Effective Harness Engineering for Algorithm Discovery with Coding Agents

Yoichi Ishibashi [1] **Taro Yano** [1] **Masafumi Oyamada** [1]

Abstract

AlphaEvolve and FunSearch have demonstrated the potential of combining large language models (LLMs) with evolutionary search for automated algorithm discovery. However, discovery success is shaped not only by model capability but also significantly by the design of the execution infrastructure, i.e., the harness. This paper investigates effective harness design through three questions: under a fixed token budget, is it better to produce many algorithms with brief thought or fewer algorithms with deeper thought? How should the harness handle evaluation hacks, where generated programs exploit the scoring function? And how can agents that require full filesystem access execute safely in parallel? Using Vesper, an algorithm discovery framework that incorporates harness improvements addressing these questions, we evaluate on Circle Packing under the same token budget. Interestingly, generating fewer algorithms while thinking more deeply about each one achieved higher scores. That is, scaling the quality of each individual is more budget-efficient than scaling the number of evolutionary generations. Surprisingly, more capable models produced evaluation hacks at higher rates, making hack detection increasingly necessary as models scale.

1. Introduction

Scientific and engineering progress is often driven by the discovery of superior algorithms. Yet such algorithm design still relies heavily on manual effort by domain experts, and AI-driven automation of discovery is beginning to change this landscape (Zheng et al., 2025). In particular, the approach of using large language models (LLMs) as mutation operators in evolutionary search, iteratively creating algorithms that surpass human designs through cycles with an automatic evaluator, has achieved remarkable results. FunSearch evolved function-level algorithms with LLMs, generating mathematical constructions surpassing the best known results for the cap set problem (Romera-Paredes et al., 2024). Beyond improving algorithms alone, alternately improving both the generated algorithms and the generating model itself through reinforcement learning has also been explored (Ishibashi et al., 2024). More recently, AlphaEvolve (Novikov et al., 2025) expanded the scope of improvement to entire codebases and improved matrix multiplication algorithms that had remained unimproved for over 50 years. This technology is applicable to any domain where algorithms play a central role, including route optimization, chip placement design, and molecular generation in drug discovery, and may replace manual expert optimization.

What determines the success of algorithm discovery is not agent capability alone; the design of the agent infrastructure, i.e., the harness, has a substantial impact. The harness refers to the entire execution infrastructure that guides agents toward discovery, encompassing prompt construction, data design, evaluation pipelines, and parallel agent management. AlphaEvolve's success is supported not only by Gemini's model capability but also by the harness design as a whole, including randomized prompt switching, evaluation cascades, and an evolutionary database combining MAP-Elites with island-based population models.

Examining existing open-source implementations from this harness perspective reveals substantial room for improvement. Representative implementations such as OpenEvolve[1] and CodeEvolve (Assumpção et al., 2025) all use LLMs as stateless code generators through single-shot API calls, without leveraging the capabilities of coding agents that have rapidly advanced since 2025. Just as coding agents such as Claude Code[2] and Codex CLI[3] have transformed productivity in software development, differences in harness implementation may fundamentally change the practicality of algorithm discovery.

[1]NEC Corporation. Correspondence to: Yoichi Ishibashi <yoichi-ishibashi@nec.com>.

Accepted by the ICML 2026 AI for Science Workshop.

[1]https://github.com/algorithmicsuperintelligence/openevolve

[2]https://github.com/anthropics/claude-code

[3]https://github.com/openai/codex

From these observations, we identify four practical challenges in existing harnesses. First, LLMs remain stateless code generators, without leveraging autonomous reasoning such as analyzing evaluation results, referencing the entire codebase, or planning multi-step corrections. Second, there is no mechanism to detect "evaluation hacking" (Amodei et al., 2016), where generated algorithms exploit flaws in the evaluation function rather than genuinely solving the problem. Third, running multiple search agents in parallel on a shared filesystem risks file conflicts and race conditions that compromise reproducibility. Fourth, each iteration starts from a blank slate, without systematically leveraging knowledge of successes and failures from past trials.

Beyond these challenges, a fundamental question about search strategy remains unanswered. Algorithm discovery is inherently a search problem whose success scales with the budget invested. The practical constraint is that every search step costs tokens, so token efficiency directly determines the scalability of discovery. Under a fixed token budget, is it better to produce many algorithms with brief thought, or fewer algorithms with deeper thought? Whether investing more thought per algorithm at the expense of fewer generations improves overall discovery performance requires empirical verification.

This paper proposes Vesper, a harness for LLM-driven algorithm discovery that integrates several candidate improvements, including coding agent integration (§ 4.2), evaluation hack detection (§ 4.3), and Git worktree isolation (§ 4.5), and empirically investigates which components matter and why. Interestingly, scaling the quality of each individual proved more budget-efficient than scaling the number of evolutionary generations (§ 5). We also find, surprisingly, that more capable frontier models such as GPT-5.2 are more prone to evaluation hacking.

2. Related Work

LLM-Driven Algorithm Discovery The dominant paradigm in LLM-driven algorithm discovery shares the common approach of incorporating LLMs into evolutionary operators as stateless generators that return single-shot code completions in response to prompts. FunSearch (Romera-Paredes et al., 2024) surpassed the best known results on the cap set problem, and AlphaEvolve (Novikov et al., 2025) improved matrix multiplication algorithms that had remained unimproved for over 50 years. Numerous specialized frameworks have also emerged, including EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), LLaMEA (van Stein & Bäck, 2025), and AEL (Liu et al., 2023). In open source, OpenEvolve reproduces AlphaEvolve's workflow, followed by CodeEvolve (Assumpção et al., 2025) and DeepEvolve (Liu et al., 2025).

Coding Agents Since 2025, command-line coding agents that operate autonomously have emerged in rapid succession, including Codex CLI, Claude Code, Gemini CLI[4], and Qwen Code[5]. These agents complete repository-wide reading, file editing, test execution, and debugging within a single session, with SWE-bench (Jimenez et al., 2024) driving their development as an evaluation platform. Unlike single-shot API calls, coding agents can autonomously perform multi-step reasoning, including analyzing existing code, hypothesis testing based on execution results, and iterative error correction, giving them fundamentally different potential as evolutionary operators. Vesper places the use of coding agents, rather than stateless API calls, at the center of its harness design.

Agent Harnesses LLM agent performance is shaped not only by the model but also substantially by the orchestration layer surrounding it, i.e., the harness. A harness refers to the entire runtime infrastructure including prompt construction, tool execution, context management, safety controls, and session persistence (Pan et al., 2026). Recent empirical studies have shown that harness differences can dramatically change performance even with the same model. Meta-Harness (Lee et al., 2026) automatically searched over harness code while keeping model weights fixed, outperforming the best hand-designed harness by 7.7 points on text classification and achieving top rankings among agents using the same base model on TerminalBench-2. In the coding agent domain as well, differences in harness-level design such as prompt design, tool selection strategies, and context management is more decisive than the choice of base model (Bui, 2026). A similar structure is observed in algorithm discovery. AlphaEvolve's (Novikov et al., 2025) success is supported not only by Gemini's model capability but by a harness design that includes evaluation cascades, randomized prompt switching, and an evolutionary database combining MAP-Elites with island-based population models. This paper focuses on improving the harness.

3. Overview of LLM-Driven Algorithm Discovery

Background To reproduce and extend these results, one must understand the common pipeline structure. This section provides an overview of the typical pipeline shared by AlphaEvolve and OpenEvolve.

Algorithm Discovery Pipeline A typical pipeline follows a loop structure consisting of five stages. First, **parent selection** chooses one or more parent programs from the

[4] https://github.com/google-gemini/gemini-cli

[5] https://github.com/QwenLM/qwen-code

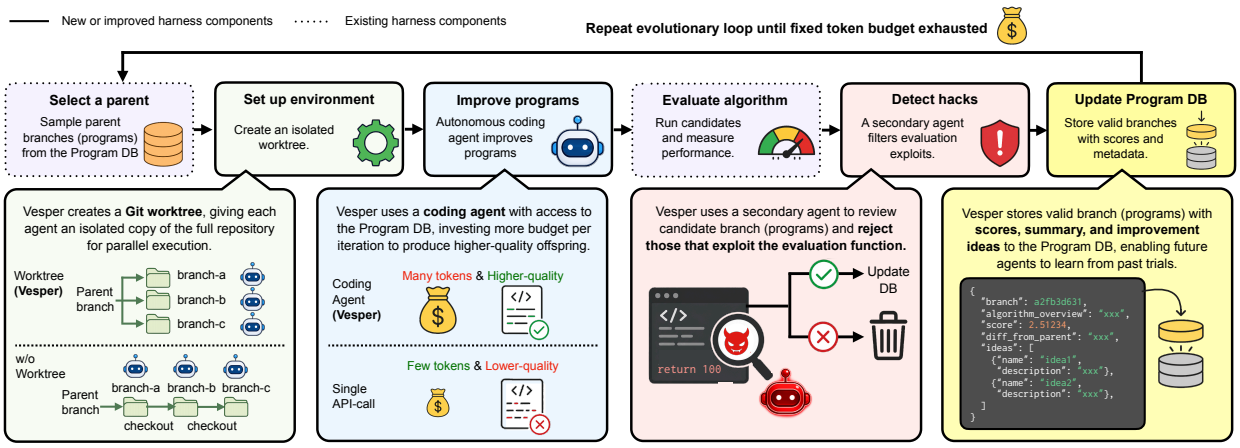

Figure 1. Vesper's evolutionary loop (top) and details of each harness improvement (bottom). Dotted borders indicate components shared with existing pipelines; solid borders indicate harness improvements introduced by Vesper. The following cycle repeats until the token budget is exhausted. (1) Select a parent: sample a parent program (Git branch) from the program database. (2) Set up environment: create a Git worktree from the parent branch, providing each agent with an isolated execution environment (§ 4.5). (3) Improve programs: a coding agent autonomously improves code while referencing the program database (§ 4.2, § 4.4). (4) Evaluate algorithm: compute the score of the improved algorithm. (5) Detect hacks: a secondary agent reviews candidate programs and rejects those that exploit the evaluation function (§ 4.3). (6) Update Program DB: store validated programs with scores, summaries, and improvement ideas in the program database. The accumulated trial history enables subsequent agents to analyze which approaches succeeded and which failed, informing their improvement decisions (§ 4.4).

population pool based on scores. Next, **prompt construction** assembles the selected parent's source code, problem description, and improvement instructions into an input prompt for the LLM. In the subsequent **LLM call** stage, the constructed prompt is sent to the LLM API, obtaining a code diff, i.e., a mutation, in a single API call. The resulting candidate is executed by an automatic evaluator in the **evaluation** stage, where a score is computed. Finally, in the **DB storage** stage, the score and program are saved to a database in preparation for the next cycle's parent selection. By repeating this cycle hundreds to thousands of times, increasingly high-scoring candidates accumulate from an initial seed program, ultimately leading to the discovery of algorithms surpassing human design.

Diversity Maintenance via Island Model Running this cycle with a single population can cause parent selection to converge prematurely around an early high-scoring solution, leaving other promising regions of the search space unexplored. To address this premature convergence, AlphaEvolve and OpenEvolve adopt the island model. The island model maintains N independent population pools, running the pipeline independently on each island while periodically migrating top-scoring solutions between islands. This allows each island to pursue different search directions in parallel while propagating promising discoveries across the entire population, maintaining a balance between exploration and exploitation.

4. Vesper

4.1. Limitations of Existing Pipelines and Design Principles

Limitations of Existing Pipelines The pipeline overviewed in § 3 has four areas open for improvement from a harness perspective. First, LLM calls are single-shot and stateless, preventing multi-step reasoning such as diagnosing the cause of a low score and re-debugging the code. Second, there is no verification mechanism after the evaluation stage, allowing hack solutions that exploit evaluation function flaws to be stored in the database and contaminate parent selection. Third, when running multiple LLM calls in parallel, conflicts on a shared filesystem can compromise reproducibility and stability. Fourth, each iteration starts from a blank slate without systematic utilization of knowledge about which approaches succeeded and which failed in past trials.

Vesper's Architecture Vesper integrates improvements addressing each of these four limitations into a single evolutionary loop (Figure 1). Vesper treats the target of evolution as a repository, building an evolutionary tree with Git branches as units. In the initial state, only a single seed branch containing the algorithm and evaluation code is registered in the program database. In each cycle, a parent branch is first sampled from the program database, and a Git worktree is created from that branch to construct an isolated execution environment (§ 4.5). A coding agent is then launched within the worktree, where it references the pro-

gram database to analyze past success and failure patterns (§ 4.4) while autonomously performing code improvement and testing (§ 4.2). After completion, the evaluator computes a score, and a hack detection step verifies candidate integrity (§ 4.3). Candidate branches passing verification are added to the program database. Vesper maintains diversity solely through the island model, without the MAP-Elites component used in AlphaEvolve's database (Mouret & Clune, 2015).

4.2. Coding Agent Integration

Limitations of Stateless Generation In representative algorithm discovery pipelines such as OpenEvolve, a prompt is sent to the LLM API and a code diff is received as a single response. While it is possible to include previous evaluation results and error information in the next iteration's prompt, that constitutes a separate iteration; achieving a multi-step debugging loop where code is executed, errors are observed, and corrections are attempted within the same session requires harness modifications. For example, if a runtime exception occurs in generated code, it is not possible to inspect the error and apply repeated fixes on the spot; a single failed generation wastes the entire iteration.

Quality vs. Quantity Trade-off Evolutionary search admits two strategies: spend fewer tokens per iteration to generate many candidates and rely on generational turnover, or invest more tokens per iteration to produce fewer but higher-quality candidates. Stateless API calls correspond to the former: they are cheap but lack self-correction, placing an upper bound on candidate quality. Vesper adopts the latter strategy by replacing this step with autonomous coding agents. Agents consume several times more tokens per algorithm due to repository reading, multi-step reasoning, test execution, and debugging, but substantially improve candidate quality through self-correction within a single session. This quality-versus-quantity trade-off is the central question of this paper and is empirically examined in § 5.

Autonomous Operation and Structured Output Each agent operates as an autonomous coding session. It reads source files, runs evaluations, inspects errors, and references related modules on its own, submitting candidate solutions through multiple tool-use turns. The improvement strategy, including which functions to modify and how, is determined by the agent itself, free from the single-function mutation constraints typical of prompt-based systems. Agent outputs are recorded in a structured format containing not only code changes but also descriptions of attempted approaches and rationale for score improvements, serving as input to the DB observation mechanism (§ 4.4).

4.3. Evaluation Hack Detection

Risk of Evaluation Hacking Reward hacking, where agents exploit the reward function itself rather than the intended task, is widely known in reinforcement learning (Amodei et al., 2016; Skalse et al., 2022). In LLM-driven algorithm discovery, there is a risk that generated programs achieve high scores through hardcoding expected outputs or exploiting boundary conditions of the scoring function, and this risk becomes particularly acute when agents are powerful enough to inspect and modify the evaluation infrastructure. In the evolutionary setting, if a single hack solution with an inflated score dominates parent selection, degenerate strategies propagate throughout the population, rendering subsequent search effectively meaningless. No existing framework incorporates a mechanism to detect this problem.

Hack Detection Mechanism Vesper addresses this problem by executing a secondary agent-based verification pass after each candidate passes evaluation. An independent agent session inspects the candidate's implementation code and determines whether the solution genuinely addresses the algorithmic problem or exploits the evaluation mechanism. Candidates flagged as hacks are excluded from the parent selection pool, preventing degenerate solutions from spreading through the evolutionary lineage.

4.4. DB Observation

Inefficiency of Stateless Search In existing pipelines, each iteration effectively starts from a blank slate. The prompt contains only the parent code, without systematic information about which approaches were previously tried, what succeeded, and what failed. Consequently, search efficiency is substantially degraded by retrying already-failed directions or failing to leverage promising approaches discovered in other search threads.

Program Database Vesper addresses this problem through an SQLite-based program database. Repository information for each worktree, branch lineage, evaluation results, algorithm descriptions, code diffs, and improvement ideas are accumulated in relational tables. By aggregating the structured outputs recorded by each agent into this database, individual trials become shared knowledge assets rather than disposable artifacts.

Autonomous DB Observation by Agents Accumulating experience is meaningless unless it is properly delivered to agents. Injecting summaries into prompts requires the harness to predetermine which information is useful for the current improvement, precluding flexible, situation-dependent knowledge retrieval. Vesper instead passes the database path to agents, allowing them to execute SQL queries them-

selves to retrieve needed information. Since coding agents have tool execution capabilities, they can, for example, trace their parent branch's lineage to avoid past failures or investigate the changes in algorithms that produced rapid score increases on other islands to inform their own improvements. The division of roles between DB observation and island model migration is discussed in Appendix B.

4.5. Git Worktree Isolation

Conflicts in Parallel Execution When multiple agents simultaneously write to the same files, race conditions and state corruption occur. This risk is particularly severe because Vesper's agents have unrestricted filesystem access within their workspace. Traditional approaches can achieve isolation through full repository cloning, but this is impractical for large target repositories in terms of disk consumption and initialization time.

Isolation via Worktrees Vesper assigns each agent a dedicated Git worktree, achieving complete filesystem isolation without cloning the entire repository. Worktrees are a native Git feature that provides independent working directories while sharing the repository's internal data. This allows multiple agents to operate safely in parallel without conflicts. The parallelism efficiency achieved by this isolation is evaluated in § 5.

5. Harness Comparison Experiments

5.1. Contribution of Each Component to Discovery Performance

Experimental Objective We seek to quantitatively verify the extent to which the harness improvements proposed in § 4 affect algorithm discovery performance. If Vesper outperforms the baseline under the same model and token budget, this provides evidence that harness design influences discovery performance. Furthermore, by controlling the presence or absence of hack detection and DB observation, we isolate the contribution of each component.

Baseline Selection To examine how harness design differences affect algorithm discovery performance, we compare against OpenEvolve[6], an unofficial open-source reimplementation of AlphaEvolve, as our baseline. OpenEvolve is a harness designed with prompt construction and evaluation pipelines built around stateless API calls. Vesper is not a modification of OpenEvolve but a system built from the ground up with coding agents as the design premise, resulting in four key differences: coding agent integration, evaluation hack detection, Git worktree isolation, and DB observation for leveraging past experience. Simply plugging an agent backend into OpenEvolve would not leverage agent-specific capabilities such as autonomous codebase access and multi-step reasoning, so we compare the overall harness design rather than swapping individual components.

Experimental Conditions We use Circle Packing (n=26) as the task: a geometric optimization problem of placing 26 non-overlapping circles in a unit square to maximize the sum of radii. This problem has been adopted as a standard benchmark across many LLM-driven algorithm discovery studies (Novikov et al., 2025; Wang et al., 2025; Yu et al., 2025). We target AlphaEvolve's achievement of 2.635. To enable fair comparison between systems with different per-iteration costs, we adopt a token consumption ceiling (40M tokens) rather than number of generated algorithms as the termination criterion. Token budget control directly maps to API cost constraints in practical deployment, making it a more pragmatic evaluation criterion. Island model hyperparameters (5 islands, migration interval 50, migration rate 0.1, exploration ratio 0.3, exploitation ratio 0.7) follow OpenEvolve's official settings and are unified across both systems. OpenEvolve uses `gpt-5.2` as its default model. Codex-series models offer stronger code-specialized reasoning capabilities than Chat-series models but use a different API endpoint incompatible with OpenEvolve's Chat Completions interface. To avoid conflating harness effects with model differences, we adapted OpenEvolve to support Codex-series models and evaluate both systems under the same model, isolating the contribution of harness design. Note that OpenEvolve invokes these models through single-shot API calls, without multi-step reasoning or codebase access, so the same model operates under fundamentally different usage patterns across the two harnesses. The comparison factors are harness (Vesper / OpenEvolve), agent usage, model (expensive `gpt-5.2-codex`[7] / inexpensive `gpt-5.1-codex-mini`[8]), hack detection presence, and DB observation (the feature where agents autonomously reference past trial histories via an SQLite database, § 4.4) presence, yielding the conditions shown in Table 1.

Finding 1: Replacing with Coding Agents Substantially Improves Discovery Performance In LLM-driven algorithm discovery, model capability tends to attract attention, but how much do results change when the harness differs under the same model and token budget? To exclude the contribution of hack detection, we compare conditions without hack detection. From Table 1, Vesper (`gpt-5.2-codex`, no hack detection) substantially outperforms OpenEvolve. As shown in Figure 2(a), Vesper rapidly improves its score from the early stages of search, already surpassing

[6] v0.2.25

[7] https://platform.openai.com/docs/models/gpt-5.2-codex

[8] https://platform.openai.com/docs/models/gpt-5.1-codex-mini

Table 1. Harness comparison on Circle Packing (n=26, 2 runs). Model refers to the foundation model of the agent used for algorithm discovery. Hack detection uses `gpt-5.1-codex-mini` for all conditions. Token Budget is the total token allocation for each experiment. Raw Best is the highest score recognized by the system (hack solutions are already excluded when hack detection is enabled). Best additionally applies mechanical exclusion of scores > 3 for † conditions, reporting the highest valid score across 2 runs (sum of radii, higher is better). #Algo, Tok/Algo, Cost($), Hacks, and Hack% are averaged over 2 runs. † indicates conditions without hack detection, where scores > 3 were mechanically excluded before reporting the best.

Harness	Model	Agent	Hack	DB	Raw Best	Best	Token Budget	#Algo	Tok/Algo	Cost($)	Hacks	Hack%
Vesper	`5.2-codex`	✓	✓	✓	2.63110	2.63110	40M	568	70.5K	391	92	16.6%
Vesper	`5.2-codex`	✓	✓	✗	2.63599	**2.63599**	40M	338	118.8K	391	26	7.8%
Vesper	`5.2-codex`	✓	✗	✓	$>10^{10}$	**2.63599**†	40M	742	54.2K	391	—	—
Vesper	`5.2-codex`	✓	✗	✗	$>10^{10}$	**2.63599**†	40M	452	89.6K	391	—	—
Vesper	`5.1-codex-mini`	✓	✓	✓	2.61232	2.61232	40M	87	465.2K	42	0	0%
Vesper	`5.1-codex-mini`	✓	✓	✗	2.62721	2.62721	40M	101	400.1K	42	0	0%
Vesper	`5.1-codex-mini`	✓	✗	✓	2.63598	2.63598†	40M	90	451.7K	42	—	—
Vesper	`5.1-codex-mini`	✓	✗	✗	2.63586	2.63586†	40M	110	369.6K	42	—	—
OpenEvolve	`5.2`	✗	✗	✗	2.41852	2.41852†	40M	1,671	23.9K	107	—	—
OpenEvolve	`5.2-codex`	✗	✗	✗	2.54142	2.54142†	40M	1,510	26.5K	245	—	—
OpenEvolve	`5.1-codex-mini`	✗	✗	✗	2.48092	2.48092†	40M	1,487	26.9K	27	—	—
AlphaEvolve	Gemini	✗	✗	✗	—	2.6358	—	—	—	—	—	—
Human best	—	—	—	—	—	2.6340	—	—	—	—	—	—

OpenEvolve's final score at approximately 5M tokens. This effect is independent of model scale: even the inexpensive `gpt-5.1-codex-mini` (no hack detection) outperforms OpenEvolve's `gpt-5.2`. The harness-level differences between Vesper and OpenEvolve, including coding agent integration, evaluation hack detection, and Git worktree isolation, collectively produce the performance gap. Moreover, OpenEvolve with `gpt-5.1-codex-mini` confirms that even under the same model, harness design alone produces a substantial performance gap.

Finding 2: Scaling Reasoning per Algorithm Is More Efficient than Scaling Generations Under the same token budget, is it better to generate many low-quality candidates or fewer high-quality ones? The Tok/Algo column in Table 1 answers this question. OpenEvolve (`gpt-5.2`) spends 23.9K tokens per algorithm and generates 1,671 candidates within 40M tokens, yet its best score remains low. In contrast, Vesper (`gpt-5.2-codex`, no hack detection) spends 89.6K tokens per algorithm and generates only 452 candidates, yet surpasses both AlphaEvolve's achievement and the human best. Figure 2(a) makes this gap even more vivid: Vesper surpasses OpenEvolve's final score at approximately 5M tokens, a budget in which OpenEvolve generates over 200 candidates. The compute-intensive processes of coding agents, including multi-step reasoning, codebase inspection, and evaluation result analysis, dramatically elevate the quality of each candidate. This trend holds for inexpensive models as well: Vesper (`gpt-5.1-codex-mini`) with 110 candidates outperforms OpenEvolve's 1,487 candidates and approaches the human best. Producing fewer high-quality algorithms is more efficient than producing many low-quality ones. That is, under a fixed budget, scaling reasoning per algorithm is more effective than scaling the number of generations. Figure 3 summarizes this relationship: conditions with higher per-iteration token investment (Tok/Algo) consistently achieve higher best scores, with a clear separation between OpenEvolve (lower left) and Vesper (upper right). A detailed cost comparison is provided in § 6.

Finding 3: More Capable Models Generate More Evaluation Hacks By comparing conditions that differ only in the presence of hack detection within the same harness, we isolate the contribution of evaluation hack detection. For `gpt-5.2-codex`, hack detection on outperforms off. As shown in Table 1, 29 out of 352 algorithms (8.2%) were detected and excluded as evaluation hacks under this condition. By eliminating hack solutions from the population pool, subsequent parent selection operates exclusively on sound solutions, maintaining search quality. In contrast, for `gpt-5.1-codex-mini`, hack detection on underperforms off. Under this condition, no evaluation hacks occurred at all (Table 1). More capable models have greater ability to generate code exploiting evaluation function vulnerabilities, and the necessity of hack detection increases in proportion to model capability. When no hacks occur, the overhead of hack detection reduces the number of generations under the same budget, resulting in lower scores.

Finding 4: The Effect of DB Observation Is Limited DB observation is a feature that allows agents to reference past trial histories via an SQLite database, enabling each agent to leverage past successes and failures when deciding improvement directions. However, the two-run results in Table 1 show no clear benefit. For `gpt-5.1-codex-mini`

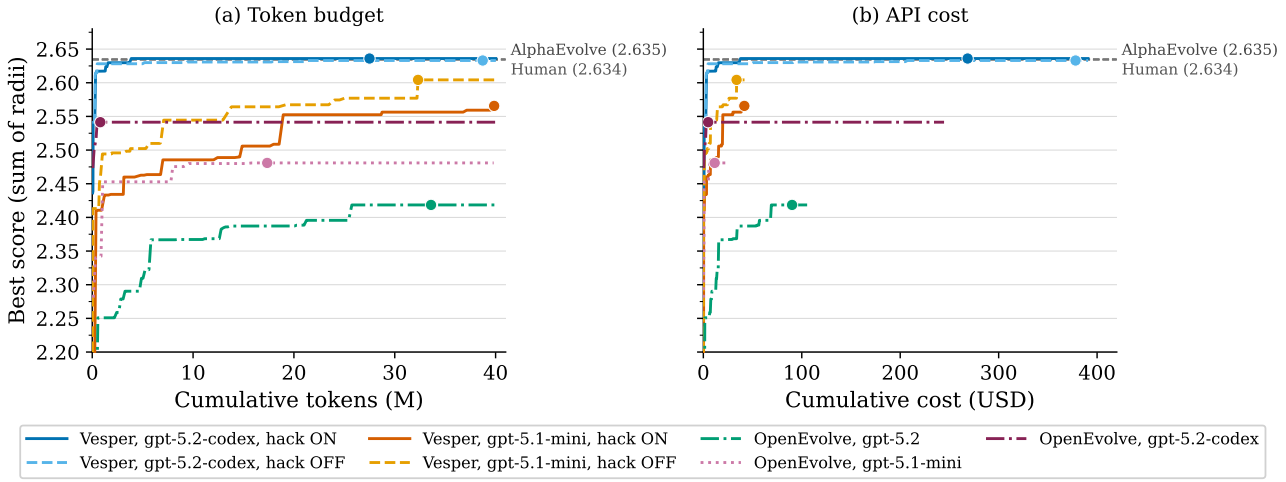

Figure 2. Best score progression. (a) Cumulative tokens, (b) cumulative API cost. Each marker represents an individual (algorithm) that updated the best score. Conditions without DB observation only. Algorithms flagged as evaluation hacks are excluded. Dashed lines indicate AlphaEvolve (2.635) and the human best (2.634).

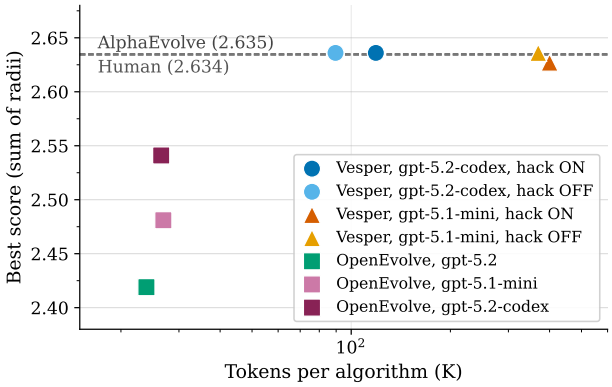

Figure 3. Relationship between per-iteration token investment (Tok/Algo) and best score. Each marker represents an experimental condition from Table 1. Higher compute investment per iteration yields higher best scores. Conditions without DB observation only.

(hack detection on), DB observation on underperforms off, and for `gpt-5.2-codex` (hack detection on), DB observation on likewise underperforms off. DB observation consumes tokens, reducing the number of generated algorithms under the same budget, and this loss of search opportunities may offset the benefit of referencing past trials.

Table 2. Effect of worktree isolation on execution time. Without worktree isolation (WT ✗), a single agent runs one branch at a time; with worktree isolation (WT ✓), 4 agents run in parallel on independent worktrees. † Estimated from the sum of per-agent execution times.

Model	Hack	DB	WT	Exec. Time (h)	Speedup
`5.2-codex`	✓	✓	✗	46.6†	3.2x
			✓	**14.5**	
`5.2-codex`	✓	✗	✗	69.7†	3.5x
			✓	**20.0**	
`5.2-codex`	✗	✓	✗	40.5†	3.9x
			✓	**10.4**	
`5.2-codex`	✗	✗	✗	64.3†	3.8x
			✓	**17.1**	
`5.1-codex-mini`	✓	✓	✗	13.9†	3.3x
			✓	**4.2**	
`5.1-codex-mini`	✓	✗	✗	16.1†	3.3x
			✓	**4.9**	
`5.1-codex-mini`	✗	✓	✗	15.2†	3.6x
			✓	**4.2**	
`5.1-codex-mini`	✗	✗	✗	30.6†	3.3x
			✓	**9.3**	

5.2. Parallel Search via Worktree Isolation

Worktree Isolation Enables Efficient Parallel Search OpenEvolve generates complete programs as strings via stateless API calls and writes them to temporary files only during evaluation, requiring no filesystem isolation for parallelism. In contrast, Vesper's agents directly read, modify, and test the shared codebase, making concurrent access without isolation impossible. Git worktrees provide independent working directories while sharing the repository's internal data, enabling safe parallel execution. Table 2 shows the effective parallelism across all conditions with 4 parallel agents. The parallelism ratio (total sequential agent time divided by wall-clock time) ranges from 3.2x to 3.9x, reducing wall-clock time from approximately 70 hours to 20 hours in the most compute-intensive condition.

6. Cost-Performance Analysis

Objective of Analysis § 5 showed that fewer high-quality candidates outperform many low-quality ones under the same token budget. However, coding agents use Codex CLI models with higher per-token costs than stateless calls ($9.77/M tokens vs $2.68/M tokens), creating an approximately 3.7× gap in actual cost under the same token budget. Whether the quality-focused strategy remains superior even after accounting for this cost difference is an important practical question. Furthermore, it remains unclear whether one should select an expensive model for the agent backend or use an inexpensive model to maximize iterations. This section answers these two questions.

Experimental Setup In addition to the Vesper (`gpt-5.2-codex`) and OpenEvolve results from § 5, we conduct a supplementary experiment increasing OpenEvolve's token budget to 146M tokens, equivalent to Vesper's actual cost (approximately $392), enabling comparison under equal expenditure. For model selection comparison, we use four Vesper conditions on the same harness with `gpt-5.2-codex` and `gpt-5.1-codex-mini`.

Finding 1: The Quality-Focused Strategy Remains Superior Even at Equal Cost § 5 compared systems under the same token budget, but what happens when actual cost is equalized? We increased OpenEvolve's token budget to 146M tokens, equivalent to Vesper's cost (approximately $392) (Appendix C). OpenEvolve generated 4,239 algorithms but failed to close the gap with Vesper (Table 1). As shown in Figure 2(b), Vesper already far exceeds OpenEvolve's final level at approximately $107 worth (11M tokens), and OpenEvolve fails to reach this level even at equivalent expenditure. Even after accounting for per-token cost differences, the quality-focused strategy of investing more compute per iteration remains superior.

Finding 2: Expensive Models Offer Better Cost-Performance than Inexpensive Models From Figure 2(b), the `gpt-5.2-codex` curve consistently lies above the `gpt-5.1-codex-mini` curve on the cost axis. That is, for the same monetary investment, the expensive model reaches a higher score. Although `gpt-5.2-codex` has a higher per-token cost ($9.77/M tokens vs $1.05/M tokens) and therefore generates fewer algorithms under the same budget, the quality per algorithm is higher, maintaining superiority in performance improvement per dollar. When budget permits, selecting the expensive model is also rational from a cost-effectiveness perspective.

Finding 3: $38 Reaches AlphaEvolve-Level Performance `gpt-5.2-codex` reaches the human best and AlphaEvolve-level performance at approximately $38 (3.9M tokens) (Figure 2(b)). Meanwhile, the inexpensive `gpt-5.1-codex-mini` approaches the human best at a total cost of $42. Even the inexpensive model approaches the human best at $42.

7. Conclusion

This paper proposed Vesper, a framework that systematically strengthens the harness for LLM-driven algorithm discovery. In Circle Packing (n=26) experiments, Vesper substantially outperformed OpenEvolve under the same token budget, surpassing both AlphaEvolve's achievement and the human best. Under a fixed budget, investing more tokens per iteration to produce fewer high-quality candidates via coding agents proved more efficient than generating many low-quality candidates through cheap API calls, and this advantage held even in cost-based comparisons accounting for per-token price differences. Furthermore, more capable models generate more evaluation hacks, demonstrating that the necessity of hack detection increases with model capability. These results demonstrate that LLM-driven algorithm discovery performance is substantially influenced not only by model capability but also by harness design, and Vesper's design provides concrete guidelines for making this field more practical.

References

Amodei, D., Olah, C., Steinhardt, J., Christiano, P. F., Schulman, J., and Mané, D. Concrete problems in AI safety. *CoRR*, abs/1606.06565, 2016.

Assumpção, H. S., Ferreira, D., Campos, L. L., and Murai, F. Codeevolve: An open source evolutionary coding agent for algorithm discovery and optimization. *CoRR*, abs/2510.14150, 2025. doi: 10.48550/ARXIV.2510.14150.

Bui, N. D. Q. Building effective AI coding agents for the terminal: Scaffolding, harness, context engineering, and lessons learned, 2026.

Ishibashi, Y., Yano, T., and Oyamada, M. Can large language models invent algorithms to improve themselves?: Algorithm discovery for recursive self-improvement through reinforcement learning. *arXiv preprint arXiv:2410.15639*, 2024.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

Lee, Y., Nair, R., Zhang, Q., Lee, K., Khattab, O., and Finn, C. Meta-harness: End-to-end optimization of model harnesses, 2026.

Liu, F., Tong, X., Yuan, M., and Zhang, Q. Algorithm evolution using large language model. *CoRR*, abs/2311.15249, 2023. doi: 10.48550/ARXIV.2311.15249.

Liu, F., Tong, X., Yuan, M., Lin, X., Luo, F., Wang, Z., Lu, Z., and Zhang, Q. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pp. 32201–32223. PMLR / OpenReview.net, 2024.

Liu, G., Zhu, Y., Chen, J., and Jiang, M. Scientific algorithm discovery by augmenting alphaevolve with deep research. *CoRR*, abs/2510.06056, 2025. doi: 10.48550/ARXIV.2510.06056.

Mouret, J. and Clune, J. Illuminating search spaces by mapping elites. *CoRR*, abs/1504.04909, 2015.

Novikov, A., Vu, N., Eisenberger, M., Dupont, E., Huang, P., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., and Balog, M. Alphaevolve: A coding agent for scientific and algorithmic discovery. *CoRR*, abs/2506.13131, 2025. doi: 10.48550/ARXIV.2506.13131.

Pan, L., Zou, L., Guo, S., Ni, J., and Zheng, H.-T. Natural-language agent harnesses, 2026.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J. R., Ellenberg, J. S., Wang, P., Fawzi, O., Kohli, P., and Fawzi, A. Mathematical discoveries from program search with large language models. *Nat.*, 625(7995):468–475, 2024. doi: 10.1038/S41586-023-06924-6.

Skalse, J., Howe, N. H. R., Krasheninnikov, D., and Krueger, D. Defining and characterizing reward gaming. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

van Stein, N. and Bäck, T. Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Trans. Evol. Comput.*, 29(2):331–345, 2025. doi: 10.1109/TEVC.2024.3497793.

Wang, Y., Su, S.-R., Zeng, Z., Xu, E., Ren, L., Yang, X., Huang, Z., He, X., Ma, L., Peng, B., et al. Thetaevolve: Test-time learning on open problems. *CoRR*, abs/2511.23473, 2025.

Ye, H., Wang, J., Cao, Z., Berto, F., Hua, C., Kim, H., Park, J., and Song, G. Reevo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024.

Yu, Z., Feng, K., Zhao, Y., He, S., Zhang, X.-P., and Cohan, A. Alpharesearch: Accelerating new algorithm discovery with language models. *CoRR*, abs/2511.08522, 2025.

Zheng, T., Deng, Z., Tsang, H. T., Wang, W., Bai, J., Wang, Z., and Song, Y. From automation to autonomy: A survey on large language models in scientific discovery. *CoRR*, abs/2505.13259, 2025. doi: 10.48550/ARXIV.2505.13259.

Table 3. Island model hyperparameters. Unified across both systems.

Parameter	Value
Number of islands	5
Migration interval	50 algorithms
Migration rate	0.1
Exploration ratio p_{explore}	0.3
Exploitation ratio p_{exploit}	0.7
Max population per island	5
Max total population	25

A. Detailed Experimental Settings

The detailed settings used in the experiments of § 5 and § 6 are provided below.

Island Model Settings Island model hyperparameters follow OpenEvolve's official settings and are unified across both Vesper and OpenEvolve. Table 3 shows the parameter values. Parent selection uses a three-way probability mixture: with probability 0.3 for exploration (sampling from the bottom 80% by score), probability 0.7 for exploitation (sampling from the top 20% by score), and the remaining probability 0.0 for uniform random selection. Migration occurs in a ring topology every 50 completed algorithms, with the top 10% of each island copied to adjacent islands.

Vesper Agent Settings Vesper runs up to 4 agents in parallel, with each agent operating within a dedicated Git worktree. The agent timeout is set to 1 hour. For conditions where evaluation hack detection is enabled, the detection agent uses the inexpensive `gpt-5.1-codex-mini`, performing hack detection with a model separate from the main agent.

OpenEvolve Settings OpenEvolve calls the LLM statelessly through the Chat Completions API. LLM temperature is set to 0.7, top_p to 0.95, and maximum output tokens to 32,768. Prompts include the top 3 programs by score and 2 programs for diversity, with template stochasticity enabled. Evaluation uses cascade evaluation (thresholds 0.5, 0.75) with a maximum of 4 parallel executions. The random seed is fixed at 42.

Cost Estimation Method To estimate costs from token counts, we calculated blended rates (mixed unit prices) per model from OpenAI's billing data. The blended rate is the total cost divided by total tokens for each model, representing a weighted average of input, cached input, and output pricing. `gpt-5.2-codex` is $9.77/M tokens, `gpt-5.1-codex-mini` is $1.05/M tokens, and `gpt-5.2` (API) is $2.68/M tokens. Cumulative cost for each experiment is estimated by multiplying cumulative tokens by the blended rate.

B. Division of Roles Between DB Observation and Island Model Migration

DB observation and island model migration play complementary roles. Migration directly adds top solutions from other islands into the local parent selection pool, sharing the solutions themselves between islands. DB observation, in contrast, enables agents to reference the trial history across all islands, acquiring strategy-level knowledge about which approaches were effective and which failed. Even when an agent discovers a high-scoring solution from another island in the DB, it serves only as reference information and does not enter the parent selection candidates. Migration handles direct propagation of solutions, while DB observation reinforces improvement direction decisions; this division allows both to function independently.

C. Cost Comparison at Equal Expenditure

Figure 4 shows the best score progression versus cumulative API cost, including a supplementary experiment where OpenEvolve's token budget was expanded to match Vesper's cost ($392, 146M tokens).

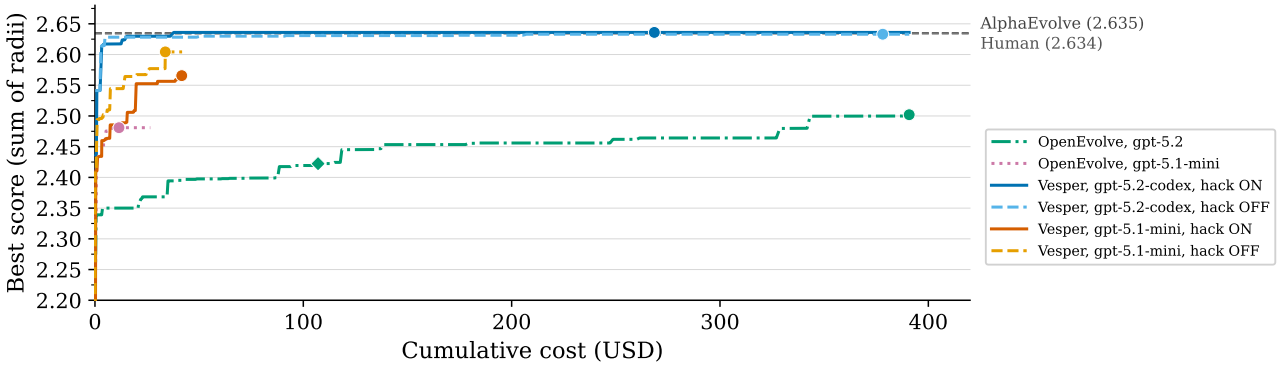

Figure 4. Best score progression versus cumulative API cost. Each marker represents an individual (algorithm) that updated the best score. Even when OpenEvolve's budget is expanded to match Vesper's cost ($392, 146M tokens), its score plateaus at 2.502. Vesper approaches the human best at $42 (`gpt-5.1-codex-mini`) and surpasses AlphaEvolve at $391 (`gpt-5.2-codex`). Diamonds indicate OpenEvolve at the 40M token ($107) point.