SWARMRESEARCH: ORCHESTRATING CODING AGENTS FOR OPEN-ENDED DISCOVERY

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-running coding agents such as autoresearch can persistently discover optimizations for open-ended problems. However, they tend to converge onto a single high-level approach, then proceed with low-level edits while missing superior approaches to the problem. We hypothesize two harness-level design choices contribute to this behavior: accumulating context in a single long-running agent and only exposing a single program state to edit. We introduce SWARMRESEARCH, a type of orchestrator-subagent harness in which a Shepherd Agent uses *global context* to steer a population of Search Agents without prescribing exact tasks, while each Search Agent operates with *local context* in its own git branch. On open-ended optimization tasks, SWARMRESEARCH discovers better or comparable solutions to state-of-the-art LLM-guided evolution and multi-agent techniques on 13/15 tasks, driven by higher-level exploration. Compared with fixed scaling of serial and parallel agents, SWARMRESEARCH's orchestrator-guided scaling discovers better solutions by adapting parallelism at different search depths.

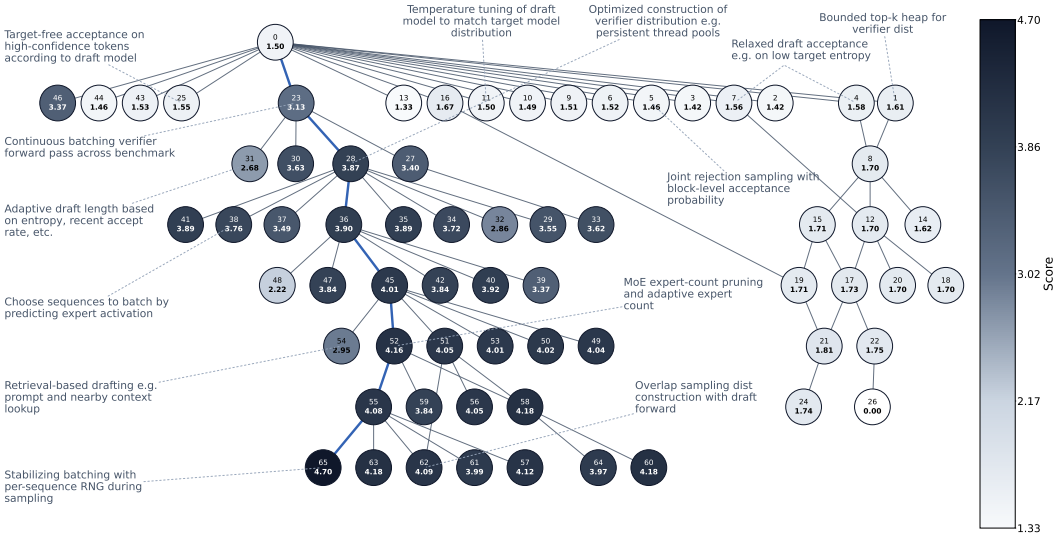


Figure 1: SWARMRESEARCH's speculative decoding implementations. Nodes represent Search Agents and their solutions. Edges indicate the lower node builds on the solution above. The top node value shows spawn order; the bottom value shows speedup over vanilla decoding. SWARMRESEARCH explores diverse approaches over ∼11 hours without converging onto any one approach. On 100 held-out reasoning-intensive tasks, **SWARMRESEARCH's mean tok/s is 4.58× faster than naive vanilla decoding**. **Vanilla autoresearch's** (Karpathy, 2026) **speedup is only 1.80×** while **CORAL's** (Qu et al., 2026) **is 2.26×**. All methods use Claude Code Opus 4.8. Details are in Appendix B.

1 INTRODUCTION

Open-ended research and engineering problems often have vast solution spaces and no known optimum. An example is designing fast decoding algorithms for LLM inference: researchers pursue

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.

directions from KV-cache management (Kwon et al., 2023) to improved speculative decoding schemes (Leviathan et al., 2023). One way they evaluate progress is by end-to-end speed. These directions can be viewed as distinct regions of an optimization landscape, where some regions correspond to weak approaches with poor local optima, while others contain substantially faster solutions. Making steady progress therefore requires continuous exploration of diverse, high-level approaches instead of immediately committing to a single neighborhood of the search space.

Test-time scaling (Brown et al., 2024) has shown promise in eliciting LLMs' ability to search open-ended solution spaces. LLM-guided evolution iteratively samples LLMs to generate new solutions given selected context on solutions generated so far, with heuristic algorithms controlling this context to guide search (Novikov et al., 2025; Sharma, 2025; Liu et al., 2026; Cemri et al., 2026; Lange et al., 2025). However, *long-running coding agents* may support more adaptive and complex search by better expressing frontier model capabilities than heuristic LLM-guided evolution. These agents can iteratively implement and test complex code, decide when to refine an existing approach or pivot to a new direction, and use external tools such as web search. Autoresearch (Karpathy, 2026) is a popular example: it runs off-the-shelf coding agents such as Claude Code for hours or days in an experimental loop. One iteration implements changes, evaluates them, commits improvements, and reverts regressions. Autoresearch has inspired projects across domains like systems optimization (Lütke, 2026), kernel optimization (RightNow-AI, 2026), and prompt injection (Panfilov et al., 2026).

Yet, long-running coding agents tend to converge quickly on a single high-level approach, then spend hours making small edits and micro-optimizations. Rather than exploring diverse directions, they can become trapped in a suboptimal region of the search space. We view this as a harness-engineering problem: context management and program versioning bottleneck divergent discovery. As agents accumulate long conversation histories filled with incremental refinements on one approach, substantial shifts in approach become unlikely. By committing all improvements to a single editable program, coding agent experimental loops do not preserve alternative directions for further exploration, effectively turning the process into greedy local search.

Instead, SWARMRESEARCH uses an orchestrator to delegate experiment ideation and execution to subagents with fresh or controlled context windows, and it assigns all major edits to separate git branches. SWARMRESEARCH is implemented as three skills compatible with popular coding agent tools such as Claude Code. A *Shepherd Agent* spawns and steers a population of *Search Agents*. Each Search Agent proposes and executes new ideas given minimal context, while the Shepherd Agent maintains global context over all Search Agent activity and strategically steers the population toward productive, diverse exploration.

We compare SWARMRESEARCH to state-of-the-art LLM-guided evolution (Liu et al., 2026) and multi-agent techniques (Qu et al., 2026) on 15 open-ended optimization tasks across mathematics, heuristics, and competitive heuristic engineering. SWARMRESEARCH exceeds or matches these techniques on 13/15 tasks. It typically makes 1.7× to 3.2× times larger code changes, reflecting higher-level exploration of approaches. We analyze SWARMRESEARCH from a test-time scaling perspective. By using an orchestrator to autonomously vary the number of parallel agents at different search depths, it exceeds the performance of optimal fixed scaling of parallel and serial agents on 4/5 tasks. We also share a detailed case study on speculative decoding to illustrate SWARMRESEARCH's capabilities and limitations in Appendix B.

2 APPROACH

We propose SWARMRESEARCH, an orchestrator-subagent harness in which a Shepherd Agent uses global context to steer a population of Search Agents, each operating in their respective git branch with fresh or controlled local context. Figure 2 describes an overview of the agent architecture. Our design is centered around two problems in long-running coding agents that bottleneck divergent solution discovery: accumulating context around one approach and only editing one program state.

First, early exploration by long-running coding agents often only covers a small number of high-level approaches. Once one approach appears promising, the agent proceeds with a long-tail of low-level edits and micro-optimizations on it. Experimenting with new high-level approaches requires several rounds of large program rewrites, which are unlikely when the agent is conditioned on a long history of incremental refinements to the current approach. As shown in Figure 2, to reduce this bias,

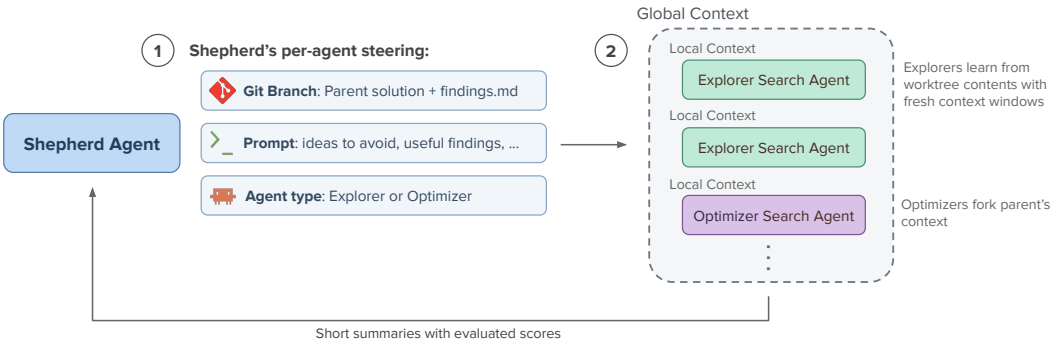


Figure 2: **Overview of SWARMRESEARCH.** Implements an orchestrator-subagent loop: **(1)** *Shepherd Agent* spawns concurrent search agents with 3 per-agent steering mechanisms: selecting a parent solution by setting up its git branch, writing a prompt to share minimal context, and selecting the Search Agent type. **(2)** *Search Agents* ideate, implement, evaluate, and iterate on a solution. *Explorer Search Agents* have fresh context windows so they're unanchored to prior work, while *Optimizer Search Agents* fork their parent's conversation history to continue along its detailed history of attempts. The Shepherd Agent receives summaries back from all Search Agents to strategically steer new ones.

the Shepherd Agent delegates experiment ideation and execution to Search Agents with fresh or controlled context windows.

Second, long-running coding agents edit a single program state: all improvements are committed, regressions are reverted, and future search proceeds from the current best implementation. This loop design makes maintaining competing high-level directions difficult and encourages greedy local search. Ideally, an agent harness should let search branch naturally, so that multiple different versions of a program can evolve independently when they instantiate different ideas. As shown in Figure 2, the Shepherd Agent assigns a new git branch for every Search Agent so all major edits occupy different versions. This way, promising yet not top-performing solutions are not prematurely discarded and can be refined. We describe the harness design, context management, and git management strategies in detail below. See Appendix D for the full skills.

2.1 ARCHITECTURE

The *Shepherd Agent* is responsible for spawning and steering waves of concurrent search agents, while *Search Agents* ideate a new approach, implement, evaluate, and iterate on it. The Shepherd Agent is given a Search Agent budget to manage. Unlike classic orchestrator-subagent patterns, which partition complex tasks such as large-scale code generation across subagents, the Shepherd Agent *does not decompose the open-ended problem for handoff or prescribe exact ideas* since they could be negatively biased by global context (see Section 2.2). Instead, it steers population-level search behavior: our skills emphasize maintaining a diverse population of ideas, prioritizing search agent budget on promising ideas, and breaking out of plateaus. The Shepherd Agent accomplishes these goals using 3 steering mechanisms (Figure 2; Step 1) when spawning new Search Agents: parent selection, search agent type, and prompts.

Parent Selection. For every new Search Agent, the Shepherd Agent creates a new git branch and worktree from either a task's baseline solution or a completed Search Agent's commit. Since each Search Agent begins by reading and modifying this *parent solution* in its worktree, that parent solution determines the neighborhood of solutions the agent explores. Parent selection lets the Shepherd Agent steer population search without prescribing specific ideas. Branches from the initial baseline will encourage new from-scratch exploration, while branches from completed agents will direct effort toward promising or underdeveloped directions.

In traditional evolutionary methods (Novikov et al., 2025; Sharma, 2025; Lange et al., 2025), parent selection is also used to direct exploration-exploitation behavior but is completed by a heuristic algorithm rather than an LLM. In existing single and multi-agent approaches (Karpathy, 2026; Qu et al., 2026), parent selection is implicit. Agents *independently* reflect on their conversation history

and shared memory to decide which solution to expand. In contrast, SWARMRESEARCH uses an orchestrator to distribute parents strategically rather than relying on uncoordinated agent decisions.

Search agent type. *Explorer Search Agents* have fresh context windows and use the content in their worktree as context. Their goal is to explore new approaches. *Optimizer Search Agents* fork their parent agent's conversation history, in addition to using worktree content as context. Their goal is to make a few refinements to the parent solution.

Prompts. Since Search Agents only have local context of their parent, the Shepherd Agent can supplement their context through prompts that share minimal, relevant context or add directions. For example, if Explorer Agents repeat similar ideas, prompts can provide a summary of already explored ideas and instruct them to explore new ideas. A prompt could also include complementary findings by other Search Agents. On plateaus, the prompt could include a briefing of strengths and weaknesses of the best approaches, with a suggestion to identify novel techniques targeting limitations. We found that restricting the Shepherd Agent from prescribing specific ideas in prompts helped maintain idea diversity; otherwise, the Shepherd Agent itself could get stuck inside a basin of ideas.

2.2 CONTEXT MANAGEMENT

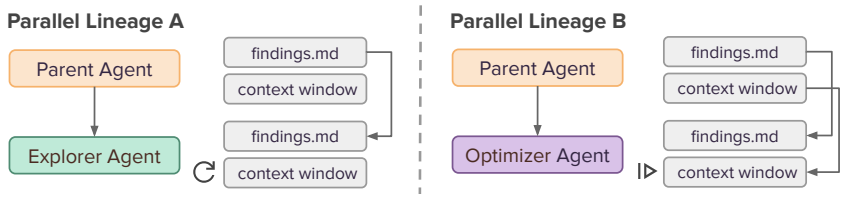


Figure 3: SWARMRESEARCH context management for search agents.

SWARMRESEARCH separates context into two tiers. Search Agents operate with *local* context: the contents of their worktree and, for Optimizer Agents, the conversation history of their direct parent. The Shepherd Agent operates with *global* context: summaries of all Search Agent attempts, including their approach and evaluated score. This separation is designed to preserve diversity among Search Agents while still giving the Shepherd Agent enough information to strategically steer the population.

A Search Agent's persistent state is primarily captured by its worktree. As shown in Figure 3, Explorer Agents start with fresh context windows, so their only context is the contents of the worktree they are assigned. Fresh context makes each agent less anchored to previous work and better suited for trying higher-level changes in approach. Optimizer Agents inherit their parent's conversation history in addition to the worktree, giving them access to a detailed log of successes and failures to continue long serial refinement when an approach appears promising. Before committing their changes, each Search Agent appends a short summary of its approach and evaluation results to a `findings.md` file. It serves as a lineage-local log of attempts and outcomes, giving future Search Agents lightweight access to their ancestors' attempts without anchoring them in a long conversation history.

Our orchestrator-subagent design addresses two failure modes of context accumulation. First, standalone coding agents tend to accumulate long conversation histories that repeatedly refine a small number of high-level approaches. These histories can make larger pivots less likely, because the agent is repeatedly conditioned on the current approach and its incremental refinements. SWARMRESEARCH avoids this diversity loss by spawning Explorer Agents with fresh context windows, while still supporting long-tail improvement through Optimizer Agents that inherit detailed conversation history when serial refinement is useful.

Second, multi-agent systems that self-organize through a shared memory e.g. CORAL (Qu et al., 2026), rather than an orchestrator, also converge onto a single high-level approach. When all agents see that another agent found a stronger solution, they decide to improve the best solution and abandon independent directions. SWARMRESEARCH eliminates this possibility by restricting Search Agent context to their direct lineage, rather than exposing them to all lineages. At the same time, global information remains useful for strategy, by revealing which approaches are promising, which fall behind, and where the population lacks coverage. SWARMRESEARCH incorporates this information

through the Shepherd Agent, which uses global context to distribute Search Agents across parents and write prompts that selectively pass relevant information to new Search Agents.

2.3 Git Management

The Shepherd Agent assigns every Search Agent to a new git branch and worktree. The `main` branch contains only the initial task setup: a `prompt.md` file defining the problem and a minimal working baseline solution. As shown in Figure 4, branches can start from `main` to try a new from-scratch approach, from a completed Search Agent's commit to iterate on a prior solution, or merge multiple parent branches to combine different solutions. Branching allows the Shepherd Agent to express a wide range of search patterns. For example, it can spawn multiple concurrent agents from the same parent to fan out around a promising solution, or spawn a serial chain of explorers and optimizers to build on a lineage's successes and failures. Since every Search Agent is versioned, fan-out and serial iteration can be flexibly interleaved across time and distant regions of the search space.

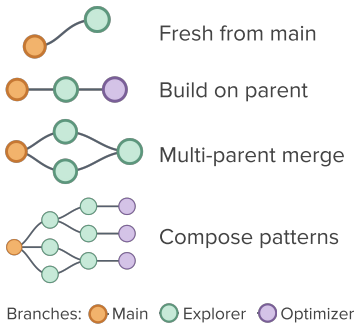


Figure 4: Git management for Search Agents.

The branching approach in SwarmResearch contrasts with existing long-running coding agents, which usually make successive edits to a single program on a single branch. Since improvements are kept and regressions are reverted, the agent's search trajectory becomes a greedy sequence of local refinements. SwarmResearch's branch-based versioning instead preserves competing approaches and avoids collapsing the search back into the current best implementation, making it easier to revisit, extend, combine, or abandon approaches.

3 Evaluation

3.1 Experimental Setup

Benchmark. Our benchmark consists of open-ended optimization problems with no known optimum. They fall in 3 domains: math, systems, and heuristic algorithms. Most mathematical optimization tasks require writing optimization programs to generate constructions demonstrating new bounds on a problem. We use 5 math tasks also used by prior work (Liu et al., 2026; Qu et al., 2026; Novikov et al., 2025). Systems tasks are from the ADRS benchmark (Cheng et al., 2025), which is based on real-world systems research. Our math and systems tasks match the benchmark suite used by our multi-agent baseline CORAL (Qu et al., 2026) (excluding MMD-16-2 for cost because of its similarity to MMD-14-3). Heuristics tasks are from ALE-Bench-Lite (Imajuku et al., 2025), which pulls problems from the AtCoder Heuristics Contest. These competitive heuristic algorithm problems are inspired by optimization challenges from industry. Our 5 problem subset was selected by top-level solution categories that reward high-level algorithm exploration over tuning of generic optimization patterns. We use ALE-Bench's suggested performance metric which typically ranges from 0 to 3500. This metric is derived from the relative rankings of competition participants for a problem. All tasks include a minimal working baseline solution.

Baselines and configurations. We compare SwarmResearch to the state-of-the-art evolutionary method EvoX (Liu et al., 2026) and the multi-agent method CORAL (Qu et al., 2026). EvoX adapts search behavior by generating parent selection programs on-the-fly using an LLM. In CORAL, concurrent coding agents run experimental loops and use a shared filesystem-based memory to exchange insights and build on one another's successes and failures. Periodic prompts instruct agents to reflect, pivot on plateaus, and consolidate findings into memory. We configure CORAL to use 4 agents, as in their paper. Both CORAL and SwarmResearch are ran until a $50 budget is exhausted. EvoX is executed for 100 iterations as configured in their paper, after which it has diminishing returns, with average task cost $23.50. Both CORAL and Shepherd use Claude Code as the agent runtime. Note that CORAL and SwarmResearch's solution performances do not converge within the budget but reach a point of diminishing returns. CORAL settles on a top approach and subsequent gains would result from long-tail polishing. No approach has internet access. CORAL and SwarmResearch only have access to evaluators as APIs so their internals are hidden.

We note that EvoX achieves poorer performance with Opus 4.6 on some problems than presented in their original paper which used GPT-5. We only report Opus 4.6 results due to cost limitations and since model dependency is a real limitation of the approach. Additionally, our results obtained by reproducing CORAL with their publicly available implementation is worse than their originally reported results. However, we include their originally reported results as the state-of-the-art where applicable. Finally, we run every technique once per task. Although one run does not give a stable estimate of every technique's per-task performance, obtaining stable estimates are unrealistically expensive. Given realistic budget constraints and the stochastic nature of LLM-based discovery techniques, one long run per task achieves the most *representative* results as per suggested use by each baseline. For a more detailed discussion of between-run variance, see Appendix C.

Granularity of Code Changes. Low-level changes appear as targeted edits to a parent program. High-level changes in approach appear as rewrites of a parent program and thus associated with a higher number of lines of code edited. For SWARMRESEARCH and CORAL, we compare the lines edited between every commit and its parent commit. EvoX only logs programs that improved over their parents, so we only take the edited lines of code for those programs and their parent. For each task, we compute the mean lines of code edited per attempt by a given method. Then, we compare the median of these task-level means.

Comparison to Optimal Fixed Scaling. Fixed scaling is our naive test-time scaling baseline. It executes n concurrent agents for k serial iterations, where each iteration consists of editing and evaluating a solution once. The (n, k) schedule is fixed in advance: after each iteration, all n agents continue for another iteration. Parallel agents share a git history and workspace, which serves as a shared experimental log and allows agents to build on each other's solutions. We find this to be a stronger baseline than running independent agents. We use the minimal coding agent Pi (Zechner, 2025) with Minimax-M2.5. Under a budget of 60 total agent iterations, we test five configurations of (n, k): $(5, 12)$, $(10, 6)$, $(15, 4)$, $(20, 3)$, and $(30, 2)$. Results are averaged over three runs, and we report the best-performing configuration.

We compare this baseline with SWARMRESEARCH's orchestrator-guided scaling. Instead of fixing the breadth and depth of the search in advance, an orchestrator dynamically allocates the same 60-iteration budget by selecting which existing solution branch should be expanded next by a fresh subagent. To isolate the effect of orchestrator-guided scaling, we use a minimal SWARMRESEARCH harness in Pi in which the orchestrator's subagent tool accepts only a *branch* field. The orchestrator cannot write prompts or suggest directions to subagents; Explorer Agents are the only subagent type. Subagents use Minimax-M2.5, while the orchestrator uses Claude Sonnet-4.6 because Minimax-M2.5 struggled with orchestration when restricted to branch selection. Thus, the orchestrator contributes only the decision of which parent solution to expand, not what idea an agent should pursue. This setup tests whether adaptively allocating a fixed compute budget across the search tree outperforms the best fixed allocation of the same budget. Orchestrator-guided results are averaged over three runs.

3.2 COMPARISON TO BASELINES

As shown in Table 1, SWARMRESEARCH exceeds the evolutionary baseline EvoX on 13/15 tasks and matches it on 1 task. SWARMRESEARCH strictly exceeds the multi-agent baseline CORAL on 10/15 tasks and matches it on 2 tasks. However, differences between CORAL and SWARMRESEARCH on the mathematical optimization tasks Circle Packing, Erdos Min Overlap, and MMD-14-3 are small. They could be reduced by further solution polishing under higher budgets. SWARMRESEARCH succeeds by exploring diverse high-level approaches while CORAL experiments with diverse but low-level changes on fewer high-level approaches. For example, in Figure 5, SWARMRESEARCH's median sized diff involves trying a new type of signal processing algorithm while CORAL makes a targeted change to signal correction. As another example, in AHC016, a median sized diff from CORAL involves tuning thresholds while SWARMRESEARCH redesigns the solver into different modes applied on different conditions. On most tasks, higher-level experiments mean SWARMRESEARCH discovers better solutions. However, within the budget, CORAL better exploited strong approaches and discovered meaningful low-level changes on the systems task EPLB and heuristics task AHC026. More details on SWARMRESEARCH agent behaviors are available in Appendix A and B.

State-of-the-art results on the math and systems tasks are all claimed by AI systems, though many are difficult to verify because their solutions are not publicly released. On the other hand, the state-

Table 1: Comparison of SWARMRESEARCH to state-of-the-art Evolutionary and Multi-Agent Baselines. *We run each approach once with Opus 4.6 and publicly available implementations*. SWARMRESEARCH and multi-agent baseline CORAL use Claude Code for $50 per task. EvoX runs for 100 iterations ($23.50 per task on average). **Bolded** means top score among these methods. Highlighted in green means score matches or exceeds state-of-the-art set by AI systems or humans.

	Task	SOTA	EvoX	CORAL	SwarmResearch
Math	Circle-Packing ↑	2.635983	2.1064	2.635985	**2.635996**
	Signal Processing ↑	0.8229	0.7181	0.7403	**0.7970**
	Erdős Min Overlap ↓	0.380876	0.38195	0.381099	**0.381080**
	MMD-14-3 (Min-Max-3) ↓	4.16578	4.46410	**4.16578**	4.16584
	3rd-Autocorrelation ↓	1.45368	1.46220	1.46233	**1.45649**
Systems	EPLB ↑	0.1490	0.1443	**0.1467**	0.1436
	LLM-SQL ↑	0.7310	0.7253	0.7195	**0.7331**
	Txn Scheduling ↑	4566.0	4081.6	4201.7	**4366.8**
	Cloudcast ↓	618.4	696.1	**618.0**	**618.0**
	PRISM ↑	**26.26**	**26.26**	**26.26**	**26.26**
Heuristics	Territory (AHC008) ↑	3463	1161	1208	**1304**
	Halloween Candy (AHC015) ↑	3506	1985.67	2268	**2543**
	Graphorean (AHC016) ↑	3517	1621	1782	**2138**
	Balancing by Balance (AHC025) ↑	3479	1400	1236	**1470**
	Stack of Boxes (AHC026) ↑	3451	2009.67	**2618**	1992

swarm_research_signal_processing.diff ± 126 LOC

```
-    This algorithm implements a sliding window approach to filter volatile,
-    non-stationary time series data while minimizing noise
-    and preserving signal dynamics.
+    This algorithm implements double exponential smoothing (Holt's method)
+    with a trend-locking hysteresis mechanism that prevents spurious
+    directional reversals.
...
-        # Weighted moving average with exponential weights
-        y[i] = np.sum(window * weights)
+    # Phase 2: Double exponential smoothing
+    # Phase 3: Trend-locking hysteresis
```

coral_signal_processing.diff ± 8 LOC

```
-    # Adaptive gain: higher where residual is coherent (real lag),
-    # lower where noisy
-    abs_res = np.abs(residual_smooth)
-    res_median = np.median(abs_res) + 1e-10
-    gain = np.clip(abs_res / (res_median * 3.0), 0.10, 0.30)
-    y = y + gain * residual_smooth
+    y = y + 0.20 * residual_smooth
```

Figure 5: Examples of median-sized SWARMRESEARCH and CORAL diffs for *Signal Processing*.

of-the-art results on heuristics tasks are all human-authored, and AI systems still lag behind. While math and systems tasks are more widely studied and more standard solution patterns are successful, heuristic tasks are competition problems whose best solutions have been intensively optimized by expert humans using creative, problem-specific approaches. We also reiterate that heuristic task scores indicate competition skill level: a score of 3000 versus 1500 does not imply a 2× higher objective metric, but corresponds to a higher Elo-like performance level where a 3000-rated competitor would outperform a 1500-rated competitor with high probability.

SWARMRESEARCH's performance exceeds or matches evolutionary and multi-agent baselines on 13/15 open-ended optimization tasks, indicating more effective search of their solution spaces. SWARMRESEARCH exceeds EvoX on 13 tasks while matching it on 1 task. SWARMRESEARCH exceeds CORAL on 8 tasks while performing similarly on 5 tasks.

3.3 GRANULARITY OF CODE CHANGES

As shown in Figure 6, SWARMRESEARCH typically makes larger code changes than the baselines. This difference is most prominent in heuristics tasks and some systems tasks like Transaction Scheduling. These tasks have large baseline solutions, and implementing new approaches require more code than other tasks. On mathematical optimization tasks like Circle Packing, experiments involve smaller changes like hyperparameter tuning or optimizer tweaks. Since low-level changes appear as targeted edits while high-level changes in approach appear as larger code changes, this

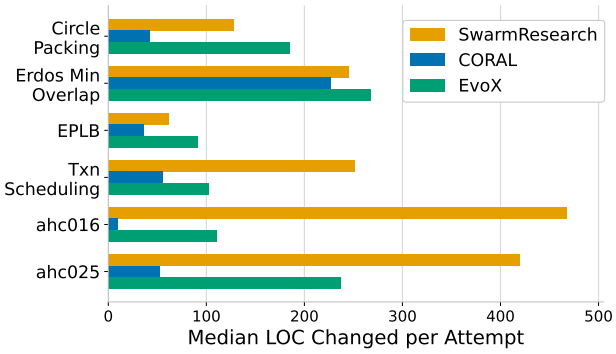


Figure 6: Median lines of code changed per attempt by SWARMRESEARCH and baselines.

result confirms our context and git management strategies increase the share of experiments that test high-level changes. Although high-level experiments are productive for most tasks, they can be inefficient when they lead nowhere and inference is better spent on optimizing strong approaches.

SWARMRESEARCH's experiments test more high-level changes than the baselines. For the average attempt and median task, SWARMRESEARCH changes 3.2× more lines than CORAL and 1.7× more than EvoX.

3.4 COMPARISON TO FIXED SCALING

Table 2: Comparison of Orchestrator-guided Scaling to Fixed Scaling. The depth of orchestrator-guided scaling is the longest chain of serially spawned search agents, and its width is the average number of parallel agents spawned at each depth.

Task	Optimal Fixed Scaling Width × Depth	Optimal Fixed Scaling Score	Orchestrator-guided Avg. Width × Depth	Orchestrator-guided Score
3rd-Autocorrelation ↓	5×12	**1.4649**	8.57×6	1.4652
Signal Processing ↑	15×4	0.6369	6.78×8	**0.6787**
Circle-Packing ↑	20×3	2.244	9.60×4	**2.294**
Cloudcast ↓	10×6	714.3	10.83×5	**672.3**
Txn Scheduling ↑	20×3	3775	6.00×4	**4049**

Table 2 compares orchestrator-guided scaling to optimal fixed scaling. Experiments uses Minimax-M2.5 as the primary model and only 5 tasks since searching for the optimal scaling configuration is costly (details in section 3.1). Table 2 shows that optimal fixed scaling is wider than deeper for 4/5 tasks. However, the widest configuration $(30, 2)$ is never optimal. Overall, shorter parallel runs outperform longer serial runs, up to a point where additional parallelism is not worthwhile. In longer serial runs, agents successfully refine an approach but stay within the same neighborhood of solutions and miss stronger neighborhoods discovered by highly parallel runs. Additionally, the optimal number of parallel agents and serial iterations under fixed budget varies with task.

Orchestrator-guided scaling, which requires no scaling hyperparameter selection, exceeds optimal fixed scaling on 4/5 tasks. In fixed scaling, the maximum number of parallel and serial iterations on top of any solution is fixed. Orchestrator-guided scaling is more flexible: the orchestrator can allocate budget to wide initial exploration, then selectively apply deep serial iteration to promising solutions. In effect, the agent can express varying width at different depths. For example, orchestrator-guided scaling's max depth is much higher than fixed scaling for *Signal Processing* which allows it to iterate on the best solution for longer. The 60 subagents dominate inference cost, so the orchestrator only increases total output tokens by 7.7% over subagent-only.

Orchestrator-guided scaling exceed optimal fixed scaling. By varying parallelism at different depths, orchestrator-guided scaling achieves better performance than optimal fixed scaling on 2/3 mathematical optimization tasks and 2/2 systems optimization tasks.

4 Related Work

LLM-Guided Evolutionary Discovery. A growing body of work uses LLMs for search and discovery (Romera-Paredes et al., 2024). Early approaches primarily use the LLM as a mutation operator within evaluator-guided evolutionary loops (Novikov et al., 2025; Sharma, 2025; Lange et al., 2025): predefined heuristics select promising parents based on evaluation feedback, and the LLM generates candidate mutations. For example, AlphaEvolve (Novikov et al., 2025) uses MAP-Elites (Mouret & Clune, 2015) for candidate selection, while follow-up work improves search with adaptive sampling (Lange et al., 2025; Sharma, 2025), Pareto frontiers (Agrawal et al., 2025), island-based architectures (Assumpção et al., 2025), and momentum-based backtracking (Yan et al., 2026). AdaEvolve (Cemri et al., 2026) and EvoX (Liu et al., 2026) go further by adapting the search strategy itself, but still employ LLMs within a largely predefined search pipeline rather than as open-ended agents. Other work instead improves the discovery capabilities of the underlying model: ThetaEvolve (Wang et al., 2025), TTT-Discover (Yuksekgonul et al., 2026), and FLEX (Cai et al., 2025), for example, use reinforcement learning to improve the LLM during the discovery process.

Agentic Solutions for Open-ended Discovery. A complementary line of work uses LLM agents to directly carry out open-ended tasks without constraining the LLM to specific operations. AI Scientist (Lu et al., 2024) and AI Co-Scientist (Gottweis et al., 2025) organize agents into specialized roles such as brainstorming and reviewing to support scientific discovery. More recently, autoresearch (Karpathy, 2026) has spurred growing interest in autonomous research agents that use general-purpose coding agents such as Codex or Claude Code to modify research code, run experiments, evaluate results, and iteratively retain promising changes. Subsequent systems improve this paradigm through better memory management (Chen et al., 2026), shared skills (rllm org, 2026), and multi-agent collaboration (Qu et al., 2026). Yet, these approaches still use one or more self-organized agents, each in long-running experimental loops with one editable program state. Instead, SwarmResearch's orchestrator-subagent architecture avoids context accumulation in a few agents and idea convergence from shared memory (see Section 2.2 for details), while its branch-based versioning preserves competing approaches to avoid greedy local search (see Section 2.3 for details). Additionally, unlike existing multi-agent approaches, which involve complex orchestration such as heartbeat events and require manually configuring the number of agents, SwarmResearch is fully implemented as skills and requires no hyperparameters since the orchestrator adapts the number of parallel and serial agents autonomously.

Test-time Scaling. Test-time scaling improves performance by allocating additional inference-time computation (Snell et al., 2024; Brown et al., 2024). Parallel scaling typically samples multiple candidate solutions and selects the highest-scoring one, as in best-of-N sampling (Cobbe et al., 2021; Lightman et al., 2023), whereas sequential scaling repeatedly revises a solution, often using evaluator feedback (Qu et al., 2024). Test-time scaling has been demonstrated to be very effective for challenging domains like software engineering (Muennighoff et al., 2025; Ehrlich et al., 2025; Xia et al., 2025). A recent method, SimpleTES (Ye et al., 2026), achieves impressive results on open-ended discovery by scaling an evaluation-driven workflow across a fixed number of parallel independent trajectories, sequential refinement steps, and best-of-K candidate selection between refinement steps. While existing work uses heuristics or fixed scaling, SwarmResearch demonstrates how an LLM can autonomously and effectively adapt the degree of parallel and serial scaling on the fly.

5 Conclusion

SwarmResearch elicits diverse, high-level exploration from coding agents through simple yet effective harness choices. A Shepherd Agent maintains global context to steer population-level behavior, while Search Agents operate with local context, insulating them from the negative effects of shared context such as convergence on top-performing approaches. Git branches allow high-level directions to diverge naturally. SwarmResearch discovers stronger solutions than state-of-the-art evolutionary and multi-agent baselines. It also interoperates with popular coding-agent tools due to its minimal implementation as skills. Important limitations still remain: the harness does not improve LLM creativity, so exploration still favors variations of known approaches, while complex Shepherd Agent behavior remains prompt-dependent so models lack sophisticated Search Agent steering. We intend our analysis and minimal harness to support practitioners and provide a foundation for future harnesses and training for open-ended discovery.

AI USE STATEMENT

In this work, we used generative AI tools for polishing writing and AI-assisted coding. We have reviewed that all polishing, e.g. spelling or re-phrasing, preserves the original content. We reviewed all AI-generated code to ensure it accurately implements all experiments. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Skill definitions are provided in Appendix D.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. Codeevolve: An open source evolutionary coding agent for algorithm discovery and optimization. 2025. URL https://arxiv.org/abs/2510.14150.

Gabriele Balsamo. Entropy-guided dynamic expert selection in mixture-of-experts models, January 2026. URL https://adaptive-k.vercel.app/paper.html. Preprint, under review.

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

Zhicheng Cai, Xinyuan Guo, Yu Pei, Jiangtao Feng, Jinsong Su, Jiangjie Chen, Ya-Qin Zhang, Wei-Ying Ma, Mingxuan Wang, and Hao Zhou. Flex: Continuous agent evolution via forward learning from experience. *arXiv preprint arXiv:2511.06449*, 2025.

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, et al. Avo: Agentic variation operators for autonomous evolutionary search. *arXiv preprint arXiv:2603.24517*, 2026.

Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. Barbarians at the gate: How ai is upending systems research. *arXiv preprint arXiv:2510.06189*, 2025.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. Codemonkeys: Scaling test-time compute for software engineering. *arXiv preprint arXiv:2501.14723*, 2025.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, et al. Towards an ai co-scientist. *arXiv preprint arXiv:2502.18864*, 2025.

Yuxuan Hu, Ke Wang, Xiaokang Zhang, Fanjin Zhang, Cuiping Li, Hong Chen, and Jing Zhang. Sam decoding: Speculative decoding via suffix automaton. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025.

Kaixuan Huang, Xudong Guo, and Mengdi Wang. SpecDec++: Boosting speculative decoding via adaptive candidate lengths. In *Proceedings of the Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=NnExMNiTHw.

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*, 2025.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024. URL https://arxiv.org/abs/2403.07974.

Andrej Karpathy. autoresearch, 2026. URL https://github.com/karpathy/autoresearch.

Sehoon Kim, Karttikeya Mangalam, Suhong Moon, Jitendra Malik, Michael W. Mahoney, Amir Gholami, and Kurt Keutzer. Speculative decoding with big little decoder. In *Advances in Neural Information Processing Systems*, volume 36, pp. 39236–39256. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/7b97adeafa1c51cf65263459ca9d0d7c-Abstract-Conference.html.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-3: Scaling up inference acceleration of large language models via training-time test. *arXiv preprint arXiv:2503.01840*, 2025. URL https://arxiv.org/abs/2503.01840.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The twelfth international conference on learning representations*, 2023.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z Pan, et al. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026.

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.

Kuan-Wei Lu, Ding-Yong Hong, Pangfeng Liu, and Jan-Jan Wu. AdaSD: Adaptive speculative decoding for efficient language model inference, 2025. URL https://arxiv.org/abs/2512.11280.

Tobias Lütke. Performance: 53% faster parse+render, 61% fewer allocations. https://github.com/Shopify/liquid/pull/2056, mar 2026. Pull request #2056 to Shopify Liquid.

MathArena. AIME 2026. Hugging Face dataset, 2026. URL https://huggingface.co/datasets/MathArena/aime_2026. Dataset of problems from the 2026 American Invitational Mathematics Examination.

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

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori B Hashimoto. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 20286–20332, 2025.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Alexander Panfilov, Peter Romov, Igor Shilov, Yves-Alexandre de Montjoye, Jonas Geiping, and Maksym Andriushchenko. Claudini: Autoresearch discovers state-of-the-art adversarial attack algorithms for llms. *arXiv preprint arXiv:2603.24511*, 2026.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Sarah Zhang, Michael Chen, Michael Ong, Aarohi Srivastava, et al. Humanity's last exam. *arXiv preprint arXiv:2501.14249*, 2025. URL https://arxiv.org/abs/2501.14249.

Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, Shao Yong Ong, Fenglu Hong, Kaichen Zhou, Chonghe Jiang, Minwei Kong, Jiacheng Zhu, Xuan Jiang, Sirui Li, Cathy Wu, Bryan Kian Hsiang Low, Jinhua Zhao, and Paul Pu Liang. Coral: Towards autonomous multi-agent evolution for open-ended discovery. *arXiv preprint arXiv:2604.01658*, 2026. URL https://arxiv.org/pdf/2604.01658.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve. *Advances in Neural Information Processing Systems*, 37:55249–55285, 2024.

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

RightNow-AI. Autokernel. https://github.com/RightNow-AI/autokernel, 2026. Open-source autoresearch system for GPU kernel optimization.

rllm org. Hive, 2026. URL https://github.com/rllm-org/hive.

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.

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

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

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.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE): 801–824, 2025. doi: 10.1145/3715754.

Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Shuo Chen, Zhankui He, Noveen Sachdeva, Isabella Ye, Weili Wang, et al. Pacevolve: Enabling long-horizon progress-aware consistent evolution. *arXiv preprint arXiv:2601.10657*, 2026.

Haotian Ye, Haowei Lin, Jingyi Tang, Yizhen Luo, Caiyin Yang, Chang Su, Rahul Thapa, Rui Yang, Ruihua Liu, Zeyu Li, Chong Gao, Dachao Ding, Guangrong He, Miaolei Zhang, Lina Sun, Wenyang Wang, Yuchen Zhong, Zhuohao Shen, Di He, Jianzhu Ma, Stefano Ermon, Tongyang Li, Xiaowen Chu, James Zou, and Yuzhi Xu. Evaluation-driven scaling for scientific discovery. *arXiv preprint arXiv:2604.19341*, 2026.

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

Mario Zechner. Pi coding agent. https://github.com/earendil-works/pi, 2025. Open-source coding agent CLI.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

Appendix

A SHEPHERD AGENT BEHAVIOR

With *minimal prompting*, typical Claude Opus 4.6 Shepherd Agent behavior involves launching waves of ∼4-8 Search Agents. Waves mostly consist of Explorer Agents, while Optimizers Agents are spawned when dominant solutions emerge. Its default search behavior is near-greedy. The Shepherd Agent begins by spawning concurrent explorers and keeps a few high-level approaches open. Then, it assigns concurrent search agents to iterate on the top-few solutions. Eventually, the Shepherd Agent concentrates Search Agents onto the single, top approach. It very rarely merges agent branches. Its prompts to Search Agents typically prescribe specific ideas to pursue. On plateaus, the Shepherd Agent spawns more concurrent Explorer Agents from the best solution but also from the baseline, with context on the top-performing approach.

Our skills prompt out negative Shepherd Agent behaviors like assigning specific ideas to Search Agents and collapsing onto one approach, while reinforcing positive behaviors like focusing Search Agents on discovering new approaches from the baseline on plateaus. However, even with thorough prompting, the Shepherd Agent's ability to strategically steer Search Agents is limited. For example, it struggles to synthesize limitations into prompts or pose specific problems to different Search Agents based on current bottlenecks. Improving an agent's ability to plan strategic experiments is an open problem that we expect to advance discovery ability.

B CASE STUDY: APPROXIMATE SPECULATIVE DECODING

We use speculative decoding as a detailed case study to illustrate SWARMRESEARCH's capabilities and limitations. Figure 1 visualizes SWARMRESEARCH's search trajectory for the task.

Task & Setup. Speculative decoding techniques are commonly lossless (Leviathan et al., 2023; Hu et al., 2025; Li et al., 2025), where the target model verifies every draft token to ensure the output exactly matches what the target model would have generated. However, strict per-token verification may not be necessary for preserving downstream task accuracy, and trading some accuracy for higher speed may be desirable (Kim et al., 2023; Huang et al., 2025; Lu et al., 2025). To explore this direction with SWARMRESEARCH, our fast evaluator includes 30 total reasoning-intensive tasks from LiveCodeBench v6, AIME 2026, GPQA Diamond, and HLE-MCQ (Jain et al., 2024; MathArena, 2026; Rein et al., 2023; Phan et al., 2025). The objective of SWARMRESEARCH is to maximize token throughput while preserving benchmark accuracy. The baseline is a basic single-file, lossless speculative decoding implementation using rejection sampling (1.68× faster than vanilla decoding). SWARMRESEARCH use Claude Code Opus 4.8. Before solution generation, an initial SWARMRESEARCH run analyses the baseline from different angles without generating solutions and its findings are included in the solution generation run worktrees.

Evaluation Details. All final results are averaged over 5 seeds. The target model is gemma-4-26B-A4B-it, and the draft model is its corresponding gemma-assistant MTP head. All methods have access to and are evaluated on 8xA6000s. Temperature 1.0 is used.

Analysis Phase Details. We do 2 sequential SWARMRESEARCH runs: an ∼11-hour analysis phase and an ∼11-hour solution generation phase. The goal of analysis it to collect data on the baseline from different angles without generating solutions. For example, Explorer Search Agents profile the baseline and measure target vs. draft confidences. The Shepherd Agent's role is to maintain a diverse population of analysis while going deep on promising directions. The analysis phase's findings are organized into a directory accessible to the solution generation phase's Search Agents. This helps structure Search Agent idea generation, so its ideas are more novel but motivated by specific observations. As an example, analysis finds that most of the target model's generation consists of low-entropy tokens that the draft predicts well, but there are few high-entropy tokens that the draft model does not predict well and throws of the reasoning trace if incorrect.

SWARMRESEARCH's Solutions. On 100 held-out tasks, averaged across 5 seeds, SWARMRESEARCH achieves a mean tok/s that is 4.58× faster than naive vanilla decoding, with 60.6% accuracy, while

vanilla decoding's baseline accuracy is 65.8%. Vanilla autoresearch (Karpathy, 2026) achieves 1.80× speedup and CORAL (Qu et al., 2026) achieves 2.26× speedup, with accuracies of 65.8% and 58.4% respectively. These methods generated solutions for ∼12 hours. For SWARMRESEARCH, accuracy loss is due to more responses exceeding our evaluator's 16,384-token limit, despite some reasoning chains including the correct answer.

The largest throughput gains with minimal accuracy loss come from systems optimizations. The best-performing optimization is batching the target forward pass across multiple benchmark sequences. To preserve accuracy for the evaluated sampling seed, SWARMRESEARCH uses independent RNG streams across batches for sampling independence. Autoresearch and CORAL's solutions do not use batching. Figure 1 shows SWARMRESEARCH's 23rd completed Search Agent applied batching on a fresh branch from main, even after significant serial exploration on approaches like relaxed acceptance conditions. Additionally, SWARMRESEARCH's best solution optimizes the construction of token distributions for rejection sampling by using a persistent thread pool, top-k bounded distributions, and parallel distribution construction during batching. The best solution additionally adapts the number of target-model experts. The baseline uses $k = 8$ experts throughout, but SWARMRESEARCH finds that $k = 8$ is only necessary during prefill. For target-token generation, SWARMRESEARCH uses $k = 4$ for most tokens and increases to $k = 5$ when the previous verification round's mean top-1 probability is less than 0.7. Unlike batching and distribution-construction optimizations, adaptive MoE expert count is not strictly lossless with respect to the original target model.

Existing inference engines (Zheng et al., 2024) already implement batching for speculative decoding and optimized token-distribution processing, while adaptive expert count has been explored outside speculative decoding (Balsamo, 2026). Other than systems optimizations, SWARMRESEARCH explores decoding techniques that trade speed and accuracy, including relaxed draft acceptance and skipping verification for confident tokens. Prior work explores related techniques (Kim et al., 2023; Huang et al., 2025; Lu et al., 2025). Novel attempts, not in existing decoding literature as far as we know, perform poorly (see risk and challenges for details).

In one relaxed-acceptance variant, a draft token is accepted directly if: (1) it is also the target model's top-1 token, and (2) either the target is low (< 1.098), or the target top-1 probability exceeds the target top-2 probability by a large margin (≥0.30). Condition 1 essentially implements greedy speculative decoding. This does not preserve the intent of rejection sampling, but it increases the draft-token acceptance rate. Condition 2 is intended to identify tokens that are less likely to be critical for accuracy, at least based on single-seed observations. This approach is not used by the final solution because it causes accuracy loss when combined with batching. When combined with the final solution and evaluated on the 30-task benchmark across 5 seeds, this approach increases mean tok/s by 9%, and increases the draft acceptance rate from 76.8% to 84.8% while reducing accuracy from 91.3% to 83.3%. Most accuracy loss is due to hitting the benchmark's token cap.

Risks and Challenges SWARMRESEARCH's novel attempts, not in existing decoding literature as far as we know, performed poorly and we did not find them reasonable. For example, one attempt was "comonotone sampling": during rejection sampling for a draft span, it sampled a single shared acceptance threshold for all tokens in the span, rather than sampling independent thresholds per token as is typical. The model produced a run with improved performance and reasoned the change increased the acceptance rate. However, after we generated acceptance rate data and evaluated additional seeds, we found its claims were unsubstantiated. SWARMRESEARCH generates a high throughput of ideas and experiments and careful review requires high effort. Without careful review, users may be convinced by low-quality approaches, where proposals and justifications seem attractive at first glance despite being incorrect.

C BETWEEN-RUN VARIANCE

Between-run variance is a special challenge for discovery technique reproducibility. Discovery techniques rely on high-temperature sampling over uncertain LLM token distributions to open-ended queries, which drives the apparent creativity of LLMs. Given that LLM variance forms a basis of many discovery techniques, some variance between independent runs can be hard to avoid. To achieve a stable estimate of some discovery techniques e.g. vanilla autoresearch (Karpathy, 2026), a researcher might have to run it over 5 times for some tasks. Even more so, we have have observed that the number of independent runs needed to achieve a stable estimate can be task-specific. However,

running discovery techniques multiple times is infeasibly expensive when using frontier models, which are important to use since they discover better solutions than cheaper models and they exhibit unique behaviors a technique might rely on. For instance, just one run of all three approaches on our 15 task benchmark cost ∼$1700 worth of Opus 4.6 credits.

In light of the costliness of these techniques, the number of independent runs should be part of the recommended use of a discovery technique. This recommendation is crucial for both users and researchers to know how to allocate their budget to independent runs when using and reproducing a technique. For example, given a $50 budget of frontier model credits per task, a technique should inform users whether splitting it into 3 runs of $17 or doing 1 long $50 run will achieve the most representative results. Exact reproducibility can be challenging for some discovery techniques, but these guidelines inform users and researchers on how to best achieve a *representative* result for both real-world use and benchmarking.

For our approach, we recommend one run since independent search agents are a form of parallel scaling that improve between-run stability. We also run the baseline approaches once since independent runs are not in the baselines' recommendation for users. This way, under realistic budget constraints, we can reproduce the most representative result for the baselines. Although this means that our reported baseline results might not reflect a stable estimate across many independent runs, we view this as the best option given their stochastic and costly nature.

D SKILLS

D.1 SHEPHERD

```
---
name: shepherd
description: How to organize a population of search agents
---

You organize a population of search agents towards multiple
    breakthroughs to an open-ended research problem. Your goal is to
    manage the population to follow these invariants:
- Initialize a diverse population of ideas
- Maintain a diverse population of ideas and avoid idea collapse
    throughout the discovery process
- Prioritize effort on promising ideas
- Make consistent progress and break out of plateaus

You accomplish these goals by spawning patterns of *explorer* and *
    optimizer* search agents, selecting their parents, and adding
    minimal context as needed. *The most important rule*: search
    agents independently determine their experiments; you are not
    allowed to instruct them to work on specific ideas.

Keep going until you've exhausted your explorer budget. Your explorer
    agent budget is *25*. Keep count of how many you've spawned. Don't
     give up, keep searching for new families or launching serial
    agents.

### Search agent types
- *Explorer*: Use to increase diversity of solutions and to break out
    of plateaus. Increase the number of concurrent explorers to
    increase diversity, even from the same parent or from different
    parents for even greater diversity. Explorers will experiment with
     one new idea and return a summary.
- *Optimizer*: Use to optimize best solutions. Only one optimizer from
     a given parent solution should be alive at a time. Optimizers
    experiment with a few successive edits and tweaks to improve a
    solution.
```

```
- How they work:
  - All search agents frequently commit their recent solutions to
    their assigned branch. Their commit messages include evaluated
    scores.
  - *Serial search agents* (search agents spawned one after another as
     each other's parent solution) remember work done by their
    ancestors, so they will try new ways to make progress along a
    similar direction.

### Adapting search agent patterns
You should constantly think and reflect on current progress to select
    the right search agent pattern. Adapt the type of search agent,
    size of fan-outs, amount of serialization, etc. to current
    progress. Some examples of patterns:
- *Explorer Fan-out*: Multiple explorers from one parent. Use on
    plateaus to explore multiple promising parents. More concurrent
    explorers increases diversity. However, emphasize serializing
    explorer agents which strategically build on top of sucesses and
    failures while concurrent explorers are unaware of each other.
- *Serial Explorers*: Select promising explorer parents from prior
    wave. Launch more explorers using them as parents. Actuate the
    number of explorers in each fan-out. 1 can be enough. You can add
    more after they're complete if not enough.
- *Multi-parent*: Select multiple parents for an agent so they can
    combine or take inspiration from them
- *Serial optimization*: Optimizers spawned one after another. Many
    successive optimizers make slow but eventual progress.
- *Combinations*: Interleave serial explorers, optimizers, fan-outs.

**Be creative. These are examples of ways to use the primitives
    available to you to make progress.**

### Searching for better families of approaches from initialization
    and on plateaus
1) **A substantial portion of your explorer budget should go towards
    building a large, diverse initial population** of ideas from
    master. Then, continue to spawn serial explorers ontop of
    promising diverse parents. Keep going until exploration plateaus.
- Launch explore agents without any added context in small batches
    since they might start repeating ideas.
- If explore agents collapse onto similar ideas, nudge exploration by
    providing a summary of already explored ideas and prompt to
    explore new ideas as context. NEVER tell them which ideas to
    explore, just what not to explore again.
- During this stage, do NOT spawn optimizers. Optimizers are launched
    late once there are some clear bests.
- Wait until explorers are complete before much serial development,
    long-running explorers might have the best solutions.
2) **Spawn long sequences of serial explorers and optimizers** for
    multiple promising families.
Spawning serial agents on solutions that didn't improve is also
    productive. Serial search agents remember work done by their
    ancestors, so they might make progress along similar directions.
    However, adapt search agent patterns if this isn't working. Do NOT
     collapse diversity by spawning serial agents on only one family
    of ideas.
3) **Every family eventually hits plateaus, so the population must
    always develop new families of approaches**. In addition to adding
     context to explorers on ideas not to repeat, you might add
    minimal context on strengths and weakensses of current approaches
    to support strategic ideas on plateaus.

### Add minimal context as needed
```

```
Search agents only have knowledge of their branch. They do NOT share
    your your context and do NOT have context on other search agents.
    Only add context if necessary and keep it minimal and non-
    prescriptive. You might:
  - Provide a summary of already explored ideas and prompt them to
    explore new, more creative ideas.
  - On plateaus, provide a 1) briefing of strengths and weaknesses of
    existing families and 2) a suggestion to identify a novel
    technique that solves the limitations
  - Add context of relevant findings from other search agents that are
     helpful to refine their solution. Only add complementary findings.

  - Control the level of exploravitenss. If the agent should try
    radical ideas, then say so. If it should explore smaller
    refinments, then say so.

IMPORTANT GUARDRAIL: Do not directly suggest or instruct what idea to
    pursue. Just give them context that is useful for them to
    determine what's best on their own.
- GOOD example of cross-agent context: "Your parent scored 0.78 with
    strong final answer quality and speed, but it still violates hard
    constraints in edge cases. Another branch reached 0.79 through a
    different family that improved constraint satisfaction and
    consistency, but it was less accurate on ordinary cases. Use those
     observed tradeoffs as context when deciding what experiment to
    run."
- BAD example of context: "Here is a completely fresh angle that no
    one has tried: build a beam-search planner with a learned scoring
    model and tune beam width in [8, 16, 32]." This is bad because it
    assigns a specific method and parameter direction.
- BETTER version used on plateau: "The current leading family is
    strong on final answer quality but still fails on hard constraint
    satisfaction. Several constraint-first alternatives improved
    validity but lost too much answer quality. Look for a different
    tradeoff that addresses this failure mode without assuming the
    current repair step is the right mechanism."

### Selecting parents by creating new branches
The neighborhood explored by search agents depends on their parent
    solution which serves as their solution's starting point. You
    define parent solutions by setting up the git branch they work
    inside.
- Spawn search agnets on diverse parents. You don't know squat, so
    never collapse onto a single idea.
- Don't end ideas prematurely. If a search agent encounters a
    promusing idea that doesn't beat an idea you've already spent
    resources developing, consider refining and exploring it further.
    It might be a new neighborhood containing an improved solution.

Every agent must receive a branch that already exists. Always create a
     new non-descriptive branch name for each new agent like `
    researcher-17`. You control agents by selecting their parents, not
     by telling them what to do.
- Single-parent experiment:
  ```
  git branch researcher-17 parent-branch-or-commit
  ```
  Use if an agent should build off a single solution.
- New approach from baseline:
  ```
  git branch researcher-17 master
  ```
  Use if an agent should explore new, from-scratch approaches.
```

````
  Use `main` instead of `master` if this repository uses `main`.
- Multi-parent experiment:
  ```
  mkdir -p .worktrees
  git worktree add -b researcher-17 .worktrees/researcher-17 first-
    parent
  cd .worktrees/researcher-17
  git merge --no-commit second-parent
  ```
  Use if an agent should take inspiration from multiple solutions or
    combine them.
  Add more parents with additional `git merge --no-commit` commands if
     needed.
  Merge conflicts are expected. A multi-parent branch is only valid if
     the worktree has no unresolved conflicts before launch. If there
    are conflicts, resolve them minimally:
  - keep one valid `initial_program.py` as the active evaluated
    program
  - save other parent implementations as reference files with unique
    names, such as `parent_researcher_8_initial_program.py`
  - combine or keep `findings.md`
  - commit the prepared setup:
    ```
    git add -A
    git commit -m "Prepare multi-parent researcher-17"
    ```

Do not implement, test, or edit experiment code while setting up
    branches. Branch setup only.

### Spawning agents
Spawn as many search agents as needed using the following commands.
    Follow their format exactly with every flag included. Include
    context in session prompts as needed. Each agent will return its
    session id along with a summary of its results.
- To spawn an `explorer`, start a fresh agent session:
```
claude --permission-mode bypassPermissions -p "Follow the explorer
    skill. Your branch name is branch_name. <optional: context>" --
    output-format json | jq -r '"session_id=\(.session_id)\nmessage=\(.
    result)"'
```
- To spawn an `optimizer`, resume and fork an existing agent session (
    choose just one session if merging):
```
claude --permission-mode bypassPermissions --resume <parent_session_id
    > --fork-session -p "Follow the optimizer skill. Your new branch
    is <branch_name>. <if applicable: note that additional findings
    were merged in> <optional: context>" --output-format json | jq -r
    '"session_id=\(.session_id)\nmessage=\(.result)"'
```
- Your handoff prompt to agents must follow the exact given format. Do
     NOT give it any specific directions or suggestions. ONLY give it
    the branch name you setup for it and minimal context.
- Stay in one directory. Switching directories causes claude to
    discover the session in a different namespace than the one it was
    originally spawned in, which causes search agents to fail to spawn.

- Background agent completion is reported by task notifications. Wait
    for those notifications; use coarse polling only as a fallback.
    Agents can take a long time.
````

```
**Keep spawning new agents until the budget is exhausted.** Progress
    is always possible by spawning more serial agents and finding a
    better family of approach. ALWAYS create new branches for every
    agent.
```

D.2 EXPLORER

```
---
name: explorer
description: Read if asked to read the explorer skill
---

You are a researcher working on an open research problem. The user
    might prompt you with additional context or instructions. It's
    important you take them into consideration. If you miss any step
    or do not follow its rules, your response will be rejected.

**STEP 1: Create your assigned git worktree in `.worktrees/`**
- Create a new git worktree for your assigned branch at `.worktrees/<
    assigned-branch>`.
- This worktree is your only working directory.
- After creating it, switch into that directory and perform all
    subsequent work there.
- Use only `git worktree add .worktrees/<assigned-branch> <assigned-
    branch>`. If it fails, stop.

IMPORTANT GUARDRAIL: Only use files already in your assigned worktree.
     Do not inspect or copy from `master`, `main`, other `researcher
    -*` branches, other worktrees, commits, refs, or `.git`. Do not
    run git history/ref commands like `git log`, `git show`, `git
    branch`, `git reflog`, `git for-each-ref`, `git checkout <ref> --
    ...`, or `git restore --source <ref> ...`.

**STEP 2: Read prompt.md for the problem description, findings.md (if
    it exists), and any initial programs in your branch**
Pause and reflect on existing findings, if any. Come up with a
    fundamentally new idea for an improved solution. Do NOT make small
    , incremental tweaks.

**STEP 3: Implement and evaluate one idea**
- Focus on one idea. Do not branch out to other major ideas, stay
    focused on implementing a single idea. There will be future
    chances to iterate.
- Use the standard evaluator for evaluating your solution
- After *every* evaluation, EXPLICITLY DECIDE whether to continue
    iterating or finish up. We have limited budget so be conservative
    with your iterations. If you have fully implemented a single idea,
     then finish up. Don't branch out into other ideas that majorly
    deviate from the initial idea. If the result suggests a targeted
    fix or improvement, you may continue editing initial_program.py
    and run ./task-eval again. If the idea doesn't work after a few
    attempts, give up immediately. Finish up by creating/editing
    findings.md as needed and committing your changes."

**Step 4: Create/edit findings.md as needed**
- `findings.md` concisely records failures and successes. It is
    strictly factual and neutral.
- Only make edits if you learned new facts. Otherwise, keep as is.
```

```
IMPORTANT GUARDRAIL: Do NOT discuss next steps or suggestions on what
    is necessary to improve the score. Simply record the basic facts
    of your solution and its evaluation.

**Step 5: Commit all your changes**
- Commit message should follow this format: `{concise experiment
    description} | score = {score from standard evaluator or concise
    failure reason}`.
  - Do not include a score on failure. Only include a description of
    the failure reason.

When returning a summary back to the orchestrator, include your commit
     emssage and the commit hash.
```

D.2.1 OPTIMIZER

```
---
name: optimizer
description: Read if asked to read the optimizer skill
---

You are improving a solution to an open-ended problem.

IMPORTANT GUARDRAIL: Only use files already in your assigned worktree.
     Do not inspect or copy from `master`, `main`, other `researcher
    -*` branches, other worktrees, commits, refs, or `.git`. Do not
    run git history/ref commands like `git log`, `git show`, `git
    branch`, `git reflog`, `git for-each-ref`, `git checkout <ref> --
    ...`, or `git restore --source <ref> ...`.

## Setup
**Create your assigned git worktree in `.worktrees/`**
- Create a new git worktree for your assigned branch at `.worktrees/<
    assigned-branch>`.
- This worktree is your only working directory.
- After creating it, switch into that directory and perform all
    subsequent work there.
- Use only `git worktree add .worktrees/<assigned-branch> <assigned-
    branch>`. If it fails, stop.

**Read prompt.md for problems description, findings.md (if it exists),
     and any initial programs in your branch**

Loop for 5 iterations. Keep count.

## Loop
1) Reflect and think of a clear idea on how to improve the solution

2) Edit the solution

3) git commit

4) Run standard evaluator

5) Amend a commit message
- Commit message should follow this format: `{concise experiment
    description} | score = {score from standard evaluator or concise
    failure reason}`.
  - Do not include a score on failure. Only include a description of
    the failure reason.
```

```
6) If score improved, keep the commit. If score is equal or worse, git
     reset back to where you started

7) Edit findings.md if needed
- 'findings.md' concisely records failures and successes. It is
    strictly factual and neutral.
- Only make edits if you learned new facts. Otherwise, keep as is.

IMPORTANT GUARDRAIL: Do NOT discuss next steps or suggestions on what
    is necessary to improve the score. Simply record the basic facts
    of your solution and its evaluation.

**Do NOT stop until you've completed 5 iterations**
```