator with minimal SEARCH/REPLACE edits instead of
rewriting it from scratch.

PROBLEM DESCRIPTION:
<PROBLEM_DESC>

INPUT VALIDITY CONSTRAINTS:
<CONSTRAINTS_TEXT>

LATEST FAILURE TYPE: <FAILURE_KIND>
LATEST FAILURE REASON:
<FAILURE_REASON>
<INPUT_SECTION>
REPAIR CHECKLIST:
<CHECKLIST_JSON>

CURRENT GENERATOR CODE:
<LAST_GENERATOR_CODE>

PATCH RULES:
1.  VALIDITY-FIRST: fix the latest failure before increasing attack pressure.
```

```
2.  Preserve already-correct large-case generation logic when possible.
3.  Keep the generator large, randomized, and boundary-oriented after the repair.
4.  Output only SEARCH/REPLACE blocks.
5.  Each SEARCH block must match the current generator code exactly once.
6.  Make the patch minimal and surgical.  Do NOT replace the whole file.

Required patch format:
<PATCH_FORMAT>
```

Hacker — anti_hash.generator

```
You are the Anti-Hash Generator, the cryptography/math specialist for an adversarial
Hacker System.
Your job is to read the Code Analyst's report identifying a polynomial rolling hash
susceptibility, and write a standalone C++ program to generate colliding strings.

PROBLEM DESCRIPTION:
<PROBLEM_DESC>

CONSTRAINTS (Output MUST SATISFY ALL string/length constraints):
<CONSTRAINTS_TEXT>

VULNERABILITY REPORT (from Code Analyst):
<REPORT_JSON>

INSTRUCTIONS FOR C++ GENERATOR:
1.  Write a complete, compilable C++17 program (`int main()`).
2.  VALIDITY-FIRST: the generated collision input MUST pass the problem validator and
satisfy all structural constraints.
3.  Implement a collision derivation algorithm (e.g., Thue-Morse sequence, Birthday
attack, or specific moduli exploitation) matching the `input_hypothesis`.
4.  Output EXACTLY the string(s) needed to trigger the collision to standard out.
5.  Keep the target constraints in mind.  If max length is `N=10^5`, the collision
strings must not exceed this length.

CRITICAL FORMATTING RULES:
1.  Return ONLY the C++ code.
2.  DO NOT wrap the code in markdown blocks (e.g., "`cpp ...  "`).
3.  The very first line should be `#include <...>` or similar valid C++.

Write the C++ Collision Generator code now:
```

When a generator's output is rejected, a per-route *checklist* prompt produces a JSON repair plan (`must_fix`, `do_not_regress`, `attack_goal`); a per-route *patch* prompt then applies minimal SEARCH/REPLACE edits with the same format as the Solver patch prompt. The full reward shaping for the Hacker (severity weights, valid/break ratios, generator-failure penalty) is given in Appendix D.

G Experimental Configuration

This appendix collects the exact backbone, infrastructure, and budget settings used in all reported experiments.

G.1 Backbones and inference

We use GPT-5.4 (OpenAI, 2026), Claude Opus 4.6 (Anthropic, 2026), Qwen3.6-Plus (Qwen Team, 2026), DeepSeek V4 Pro (DeepSeek-AI, 2026), and Grok 4 (xAI, 2025). All five backbones are accessed through a unified Azure OpenAI–compatible gateway with AAD authentication. Within each row of the main comparison and each reported ablation subset, every method is driven by the same backbone deployment so that comparisons isolate the agent framework rather than the underlying model.

Backbone deployments and decoding

Models used:

GPT-5.4	`gpt-5.4` (default and per-role for code/generator/validator/checker/output/hacker)
Claude Opus 4.6	`claude-opus-4-6`
Qwen3.6-Plus	`qwen3.6-plus` via the unified OpenAI-compatible gateway
DeepSeek V4 Pro	`deepseek-v4-pro`
Grok 4	`grok-4`

Decoding: temperature $= 0.1$, max_tokens $= 65{,}536$ for all backbones and methods.
Embeddings: runtime retrieval uses `text-embedding-3-small` via the same gateway, with a 32,768-entry LRU cache and up to 5 HTTP retries; offline corpus deduplication uses `text-embedding-3-large` as described in Section 2.3; both fall back to local Sentence-Transformers when the gateway is unavailable.
API: OpenAI-compatible gateway; provider-specific endpoint and authentication details are released with the public code.

G.2 Pipeline budgets

The closed-loop pipeline (Section 3.1) runs under the fixed iteration budgets below; token costs of all methods are compared in Figure 4a.

Per-problem pipeline budgets

Solver inner loop: `max_iterations` $= 8$ patch/regenerate rounds before the candidate is declared failed.
Hacker outer loop: `max_hack_rounds` $= 3$ rounds; each round runs the full Code-Analyst $\rightarrow$ generator $\rightarrow$ validator cascade.
Oracle certification: $N_{\text{target}} = 200$ generated inputs per problem when a dataset-provided accepted solution is available (offline training only) and $N_{\text{target}} = 50$ otherwise, including all evaluation runs; acceptance gate $\rho \geq \tau$ with $\tau = 0.9$ (Eq. 3). On rejection the Oracle retries with a different solver family selected by the bandit.
Output token cap: 65,536 tokens per LLM call for all methods.

G.3 Knowledge-network defaults

Each backbone trains its own set of networks on the same cold-start corpus and training stream. Default settings for QMS and bandit memories:

Solver QMS network (config/solver_network.yaml)

`enabled: true`, `graph_dir: artifacts/solver_network/latest/graph`
Top-k similar Q nodes per query: 4
Skill candidate pool: top 20 by ϕ; sample $K = 5$ sequentially without replacement with temperature $T = 0.2$ in Eq. 2 (`skill_selection_temperature`), renormalizing after each draw.
QMS training: one sampled group per problem, retained throughout the Solver run; learning rate $\beta = 0.005$ in Eq. 2; Q–M edge weights initialized to 1.0; edge weights clipped below at 10^{-6}, with M–S weights renormalized within each M group after each update.
LLM selection: `min_llm_skills = 1`, `max_llm_skills = 5`; an empty selection is allowed when no candidate applies.
Augmentation block: skill text only by default; full templates injected only when `include_skill_templates_in_augmentation` is on.
Ensemble: enabled for all Solvita results in Tables 1–6. After test generation, 3 independent skill-plan + CodeGen/Hacker branches run in parallel; the run submits the branch with the highest pass rate on its internal tests (Oracle-certified and Hacker regression tests), breaking ties by branch order. Hidden benchmark tests are never consulted, and reported token counts include all branches.

Trainable bandit memory (config/trainable_memory.yaml)

`enabled: true`, `data_dir: artifacts/trainable_memory`
Solver advice: enabled at deployment; disabled during QMS training.
Per-namespace top-k injected as advice: `solve_top_k = 3`, `test_top_k = 3`, `hack_top_k = 3`, `oracle_top_k = 3`.
Bandit learning rate $\alpha = 0.01$; tag-overlap prior $+0.05$ per matching tag.
Items with average reward < -0.3 after ≥ 20 uses are auto-deprecated.
Persistence: SQLite-backed JSON store with atomic file-locking writes.

G.4 Datasets and benchmarks

Benchmarks and evaluation metrics

CodeContests (CC): the 165-problem DeepMind CodeContests test split.
APPS: 1,000 problems from the APPS competition test split.
AetherCode (AC): 400 problems from the AetherCode v1 2024 open split, combining novel algorithmic tasks with expert-validated test suites.
Codeforces (Fig. 5b): ten numbered rounds in the interval 1093–1104, totaling 79 distinct problems. For each Solvita backbone, each round is one serial session; separate Div. 1/Div. 2 pages are used only for score insertion.
Benchmark isolation: The filtered source pool and all downstream stores are built only from the APPS train split, the CodeContests train split, and the train split of the CodeContests+ `1x` configuration. Overlap with the APPS competition test split, the full CodeContests test split, and AetherCode is removed before training. Every evaluation run starts from a read-only copy of its configuration's frozen

checkpoint; bandit weights, memory items and their usage counters, running rewards, and deprecation flags, the QMS graph, and stored failure cases are not updated during evaluation (Table 17), so no information carries across problems or runs.
Filtered source pool: 12,017 problems retained after the four-stage pipeline in Section 2 and partitioned without overlap into 6,699 cold-start and 5,318 online-training problems.
Solve metric: Tables 1–6 report pass@3 for every method. Each method runs three times per problem; a problem counts as solved if any run succeeds. Single-pass uses one generation per run. Codeforces uses pass@1 and CodeElo rating estimates.

G.5 Baseline protocols

Every method in Table 1 uses the same backbone deployment, decoding settings, and output token cap as Solvita (Appendix G), and its submissions are judged by the same harness against the benchmark's hidden tests, using the benchmark's official checker where one is provided; neither is exposed to any method.

Baseline execution

Single-pass: one generation per run from the problem statement and public samples; the C++ program in the response is submitted.
Codex CLI and Claude Code: Codex CLI 0.142.0 and Claude Code 2.1.168, run headless with default tool and agent settings apart from decoding, with model calls routed through the same gateway as Solvita and no other network access; both can compile and run code locally, including on the public samples.
AlphaCodium and MapCoder: official implementations at the latest public commits (`eb7577d` and `c87e6d9`, respectively) with their default iteration budgets, configured to produce C++.
Submission: each run submits one C++ program; each method runs three times per problem and is scored by pass@3.

Table 14: Information available to each method on evaluation problems. Official artifacts from training splits are used only to build Solvita's checkpoint (Section 2).

Method	Statement and public samples	Hidden tests	Official solutions and checkers	Internet access	Cross-problem memory	Runs
Single-pass	✓	–	–	–	–	3
Codex CLI	✓	–	–	–	–	3
Claude Code	✓	–	–	–	–	3
AlphaCodium	✓	–	–	–	–	3
MapCoder	✓	–	–	–	–	3
Solvita	✓	–	–	–	frozen, read-only	3

G.6 Codeforces Rating Estimation Protocol

We use the rating estimator published with CodeElo (Quan et al., 2025), rather than the fixed CodeElo problem set. The result is therefore a CodeElo-style estimate, not an official Codeforces account rating.

Runs and outcomes. We replay ten eligible Codeforces rounds in the prespecified interval 1093–1104, excluding Rounds 1096 and 1103, which contain only Div. 3 contests. For each Solvita backbone, a single agent processes all distinct problems serially within each round. Each problem is evaluated with a single Solvita run and one Codeforces submission (pass@1), without multi-candidate selection. The bare control uses the same backbone, problem order, one-submission rule, and score insertion, but disables the Solvita loop. Rounds 1093 and 1097 have separate Div. 1 and Div. 2 pages; the same distinct-problem outcomes are projected onto both official pages for scoring, not rerun as separate sessions.

Table 15: Per-page Solvita results. Each model cell reports page score / CodeElo estimate. Overall solved counts deduplicate shared problems; final estimates and SEs weight the ten execution rounds equally.

Contest	Opus 4.6	DeepSeek V4 Pro	GPT-5.4
2219	5,750 / 3,829.3	5,750 / 3,829.3	4,750 / 3,829.3
2220	10,500 / 3,057.1	10,500 / 3,057.1	5,000 / 2,359.9
2222	14,500 / 3,853.2	14,000 / 3,637.3	14,500 / 3,853.2
2226	9,750 / 3,106.6	9,750 / 3,106.6	9,750 / 3,106.6
2223	9,750 / 3,754.6	9,750 / 3,754.6	13,250 / 3,754.6
2224	9,250 / 2,918.6	9,250 / 2,918.6	9,250 / 2,918.6
2228	13,250 / 3,130.6	12,250 / 3,130.6	13,250 / 3,130.6
2231	10,000 / 3,137.5	10,000 / 3,137.5	10,000 / 3,137.5
2229	18,500 / 3,892.7	22,500 / 3,892.7	13,500 / 3,467.3
2232	11,250 / 3,142.0	11,250 / 3,142.0	11,250 / 3,142.0
2234	9,250 / 3,134.1	9,250 / 3,134.1	8,500 / 3,013.6
2237	23,500 / 3,759.2	23,500 / 3,759.2	23,500 / 3,759.2
Solved	**73/79**	**73/79**	**72/79**
Mean $\pm$ SE	$\mathbf{3,394 \pm 103}$	$\mathbf{3,372 \pm 94}$	$\mathbf{3,304 \pm 94}$

Table 16: Per-page bare-backbone results, using the same score / CodeElo-estimate format and round-weighted final estimates as Table 15.

Contest	Opus 4.6	DeepSeek V4 Pro	GPT-5.4
2219	5,000 / 3,829.3	4,500 / 3,322.2	4,500 / 3,322.2
2220	9,500 / 3,057.1	8,500 / 2,936.6	8,500 / 2,936.6
2222	9,000 / 2,971.8	9,000 / 2,971.8	6,000 / 2,150.5
2226	7,250 / 2,915.4	7,250 / 2,915.4	7,250 / 2,915.4
2223	4,500 / 3,612.8	3,000 / 2,770.8	3,000 / 2,770.8
2224	8,000 / 2,918.6	5,750 / 2,727.3	5,750 / 2,727.3
2228	6,750 / 2,567.7	6,750 / 2,567.7	6,750 / 2,567.7
2231	4,750 / 2,316.5	4,750 / 2,316.5	7,250 / 2,775.3
2229	13,500 / 3,467.3	13,500 / 3,467.3	9,500 / 2,847.4
2232	5,250 / 2,572.6	3,250 / 1,984.4	5,250 / 2,572.6
2234	6,750 / 2,716.3	6,750 / 2,716.3	6,750 / 2,716.3
2237	15,750 / 3,759.2	13,500 / 3,759.2	10,000 / 3,356.6
Solved	**55/79**	**51/79**	**50/79**
Mean $\pm$ SE	$\mathbf{3,000 \pm 149}$	$\mathbf{2,858 \pm 165}$	$\mathbf{2,778 \pm 104}$

Different page scores can produce the same estimate when they induce the same inserted rank in the official standings.

CodeElo scoring and inversion. For page c, an accepted problem with initial value w_j on official attempt a_j contributes $\max(0, w_j - 50(a_j - 1))$ points; an unsolved problem contributes zero. Here $a_j = 1$ for every accepted problem, so no failed-attempt deduction applies. Following CodeElo, inference time is omitted. We retain rated participants present in both the official standings and pre-contest rating records, insert the agent by score and penalty, and denote its one-based rank by m_c.

For retained human ratings $R_{c,i}$, the published CodeElo equation defines the page-level estimate $\hat{r}_c$ by

$$m_c = \sum_{i \in H_c} \frac{1}{1 + 10^{(\hat{r}_c - R_{c,i})/400}}. \tag{8}$$

We solve this monotone equation numerically for each backbone and page.

Round aggregation. Let D_q contain the official page or pages associated with round q. For backbone a, we first compute $\hat{r}_{a,q} = |D_q|^{-1} \sum_{c \in D_q} \hat{r}_{a,c}$ and then plot the chronological running mean $\bar{r}_{a,t} = t^{-1} \sum_{q=1}^{t} \hat{r}_{a,q}$. This gives equal weight to each of the ten execution rounds. The final estimates for Claude Opus 4.6, DeepSeek V4 Pro, and GPT-5.4 are 3394, 3372, and 3304, respectively; their matched bare estimates are 3000, 2858, and 2778. We report the across-round

standard error

$$\mathrm{SE}(\bar{r}_a) = \sqrt{\frac{1}{K(K-1)} \sum_{q=1}^{K} (\hat{r}_{a,q} - \bar{r}_a)^2}, \qquad K = 10, \tag{9}$$

which gives 103, 94, and 94 points for Solvita and 149, 165, and 104 for the bare controls, in the same model order. The 12 underlying page estimates remain listed in Tables 15 and 16. The tier bands in Figure 5b are included only for interpretation.

H Knowledge-Network and Pipeline Implementation Details

This appendix documents implementation details of the three knowledge networks and the closed-loop control flow that sit behind Section 3.

H.1 Shared item schema

Advice and strategy items in the three namespaces (Solve, Test/Oracle, Hack) share an SQLite-backed item table. Each entry stores: a stable id (primary key), a namespace tag, a human-readable summary, a role-specific JSON payload, a list of searchable tags, a usage counter, a running average reward, a deprecation flag, and creation/last-used timestamps. Reads are served from an in-memory index; writes go through atomic file-locked transactions so that concurrent benchmark workers do not corrupt the store. For reported benchmark results, these tables are initialized from frozen checkpoints built on the benchmark-isolated training corpus, and evaluation-time insertion of new items, traces, and verdict artifacts is disabled.

H.2 Featurizer and bandit scoring

At inference time, a per-namespace featurizer maps the current pipeline state to a sparse set of feature keys $\Phi(x)$. The keys cover three orthogonal axes:

- **FSM position:** `FSM:SOLVE_DRAFT`, `FSM:SOLVE_PATCH`, `FSM:HACK_SEMANTIC`, etc.
- **Failure type from the previous iteration:** `FAIL:WA`, `FAIL:TLE`, `FAIL:RE`, `FAIL:MLE`, `FAIL:OFF_BY_ONE`, ...
- **Problem-level tags:** `TAG:dp`, `TAG:graphs`, `TAG:strings`, ...

For each advice or strategy item i, let m_i be the number of matching tags. The selector computes

$$s_i = b_i + \sum_{f \in \Phi(x)} W_{f,i} + 0.05 m_i, \qquad \eta_i \sim \mathrm{Uniform}(0, 0.01),$$

ranks items by $s_i + \eta_i$, and selects the top $k = 3$. Solver advice is populated in the prompt through `<MEMORY_ADVICE>`. After receiving module reward r, each selected item is updated by

$$b_i \leftarrow b_i + \alpha r, \qquad W_{f,i} \leftarrow W_{f,i} + \alpha r \quad (f \in \Phi(x)), \qquad \alpha = 0.01.$$

Unselected items and inactive features are unchanged, and missing parameters are initialized to zero.

H.3 Solver knowledge network storage and dynamics

The Solver knowledge network is persisted under `artifacts/solver_network/latest/graph` as a node–edge bundle plus a checkpoint of the learned w_{qm} and w_{ms} matrices. Each Q node stores the canonical problem JSON and tag list; each M node stores either an analysis trajectory (single solution decomposed into a function-block DAG) or a contrastive pair (correct + incorrect solution sharing the same approach, with the divergence point annotated); each S node stores an annotated skill with a C++ template and usage notes. Q–M edges are created at corpus-build time from authored or extracted decompositions; M–S edges use either deterministic function-block identifier matches (preferred) or an embedding-similarity fallback when no deterministic match exists. For reported benchmark results, all Q/M/S nodes are drawn from the benchmark-isolated training

subset or skill subset. Consequently, evaluation editorials, solution decompositions, code templates, and problem-specific analyses are not retrievable at inference time. Edge weights are renormalized within each MS group after every REINFORCE update so that $\sum_s w_{\mathrm{ms}}^{(j,s)} = 1$.

H.4 Failure-event propagation

When the Hacker breaks a candidate during training, the failure event is broadcast to the three trainable namespaces with role-specific payloads: the Solve namespace creates a new contrastive M node pairing the broken solution with the closest correct sibling, the Oracle namespace records the missed input pattern as a generator hint, and the Hack namespace bumps the successful route's score. At evaluation these writes are disabled: a break only adds the failing input to the current run's regression suite and returns it to the Solver.

H.5 Persistent state in training and evaluation

For each problem in the online training stream, Solvita runs the same pipeline as at evaluation (Figure 2), except that the Solver is executed twice, with and without QMS augmentation (Section 3.3), and the resulting rewards update the persistent state in Table 17. Evaluation reads this state from a frozen checkpoint and changes none of it. The only state created during evaluation is per run: the problem's certified tests and the regression suite built from Hacker failures, both discarded when the run ends.

Table 17: Persistent state, its training update, and its status during evaluation.

State	Training update	Evaluation
QMS edge weights $w_{\mathrm{qm}}, w_{\mathrm{ms}}$	REINFORCE with ΔR (Eq. 2); MS groups renormalized	read-only
QMS nodes and edges	growth from paired outcomes (Section 3.3) and Hacker failure events (Appendix H.4)	fixed
Oracle family parameters $b_f, W_{k,f}$	bandit update with r_{oracle} (Eq. 6)	read-only
Hacker route parameters	bandit update with r_{hack} after each round (Eq. 4)	read-only
Advice item parameters $b_i, W_{f,i}$	bandit update of the selected items	read-only
Usage counters, running rewards, deprecation flags, last-used timestamps	updated after each use; deprecation below -0.3 after ≥ 20 uses	frozen
Embedding cache (Appendix G)	caches text embeddings only; no outcomes or verdicts	cache only
Oracle generator hints and stored failure cases	written from Hacker failure events (Appendix H.4)	not written

H.6 Tool preparation and judge resolution

Each diagnostic configuration separately prepares its validator, checker, and reference program, generating them from the problem statement, constraints, and public samples as in evaluation. The configurations use matched backbones, preparation prompts, retry limits, and token budgets. Within Both, the Oracle and Hacker share one prepared set; the common preparation rules and judging priority do not imply reuse of the same files across configurations. The reference program is a judging artifact and is distinct from the Solver agent that generates candidate programs.

All compilation and execution happens in a per-process sandbox (`src/sandbox/`) that wraps `g++ -std=c++17 -O2` with wall-time and memory limits. During offline training, per-test verdicts use, in order, (1) the dataset `correct_solution`, (2) a custom checker, (3) execution of the reference program and token comparison, or (4) exact matching against the available expected output. Evaluation and deployment do not use the dataset solution and begin at the custom checker. This priority applies to both the Oracle certification gate and the Hacker break check and defines the verdicts that feed the Oracle certification ratio ρ in Eq. 3 and the Hacker break-rate g_{break} in Eq. 4.

Table 18: Controls separating graph coverage from routing weights. Retrieval slots and pipeline budgets are matched throughout. Candidate matching in the weight-only comparison preserves skill identity and content as well as graph-structure conditions.

Control	Graph	Weights	Candidate condition
Graph growth only	Grown Q/M/S nodes	Frozen uniform Q–M and M–S	Same retrieval slots
Weight learning only	Matched node counts; fixed connectivity and other graph conditions	Trained edge weights only	Identical reachable skill IDs and content
Complete Solver QMS	Full grown graph	Trained	Learned two-hop routing

I Additional Ablations

This appendix expands on the secondary ablations referenced from the experiments section. The main paper reports the headline numbers in Table 6 (patch vs. regenerate) and Figure 5a (Oracle/Hacker diagnostic configurations); here we describe their protocols and the default settings of related components.

Cost and error profiles. Figure 4a reports average token consumption per problem and run by backbone and framework. Solvita counts include every LLM call made by its agents across all three ensemble branches; baseline counts include every model call issued by the framework within the run. Light and dark segments represent prompt and completion tokens, respectively; their sum is the total consumption, and K denotes thousands of tokens. The three panels use GPT-5.4, Claude Opus 4.6, and DeepSeek V4 Pro.

Figure 4b reports GPT-5.4 error rates on CodeContests, APPS, and AetherCode. Rates use the full benchmark sizes (165, 1,000, and 400 problems) as denominators, not the number of failed problems. Light segments show Solvita's error rates, and dark segments show the bare single-pass model's excess over Solvita; the full bar height is therefore the bare model's error rate. Summed over categories, each model's rates equal its failure rate in Table 1, up to display rounding. Alg. WA denotes algorithmic wrong answers; Edge/Spec denotes errors involving edge cases or specification mismatches; TLE, MLE, and RE denote time limit exceeded, memory limit exceeded, and runtime error, respectively. Rates are computed from counts before rounding for display.

Graph-growth and routing controls. Table 18 specifies the three controls reported in Table 3.

Diagnostic configurations. All three configurations retain the tool preparation pipeline. Oracle-only enables conventional test generation and internal certification while disabling adversarial testing. Hacker-only disables the Oracle's conventional generation and certification but enables vulnerability analysis, adversarial input generation, and execution diagnosis using the prepared judging tools. Both runs the Oracle before the Hacker, which then uses the tests and judging artifacts retained by the Oracle. The Oracle's conventional test generator and the Hacker's adversarial input generators are separate components. A valid failure found by either enabled stage rejects the candidate.

All three configurations are evaluated on the same candidate solutions. Each configuration prepares its artifacts independently under matched backbone, preparation prompts, retry limits, and token budgets, whereas the Oracle and Hacker share the same artifacts within Both. Because the Hacker in Both uses the artifacts retained by the Oracle, its decisions can differ from those of Hacker-only, so Both is not the union of the two single-module decisions. The comparison therefore measures complete diagnostic configurations rather than the marginal effect of adding a module to an identical artifact set.

Diagnostic metrics for Figure 5a. Figure 5a evaluates each diagnostic configuration against held-out candidate solutions with known official verdicts. Solvita is said to *accept* a candidate if the diagnostic procedure does not find a failure, and to *reject* it if the procedure flags the candidate as

incorrect. For candidate-level accounting, we use the following four outcome counts:

	Officially accepted	Officially rejected
Solvita accepts	TP	TN
Solvita rejects	NP	NF

Here, TP denotes candidates accepted by both Solvita and the official judge; TN denotes candidates accepted by Solvita but rejected by the official judge; NP denotes candidates rejected by Solvita but accepted by the official judge; and NF denotes candidates rejected by both Solvita and the official judge.

The rejection detection rate, reported as Det'd, measures the fraction of officially rejected candidates that Solvita also rejects:

$$\text{Det'd} = \frac{NF}{TN + NF} \times 100\%.$$

The official acceptance preservation rate, reported as Pres'd, measures the fraction of officially accepted candidates that Solvita does not reject:

$$\text{Pres'd} = \frac{TP}{TP + NP} \times 100\%.$$

Confirmed stronger-test cases remain disagreements in Pres'd because this metric measures preservation of official acceptance rather than semantic correctness.

The third metric, reported as Str. Rate, is computed only over Solvita-rejects/official-accepts disagreements and is aggregated at the problem level rather than the candidate level. These disagreements correspond to the NP cell above, but we do not automatically treat them as Solvita errors: in some cases Solvita may have generated stronger diagnostic tests that expose hidden failures in solutions accepted by the official test suite. Let $\mathcal{D}_{NP}$ be the set of problems for which Solvita rejects at least one officially accepted candidate. For each problem $d \in \mathcal{D}_{NP}$, we collect a pool $\mathcal{A}_d$ of standard accepted candidate solutions and run every solution in $\mathcal{A}_d$ against Solvita's diagnostic tests for d. Let $\mathcal{B}_d \subseteq \mathcal{A}_d$ be the subset of these accepted candidates that are rejected by Solvita's diagnostic tests:

$$\mathcal{B}_d = \{a \in \mathcal{A}_d : a \text{ fails Solvita's diagnostic tests for } d\}.$$

A disagreement problem d is counted as a confirmed stronger-test case only when both of the following conditions hold:

$$\frac{|\mathcal{B}_d|}{|\mathcal{A}_d|} > 0.10,$$

and manual inspection confirms that the failing diagnostic examples have valid inputs under the problem statement and constraints, with correct expected outputs or correct checker verdicts. The first condition rules out isolated failures of a single accepted candidate; the second condition rules out invalid inputs, incorrect expected outputs, checker mistakes, and Oracle/certification errors. Let $\mathcal{D}_{\text{strong}} \subseteq \mathcal{D}_{NP}$ be the set of disagreement problems satisfying both conditions. We define the confirmed stronger-test rate as

$$\text{Str. Rate} = \frac{|\mathcal{D}_{\text{strong}}|}{|\mathcal{D}_{NP}|} \times 100\%.$$

Thus, Str. Rate measures the share of Solvita-rejects/official-accepts disagreement problems that are confirmed, through accepted-solution cross-checking and manual validation, to reflect genuinely stronger diagnostic tests rather than unsupported Solvita rejections.

Table 19 reports the underlying counts. The diagnostic set consists of 500 historical Codeforces problems disjoint from the cold-start corpus, the online training stream, the three evaluation benchmarks, and the Codeforces rounds of Section 4.7. Each problem contributes 7 officially accepted and 9 officially rejected submissions, and every configuration is evaluated on the same 3,500 accepted and 4,500 rejected candidates.

Table 19: Counts behind Figure 5a. Det'd $= NF/(TN + NF)$, Pres'd $= TP/(TP + NP)$, and Str. Rate $= |\mathcal{D}_{\text{strong}}|/|\mathcal{D}_{NP}|$.

Backbone	Configuration	TP	TN	NP	NF	$\lvert\mathcal{D}_{NP}\rvert$	$\lvert\mathcal{D}_{\text{strong}}\rvert$
Claude Opus 4.6	Oracle-only	3,389	793	111	3,707	71	6
	Hacker-only	3,252	560	248	3,940	228	26
	Both	3,361	326	139	4,174	92	18
GPT-5.4	Oracle-only	3,365	911	135	3,589	121	9
	Hacker-only	3,212	668	288	3,832	264	27
	Both	3,338	422	162	4,078	117	21
DeepSeek V4 Pro	Oracle-only	3,331	1,040	169	3,460	111	7
	Hacker-only	3,168	778	332	3,722	272	24
	Both	3,292	535	208	3,965	135	21

Paired comparisons with commercial agents. Table 20 compares Solvita with Codex CLI and Claude Code problem by problem. For each problem, the pass@3 outcome of each method is a binary indicator; Δ is the mean difference in percentage points, and the 95% interval comes from 20,000 paired bootstrap resamples over problems. The intervals reflect variation over problems for the reported runs and checkpoints, not variation from retraining the knowledge networks. On their native backbones, the two agents are statistically indistinguishable from Solvita in all six settings (exact McNemar $p \geq 0.68$). Solvita leads in all 24 remaining comparisons; 19 bootstrap intervals exclude zero, and exact McNemar tests give 18 comparisons with $p < 0.05$, 17 after Benjamini–Hochberg correction over all 30 comparisons, and 8 under the more conservative Holm correction.

Table 20: Paired pass@3 comparison of Solvita with commercial agents. Δ: Solvita minus baseline (points); S/B: problems solved only by Solvita / only by the baseline.

		vs. Codex CLI			vs. Claude Code		
Backbone	Bench	Δ	95% CI	S/B	Δ	95% CI	S/B
GPT-5.4	CC	$+0.61$	$[-4.24, +5.45]$	8/7	$+11.52$	$[+4.24, +18.79]$	30/11
GPT-5.4	APPS	$+0.60$	$[-1.80, +3.00]$	78/72	$+7.30$	$[+4.20, +10.40]$	162/89
GPT-5.4	AC	$+0.75$	$[-2.50, +4.00]$	24/21	$+6.50$	$[+2.25, +11.00]$	53/27
Claude Opus 4.6	CC	$+10.30$	$[+3.64, +16.97]$	25/8	$+0.61$	$[-3.03, +4.85]$	6/5
Claude Opus 4.6	APPS	$+9.50$	$[+6.90, +12.10]$	142/47	$+0.30$	$[-1.40, +2.00]$	41/38
Claude Opus 4.6	AC	$+9.50$	$[+5.25, +13.75]$	60/22	-0.50	$[-4.50, +3.50]$	32/34
Qwen3.6-Plus	CC	$+9.70$	$[+3.64, +15.76]$	22/6	$+10.91$	$[+4.24, +17.58]$	25/7
Qwen3.6-Plus	APPS	$+7.20$	$[+4.50, +9.90]$	135/63	$+8.80$	$[+6.10, +11.50]$	140/52
Qwen3.6-Plus	AC	$+3.50$	$[+0.25, +6.75]$	29/15	$+4.50$	$[+1.50, +7.50]$	29/11
DeepSeek V4 Pro	CC	$+9.70$	$[+3.64, +15.76]$	21/5	$+12.12$	$[+5.45, +18.79]$	27/7
DeepSeek V4 Pro	APPS	$+1.80$	$[-0.60, +4.20]$	85/67	$+3.60$	$[+1.20, +6.00]$	90/54
DeepSeek V4 Pro	AC	$+6.50$	$[+2.00, +11.00]$	55/29	$+7.75$	$[+3.75, +12.00]$	53/22
Grok 4	CC	$+4.85$	$[-1.21, +10.91]$	17/9	$+6.06$	$[+0.61, +11.52]$	16/6
Grok 4	APPS	$+1.30$	$[-0.70, +3.30]$	60/47	$+3.00$	$[+0.70, +5.20]$	81/51
Grok 4	AC	$+2.00$	$[-1.25, +5.25]$	26/18	$+3.75$	$[0.00, +7.50]$	36/21

Table 21 reports mean single-run success. Each Solvita run already selects one of three internal branches (Appendix G). On their native backbones, Codex CLI and Claude Code exceed Solvita's single-run success by 0.2 to 1.8 points in five of six settings, so Solvita's pass@3 lead there comes from greater complementarity across runs. On the other backbones, Solvita has higher single-run success in 22 of 24 comparisons.

Table 21: Mean single-run success (%) over the three runs used for pass@3 in Table 1.

	GPT-5.4			Claude Opus 4.6			Qwen3.6-Plus			DeepSeek V4 Pro			Grok 4		
Method	CC	APPS	AC	CC	APPS	AC	CC	APPS	AC	CC	APPS	AC	CC	APPS	AC
Codex CLI	72.93	54.47	37.42	60.61	46.77	35.08	48.69	36.83	17.42	66.87	55.63	36.50	64.85	47.80	22.25
Claude Code	59.19	47.30	32.42	67.68	58.47	42.00	45.66	35.83	16.33	67.27	52.43	34.25	61.82	42.80	22.75
Solvita	71.11	54.30	37.00	66.87	57.63	45.08	57.98	43.63	19.58	75.96	54.30	38.92	64.65	48.53	25.00

Patch repair vs. full regeneration. We rerun the Solver inner loop with two settings holding everything else fixed: (i) *patch*, the default, which routes through `generate_code.patch_decision` and emits SEARCH/REPLACE blocks (Appendix F); (ii) *regenerate*, which always falls through to `generate_code.regenerate` and rewrites the full program. Both modes share the same iteration cap ($N_{\max} = 8$) used end-to-end (Appendix G) and the same failure-analysis prompt.

The token savings in Table 6 are converted from the two modes' common-reference summaries as $100[1 - (1 - s_{\text{patch}})/(1 - s_{\text{regen}})]$, where s is the fractional saving against their shared reference cost. The common denominator cancels, yielding patch savings relative to actual full regeneration. These values are approximate because the source percentages were rounded to one decimal place.

LLM selection and planning within the sampled group. The default invokes `solver_skill_selection` (Appendix F) to select a subset of the QMS candidate group and produce a subproblem DAG.

Oracle acceptance threshold τ. The Oracle gate (Eq. 3) accepts an artifact only when the certification ratio $\rho \geq \tau$, with default $\tau = 0.9$. Lower τ admits more tests at the risk of admitting incorrect ones.

Hacker round budget. `max_hack_rounds` is set to 3 by default (Appendix G), so that each of the three routes (semantic, stress, and anti-hash) can receive a round on problems where it applies.

Paired Solver reward. To characterize the local QMS update in Eq. 2, we inspect saved training checkpoints and reward logs. Each problem uses 128.6 Solver-side tests on average, so R is a test-pass fraction rather than a single binary verdict. Pairing runs with and without QMS augmentation on the same problem yields

$$\begin{aligned} \mathrm{Var}(\Delta R) &= 0.031, \\ \frac{\mathrm{Var}(R_{\text{with}} - R_{\text{without}})}{\mathrm{Var}(R_{\text{with}}) + \mathrm{Var}(R_{\text{without}})} &= 0.37. \end{aligned} \tag{10}$$

The denominator is the variance of a difference with the same marginal reward variances but zero covariance. The ratio therefore indicates cancellation of shared variation through pairing, not reduced variance of the full policy-gradient estimator or a comparison with $\mathrm{Var}(R_{\text{with}})$ alone. The return is local to the Solver stage and excludes later Oracle certification, adversarial search, and patch-drift decisions. Graph-growth, weight-learning, and retrieval controls are reported separately in Section 4.3.

J Limitations and Failure Cases

We discuss training and initialization costs, limits of adversarial coverage, and representative failure modes observed during evaluation.

Training and initialization cost. The agentic loop is more expensive than direct generation, and QMS training requires paired runs with and without QMS augmentation. Section 4.4 shows learning from an empty graph but also sensitivity to the cold-start corpus under a fixed training budget. Longer learning curves are needed to distinguish sample-efficiency gains from changes in asymptotic accuracy.

Scope of mechanism evidence. The routing and retrieval controls in Section 4.3 isolate learned routing for the Solver only. Table 2 shows that trained Oracle and Hacker networks help, but does not separate their memory content from their learned routing, and we do not isolate how failure events propagated across roles during training affect later problems. These controls use a single training run per condition and report no intervals.

Cold-start retrieval misfires. On problems whose paradigm is poorly represented in the cold-start corpus, the Solver QMS network retrieves structurally similar but semantically misleading skills. The selection LLM (Appendix F, `solver_skill_selection`) is permitted to return an empty list

in this case, but it does not always do so, and the resulting skill block can bias the initial draft toward the wrong paradigm. Rewards for candidate groups provide limited feedback on the contribution of individual skills.

Oracle false certification. Despite the gate in Eq. 3, internally certified tests can preserve errors shared by generated judging artifacts. Hacker adversarial testing can expose some program defects missed by the Oracle's conventional tests, but remains limited by errors in shared judging tools. Crashes and timeouts on valid inputs are directly observable, whereas a wrong-answer claim requires reliable checker or reference evidence; disagreement alone does not identify which program is wrong. Hacker feedback repairs or regenerates the candidate program rather than automatically correcting Oracle artifacts.

Hacker scope limitations. Problems involving number-theoretic invariants, combinatorial identities, or geometric tolerances frequently survive all three Hacker routes because the Code Analyst fails to identify the bug class; anti-hash and lattice-based attacks are likewise bounded by the backbone's reasoning capacity. We currently do not have a systematic mitigation; warm-starting the Hack network from human-authored counterexamples on a held-out math subset is the most promising next step.

Patch repair drift on global flaws. The `patch_decision` prompt (Appendix F) is meant to route systemic flaws to `full_regen`, but on borderline cases it can mislabel a global flaw as localized and start patching. The result is a sequence of small edits that each fix the immediate failure but accumulate state inconsistencies, eventually exhausting `max_iterations` (Appendix G) without converging. Detecting such runs, for example by tracking how often accepted patches are followed by new failures, and feeding that signal into `patch_decision` is future work.

K Future Work

Three directions follow naturally. First, improving cold-start corpus selection and graph initialization may reduce the training examples needed to reach a target accuracy, motivated by the diminishing returns in Section 4.4. Second, transferring the four-agent decomposition to other verifiable reasoning domains—formal theorem proving, where the Oracle becomes a proof checker and the Hacker searches for counter-models; mathematical olympiad problems, where certified test cases are replaced by symbolic verification; and scientific reasoning with executable simulators—is a direct port of the same closed-loop interface to settings that share competitive programming's verifier-grounded reward structure. Third, the per-step adversarial signal produced by the Hacker is a candidate fine-tuning signal beyond prompt-level updates: investigating whether REINFORCE on knowledge-network weights can be lifted into model-weight updates without losing the role-aligned credit assignment is, to us, the most interesting open question.