AGENTDISCOVER: AUTONOMOUS DISCOVERY WITH MINIMAL SEARCH SCAFFOLDING

Anonymous authors
Paper under double-blind review

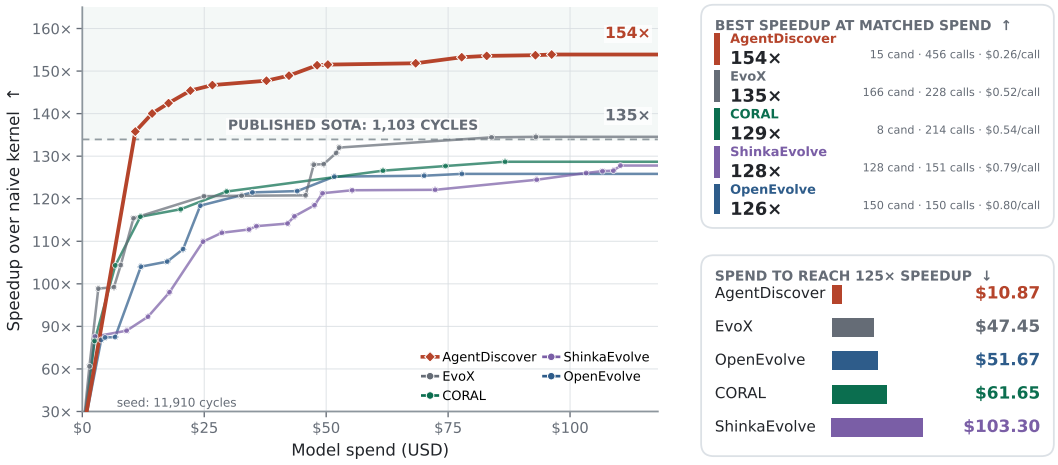


Figure 1: On the kernel builder task, AgentDiscover reaches 960 cycles, a new state of the art against the best published result of 1,103 and the best publicly reported one of 1,063, and it reaches every speedup level at a lower token cost than the frameworks we compare against. All platforms in the figure run on Claude Opus 5, and all curves are truncated at the common budget of $118.

ABSTRACT

Frameworks that use large language models for scientific discovery typically rely on a fixed, human-designed algorithm that decides what the model sees at each step, leaving the model only the role of proposer. The model knows nothing of the search beyond what it is shown. As models grow more capable, a question arises: does a search strategy chosen by a human before the run scale better than promoting the model from proposer to planner and letting it own the search? *The Bitter Lesson* suggests that choosing the strategy in advance is the kind of hand-designed structure that general methods eventually outscale. We introduce AgentDiscover, in which a coding agent plans the search using its context as working memory, runs experiments, and records every attempt in a database of ideas, candidates, and their relations. This database serves as the agent's long-term memory and is structured so that the selection rules of classical algorithms such as MAP-Elites and Monte Carlo tree search each reduce to a single query, which the agent is free to use, combine, or replace. A server maintains the database and steers the agent after every submission, keeping it on course over long runs. In our experiments, AgentDiscover is more cost-efficient than existing frameworks, reaching better scores at lower cost. It achieves the best result among existing discovery frameworks on the kernel builder task, and on seven past AtCoder heuristic contests its programs would have placed first among human competitors in every contest. On eleven mathematical and systems optimization tasks, it matches or exceeds every baseline that uses the same model. Our code is available here.

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

1 INTRODUCTION

Large language models (LLMs) are increasingly capable problem solvers, but their solutions must be verified. Returning each verdict to the model lets it improve its solutions, closing a loop of proposal and verification. When both the candidate solution and its evaluator are programs, this loop needs no human intervention: the model writes a program that either encodes a solution or searches for one, and an evaluator program runs it and scores the result. This recipe, typically run as an evolutionary search, has already produced genuine mathematical and scientific discoveries. For instance, FunSearch (Romera-Paredes et al., 2024) found new constructions in extremal combinatorics, and AlphaEvolve (Novikov et al., 2025) improved the best known results on roughly a fifth of more than fifty open mathematical problems. A growing family of open systems follows the same recipe (Sharma, 2025; Lange et al., 2026; Cemri et al., 2026; Liu et al., 2026a), and some go further by training the model on each problem at test time (Wang et al., 2025; Yuksekgonul et al., 2026).

These systems share a common architecture with three roles. A *search algorithm* selects past candidates for the model's context, a *proposer*, a language model, writes a new candidate from them, and an *evaluator* scores it; the result will then be recorded, typically in an archive of past candidates. In most implementations, the selection rule is designed by humans and fixed before the run: AlphaEvolve combines MAP-Elites (Mouret & Clune, 2015) with island models, ShinkaEvolve (Lange et al., 2026) extends a prescribed population loop with novelty filtering, and MCTS-AHD (Zheng et al., 2025) organizes candidates in a tree and selects among them with the upper confidence bound for trees (UCT) (Kocsis & Szepesvári, 2006). More recent systems adapt this rule during a run, but only within a space prescribed beforehand by their designers (Cemri et al., 2026; Liu et al., 2026a). In every case, the search is run by a human-designed algorithm and the LLM only proposes: it sees the candidates that the algorithm puts in its context, but has no say in which ones those are.

Leaving the search to a fixed human-designed algorithm becomes harder to justify as LLMs improve. In *The Bitter Lesson*, Sutton observes that methods built on human knowledge are eventually overtaken by general methods that exploit growing computation (Sutton, 2019). Choosing the search strategy before the run is such a use of human knowledge; it bets that the designer understands the problem better than the language model will. The LLM, meanwhile, could run the search itself: classical algorithms such as MAP-Elites and UCT are well documented in the text it is trained on, and *coding agents*, which let an LLM write and run code with tools, can implement them when needed. Knowing the alternatives matters because no single search strategy is best across problems (Wolpert & Macready, 1997). In automated evolutionary discovery specifically, a budget-matched comparison of 30 fixed search strategies ("harnesses") found none reliably better than a simple baseline across models and problems, and the full OpenEvolve recipe ranked last (Gupta et al., 2026). The best strategy can also change within a run (Cemri et al., 2026; Liu et al., 2026a); fixing one for the whole search leaves the model's knowledge of the alternatives unused. This raises a natural question: *can coding agents fill both roles, one planning the search and another proposing candidate solutions with minimal human-engineered search heuristics?*

Handing the search to a coding agent is not enough on its own. The agent must remember what it has tried long after those attempts have left its context, retrieve the relevant part of that history cheaply, stay on course over hours of work, and do all of this without being able to tamper with the evaluator that scores it. Without a structured memory, the agent must reconstruct the state of the search from raw logs before it can apply any strategy to it. Storing such a memory is cheap; reading it is not. Every attempt can be recorded, with its score, its lineage, and the ideas it builds on, at negligible cost next to the compute a run already spends, but an agent pays in context for every record it reads. Databases have long addressed this: a query states what is wanted, and indexes and a query planner find it, so the reader receives only the answer. This raises a second question: *can the history of a search be kept so that the agent retrieves only what it needs, at the cost of a query?*

We answer these questions affirmatively by developing AgentDiscover, a novel framework for scientific discovery in which a coding agent both directs the evolutionary search and proposes its candidates, with every past attempt one query away in a graph database (Figure 2b, §3). A coding agent works on the problem inside a sandbox, and its context is its working memory: it runs experiments, writes candidates, and decides what to try next. Each agent session eventually ends and a fresh one takes over, for two reasons: a fresh agent is not conditioned on its predecessor's reasoning; and beyond some length, re-reading an ever longer context at every step costs more than starting over. What a session learns outlives it in a long-term memory: a graph database of ideas and candidates.

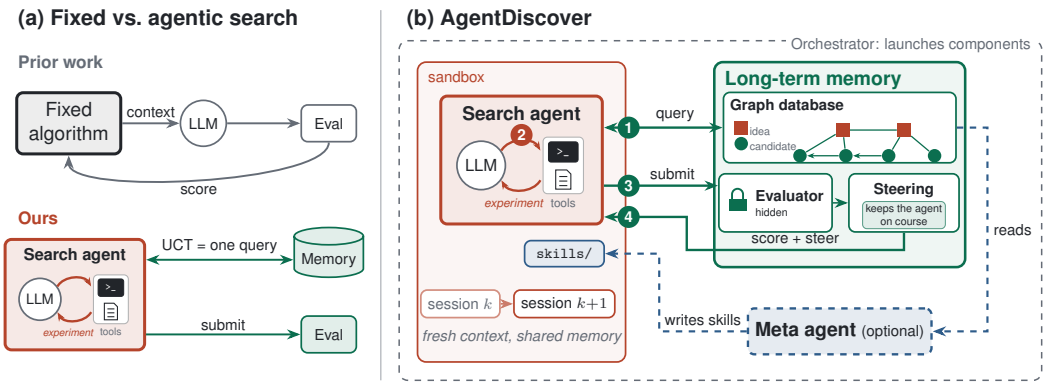


Figure 2: AgentDiscover lets a coding agent direct the evolutionary search. (a) A fixed algorithm decides what the LLM sees in prior systems; in AgentDiscover, the agent queries its memory itself. (b) The agent's loop: (1) query the graph database, (2) experiment, (3) submit a candidate, (4) receive its score and a steering message. New sessions start with a fresh context and the same memory.

Because the graph makes each candidate's ideas and lineage explicit, the selection rule of a classical algorithm becomes a single query: the best candidate per idea for MAP-Elites, or visit counts and values over the candidate lineage for Monte Carlo tree search. The agent can use these rules, combine them, or replace them. Because the search strategy is not fixed in advance, a stronger model improves not only the candidates but also the search that produces them, whereas under a fixed search algorithm it can improve only the candidates. A server mediates all access to this memory, evaluates submitted candidates in isolation from the agent's sandbox, and returns each score with a message that keeps the agent on course within and across sessions. A meta agent reads past sessions and writes skills that later sessions draw on.

We evaluate AgentDiscover on the kernel builder task (Anthropic, 2026), on eleven mathematical and systems optimization tasks (Qu et al., 2026), run with the same LLM as the published baselines (Table 1), and on seven past AtCoder heuristic contests (Imajuku et al., 2025) and the first autocorrelation inequality (Table 2 and Table 3). On the kernel builder task, AgentDiscover achieves the best result among existing discovery frameworks, 960 cycles against the previous best published result of 1,103 (Qu et al., 2026) (Figure 1). On seven past AtCoder heuristic contests, its programs would have placed first among human competitors in every contest. On the eleven optimization tasks, it matches or exceeds every baseline. It also improves the best published upper bound on the first autocorrelation inequality. In the course of this evaluation we found that several evaluators used by prior work admit reward hacking; we correct them and describe the common vulnerabilities.

In summary, our contributions are:

1. We present AgentDiscover, a framework for scientific discovery that replaces the fixed evolutionary search algorithm with a coding agent that directs the search and proposes candidates, and that manages its context efficiently through a database of past attempts in which the selection rules of classical search algorithms are single queries the agent can use, combine, or replace (§§3.1 and 3.2).
2. We show that AgentDiscover matches or improves the best published result on all of the eleven standard tasks, would have placed first against human competitors in seven past AtCoder contests, and improves the best published bound on the first autocorrelation inequality (§4).
3. We identify reward-hacking vulnerabilities in evaluators used by prior work and correct them (Appendix C).

2 Related Work

LLM-based evolutionary search. A first line of work couples an LLM with an evolutionary search loop for scientific and algorithmic discovery. FunSearch (Romera-Paredes et al., 2024) established the recipe, evolving programs inside a fixed island-based loop and discovering new constructions in

extremal combinatorics. AlphaEvolve (Novikov et al., 2025) then shifted the landscape, improving the best known results on roughly a fifth of more than fifty open mathematical problems with a fixed archive and population policy combining MAP-Elites and island models. ShinkaEvolve keeps a prescribed population outer loop but adds novelty filtering and bandit-based model selection, while MCTS-AHD replaces population evolution with a fixed UCT-organized tree (Lange et al., 2026; Zheng et al., 2025). SimpleTES designs parallel trajectories, refinement, and history selection, and sets new best-known values with an open model (Ye et al., 2026). Fine-tuning variants keep the same loop: ThetaEvolve and TTT-Discover update the model parameters themselves at test time (Wang et al., 2025; Yuksekgonul et al., 2026). In all these systems the search algorithm is designed in advance and stays fixed; they differ mainly in which algorithm they fix. A few go a step further and adapt the algorithm at runtime: AdaEvolve adjusts within-island exploration, cross-island allocation, and stagnation-triggered guidance through a prescribed three-level controller (Cemri et al., 2026); Gupta et al. (2026) run several fixed harnesses side by side and shift the remaining compute to those whose early scores lead; and EvoX evolves an explicit parent-selection, inspiration, and variation policy, but only when a fixed progress monitor triggers replacement (Liu et al., 2026a). Even here, adaptation remains confined to a predefined space.

Coding agents as proposers. A few systems instead hand the search to a coding agent. AlgoEvo lets an agent decide when to inspect, diagnose, edit, or evaluate code, but a fixed upper-confidence rule still chooses which earlier design guides each step (Qiu et al., 2026). Vesper gives repository-level coding agents direct database access, but its fixed island-model policy samples the parent and seeds the worktree before the agent is invoked (Ishibashi et al., 2026). Meta-Harness removes the selection rule entirely: a coding agent reads every earlier harness with its scores and traces and decides what to change next, though its object is the harness around a model rather than the solution to a discovery task (Lee et al., 2026). CORAL is the closest work to ours: its long-running agents control retrieval and search direction across shared persistent state (Qu et al., 2026). The two frameworks differ, however, in ways that matter for long-horizon search. CORAL shares attempts, notes, and skills as file-system artifacts, whereas AgentDiscover keeps long-term memory in a graph database of reusable ideas and many-to-many candidate lineage, making retrieval a structured query rather than a file scan. AgentDiscover also periodically resets the agent's context, which helps the agent escape local minima and avoids the token cost of very long agent histories. Steering differs as well: CORAL interrupts and restarts the session on a heartbeat, whereas AgentDiscover's server injects steering messages into the running session.

Adapting prompts, scaffolds, and memories. A third group adapts artifacts other than the candidate program. With frozen weights, GEPA adapts prompts, BES searches over reasoning trajectories at inference time, SICA recursively edits its coding-agent scaffold, and SkillOpt applies bounded updates to skill documents (Agrawal et al., 2025; Xu et al., 2026; Robeyns et al., 2025; Yang et al., 2026). Others persist semantic search state: SeaEvo clusters and retrieves strategy descriptions and PACEvolve decouples idea generation from code evaluation (Luo et al., 2026; Yan et al., 2026).

Autonomous research agents. A fast-growing line of work gives LLM agents autonomy over research itself. The AI Scientist automates the cycle from idea to paper (Lu et al., 2024; Yamada et al., 2025); the AI co-scientist and the Virtual Lab organize agents as a research group (Gottweis et al., 2026; Swanson et al., 2025); and long-running systems combine literature search and data analysis to reach new findings (Ghareeb et al., 2026; Mitchener et al., 2025). Agents are also changing how research is shared: Paper2Agent turns papers into agents (Miao et al., 2025), AgentRxiv lets agents build on each other's reports (Schmidgall & Moor, 2025), and concurrent projects run coding agents autonomously on optimization targets (Karpathy, 2026; Chen et al., 2026). These systems target open-ended research judged by humans or experiments. AlphaEvolve-style methods instead perform verifier-driven program evolution, but the LLM only proposes candidates inside a fixed, human-designed search loop. AgentDiscover combines the two: an autonomous agent that owns the search, scored by a verifier.

3 METHODOLOGY

3.1 PROBLEM FORMULATION AND MOTIVATION

LLM-driven scientific discovery casts a research problem as a search over programs. The problem is stated in natural language, and each *candidate* is a program x: a representation the LLM can read

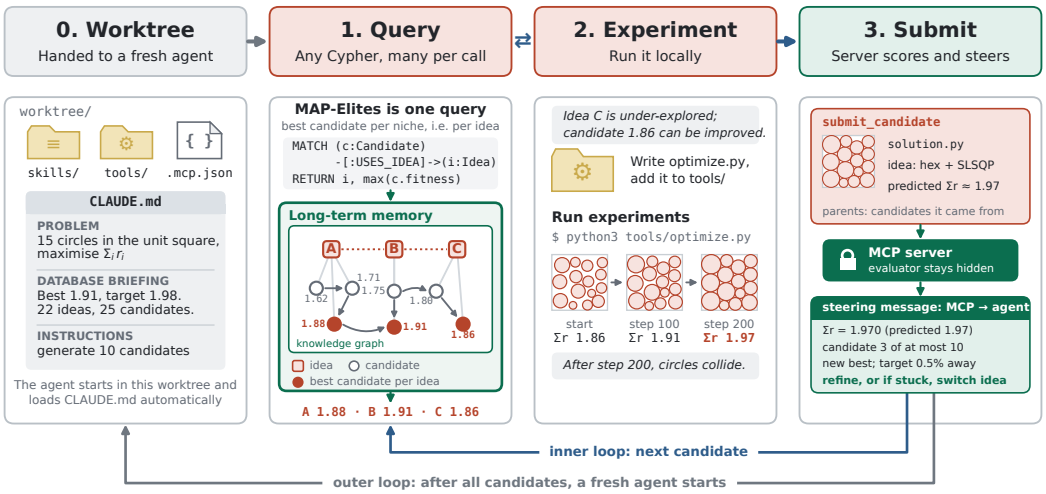


Figure 3: AgentDiscover on a toy task: place 15 circles in a square to maximize the sum of their radii. A coding agent does the search. An MCP server stores all past ideas and candidates in a graph database and scores new candidates. (0) The agent starts in its folder (worktree). A CLAUDE.md file there describes the task and a summary of the database. (1) The agent explores the database with any read-only query. (2) It runs experiments, using available tools or new ones it writes first. (3) It submits a candidate. The server scores it with the task's evaluator and replies with a steering message. The agent repeats steps 1–3 for each new candidate (inner loop). After the last one, a new agent with a fresh context starts on the same database and keeps all tools written so far (outer loop).

and revise, and a machine can execute and score. A fixed *evaluator* E scores each candidate with a scalar *fitness* $E(x)$, oriented so that higher is better, and reports any auxiliary metrics the problem defines. The goal is a candidate of high fitness within a budget of evaluations, found by an LLM that revises candidates iteratively in light of what has already been tried.

Existing systems (e.g., Romera-Paredes et al., 2024; Novikov et al., 2025; Lange et al., 2026; Ye et al., 2024) fix this iteration before the run: an evolutionary algorithm decides which past candidates the LLM sees, and the LLM only proposes the next one. This design is ironic: these systems aim to surpass human design, yet their search procedure is human-designed, and the LLM, the one component meant to outdo human design, has no say in it. Committing the whole run to one such procedure is hard to justify when no single strategy is best across problems (§1).

We therefore replace the LLM proposer with a coding agent. The agent queries past attempts to build its own context, runs its own experiments, and records what it tries in a memory that outlives its context. Which candidates to revisit, which directions to abandon, and which strategy to follow become decisions the agent makes, and revises, during the run; in the terms of metacognition, it takes over the monitoring and control of its own search (Flavell, 1979). This flexibility pays off only if it remains safe and efficient over the long horizons these tasks demand; AgentDiscover is the infrastructure that provides it.

3.2 The AgentDiscover Framework

AgentDiscover has three components. The **search agent** is a coding agent that studies previous attempts, plans, runs experiments, and proposes new candidates. The **long-term memory** is a graph database that keeps every idea and candidate tried so far. The optional **meta agent** is another coding agent that reflects on the search agents' reasoning logs and writes what it learns as skills for them. We designed these components with two properties in mind, and show below how they achieve them. First, *flexibility*: no search strategy is hard-coded, and the agent decides what to read, what to try, and when to change course. Second, *efficiency over long horizons*: a search runs for hours, so the agent must stay on course, retain what it learns beyond its context, and keep its token cost under control.

Search agent: The search agent is a coding-agent CLI, such as Claude Code, sandboxed in a version-controlled worktree. It starts from an instruction file that describes the task, summarizes the database, and explains how to access it through a Model Context Protocol (MCP) server; the worktree also holds tools and skills folders, both empty in the first session. At each step, the agent chooses whether to query the database, plan, write or run tools, or submit a candidate, and it reflects on each evaluation and plans further. After submitting its allotted number of candidates, the agent records a lesson in the database, including experiments that never became candidates, and terminates, ending the *session*. A new session then starts a fresh search agent that inherits the long-term memory, tools, and skills. Figure 3 walks through a session on a toy task.

Long-term memory: The long-term memory is a graph database of all ideas, candidates, sessions, and lessons, and the relations among them. It sits on the MCP server alongside the evaluator: the agent reads it with any query but writes only through MCP tools that validate their input arguments, keeping the database consistent. The server scores each submitted candidate and sends the agent a steering message that reports the fitness and the number of candidates left, keeping the agent on course over the long horizon. The graph also makes classical strategies native: MAP-Elites, which keeps the best candidate per niche, is one query returning the best candidate for each idea.

Meta agent: The meta agent is another coding-agent CLI, but with a different task and a different view of the database. It runs after each search-agent session and focuses on how the search was done rather than on what it found. It reads the session transcripts, including the search agents' reasoning traces, and relates them to the outcomes recorded in the long-term memory to find which ways of reasoning paid off and which did not. It then consolidates its conclusions into skill files that later sessions draw on.

The agent accumulates a long context, which causes three problems. (i) Each LLM call re-reads the full context, so past some length a new session is cheaper than continuing, even with the reduced cost of cached tokens. (ii) The agent also becomes anchored to its earlier reasoning, which makes it harder to change ideas and more likely to get stuck in a local optimum. (iii) It is more likely to lose track of the task. Fresh agents and the database together address the first two problems. The new agent explores the database from scratch and can form a new perspective on the search; we expect this to produce new ideas more reliably than prompting the old agent to try something else while its context stays the same. Restarting is cheap because retrieval is a query. Without the database, the agent would read past attempts from files: the Meta-Harness (Lee et al., 2026), for example, reads a median of 82 files per iteration, each read costing an LLM call and consuming context the agent cannot get back. For the third problem, the server answers every submission with a steering message that tells the agent where the search stands and keeps it on course.

4 EXPERIMENTS

We evaluate AgentDiscover on 20 problems spanning five systems tasks, six mathematical optimization tasks, seven AtCoder heuristic contests, the first autocorrelation inequality (AC1), and Anthropic's kernel builder task. We compare solution quality with existing search frameworks and published test-time finetuning results, including comparisons at matched model spend. We also study the effects of parallel search agents and meta-agent guidance. We provide hyperparameters and run configurations in Appendix D.

Benchmarks and baselines. We organize the evaluation into four settings: systems and mathematical optimization, AtCoder heuristic contests, comparison with test-time RL finetuning, and Anthropic kernel builder (Tables 1–4).

Systems and mathematical optimization. To test performance across different optimization problems, we consider five systems tasks and six mathematical tasks (Table 1). We compare against OpenEvolve (Sharma, 2025), ShinkaEvolve (Lange et al., 2026), EvoX (Liu et al., 2026a), and CORAL (Qu et al., 2026), using the baseline results reported by Qu et al. (2026). All methods use Claude Opus 4.6, keeping the backbone model fixed across frameworks.

AtCoder heuristic contests. To assess performance on heuristic programming problems with official tests and human competition results, we evaluate on five past AtCoder Heuristic Contests: AHC006, AHC012, AHC015, AHC025, and AHC041 (Table 2). These contests provide the task

Table 1: Results on five systems and six mathematical optimization tasks. All methods use Claude Opus 4.6 with a budget of at most 100 candidates. Published baseline results are taken from Qu et al. (2026). Arrows indicate the direction of improvement; best scores and ties at common reporting precision are bold. For signal processing, the faded row reports scores from the original evaluator, which permits reward hacking; these scores are excluded from performance comparisons. The next row uses the corrected evaluator (Appendix C).

	Task	OpenEvolve	ShinkaEvolve	EvoX	CORAL	AgentDiscover
Systems	EPLB ↑	0.127	0.129	0.146	0.149	**0.153**
	PRISM ↑	**26.26**	**26.26**	**26.26**	**26.26**	**26.26**
	LLM-SQL ↑	0.716	0.724	0.726	0.731	**0.737**
	Txn Sched. ↑	3774	3802	3984	4566	**4651**
	Cloudcast ↓	627.2	627.8	623.5	618.4	**618.0**
Math	Circle-Pack. ↑	2.6293	2.6001	2.6320	**2.6360**	**2.6360**
	Signal Proc. ↑ (original)	0.6420	0.8171	0.7306	0.8229	**0.9259**
	Signal Proc. ↑ (corrected)	0.7286	0.7635	0.7288	0.7790	**0.7948**
	Erdős Over. ↓	0.38188	0.38156	0.38125	0.38089	**0.38077**
	MMD-16-2 ↓	12.92	**12.89**	12.96	**12.89**	**12.89**
	MMD-14-3 ↓	4.21	4.46	4.46	**4.16**	**4.16**
	3rd-Autocorr. ↓	1.4731	1.4812	1.5552	1.4557	**1.4556**

Table 2: Results on five past AtCoder heuristic contests using Claude Opus 5 without web access and at matched model spend. Scores are measured on the official system tests; higher is better, and the best score in each row is bold. *Con.* gives the rank a program would have achieved among the original competitors; *Ext.* includes post-contest submissions. †AHC025 reports relative scores in units of 10^9 from a fixed standings snapshot; see Appendix E.2.

	OpenEvolve			EvoX			CORAL			AgentDiscover		
Task	Score	Con.	Ext.	Score	Con.	Ext.	Score	Con.	Ext.	Score	Con.	Ext.
AHC006 ↑	2,238,362	1st	13th	2,197,920	4th	36th	**2,317,352**	1st	1st	2,309,017	1st	1st
AHC012 ↑	99,090,668	5th	39th	98,579,333	9th	55th	99,977,717	1st	4th	**99,997,290**	1st	3rd
AHC015 ↑	144,369,058	15th	130th	151,853,188	4th	93rd	156,230,796	4th	57th	**168,660,525**	1st	5th
AHC025† ↑	92.60	37th	51st	94.46	21st	30th	98.50	2nd	6th	**98.76**	1st	3rd
AHC041 ↑	76,321,331	10th	58th	75,787,437	31st	127th	77,358,800	1st	6th	**78,044,915**	1st	1st

setting used by ALE-Bench (Imajuku et al., 2025). We run OpenEvolve, EvoX, CORAL, and AgentDiscover with Claude Opus 5 and score their programs on the official system tests.

Comparison with test-time RL finetuning. We next ask whether search with fixed model weights can compete with task-specific RL finetuning. We consider AHC039, AHC058, and the first autocorrelation inequality (AC1), for which TTT-Discover (Yuksekgonul et al., 2026) reports results obtained by finetuning gpt-oss-120b with reinforcement learning (Table 3). We compare these published results with CORAL and AgentDiscover, which we run using Claude Opus 5 with fixed weights. Since the backbone models differ, this compares the complete systems rather than isolating the effect of finetuning.

Anthropic kernel builder. Finally, we test how solution quality improves with model spend on Anthropic's Original Performance Take-Home (Anthropic, 2026) (Table 4). The task is to optimize a parallel tree-traversal kernel on a simulated VLIW SIMD processor. At each step, the kernel hashes an input value with the current node's value and uses the result to choose the next branch. The goal is to minimize execution cycles while preserving correctness. We compare AgentDiscover with all four search baselines at common model-spend budgets, using Claude Opus 5 with low reasoning effort and the same evaluator.

Protocol. In Table 1, we allow at most 100 submitted candidates per run, following the 100-iteration protocol used by Qu et al. (2026). For the AtCoder contests and AC1 (Tables 2 and 3), AgentDiscover and CORAL use the same model-spend budget. TTT-Discover results are taken from its paper under its original training and compute settings. For kernel optimization (Table 4), we compare all methods at common model-spend budgets from $25 to $118. Each main-table entry

reports the best candidate found within the stated budget in a single run. Appendix D.4 explains how model spend is estimated from token usage and model-specific prices.

We use evaluators from prior work and correct vulnerabilities that allow candidates to obtain high scores without solving the intended task. Where we report scores from both the original and corrected evaluators, we mark the original scores separately and exclude results that exploit these vulnerabilities from performance claims. The corrections preserve the scoring of valid solutions. Appendix C describes the vulnerabilities and fixes.

4.1 Results

Systems and mathematical optimization. AgentDiscover matches or outperforms every baseline on all eleven tasks in Table 1, leading on seven and tying for best on four at common reporting precision. It achieves the best score on four of the five systems tasks: EPLB, LLM-SQL, transaction scheduling, and Cloudcast; and it reaches the known optimum on PRISM. On the mathematical tasks, it achieves the best score under the corrected signal-processing evaluator and improves on all baselines for both Erdős overlap and the third autocorrelation inequality. It matches the strongest results on circle packing, MMD-16-2, and MMD-14-3. These gains are obtained using the same backbone model across all methods.

Table 3: Comparison with published test-time RL finetuning results from Yuksekgonul et al. (2026). CORAL and AgentDiscover use frozen Claude Opus 5; AgentDiscover and CORAL use web access. AC1 entries are upper bounds on C_1.

Task	TTT-Discover	CORAL	AgentDiscover
AHC039 ↑	567,062	587,675	**588,114**
AHC058 ↑	848,414,228	851,222,618	**851,373,497**
AC1 ↓	1.50287	1.50282	**1.50274**

AtCoder heuristic contests. Without web access, AgentDiscover achieves scores that would have placed first in all five original AtCoder contests (Table 2). It also outperforms the compared search frameworks on four of the five tasks; CORAL scores higher only on AHC006. When post-contest submissions are included, AgentDiscover remains in the top five on every task, ranking first on two. For AHC025, we report the relative score from a fixed standings snapshot and provide absolute scores in Appendix E.

Comparison with test-time RL finetuning. With web access, AgentDiscover achieves higher scores on both AtCoder contests and a tighter AC1 bound than the published TTT-Discover results, without updating its model weights (Table 3). It also outperforms CORAL using the same backbone. Its programs would have placed first in both original contests, and its AC1 upper bound of 1.50274 improves on CORAL's 1.50282 and TTT-Discover's 1.50287.

Anthropic kernel builder. AgentDiscover achieves the lowest cycle count at every reported budget (Table 4 and Figure 1). With just $25 of model spend, it reaches 1,016 cycles, outperforming every baseline even at their full $118 budget. By $100, it reaches 960 cycles, compared with 1,098 for the strongest baseline, EvoX, a 12.6% reduction in execution cycles. This corresponds to a $153.9\times$ speedup over the naive kernel.

4.2 Additional Experiments

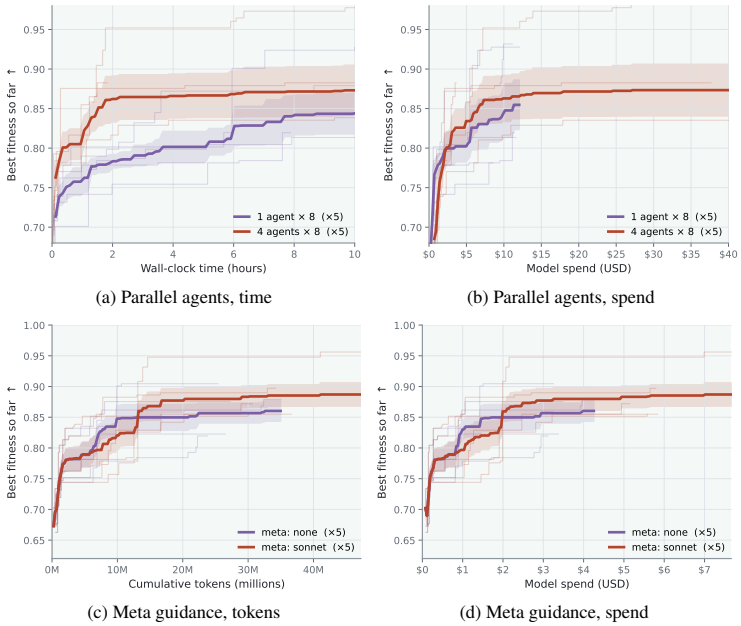
We study the effects of two components of AgentDiscover: parallel search agents, and meta-agent guidance. All experiments use transaction scheduling with Claude Haiku 4.5 as the search model.

Table 4: Kernel builder at common model-spend budgets. All methods use Claude Opus 5 with low reasoning effort and the same evaluator. Each entry is the lowest cycle count achieved at or below the indicated budget; lower is better. The best result at each budget is bold.

Budget	OpenEvolve	ShinkaEvolve	EvoX	CORAL	AgentDiscover
$25	1,248	1,344	1,225	1,257	**1,016**
$50	1,213	1,218	1,153	1,214	**976**
$75	1,178	1,210	1,119	1,157	**973**
$100	1,174	1,187	1,098	1,148	**960**
$118	1,174	1,156	1,098	1,148	**960**

Parallel search agents. AgentDiscover can run several search agents in parallel, all reading from and writing to the same candidate database through one MCP server. We compare one search agent with four, using five runs per setting. With four agents, the search converges faster in wall-clock time (Figure 4a), since the agents propose and evaluate up to four times as many candidates in the same time. At equal model spend, however, the two settings reach similar fitness (Figure 4b). Thus, parallel agents shorten the time to a good solution without reducing its cost, which makes them useful when wall-clock time matters more than model spend.

Meta-agent guidance. The meta agent periodically rewrites the guidance that search agents read before each iteration, using the shared database to summarize which approaches have and have not worked. We compare meta guidance from Claude Sonnet 5 with no meta agent, using a budget of 50 candidates per run and five paired seeds per setting. Paired seeds share the same first iteration, which removes its variance from the comparison. Meta guidance achieves a higher mean best fitness at equal token usage and equal model spend (Figure 4c,d). It wins on three of the five seeds and loses narrowly on the other two, so the improvement holds on average but not on every seed.



(a) Parallel agents, time (b) Parallel agents, spend

(c) Meta guidance, tokens (d) Meta guidance, spend

Figure 4: Parallel agents (a,b) and meta guidance (c,d) on transaction scheduling with Claude Haiku 4.5. Lines show the mean, bands the standard error, and faint lines individual runs.

5 Conclusion

We asked whether a search strategy chosen by a human before the run scales better than promoting the model from proposer to planner and letting it own the search. Our results favor the agent. The agent reached better scores at lower cost than the frameworks we compared against, would have placed first among human competitors in seven past AtCoder contests, and improved the best published bound on the first autocorrelation inequality. The improvement over fixed search algorithms comes, in our view, from the agent seeing the whole search rather than the few candidates a fixed algorithm would hand it, and from reasoning and experimenting over many steps rather than proposing each candidate in one shot. The improvement over CORAL, where an agent also runs the search, comes from managing context more efficiently over the long horizon. Along the way, we found that several evaluators inherited from prior work admit reward hacking, and corrected them (Appendix C).

Limitations. Each entry in our main tables is a single run, so we report no variance; small margins between methods should be read cautiously. Additionally, we ran AgentDiscover almost exclusively with Claude models; the framework supports other coding agents, but we have not varied the backbone model on a fixed task. Finally, we have not analyzed which search strategies the agents actually use: whether they rediscover classical rules such as MAP-Elites and UCT from the provided queries, or synthesize new ones.

AI USE STATEMENT

Coding agents built on Claude models are the object of study in this paper: they act as the search and meta agents of AgentDiscover and as the backbone of every baseline. That use is part of the method and is described in Section 3 and Appendix D.

Separately from the method itself, we used generative AI tools (Claude Code and ChatGPT) to help implement methods, namely to write and debug parts of the AgentDiscover codebase and the analysis and plotting scripts. We have not used generative AI tools to generate synthetic datasets, develop conceptual frameworks, formulate mathematical claims or proofs, propose hypotheses, design the methodology or experiments, or interpret results; translation, dataset cleaning, and qualitative data analysis are not applicable to this work. Additionally, we used generative AI tools to search for and summarize related literature, and to edit the paper for readability. We have reviewed all AI-assisted work. All AI-assisted code was read and tested by the authors, every reported number was checked against the stored run logs and evaluator outputs, and every citation was checked against the original source. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work involves no human subjects and no personal or sensitive data. All tasks are public benchmarks or past programming contests, used in line with their licenses and terms, and all baselines are run from their public releases and credited to their authors. Because AgentDiscover lets a coding agent run code autonomously for hours, we isolate execution: agents run in a sandbox without access to the evaluator, the database credentials, or web tools unless a setting states otherwise, and candidates are scored in a separate sandbox (Appendix A.4). We also found that several evaluators used in prior work can be reward hacked. We report these failure modes and our fixes (Appendix C) so that others can avoid them, and we exclude scores that exploit them from all performance claims.

REPRODUCIBILITY STATEMENT

We will release the anonymized source code of AgentDiscover, including the orchestrator, MCP server, evaluator sandbox, corrected evaluators, and run configurations, as supplementary material. Section 3 and Appendix A describe the architecture, the knowledge-graph schema, and the execution safeguards. Appendix B gives the full agent instructions, prompt templates, and MCP tool interface. Appendix C documents every evaluator correction we applied to the benchmark tasks. Appendix D lists the models, budgets, search schedules, and meta-agent settings for each experiment, as well as how the baselines were run. Since the backbone models are sampled stochastically, individual trajectories will vary between runs.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025. URL https://arxiv.org/abs/2507.19457.

Anthropic. Anthropic's original performance take-home. https://github.com/anthropics/original_performance_takehome, 2026.

Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, et al. AdaEvolve: Adaptive LLM driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

Terry Chen, Zhifan Ye, Bing Xu, Zihao Ye, Timmy Liu, Ali Hassani, Tianqi Chen, Andrew Kerr, Haicheng Wu, Yang Xu, Yu-Jung Chen, Hanfeng Chen, Aditya Kane, Ronny Krashinsky, Ming-Yu Liu, Vinod Grover, Luis Ceze, Roger Bringmann, John Tran, Wei Liu, Fung Xie, Michael Lightstone, and Humphrey Shi. AVO: Agentic variation operators for autonomous evolutionary search, 2026. URL https://arxiv.org/abs/2603.24517.

John H. Flavell. Metacognition and cognitive monitoring: A new area of cognitive–developmental inquiry. *American Psychologist*, 34(10):906–911, 1979.

Ali Essam Ghareeb, Benjamin Chang, Ludovico Mitchener, Angela Yiu, Caralyn J. Szostkiewicz, Dmytro Shved, Gavin J. Gyimesi, Jon M. Laurent, Samantha M. Wright, Muhammed T. Razzak, Andrew D. White, Silvia C. Finnemann, Michaela M. Hinks, and Samuel G. Rodriques. A multi-agent system for automating scientific discovery. *Nature*, 2026. doi: 10.1038/s41586-026-10652-y.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Petar Sirkovic, Artiom Myaskovsky, Grzegorz Glowaty, Felix Weissenberger, Alessio Orlandi, Dan Popovici, Anil Palepu, Keran Rong, Ryutaro Tanno, Khaled Saab, Fan Zhang, Jacob Blum, Andrew Carroll, Kavita Kulkarni, Nenad Tomašev, Dina Zverinski, Ivor Rendulic, Elahe Vedadi, Florian Hasler, Luka Rimanic, Marina Boia, Ivan Budiselic, Ben Feinstein, Mathias Bellaiche, Tom Sheffer, Jan Freyberg, Jeremy Ratcliff, Ottavia Bertolli, Katherine Chou, Avinatan Hassidim, Burak Gokturk, Amin Vahdat, Yuan Guan, Vikram Dhillon, Eeshit Dhaval Vaishnav, Byron Lee, Tiago R. D. Costa, José R. Penadés, Gary Peltz, Yossi Matias, James Manyika, Demis Hassabis, Yunhan Xu, Pushmeet Kohli, Annalisa Pawlosky, Alan Karthikesalingam, and Vivek Natarajan. Accelerating scientific discovery with Co-Scientist. *Nature*, 655(8122):487–496, 2026. doi: 10.1038/s41586-026-10644-y. URL https://www.nature.com/articles/s41586-026-10644-y.

Akshat Gupta, Jermaine Lei, Alexander Lu, Gopala Anumanchipalli, and Leshem Choshen. Automated discovery has no universally superior harness. *arXiv preprint arXiv:2607.18235*, 2026.

Yuki Imajuku, Kohki Horie, Yoichi Iwata, Kensho Aoki, Naohiro Takahashi, and Takuya Akiba. ALE-Bench: A benchmark for long-horizon objective-driven algorithm engineering. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025.

Yoichi Ishibashi, Taro Yano, and Masafumi Oyamada. Effective harness engineering for algorithm discovery with coding agents. *arXiv preprint arXiv:2605.15221*, 2026. URL https://arxiv.org/abs/2605.15221.

Andrej Karpathy. autoresearch: AI agents running research on single-GPU nanochat training automatically. https://github.com/karpathy/autoresearch, 2026.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European Conference on Machine Learning (ECML)*, 2006.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations*, 2026.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026. URL https://arxiv.org/abs/2603.28052.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alvin Cheung, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026a.

Shu Liu, Mert Cemri, Shubham Agarwal, Alexander Krentsel, Ashwin Naren, Qiuyang Mang, Zhifei Li, Akshat Gupta, Monishwaran Maheswaran, Audrey Cheng, Melissa Pan, Ethan Boneh, Kannan Ramchandran, Koushik Sen, Matei Zaharia, Alexandros G. Dimakis, and Ion Stoica. SkyDiscover: A flexible, adaptive framework for AI-driven scientific and algorithmic discovery. In *Proceedings of the ACM Conference on AI and Agentic Systems*, CAIS '26, pp. 1223–1227. Association for Computing Machinery, 2026b. doi: 10.1145/3786335.3813221. URL https://doi.org/10.1145/3786335.3813221.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The AI scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Sichun Luo, Yi Huang, Haochen Luo, Fengyuan Liu, Guanzhi Deng, Lei Li, Qinghua Yao, Zefa Hu, Junlan Feng, and Qi Liu. SeaEvo: Advancing algorithm discovery with strategy space evolution. *arXiv preprint arXiv:2604.24372*, 2026. URL https://arxiv.org/abs/2604.24372.

Jiacheng Miao, Joe R. Davis, Yaohui Zhang, Jonathan K. Pritchard, and James Zou. Paper2Agent: Reimagining research papers as interactive and reliable AI agents. *arXiv preprint arXiv:2509.06917*, 2025.

Ludovico Mitchener, Angela Yiu, Benjamin Chang, Mathieu Bourdenx, Tyler Nadolski, Arvis Sulovari, Eric C Landsness, Daniel L Barabasi, Siddharth Narayanan, Nicky Evans, et al. Kosmos: An AI scientist for autonomous discovery. *arXiv preprint arXiv:2511.02824*, 2025.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Junhao Qiu, Qinglong Hu, Xialiang Tong, Mingxuan Yuan, Liyong Lin, and Qingfu Zhang. AlgoEvo: Self-evolving agentic search for automated algorithm discovery. *arXiv preprint arXiv:2609.15820*, 2026.

Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, Shao Yong Ong, Fenglu Hong, Kaichen Zhou, Chonghe Jiang, Minwei Kong, et al. CORAL: Towards autonomous multi-agent evolution for open-ended discovery. *arXiv preprint arXiv:2604.01658*, 2026.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025. URL https://arxiv.org/abs/2504.15228.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Samuel Schmidgall and Michael Moor. AgentRxiv: Towards collaborative autonomous research. *arXiv preprint arXiv:2503.18102*, 2025.

Asankhaya Sharma. OpenEvolve: an open-source evolutionary coding agent. https://github.com/algorithmicsuperintelligence/openevolve, 2025.

Richard Sutton. The bitter lesson. http://www.incompleteideas.net/IncIdeas/BitterLesson.html, 2019.

Kyle Swanson, Wesley Wu, Nash L. Bulaong, John E. Pak, and James Zou. The virtual lab of AI agents designs new SARS-CoV-2 nanobodies. *Nature*, 646:716–723, 2025. doi: 10.1038/s41586-025-09442-9.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, et al. ThetaEvolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.

David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

Guowei Xu, Zhenting Qi, Huangyuan Su, Weirui Ye, Himabindu Lakkaraju, Sham M. Kakade, and Yilun Du. Self-improving language models with bidirectional evolutionary search. *arXiv preprint arXiv:2605.28814*, 2026. URL https://arxiv.org/abs/2605.28814.

Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The AI scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *arXiv preprint arXiv:2504.08066*, 2025.

Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Shuo Chen, Zhankui He, Noveen Sachdeva, Isabella Ye, Weili Wang, Chi Wang, Ed H. Chi, Fernando Pereira, Wang-Cheng Kang, Derek Zhiyuan Cheng, and Beidou Wang. PACEvolve: Enabling progress-aware consistent evolution. *arXiv preprint arXiv:2601.10657*, 2026. URL https://arxiv.org/abs/2601.10657.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026. URL https://arxiv.org/abs/2605.23904.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. *Advances in neural information processing systems*, 37:43571–43608, 2024.

Haotian Ye, Haowei Lin, Jingyi Tang, Yizhen Luo, Rahul Thapa, Caiyin Yang, Chang Su, Rui Yang, Ruihua Liu, Rundao Li, Zeyu Li, Pengwei Sun, Chong Gao, Dachao Ding, Guangrong He, Miaolei Zhang, Lina Sun, Wenyang Wang, Yuchen Zhong, Zhuohao Shen, Puheng Li, Pan Lu, Bianxiao Cui, Di He, Jianzhu Ma, Junfeng Li, Hexi Baoyin, Yejin Choi, Stefano Ermon, Xiaowen Chu, Tongyang Li, Yuzhi Xu, and James Zou. Structured scaling of AI discovery across diverse scientific domains, 2026. URL https://arxiv.org/abs/2604.19341.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in LLM-based automatic heuristic design. *arXiv preprint arXiv:2501.08603*, 2025. URL https://arxiv.org/abs/2501.08603.

A Implementation Details

This appendix adds implementation details to the components of §3.2. Prompt templates and tool descriptions are in Appendix B, and the settings of our experiments in Appendix D.

A.1 Architecture and Run Lifecycle

Processes and setup. The trusted side is the orchestrator, a Neo4j database whose data persists between runs, and the MCP server, a host process. Only this trusted code holds the database password, and the database and the server listen on loopback only. Containers are started through one sandbox module that supports Docker, Podman, Apptainer/Singularity, and Enroot. Before any agent starts, the orchestrator starts the database and the MCP server and builds the evaluation environment. Each branch takes its starting iteration from the database, so a new invocation continues a branch where it stopped. A new branch's worktree (its workspace in §3.2) is a fresh git repository, not connected to the main repository's history, holding a copy of the problem's `solution.py` and empty `tools/` and `skills/` folders.

Iterations. Each branch runs its sessions in sequence in its own loop, in parallel with the other branches; iterations are grouped into blocks, and all branches wait at a barrier at the end of a block before the meta pass. To open a session, the orchestrator renders the instruction file from a template with the branch's current guidance last, builds a short briefing from fixed queries (Appendix B), and creates the `Session` node. The coding-agent CLI then runs non-interactively (`claude -p` or `codex exec`) in the branch's container. The session ends when the CLI exits or its wall clock expires; a crash or timeout is recorded and the run continues. On close, the orchestrator stores the reasoning text from the transcript on the `Session` node, writes the exposure edges (§A.2), and commits the worktree.

Table 5: Node labels of the knowledge graph, with what each node holds.

Label	Key	Holds
`User`	`name`	The person who owns the agents.
`Agent`	`key`	One branch: a persistent search-agent identity and the path of its worktree.
`Session`	`id`	One search-agent session: its iteration, model and briefing, its reasoning text (`thinking_text`), start and end times, the fingerprint of its call token (`token_hash`), its own lesson (`lesson_learned`), and the meta agent's judgment of it (`meta_lesson`).
`Candidate`	`genome_hash`	One unique program (`genome`, the full `solution.py`), its `fitness` and evaluation status, one property per evaluator metric, and the agent's predictions, made before scoring.
`Idea`	`name`	A named idea, reused across candidates, with a description.
`Resource`	`name`	A file the problem provides, and where agents find it.
`Guidance`	`id`	One version of a branch's strategy, with a summary of what changed; the first is a seed, and later ones are written by the meta agent.
`Tool`	(`name`, `branch`)	A read-only query the meta agent saved for one branch: its description, Cypher code, and input schema.
`RunConfig`	`problem`	The settings the current run was launched with, which every process reads back.

Table 6: Knowledge-graph relationships and their writers. Endpoints are shown as source → target; *facts* are reconstructed from the session log, whereas *claims* are the counterpart declared by the search agent.

Writer / provenance	Relationships
orchestrator	`OWNS` (User→Agent), `RAN` (Agent→Session), `UNDER_GUIDANCE` (Session→Guidance), seed `HAS_GUIDANCE` (Agent→Guidance)
orchestrator (*facts*)	`RETRIEVED` {`via`} (Session→Candidate), `READ` (Session→Resource)
server	`PRODUCED` (Session→Candidate), `USED_TOOL` (Session→Tool)
search agent	`USES_IDEA` (Candidate→Idea), `RELATED_TO` {`how`} (Idea→Idea)
search agent (*claims*)	`DERIVED_FROM` {`kind`} (Candidate→Candidate), `USED_CONTEXT` (Candidate→Resource)
meta agent	`HAS_GUIDANCE` (Agent→Guidance), `HAS_TOOL` (Agent→Tool), `SUPERSEDES` (Guidance→Guidance)

State across sessions. A branch keeps its worktree (with `solution.py` as the last session left it, `tools/`, and `skills/`), the worktree's git history, its guidance, the tools the meta agent wrote for it, and the shared graph. Each session is a new CLI process with a re-rendered instruction file. One exception: a session cut off because the model provider's usage window ran out is resumed with its transcript once the window resets.

A.2 Knowledge Graph Schema

Tables 5 and 6 list the node labels and relationships. Every key has a uniqueness constraint. A candidate's key is the SHA-256 of its program text, so a resubmitted program never becomes a second node; its `fitness` is the evaluator's score, and each further metric the evaluator reports is a property of its own.

Facts and claims. Edges written by the orchestrator or the server record what happened; the agent cannot write them. Edges written by the search agent are its declarations, which the server checks only for existence. Two declarations have a logged counterpart: a candidate's `DERIVED_FROM` and `USED_CONTEXT` edges claim which candidates and files it came from, while the session's `RETRIEVED` and `READ` edges, reconstructed by code from its transcript, record which it actually fetched.

A.3 META PASS

The optional meta pass runs at the barrier between blocks; a single meta agent serves all branches, and if the pass fails, the next block keeps the guidance already in place. Its container mounts each branch's transcripts read-only and gets neither the evaluator, the database password, any `solution.py`, nor any git history. Besides the skill files of §3.2, it writes through three tools of its own: `write_meta_lesson` stores its judgment of one session on that `Session` node; `add_tool` stores a read-only saved query for one branch, which the branch's later sessions are offered as an MCP tool; and `write_strategy` stores a branch's new strategy as a `Guidance` node that supersedes the previous one and is rendered into the branch's next instruction file.

A.4 EXECUTION SAFEGUARDS

- **Evaluator isolation.** The evaluator is never mounted into an agent's container, and a worktree's git history cannot contain it. The server reads `solution.py` from the agent's worktree itself and evaluates a copy in a separate container, under limits set per problem and without network access (except on the Enroot and `none` backends, which cannot disable it). Hidden evaluation data is fetched before any candidate exists, into a cache no agent can see.
- **Agent containers.** A search agent's container holds its own worktree and the problem's public files, and never the database password. The CLI's web tools are removed by default, and the prompt forbids looking up results online (Appendix B); the container keeps the network access the CLI needs to reach its model API.
- **Read-only queries, typed writes.** `read_graph` runs in a Neo4j read transaction, so the database itself rejects any write; all writes go through the fixed tools.
- **Attribution.** The server attributes every call to a session itself, never trusting what an agent says about itself, and offers each caller only its own tools.

B AGENT PROMPTS AND TOOLS

The search and meta agents receive different instructions and tool sets. The boxes below preserve the wording of the consequential passages while condensing long lists.

B.1 SEARCH-AGENT INSTRUCTIONS

At the start of each search session, the orchestrator creates `CLAUDE.md` or `AGENTS.md` by substituting `{{PROBLEM}}`, `{{SCHEMA}}`, `{{FOLDER_LISTING}}`, and `{{STRATEGY}}` in one template. Box B.1 selects the parts that define the agent's objective, interface, persistent state, and search loop.

Box B.1: Selected search-agent instructions (abridged)

AgentDiscover
You are working on an evolutionary search over `solution.py`, building on work that is already done. A shared database holds every candidate submitted so far, the ideas behind them and the lessons written about them; your working directory holds `tools/` and `skills/` left by earlier work. Explore the database and those folders first, then either build on what you find or pursue your own idea.

Problem
`{{PROBLEM}}`

What you are judged by
Your briefing names a **target**. Beating it, preferably by a clear margin, is what this run is for. Your job is to push the best number in the database past it, as far past as you can get within this session's cap.

The target is also your yardstick. After each candidate you see your score and the distance still to go: judge from that whether your current idea can realistically reach the target, or whether it is levelling off short of it. If it is levelling off, change idea rather than refine it further.

Your tools

`read_graph(cypher)`—run any read-only Cypher query and get rows as JSON.
`add_idea(name, description, related_to)`—record the idea a candidate will use, or reuse an existing idea by exact name.
`submit_candidate(idea, linguistic_prediction, fitness_prediction, parents, resources)` —submit the current `solution.py`; the server evaluates it in a locked sandbox and returns the score.
`write_lesson(text)`—after your last candidate, write one lesson for the whole session.
Your whole working directory persists and is committed after every session. Put reusable code in `tools/`; `skills/` contains notes, written from the evidence so far, on how the search on this problem has gone.

The graph you can read

This is a graph database, not a table: `read_graph` takes any read-only Cypher, so follow relationships across several hops rather than only listing nodes.
`{{SCHEMA}}`

The session loop

Your prompt names the maximum number of candidates for this session. It is a cap, not a quota. Push the fitness with every candidate. If your improvements look like they are levelling off short of the target, switch to a different idea rather than tweak the same one further.

1. **Explore**—briefing first, then `read_graph`. Before your first candidate, read the most recent lessons; explore beyond that as much as is useful.
2. **An idea** (mandatory)—`add_idea` a new one, or pick an existing idea's exact name to reuse.
3. **A candidate** (mandatory)—edit only the marked evolve block of `solution.py`; test locally; decide its parents; write both predictions before submitting; then call `submit_candidate` and read the score it returns.
4. Loop back, building on what the scores taught you.

When you stop—at the cap, or earlier because different ideas all failed—call `write_lesson` with one lesson covering the whole session. Then stop; the orchestrator takes over from there.

Honesty rules

Your declared parents, idea, and resources are recorded as your claims about how the candidate was made. What you actually retrieved and read is recorded separately, by the platform, from your session log.

Persistent artifacts and current strategy

`{{FOLDER_LISTING}}`

`{{STRATEGY}}`

B.2 META-AGENT INSTRUCTIONS

After each block, the orchestrator renders the meta template by substituting the problem and graph schema, the block and session sizes, branch paths, and a block summary. The meta agent sees the search traces and outcomes but not the evaluator. Box B.2 selects the instructions that define its evidence, outputs, and boundary with the search agent.

Box B.2: Selected meta-agent instructions (abridged)

Role

You are the meta-learning step, not one of the search branches. You run after all search agents have paused for this block. Search agents are short-lived: each is spawned fresh with no memory, proposes a few candidate `solution.py` programs, and stops. Your job is to find **which ways of thinking paid off**, and to make them cheap for the next fresh agents.
Whatever you write for a branch stays in force for `{{META_EVERY}}` sessions, each proposing `{{PROPOSALS}}` candidates. That horizon tells you how long your claims must keep being true; it is not a budget for you to allocate.

What you can reach

The shared history graph through `read_graph`; each branch's raw session logs (read-only); and each branch's `skills/` directory (writable). You do not have any genome, `solution.py`, the evaluator, or database credentials. You shape *how* branches search, never the genomes directly.

Branches: `{{BRANCHES}}`

Briefing for this block: `{{BRIEFING}}`

What to read

The useful material is always **pairs**—thinking joined with outcome, because thinking alone is stories and fitness alone is numbers. Compare session reasoning with candidate fitness, predictions with results, idea and lineage reuse, tool use, and performance under earlier guidance.

What you write—per branch

1. `write_meta_lesson(session_id, text)`—for each session you reviewed: your judgment of how well it searched, independent of the score it got.
2. `add_tool(...)`—a **saved query**, and only that: a read-only Cypher template exposed to later sessions as a named tool.
3. **Skills**—short Markdown files of methodological guidance from behavioural evidence: how the branch tends to fail, what exploration paid for, and what need not be re-derived.
4. `write_strategy(...)`—what ways of thinking pay off for this branch and what ways do not, written to the next fresh agent.

The line between your job and theirs

Deciding what to try next is the search agent's job. Deciding how to think about the problem is yours. Do not write candidate allocations, ranked lists of approaches, or implementation into a strategy. A test before you write: *would this still make sense to an agent that finds something you did not anticipate?*

B.3 Session Briefing and Feedback

The orchestrator's command-line prompt for the search agent adds a candidate cap, the internet policy, and a briefing built by fixed database queries. With web access off (the default), the internet rule reads "Never reach out to the internet in any way (web search, curl, wget, git clone, or otherwise) to look up other results for this problem."; otherwise "You may search the internet if you want." Each submission returns a code-generated steering message from the MCP server. Box B.3 shows their stable forms and the cost note; angle-bracketed fields are runtime values, and other optional variants are omitted.

Box B.3: Search-session prompt and server feedback (abridged)

Invocation prompt

Submit up to ⟨ *proposals* ⟩ candidates with the `submit_candidate` tool.

Do not reply with a plan or a summary of what you intend to do: a reply that calls no tool ends your session immediately, and a session that submits nothing is wasted. Explore as much as you need, then edit `solution.py` and submit.

⟨ *internet rule* ⟩

Briefing

Best fitness in your branch: ⟨ *value* ⟩

Best in the database: ⟨ *value* ⟩

Best ⟨ *objective name* ⟩ in the database: ⟨ *value* ⟩ (target to beat clearly: ⟨ *target* ⟩)

The database holds ⟨ *n* ⟩ candidates and ⟨ *m* ⟩ ideas.

⟨ *k* ⟩ search agents are running in parallel.

This whole session is capped at ⟨ *wall* ⟩ s (approximately ⟨ *hours* ⟩ h)—budget your candidates against it; a session cut off at the cap loses its lesson.

Reply below the cap

⟨ *objective name* ⟩ = ⟨ *value* ⟩ (you predicted ⟨ *prediction* ⟩) (target ⟨ *target* ⟩, ⟨ *gap* ⟩% away). fitness ⟨ *fitness* ⟩. That was candidate ⟨ *n* ⟩ of at most ⟨ *cap* ⟩. Earlier this session: ⟨ *earlier values with predictions* ⟩. Best this session: ⟨ *value* ⟩. Best in the database: ⟨ *value* ⟩. You can read the graph again, and use or add to your persistent `tools/` folder. Work on your next candidate: refine what shows promise, or if these numbers say you are stuck, switch to a different idea. Stop early only if you have tried different ideas and none worked—then write the lesson. About ⟨ *hours* ⟩ h of this session's wall clock remain (⟨ *seconds* ⟩ s). ⟨ *cost note* ⟩

Cost note (from the fifth candidate, when a fresh session would be cheaper)
The token cost of continuing this session will probably be higher than the cost of starting a new one (this session's context has grown to ⟨ *L* ⟩ tokens; a fresh session would start from ⟨ *c* + *i* ⟩). Continue only if you have a great idea that is very likely to give a good result; otherwise write the lesson and terminate, and you will get another chance to try in a new session.

Reply at the cap
That was candidate ⟨ *n* ⟩ of at most ⟨ *cap* ⟩—the cap. Now write the lesson for this session with `write_lesson`—include the work that never became a candidate, since only submitted ones are in the graph—then stop.

B.4 Tools

Table 7 lists the MCP interface. The "saved query" in the meta prompt is a named, branch-specific, read-only Cypher query registered with `add_tool`; we call it a *meta-written query tool* below. Later search sessions see its name, description, and argument schema, but not its Cypher implementation.

Table 7: MCP tools available to search and meta agents.

Tool	Caller	Arguments	Effect
`read_graph`	**both**	`cypher`	Runs a read-only Cypher query and returns rows as JSON.
`add_idea`	**search**	`name`, `description`, `related_to`	Creates or reuses an idea and records its relations to existing ideas.
`submit_candidate`	**search**	`idea`, predictions, `parents`, `resources`	Evaluates the current `solution.py`, records its provenance and result, and returns steering feedback.
`write_lesson`	**search**	`text`	Stores the session's final lesson.
`write_meta_lesson`	**meta**	`session_id`, `text`	Stores the meta agent's assessment of one session's search process.
`add_tool`	**meta**	`branch`, `name`, `description`, `code`, `params`	Registers a read-only Cypher query as a tool for one branch.
`write_strategy`	**meta**	`branch`, `strategy`, `diff_summary`	Replaces a branch's guidance for later sessions.
meta-written query tool	**search**	arguments declared by its schema	Runs the branch-specific query registered by the meta agent.

C Evaluator Failure Modes and Corrections

The evaluation sandbox (Appendix A.4) keeps the evaluator and its private files outside the agent's workspace. The submitted program must nevertheless run somewhere, and the evaluator must determine whether its output is a valid solution. While porting the benchmark evaluators, we repeatedly encountered the following failure modes. We report these recurring patterns rather than attempt an exhaustive audit of every evaluator or upstream version.

Incomplete output validation. *Affected tasks: Signal Processing, LLM-SQL.* An evaluator may compute a favorable score for an output that does not satisfy the constraints of the stated task if it checks only that the output can be parsed. On Signal Processing, the required output length was

never enforced: a candidate that returned only two samples, with no slope reversals and near-perfect correlation, scored 0.9259 on the evaluator, although it is not a solution to the task at all. Even with the length enforced, the metric rewards smoothness and penalizes the hidden clean signal for its own oscillations, so an oversmoothed output can score above the exact ground truth, which scores 0.727. On LLM-SQL, the returned table was never checked to be a rearrangement of the input rows: a candidate that discarded the input and returned one long constant row n times made every row after the first a full prefix-cache hit and scored 0.994, against 0.737 for the best honest reordering we found. We therefore check the output's shape, contents, completeness, and resource constraints against authoritative task data before scoring it.

Candidate-controlled scoring. *Affected tasks: Signal Processing, 3rd-Autocorr., PRISM, and LLM-SQL result loading.* If candidate code runs in the process that computes the score, it can rebind the metric functions, inspect scorer state, or simply print a result in the evaluator's name. On Signal Processing, a candidate that replaced `pearsonr` with a function returning 10^9 was scored 2×10^8, and one that printed {`"score": 1e9`} and exited had that value reported at face value. Loading a candidate-controlled serialized object into the scoring process creates the same boundary failure: on LLM-SQL the returned DataFrame was exchanged via `pickle`, whose deserialization executes code the candidate chooses. We run candidate code separately, exchange only the proposed solution, restrict deserialization to the expected data types, and compute the score in a process the candidate did not enter.

Information leakage across the boundary. *Affected task: EPLB.* EPLB scores a placement on the *next* recorded workload, so the workload data must stay hidden: a candidate that sees the future no longer has to predict it. The agent nevertheless found two ways to obtain it. First, although the agent's own sandbox blocks the data, the candidate program it submits ran where the file was readable, and candidates that read it raised the balance score from 0.25 to 0.44; a keyword blocklist did not stop this, since a file path can be built indirectly. Second, whatever a candidate prints was returned to the agent as feedback, and the same workloads are replayed in the same order on every evaluation. Candidates printed their inputs, and later candidates hardcoded the answers, lifting the score from a best honest 0.153 to 0.201 in our strongest run. Candidate processes therefore receive only public task inputs (the private data is replaced by an empty directory), and their stdout and exception messages are withheld from search feedback.

D Experimental Details

This appendix records the model and run configuration for each evaluation setting in §4. All AgentDiscover runs use the Apptainer/Singularity backend of the sandbox described in Appendix A; only settings that vary by experiment are listed below.

D.1 Models, Access, and System Budget

Table 8 reports the model, access policy, and system-level budget. The reasoning-effort setting is shared by the search and meta agents; Table 10 reports the meta model and schedule.

Table 8: Model, access, and system-level budget ceiling by evaluation setting. All listed models are Claude models. "On" for test-time finetuning applies to both the CORAL and AgentDiscover runs.

Setting	Search model	Effort	Web	Budget ceiling
Systems + math	Opus 4.6	default	off	100 candidates
AtCoder (5 contests)	Opus 5	default	off	matched model spend
Test-time finetuning	Opus 5	default	on	matched model spend
Kernel optimization	Opus 5	low	off	matched model spend
Parallel-agent ablation	Haiku 4.5	default	off	matched wall time / model spend
Meta-agent ablation	Haiku 4.5	default	off	50 candidates; tokens / model spend

The systems and mathematical baselines are published results compiled by Qu et al. (2026); all methods use Claude Opus 4.6. For the five AtCoder contests, we run OpenEvolve, EvoX, CORAL,

and AgentDiscover with Claude Opus 5 and score their programs on the official system tests. For AHC039, AHC058, and the first autocorrelation inequality, TTT-Discover retains its published training and compute setting, while CORAL and AgentDiscover use frozen Claude Opus 5 under the same model-spend cap. All five kernel methods use Claude Opus 5 at low reasoning effort and the same evaluator.

For baseline runs we conduct ourselves, OpenEvolve, ShinkaEvolve, and EvoX run through SkyDiscover (Liu et al., 2026b) with their default configurations, and CORAL uses its default configuration. We change only the model and budget settings required by the comparison protocol; for settings without web access, we also disable web access in CORAL's sandbox configuration.

Candidate budgets count submitted programs, including those whose evaluation fails or times out; only valid candidates can be reported. Model spend is computed consistently for every method (Appendix D.4). It is not a raw token count, and includes meta-agent calls when enabled. Kernel results use the best-so-far candidate at common spend checkpoints; wall-clock time is not used as a proxy for model usage. Evaluator corrections are described in Appendix C.

D.2 Search Schedule and Stopping Rules

Table 9 reports the maximum scheduled search work. Every setting runs a single search branch, except the parallel-agent ablation, which compares one branch with four. Each iteration runs one search session on every active branch. Runs continue until the shared system-level budget in Table 8 is exhausted. The maximum-iterations setting in Table 9 is a non-binding safety ceiling, not the comparison budget.

The candidates-per-session cap controls context length, not the comparison budget. A session ends when the agent finishes, reaches this candidate cap, or reaches its timeout; reaching the candidate cap starts another session when one remains. The launcher may raise the configured search timeout when it cannot accommodate the candidate cap and evaluator's per-candidate limit. Candidate evaluation has a separate task-specific timeout.

Each main-table entry is the best valid candidate from one run, so the main comparisons report no variance. The component ablations use five repeats per condition and report means and standard errors; paired meta-ablation runs share their first session.

Table 9: Search schedule and wall-clock ceiling by evaluation setting.

Setting	Cand. / session	Max iter.	Search timeout (s)
Systems + math	25	5	28,800
AtCoder (5 contests)	20	10	36,000
Test-time finetuning	20	5	57,600
Kernel optimization	10	10	28,800
Parallel-agent ablation	8	10	14,400
Meta-agent ablation	10	5	28,800

D.3 Meta-Agent Schedule

Table 10 reports the meta model, schedule, and timeout. One meta pass runs after each completed search iteration.

Table 10: Meta-agent configuration by evaluation setting.

Setting	Meta model	Meta timeout (s)
Systems + math	Opus 4.6	7,200
AtCoder (5 contests)	Opus 5	3,600
Test-time finetuning	Opus 5	5,400
Kernel optimization	Opus 5	7,200
Parallel-agent ablation	Haiku 4.5	3,600
Meta ablation (meta)	Sonnet 5	3,600
Meta ablation (no meta)	none	n/a

The meta timeout is an independent ceiling: a pass can finish earlier, and a timeout skips that pass without terminating search. No pass is launched after the final iteration because no later session could use its guidance.

D.4 Model-spend accounting

We estimate model spend from the token usage recorded in Claude Code transcripts for each task. For every assistant message, we price input, output, cache-read, and cache-write tokens at the published rates for the model used, accounting separately for 5-minute and 1-hour cache writes. We sum message costs across all search and meta agents in a run to obtain cumulative model spend. These values are estimates at list prices, not invoice charges, because the runs used a Claude subscription.

E Additional AtCoder Results

Official submissions. The AgentDiscover programs evaluated on the five contests in Table 2 are available for AHC006, AHC012, AHC015, AHC025, and AHC041. The programs used for the test-time finetuning comparison in Table 3 are available for AHC039 and AHC058.

E.1 Additional Spend–Performance Curves

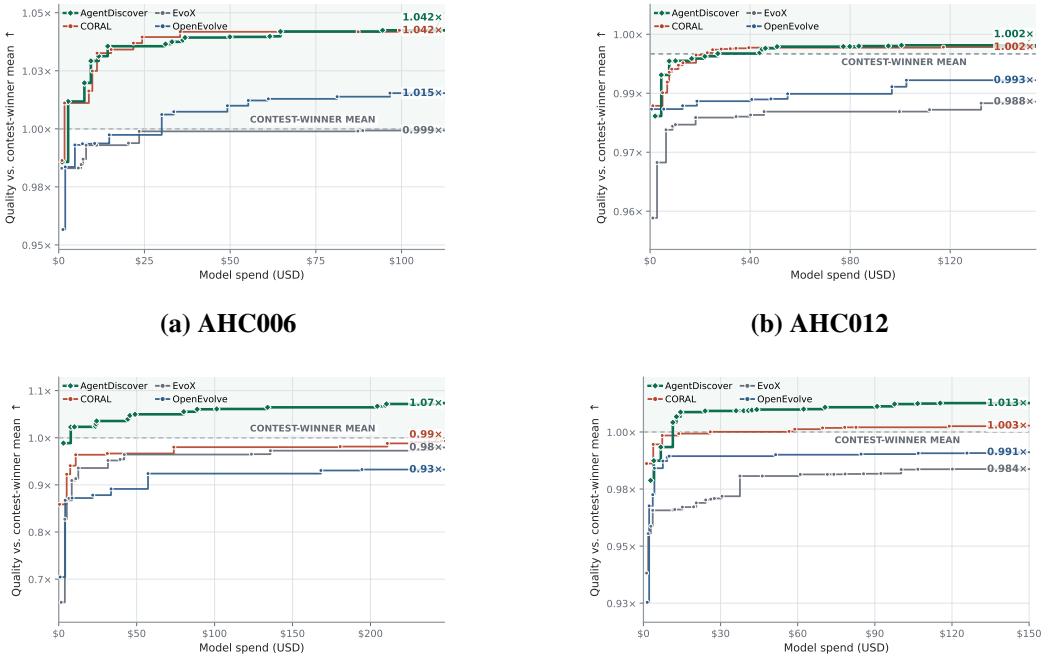
Figure 5 shows best-so-far performance against billed model spend on AHC006, AHC012, AHC015, and AHC041, measured on the public evaluation cases. AgentDiscover finishes highest on all four, and its lead is clear on AHC015 and AHC041. On AHC006 and AHC012, AgentDiscover and CORAL finish within rounding of each other, and on AHC006 CORAL's program scores higher on the official system test (Table 2).

E.2 AHC025 Scoring

AHC025 is scored differently from the other contests in Table 2 and Table 3. Each test case has a fixed *absolute score* (lower is better), but the standings rank programs within each case and sum rank-based scores, so a program's *relative score* changes as others submit. Table 2 therefore reports relative scores and ranks from a single extended-standings snapshot, while the linked submission page shows the fixed sum of absolute scores.[1]

Table 11 reports each method's fixed absolute score alongside its relative score, including ShinkaEvolve, which we ran on AHC025 only; the contest rank counts only original contest entrants whose relative score exceeds the submission's. AgentDiscover attains the lowest absolute score (19,173,103, against 25,923,844 for the strongest baseline, CORAL), and the ordering of methods is the same under both scores.

[1] Official AHC025 scoring rules: `https://atcoder.jp/contests/ahc025/tasks/ahc025_a?lang=en`.



(a) AHC006 **(b) AHC012**

(c) AHC015 **(d) AHC041**

Figure 5: Best-so-far performance against billed model spend on four further AtCoder heuristic contests, measured on the public evaluation cases. Each objective is normalized by the corresponding contest-winner mean, so higher is better and $1\times$ is the reference value.

Table 11: AHC025 submitted-program results. Absolute score is fixed and lower is better. Relative score and ranks are from the same mutable extended-standings snapshot as Table 2 and are higher-is-better.

Method	Absolute ↓	Relative ↑	Contest	Extended
OpenEvolve	95,926,386	92,596,442,685	37th	51st
EvoX	39,086,466	94,461,264,820	21st	30th
ShinkaEvolve	31,925,711	95,932,806,317	12th	18th
CORAL	25,923,844	98,498,020,580	2nd	6th
AgentDiscover	**19,173,103**	**98,762,866,188**	**1st**	**3rd**

Figure 6 compares all five methods on the AHC025 absolute score as a function of model spend. AgentDiscover reaches the contest-winner reference after \$22.34 of model spend, against \$72.97 for the strongest baseline. At the full budget, AgentDiscover finishes at $1.96\times$ the contest-winner quality against CORAL's $1.72\times$, a 12.3% lower objective.

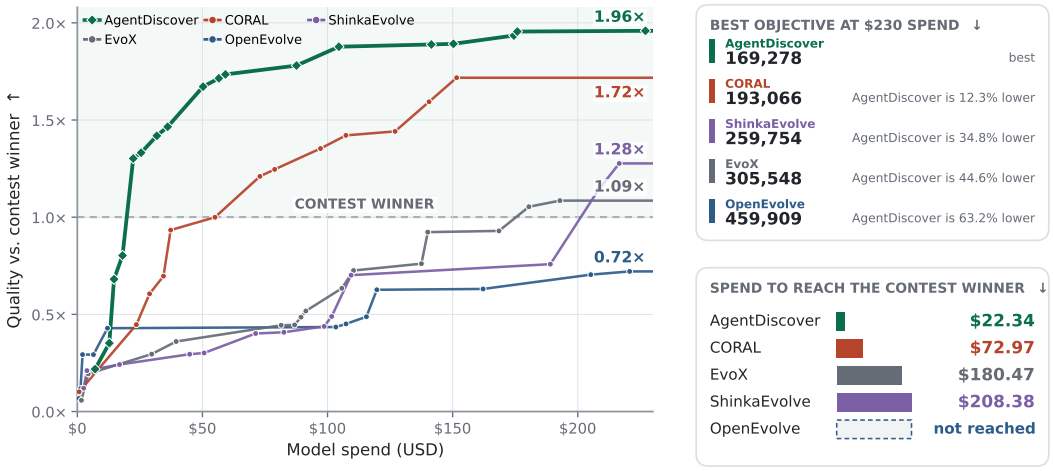


Figure 6: AHC025 fixed-objective comparison at matched model spend. The trajectories and the upper-right summary compare methods by their best mean absolute objective on the same 50 public evaluation cases (lower objective is better). The dashed $1\times$ reference uses the contest winner's mean over the 5,000 official system-test cases, whereas the trajectories use the 50 public cases. Official leaderboard (relative) scores are reported in Table 2.