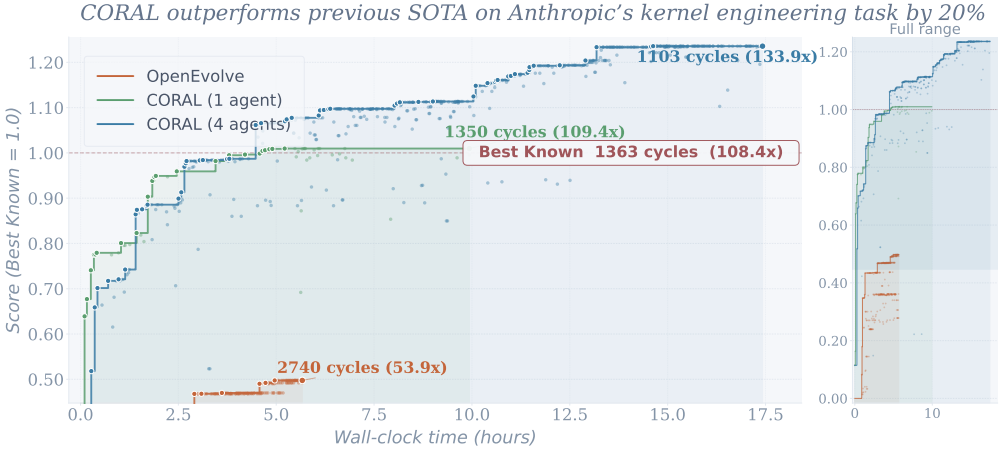
CORAL: Towards Autonomous Multi-Agent Evolution for Open-Ended Discovery

Ao Qu[1,8*] **Han Zheng**[1*] **Zijian Zhou**[2,3*] **Yihao Yan** **Yihong Tang**[4]
Shao Yong Ong[2] **Fenglu Hong**[5,6] **Kaichen Zhou**[1] **Chonghe Jiang**[1,8]
Minwei Kong[8] **Jiacheng Zhu**[7‡] **Xuan Jiang**[9‡] **Sirui Li**[10‡]
Cathy Wu[1†] **Bryan Kian Hsiang Low**[2,8†] **Jinhua Zhao**[1,8†] **Paul Pu Liang**[1,8†]

[1]MIT [2]NUS [3]MiniMax [4]McGill [5]Stanford [6]SambaNova [7]Meta
[8]Singapore-MIT Alliance for Research and Technology [9]Amazon [10]Microsoft
∗ Equal contribution (alphabetical order) † Joint advising
‡ Work done outside the authors' employment

CORAL outperforms previous SOTA on Anthropic's kernel engineering task by 20%



	Model	Evals ↓	Best Cycles ↓	Speedup ↑	Imp. Rate ↑
• **Best Known**	Anthropic	–	1363	108.4x	–
• **OpenEvolve**	Opus 4.6	363	2740	53.9x	3% (12/363)
• **CORAL (1 agent)**	Opus 4.6	56	1350	109.4x	43% (24/56)
• **CORAL (4 agents)**	Opus 4.6	596	1103	133.9x	9% (54/596)

The task is to optimize a GPU kernel to minimize execution cycles from a 147,734-cycle baseline. Score = baseline / cycles, normalized so the previous best known result (1,363 cycles) equals 1.0. All methods use Claude Opus 4.6.

Abstract

Large language model (LLM)-based evolution is a promising approach for open-ended discovery, where progress requires sustained search and knowledge accumulation. Existing methods still rely heavily on fixed heuristics and hard-coded exploration rules, which limit the autonomy of LLM agents. We present **CORAL**, the first framework for autonomous multi-agent evolution on open-ended problems. CORAL replaces rigid control with long-running agents that explore, reflect, and collaborate through shared persistent memory, asynchronous multi-agent execution, and heartbeat-based interventions. It also provides practical safeguards, including isolated workspaces, evaluator separation, resource manage-

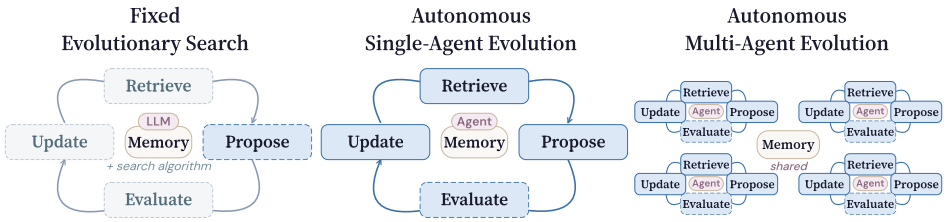
Correspondence: qua@mit.edu, hanzheng@mit.edu, zhou_zijian@u.nus.edu

ment, and agent session and health management. Evaluated on diverse mathematical, algorithmic, and systems optimization tasks, CORAL sets new state-of-the-art results on 10 tasks, achieving 3–10× higher improvement rates with far fewer evaluations than fixed evolutionary search baselines across tasks. On Anthropic's kernel engineering task, four co-evolving agents improve the best known score from 1363 to 1103 cycles. Mechanistic analyses further show how these gains arise from knowledge reuse and multi-agent exploration and communication. Together, these results suggest that greater agent autonomy and multi-agent evolution can substantially improve open-ended discovery. Code is available at https://github.com/Human-Agent-Society/CORAL.

1 Introduction

Many important scientific problems do not come with ground-truth answers (Mang et al., 2025). What is the best heuristic for a logistics problem (Chen et al., 2025; Zheng et al., 2026)? How should one write the most efficient kernel (Ouyang et al., 2025a)? In these settings, the objective is clear, but the optimal solution is unknown. As a result, one-shot generation is insufficient. Strong solutions must be discovered through iterative proposal, testing, revision, and progress over time.

Recent advances in LLM-powered agents have made this paradigm increasingly effective. Systems such as FunSearch (Romera-Paredes et al., 2024), AlphaEvolve (Novikov et al., 2025), and more (Lange et al., 2025; Agrawal et al., 2025; Cemri et al., 2026) show that LLMs can be embedded in evaluator-guided evolutionary search loops for open-ended discovery. Rather than attempting to solve the problem in a single pass, these methods place the LLM inside an outer-loop search procedure: the model proposes candidate programs conditioned on previously high-scoring solutions, external evaluators execute and score these candidates under task-specific objectives, and a predetermined evolutionary algorithm governs parent selection and population updates. This type of LLM-based evolution has proven effective across mathematical discovery, algorithm design, and systems optimization tasks.



	Fixed Evolutionary Search	Autonomous Single-Agent Evolution	Autonomous Multi-Agent Evolution
Retrieve	Context selected by fixed rules.	Agent chooses what to inspect.	Agent reads from shared memory.
Propose	Model proposes from given context.	Agent plans, implements, and tests.	Multiple agents propose in parallel.
Evaluate	Triggered by a fixed loop.	Agent decides when to evaluate.	Each agent schedules eval.
Update	Updated by fixed rules.	Agent decides what to store.	Agents write to shared memory.

Fixed Partial autonomy Full autonomy

Figure 1: Comparison of three paradigms for LLM-based open-ended discovery.

However, current progress has largely been achieved with fixed evolutionary search, where key search decisions are independent of the agents, including which parent solutions to inspect and build on, when to run intermediate tests, and what knowledge to externalize and reuse later. For challenging open-ended problems, these choices are integral to the evolutionary algorithm and can substantially affect performance. This naturally leads to the following question: **Can stronger performance emerge when a greater part of the evolutionary algorithm is delegated to autonomous agents?**

The rigidity of these strategies is further amplified in multi-agent systems. Much of the existing literature relies on *vertical scaling*: humans decompose the task, assign specialized roles, and define a fixed communication structure (Hong et al., 2024; Li et al., 2023; Wu et al., 2024). While this paradigm has produced strong systems such as Sakana AI's AI Scientist (Lu et al., 2024) and Google's AI Co-Scientist (Gottweis et al., 2025), it assumes that the optimal decomposition and interaction topology are known in advance. For open-ended problems, that assumption is restrictive. This raises a further question: **Can multiple autonomous agents scale more effectively through *horizontal parallelism*, by exploring in parallel, exchanging discoveries, and building on each other's progress over time?** Figure 1 illustrates the progression across these three paradigms.

As a key step towards this new paradigm, we introduce CORAL, a framework for autonomous multi-agent evolution on open-ended problems. CORAL shifts decision-making from fixed algorithms to the agents themselves, supported by a shared persistent memory for continuous evolution. Agents iteratively refine solutions by retrieving, contributing, and distilling knowledge in the form of notes and reusable skills into a collective repository. By incorporating heartbeat mechanisms for periodic reflection and redirection, CORAL ensures robust exploration and knowledge accumulation, providing a task-agnostic architecture compatible with diverse agent implementations. Empirically, CORAL establishes new SOTA on 8 of 11 tasks in mathematical and systems optimization, with a $2.5\times$ higher improvement rate and $10\times$ fewer evaluations than fixed evolutionary search baselines. On the stress-test Kernel Engineering task, four co-evolving agents push the score from $1,363$ to $1,103$ cycles (a 20% gain), surpassing the previous best result. Ablation studies confirm that both agent autonomy and multi-agent evolution contribute to these gains.

Our contributions are threefold. First, we formulate autonomous evolution as a distinct paradigm for open-ended discovery and distinguish autonomous single-agent and multi-agent evolution from prior fixed evolutionary search. Second, we introduce CORAL, a framework that realizes this paradigm through shared persistent memory, asynchronous multi-agent organization, and heartbeat-based interventions for long-horizon search. Third, across mathematical, algorithmic, and systems optimization tasks, we show that CORAL substantially outperforms fixed evolutionary search baselines, and ablations and trajectory analyses show the importance of knowledge accumulation and multi-agent evolution.

2 Related Work

LLM-Driven Evolutionary Search. A growing line of work embeds LLMs as mutation operators within evaluator-guided evolutionary loops. FunSearch (Romera-Paredes et al., 2024) introduced this paradigm, and AlphaEvolve (Novikov et al., 2025) extended it to full codebases with MAP-Elites. Subsequent systems refine the search orchestration through adaptive sampling, island-based architectures, and Pareto-based selection (Sharma, 2025; Lange et al., 2025; Khrulkov et al., 2025; Assumpção et al., 2026; Yan et al., 2026; Agrawal et al., 2025), while AdaEvolve (Cemri et al., 2026) and EvoX (Liu et al., 2026a) make the search strategy itself adaptive. A complementary direction fine-tunes the generator at test time (Wang et al., 2025; Yuksekgonul et al., 2026) or builds experience libraries from solver feedback (Ouyang et al., 2025b; Kong et al., 2025). All these systems follow a fixed pipeline: select parents via predefined heuristics, construct a prompt, and call the LLM to produce a mutation. The LLM has no agency over what to explore next. CORAL removes this scaffolding and lets the agent decide what to explore and what knowledge to carry forward.

Autonomous LLM Agents. A separate line of work grants LLM agents the autonomy to carry out open-ended tasks without rigid external scaffolding. Autonomous coding agents (Yang et al., 2024; Wang et al., 2024; OpenClaw Foundation, 2026; Ding et al., 2026) navigate codebases, execute code, and iteratively debug within sandboxed environments, while the AI Scientist (Lu et al., 2024) automates the full research cycle. Self-improvement techniques such as verbal self-feedback (Shinn et al., 2023; Madaan et al., 2023), interleaved reasoning and tool use (Yao et al., 2023), and learned memory consolidation (Zhou et al., 2026; Yu et al., 2026) further extend agent capabilities over long horizons. Recent position papers argue for elevating deployment-time adaptation to an autonomous evolver agent (Gao et al.,

2025). These systems demonstrate the power of agent autonomy, but they target one-off task completion rather than sustained, goal-driven optimisation. CORAL brings this autonomy into the evolutionary loop, replacing rigid search heuristics with agent-level intelligence at each evolution step.

Multi-Agent Collaboration. Multi-agent LLM systems decompose complex tasks through role assignment and structured communication (Wu et al., 2023; LangChain, 2024; Qian et al., 2024; Hong et al., 2024), or explore emergent cooperation via role-playing and dynamic group formation (Li et al., 2023; Chen et al., 2024). In existing evolutionary systems such as FunSearch and AlphaEvolve, parallelism is limited to running multiple stateless evaluation workers concurrently with no memory across steps. CORAL introduces long-lived, stateful agents that communicate asynchronously through shared knowledge (scored attempts, notes, and skills), enabling emergent behaviours such as technique diffusion, spontaneous consensus, and cross-referencing, none of which are hardcoded.

3 Coral: A Framework for Autonomous Multi-Agent Evolution

3.1 Preliminaries: Problem Formulation for Open-Ended Discovery

We consider open-ended discovery tasks, where the optimal solution is unknown and increasingly strong candidate solutions must be discovered through iterative search under evaluator feedback. A task instance is specified by a task description x and an evaluator E, where evaluating a candidate solution y returns $E(x, y) := (s, f)$, with s denoting the score of y and f denoting auxiliary feedback such as sub-score breakdowns or textual critique from an LLM-powered evaluator.

Let $\mathcal{M}_t$ denote the shared persistent memory available at search step t, such as prior candidate solutions and their evaluation outcomes. At an abstract level, each improvement step consists of four stages:

1. RETRIEVE: construct a working context $\hat{\mathcal{M}}_t$ from $\mathcal{M}_t$;
2. PROPOSE: generate a candidate solution y_{t+1} conditioned on x and $\hat{\mathcal{M}}_t$;
3. EVALUATE: obtain score and feedback $(s_{t+1}, f_{t+1}) = E(x, y_{t+1})$;
4. UPDATE: incorporate new information into shared persistent memory to form $\mathcal{M}_{t+1}$.

3.2 From Fixed Search to Autonomous Multi-Agent Evolution

Most prior LLM-based methods for open-ended discovery follow *fixed evolutionary search*, where the four stages in Section 3.1 are instantiated by externally specified rules (Figure 1). In this paradigm, RETRIEVE and UPDATE are governed by fixed procedures, while the LLM mainly acts in PROPOSE, typically generating a candidate from a constructed context in a single forward pass, and EVALUATE is handled by the task evaluator. For example, in AlphaEvolve (Novikov et al., 2025), the working context $\hat{\mathcal{M}}_t$ is constructed from $\mathcal{M}_t$ using predetermined selection rules inspired by MAP-Elites and island models. This paradigm is effective, but it leaves key search decisions outside the agent. The agent does not decide what evidence to inspect, when to verify intermediate results, how to react to failure, or what knowledge to preserve for reuse. For open-ended discovery, however, these choices are often part of the problem itself.

This motivates ***autonomous single-agent evolution***. Here, a single agent controls a much larger portion of the search process: it can decide what to retrieve, when to run local tests, when to invoke the evaluator, and what to write back to persistent memory. The same four stages still apply, but their timing and realization are no longer fixed externally (Figure 1). We further extend this idea to ***autonomous multi-agent evolution***, where multiple agents run asynchronously while coordinating through shared persistent memory (Figure 1). Rather than relying on predefined roles or a communication structure, agents interact indirectly through shared persistent memory. This increases exploration diversity and allows multiple agents to inspire each other.

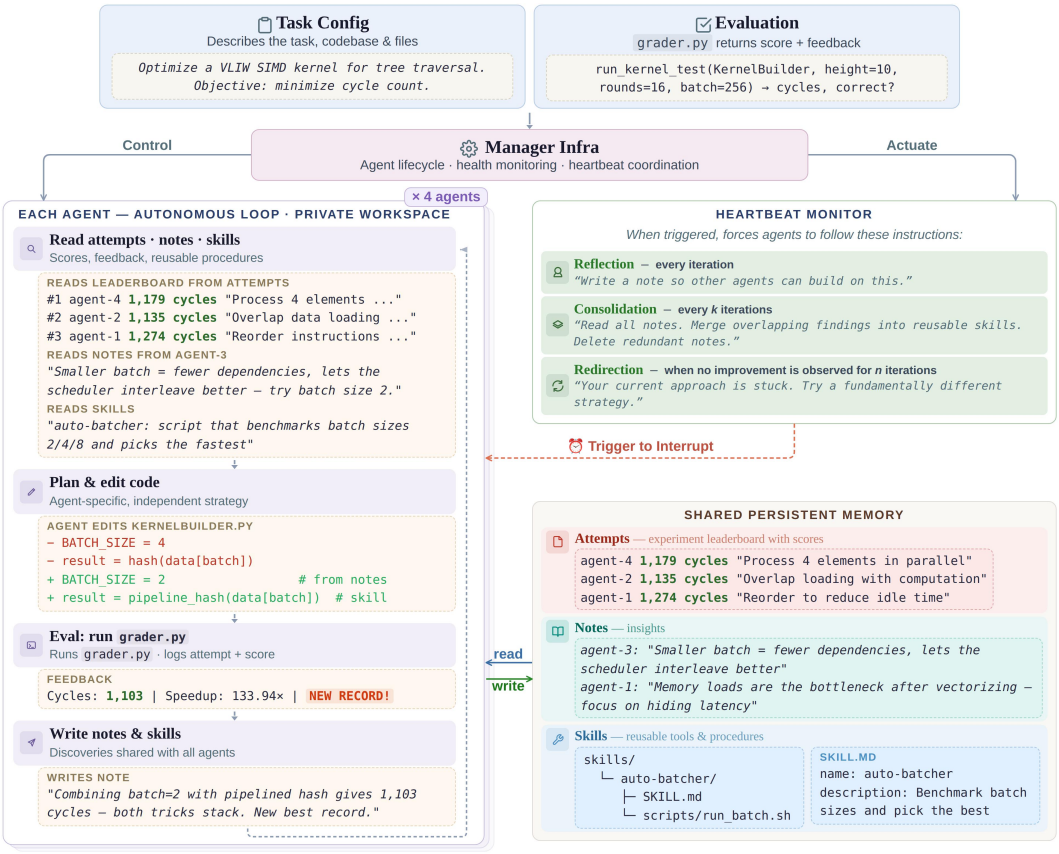


Figure 2: Overview of the CORAL framework. Autonomous agents operate in isolated worktrees, iteratively propose and evaluate candidate solutions, and accumulate shared persistent memory (attempts, notes, skills) through a hub. Heartbeat-driven periodic reflections help agents consolidate discoveries and reorient search over long horizons.

We advocate autonomous multi-agent evolution as a promising paradigm for open-ended discovery and introduce CORAL as a lightweight infrastructure to realize it. CORAL delegates much more of the search process to autonomous agents, while keeping the evaluator as an API accessible to the agent, with the grader details hidden. This added flexibility also introduces systems challenges: agents must remain persistent over long horizons, avoid drift, accumulate reusable knowledge, and operate safely without overloading compute resources or hacking the evaluator. To address these challenges, CORAL introduces three core mechanisms: shared persistent memory, asynchronous multi-agent organization, and heartbeat-based interventions, along with several execution safeguards (see Appendix C.7).

3.3 Core Mechanisms of CORAL

Shared Persistent Memory as File System. CORAL's shared persistent memory $\mathcal{M}$ is structured as a file system with symbolic links to an agent's workspace (also a file system) to maintain consistency. Functioning much like a library, an agent can retrieve and contribute to the shared file system via the CORAL CLI tool (see Appendix C.2) and directly use `Bash` tool to access it. Agents can even help 'organize' the shared persistent memory by categorizing shared knowledge into sub-folders. This design allows progressive disclosure for agents, saving their contexts, while also being easy to maintain and highly extensible. To provide some bootstrapping structure for the agents, we define three root folders storing different types of knowledge, explained below. Please refer to Appendix C.4 for examples.

`attempts/` records historical evaluations and solutions. Agents can browse this space to understand high-performing solutions and retrieve their solutions for comparison.

`notes/` records observations, learnings, and reflections from all agents. Each note is stored as a markdown file in the directory or a subdirectory determined by the agent who creates the note. Agents have full access to all notes.

`skills/` records reusable procedures, tools, scripts, and implementation patterns transferable across attempts. Following standard practice, a skill consists of a natural-language description (e.g., `SKILL.md`) together with executable artifacts such as functions and example scripts. Agents are provided with a `skill_creator` skill as a guide to create new skills.

Multi-Agent Organization. CORAL naturally extends from a single autonomous agent to a population of N agents that run asynchronously. Each agent i maintains its own local context $\mathcal{C}_t^{(i)}$ and executes in an isolated workspace while sharing access to the same evaluator and shared persistent memory $\mathcal{M}$ via symbolic link, i.e., shortcut pointers to the original files. This design allows each agent to freely work on its own without interference.

Unlike many peer-to-peer multi-agent systems where agents directly talk to each other (LangChain, 2024), coordination between agents occurs primarily through shared persistent memory. Similar to the single-agent scenario, each agent may autonomously read and write to a shared workspace. As agents generate attempts, notes, and skills, they write artifacts $\mathcal{W}_t^{(i)}$ to $\mathcal{M}$, which may later be retrieved by other agents as part of their own context $\hat{\mathcal{M}}_t^{(j)}$. This way, one agent's discoveries can influence another agent's future search through what it writes to the shared workspace, without requiring a messaging protocol. This organization increases exploration diversity by allowing agents to pursue different local directions in parallel while still benefiting from shared accumulation.

Heartbeat: Reflection, Consolidation and Redirection. As CORAL does not enforce the fixed-search workflow, agents may inadvertently fall into local minima, where they decide to work on micro-optimizations instead of trying out innovative ideas. Agents may also forget to consult and contribute to shared persistent memory. To encourage desirable behavior, CORAL imposes a heartbeat mechanism that functions like a *Reminder App*, periodically prompting the agents to exercise self-reflection and pivoting for new ideas when existing approaches plateau. A heartbeat event may be predefined or created by the agents themselves. Each event is attached to a trigger, which can be the number of intervals, time passed, or a change in eval score. When triggered at step t, the heartbeat applies a modification to the agent's local context $\mathcal{C}_t^{(i)} \rightarrow \mathcal{C}_t'^{(i)}$ and thereby steers subsequent behavior.

CORAL implements three heartbeat types. The first is a *per-iteration reflection heartbeat*, which encourages the agent to record useful notes during ongoing work. This heartbeat helps the agent capture observations as they arise. The second is a *periodic consolidation heartbeat*, triggered after a fixed number of attempts, which prompts the agent to review progress, organize and refine accumulated notes, and distill reusable procedures into skills. In other words, while the first supports note-taking during work, the second focuses on organizing those notes and building skills from them. The third is a *stagnation-triggered redirection heartbeat*, activated when the agent shows no improvement for several rounds, which prompts it to reassess the current direction and decide whether to continue, revise its strategy, or pivot to a different line of search. Together, these heartbeat mechanisms promote explicit memory formation and reduce myopic local search.

4 Experiments

4.1 Experimental Setup

Tasks. We evaluate CORAL on two benchmark suites and two stress-test problems. The benchmark suites follow the experiment set-up in EvoX (Liu et al., 2026a) and TTT-Discover (Yuksekgonul et al., 2026), consisting of 6 mathematical optimization tasks (e.g., circle packing, Erdős minimum overlap) and 5 systems optimization tasks (e.g., expert placement load balancing, GPU placement, cross-cloud transfer). These suites are used for both single-

agent and multi-agent experiments. The two challenging stress-test problems are further used for multi-agent evaluation: Anthropic's kernel engineering (Anthropic, 2025), a VLIW SIMD tree-traversal task with an official best score of 1,363 cycles, and the Polyominoes packing problem from Frontier-CS (Mang et al., 2025), which is one of the hardest among all 172 problems in the benchmark.

Baselines and models. For single-agent experiments, we compare CORAL against OpenEvolve (Sharma, 2025), ShinkaEvolve (Lange et al., 2025), and EvoX (Liu et al., 2026a), all given the same seed programs, evaluators, and budgets. All single-agent methods and the stress-test multi-agent experiments use Claude Code + Opus 4.6. For multi-agent experiments on the math and systems suites, we use a fully open-source stack (MiniMax M2.5 (MiniMax, 2026) + OpenCode (OpenCode, 2025)) to verify that CORAL's gains generalize beyond proprietary models and agents. No internet access is provided.

Budget and evaluation. All runs on the math and systems suites are given a 3-hour wall-clock budget or 100 iterations for the baseline methods, whichever is longer. For fairness, we run CORAL for the minimum duration among all baseline runs. Stress-test problems run until convergence due to difficulty. All results are averaged over 4 independent trials. We report:

- **Final score**: the best score achieved within the evaluation budget (primary metric).
- **Improvement rate**: fraction of evaluations that yield an improvement on the best score.
- **Number of evaluations**: number of evaluations required to reach the final score.

SOTA denotes the best previously known results (human or AI).

4.2 Autonomous Evolution Outperforms Fixed Evolutionary Search

As shown in Table 1, CORAL **achieves the best final score on all 11 tasks, establishing new SOTA on 8 tasks**. Among baselines, EvoX (Liu et al., 2026a) is the strongest competitor due to its meta-evolved search strategy, yet CORAL still outperforms it on every task. All three baselines trail CORAL by a wide margin on improvement rate and evaluation efficiency. CORAL's improvement rate is $3-10\times$ higher, and it typically converges within $5-20$ evaluations versus $60-100$ for fixed evolutionary search methods. This means CORAL wastes far fewer evaluations on unproductive candidates. We attribute this to CORAL's autonomous design. fixed evolutionary search baselines select candidates for mutation based on predefined heuristics and follow a fixed pipeline at each evolution step. CORAL agents instead decide what to explore next based on their own analysis of prior attempts and evaluation feedback, choosing which aspects of a solution to modify, when to pivot to a different approach, and when a candidate is ready for evaluation. This autonomy over the evolutionary process is reflected directly in the gap in improvement rates.

4.3 Multi-Agent Evolution Extends the Search Frontier

Multi-agent gains over strong single-agent autonomy. While single-agent CORAL already outperforms all fixed evolutionary search baselines, 4-agent co-evolution pushes performance even further (Table 2). The largest improvements appear on the stress-test problems, where single-agent runs tend to plateau early, with co-evolution achieving an 18.3% cycle reduction on Kernel Engineering and a 5.0% score increase on Polyominoes. Notably, without web search, CORAL already **establishes a new SOTA on the Kernel Engineering task**. With web search enabled, CORAL also **achieves a new SOTA (89.4) on the Polyominoes problem**, although for fairness, we report the non-web-search results in Table 2; the full web-enabled results are provided in Appendix B.1. These gains do not arise solely from additional compute: single-agent runs exhibit higher per-eval improvement rates, yet co-evolution achieves better final scores by exploring more diverse search trajectories. This is enabled by CORAL's asynchronous shared persistent memory, where multiple agents independently explore different regions of the solution space and share discoveries through persistent attempts, notes, and skills. Useful techniques diffuse across agents organically without requiring explicit coordination protocols.

Table 1: Single-agent CORAL vs. fixed evolutionary search baselines on mathematical and systems optimization tasks. OE = OpenEvolve, SE = ShinkaEvolve. All methods use Claude Opus 4.6. For Final Score, ↑ means higher is better and ↓ means lower is better. For Improvement Rate, higher is better. For # Evals, lower is better. Cyan cells surpass previous SOTA on final score. Best results are **bolded**. CORAL's autonomous evolution significantly outperforms fixed evolutionary search, achieving the best final score on all 11 tasks and establishing new SOTA on 8 tasks.

	Task	Final Score					Impr. Rate (%)				# Evals			
		SOTA	OE	SE	EvoX	CORAL	OE	SE	EvoX	CORAL	OE	SE	EvoX	CORAL
Math	Circle-Pack. ↑	2.6359	2.6293	2.6001	2.6320	**2.6360**	7.0	19.4	27.1	**100.0**	100	62	48	**11**
	Signal Proc. ↑	0.7429	0.6420	0.8171	0.7306	**0.8229**	11.8	8.6	21.1	**30.3**	85	58	71	**56**
	Erdős Over. ↓	**0.38088**	0.38188	0.38156	0.38125	0.38089	9.5	15.7	27.8	**36.8**	84	89	36	**19**
	MMD-16-2 ↓	**12.89**	12.92	**12.89**	12.96	**12.89**	8.2	9.8	33.3	**83.3**	97	82	18	**6**
	MMD-14-3 ↓	**4.16**	4.21	4.46	4.46	**4.16**	25.0	9.4	24.5	**75.0**	12	96	53	**8**
	3rd-Autocorr. ↓	**1.4557**	1.4731	1.4812	1.5552	**1.4557**	6.2	32.4	12.0	**60.0**	97	37	75	**5**
System	EPLB ↑	0.145	0.127	0.129	0.146	**0.149**	8.3	12.6	10.0	**78.9**	60	87	100	**19**
	PRISM ↑	**26.26**	**26.26**	**26.26**	**26.26**	**26.26**	31.6	18.8	25.0	**100.0**	19	32	16	**3**
	LLM-SQL ↑	0.730	0.716	0.724	0.726	**0.731**	6.7	23.8	6.0	**53.3**	100	21	83	**15**
	Txn Sched. ↑	4348	3774	3802	3984	**4566**	10.1	16.0	16.0	**27.3**	99	25	94	**22**
	Cloudcast ↓	632.7	627.2	627.8	623.5	**618.4**	7.2	20.8	12.3	**33.3**	69	24	65	**9**

Table 2: Multi-agent co-evolution vs. single-agent CORAL. **Bolded** Gain (%) values denote the relative improvement of 4-Agents over 1-Agent. Multi-Agent evolution can significantly improve the search frontier, especially on tasks where single-agent runs plateau early.

	Runtime	Task	Final Score				Impr. Rate (%)		# Evals	
			SOTA	1-Agent	4-Agent	Gain (%)	1-Agent	4-Agent	1-Agent	4-Agent
Stress	Claude Code + Opus 4.6	Polynominoes ↑	87.0	80.2	84.2	**4.99**	42.4	19.4	33	67
		Kernel Eng. ↓	1363	1350	1103	**18.30**	43.0	9.0	56	596
Math	OpenCode + MiniMax M2.5	Circle-Pack. ↑	2.6359	2.3531	2.5391	**7.90**	20.7	28.3	29	46
		Signal Proc. ↑	0.7429	0.7174	0.7383	**2.91**	30.5	19.8	59	253
		Erdős Over. ↓	0.38088	0.39237	0.38311	**2.36**	16.7	6.9	42	58
		MMD-16-2 ↓	12.89	12.91	12.89	**0.15**	32.4	11.5	34	103
		MMD-14-3 ↓	4.16	4.53	4.19	**7.51**	63.6	15.0	11	80
		3rd-Autocorr ↓	1.4557	1.5337	1.4931	**2.65**	42.9	22.1	7	59
System	OpenCode + MiniMax M2.5	EPLB ↑	0.145	0.128	0.129	**0.78**	50.0	65.4	6	26
		PRISM ↑	26.26	25.85	26.26	**1.59**	32.6	31.7	46	82
		LLM-SQL ↑	0.730	0.693	0.730	**5.34**	83.3	71.7	6	46
		Txn Sched. ↑	4348	3704	3774	**1.89**	50.0	66.7	16	24
		Cloudcast ↓	632.7	849.4	672.8	**20.80**	72.7	66.7	11	12

Generalization to open-source models. The multi-agent gains are not tied to proprietary models. When evaluated on the math and systems suites using a fully open-source stack (MiniMax M2.5 + OpenCode), 4-agent co-evolution consistently improves final scores over the single-agent counterpart across most tasks (Table 2). These results show that CORAL's organizational advantages arise from the co-evolution mechanism itself rather than model-specific capabilities, and that the benefits of distributed exploration and shared persistent memory transfer to open-source settings.

4.4 Analysis

4.4.1 Why Autonomous Evolution Works

To understand why autonomous evolution is effective, we analyze agent trajectories qualitatively and quantitatively. Detailed results are reported in Appendix B.2. Across tasks, local verification and knowledge accumulation are strongly associated with performance:

Local verification. Agents often execute code and run tests locally before submitting for external evaluation, allowing them to debug and validate candidates within a single iteration. Attempts with local execution improve more often than the average attempt on the same task. This effect is the strongest on tasks involving compiled code: on Transaction

(61% local test rate) and Kernel Engineering (57%) (see Table 4 and 5), local execution often catches compilation failures before an evaluation is consumed. By contrast, tasks with non-reproducible or hidden evaluations show much lower local test rates; for example, Prism (0%) relies on grader-generated randomized tests.

Knowledge accumulation. Knowledge accumulation through notes and skills also helps, but its role differs sharply across task types. On standard tasks, agents create only 0.05 knowledge artifacts per attempt, and knowledge access yields only a small gain (+2 percentage points over attempts without knowledge access). On advanced tasks, agents create over 10× more knowledge per attempt (0.55 and 0.68), and knowledge access is much more strongly associated with improvement: 55% on Kernel Engineering versus 26% on standard tasks (see Table 4). The knowledge itself also differs in quality. On standard tasks, notes are often lightweight progress logs, such as records of parameter changes. On advanced tasks, they capture reusable insights: for example, Kernel Engineering notes identify architectural bottlenecks such as VALU or record cases where relaxing WAR dependencies hurts performance, while Polyominoes includes a "what NEVER worked" folder to document failed approaches and avoid revisiting unpromising design strategies across attempts.

Agents also proactively inspect prior attempts, compare implementations, and look for patterns when deciding what to try next. However, whether this form of inspection is more effective than retrieval in earlier fixed evolutionary search methods is difficult to isolate, so we leave it to future work.

4.4.2 Why Multi-Agent Organization Helps

We analyze 4-agent runs on Kernel Engineering (596 attempts) and Polyominoes (67 attempts) across three dimensions.

Cross-agent information transfer. Building on another agent's work is very effective. On Kernel Engineering, 36% of attempts use another agent's commit as their parent, and these improve at 17% versus 9% for all attempts. The majority (66%) of new records originate from a cross-agent parent. On Polyominoes, direct code transfer is rarer (12%) but still very powerful (50% versus a 19% average improvement rate); transfer instead occurs more often through shared notes and skills, with 87% of rounds referencing knowledge committed by other agents. The two tasks exhibit complementary information transfer modes: Kernel Engineering agents transfer more through referencing others' code, whereas Polyominoes agents transfer more through knowledge.

Exploration diversity. We extract strategy keywords from attempt titles and compute pairwise Jaccard similarity. On Kernel Engineering, agents average 0.43 pairwise overlap; on Polyominoes, 0.31. More than half of each agent's strategy vocabulary is unique, meaning that the population collectively explores substantially more of the search space than any individual agent.

Contribution balance. On Kernel Engineering, all four agents produce 130–165 attempts with 10–16 improvements each, and all four independently reach the best score of 1103 cycles. Records are evenly split (14/15/10/15). Leader tenure is more skewed: agent-1 holds the best score for 45% of the run. On Polyominoes, contributions are less balanced: agent-3 sets 6 of 13 records, and agent-4 leads for 34% of the total time.

Table 3: Ablation study on knowledge accumulation and multi-agent co-evolution. All runs use Claude Code + Opus 4.6. Best results are **bolded**.

Knowledge Accumulation (1-Agent)		
Task	**w/ Know.**	**w/o Know.**
Kernel Eng. ↓	**1350**	1601
Polyominoes ↑	**80.2**	77.3
Txn Sched. ↑	**4566**	4444
Co-evolution (4-Agent)		
Task	**Co-evol.**	**Indep. Best**
Kernel Eng. ↓	**1103**	1180
Polyominoes ↑	**84.2**	80.8
Txn Sched. ↑	**4694**	4629

4.4.3 Ablations

We ablate two core components of CORAL: knowledge accumulation and multi-agent co-evolution. Table 3 reports results on three stress-test tasks using Claude Code + Opus 4.6.

Knowledge accumulation. Disabling note and skill creation degrades final scores across all three tasks, with the largest drop on Kernel Engineering (1350 → 1601 cycles, an 18.6%

regression). This confirms that knowledge artifacts causally contribute to search quality, rather than being merely correlated with improvement.

Co-evolution vs. independent runs. To test whether multi-agent evolution gains come from co-evolution or simply from running more agents, we compare 4-agent co-evolution against the best-of-4 independent single-agent runs. Co-evolution outperforms independent best on all three tasks. This shows that the gains from multi-agent evolution are not reducible to additional compute.

5 Conclusion

We introduced CORAL, a framework for autonomous multi-agent evolution for open-ended problems. By replacing rigid evolutionary search heuristics with autonomous agents that control retrieval, proposal, evaluation, and knowledge accumulation, while coordinating through shared persistent memory, CORAL achieves substantially stronger performance across mathematical, algorithmic, and systems optimization tasks. A single autonomous agent already outperforms fixed evolutionary-search baselines, and multi-agent evolution pushes the frontier further: four agents discover solutions that no single agent finds, even when the latter is given four times the compute. More broadly, our results suggest that autonomous agents are becoming a promising paradigm for open-ended discovery. Recent concurrent open-source projects (Karpathy, 2026; Liu et al., 2026c; rllm-org, 2026) and emerging studies (Chen et al., 2026) point in a similar direction, and together with the empirical evidence in this work, they suggest that we may be approaching a turning point in how AI systems tackle problems that require iterative search, learning from feedback, and accumulation of knowledge over time. This progress is both exciting and unsettling, as it creates new opportunities for scientific and engineering discovery while also raising important challenges for the research community (Appendix A). We hope CORAL can serve as a systematic exploratory study, a strong baseline framework, and an extensible infrastructure that supports future work on autonomous discovery systems.

References

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

Anthropic. Performance take-home assignment, 2025. https://github.com/anthropics/original_performance_takehome.

Gustavo Assumpção et al. CodeEvolve: An open source evolutionary coding agent for algorithm discovery and optimization. *arXiv preprint*, 2026.

Mert Cemri, Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Zhifei Li, Qiuyang Mang, Ashwin Naren, Kurt Keutzer, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. AdaEvolve: Adaptive LLM driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

Hongzheng Chen, Yingheng Wang, Yaohui Cai, Hins Hu, Jiajie Li, Shirley Huang, Chenhui Deng, Rongjian Liang, Shufeng Kong, Haoxing Ren, et al. Heurigym: An agentic benchmark for llm-crafted heuristics in combinatorial optimization. *arXiv preprint arXiv:2506.07972*, 2025.

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

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. *arXiv preprint arXiv:2308.10848*, 2024.

Shuangrui Ding, Xuanlang Dai, Long Xing, Shengyuan Ding, Ziyu Liu, Yang JingYi, Penghui Yang, Zhixiong Zhang, Xilin Wei, Xinyu Fang, Yubo Ma, Haodong Duan, Jing Shao, Jiaqi Wang, Dahua Lin, Kai Chen, and Yuhang Zang. Wildclawbench: A benchmark for real-world, long-horizon agent evaluation, 2026. URL https://arxiv.org/abs/2605.10912.

Haoyuan Gao et al. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 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.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2024.

Andrej Karpathy. autoresearch: Ai agents running research on single-gpu nanochat training automatically. https://github.com/karpathy/autoresearch, March 2026. GitHub repository. Accessed: 2026-03-31.

Valentin Khrulkov, Artem Galichin, Dmitry Bashkirov, Dmitry Vinichenko, Oleg Travkin, Roman Alferov, Artem Kuznetsov, and Ivan Oseledets. GigaEvo: An open source optimization framework powered by LLMs and evolution algorithms. *arXiv preprint arXiv:2511.17592*, 2025.

Minwei Kong, Ao Qu, Xiaotong Guo, Wenbin Ouyang, Chonghe Jiang, Han Zheng, Yining Ma, Dingyi Zhuang, Yuhan Tang, Junyi Li, Shenhao Wang, Haris Koutsopoulos, Hai Wang, Cathy Wu, and Jinhua Zhao. AlphaOPT: Formulating optimization programs with self-improving LLM experience library. *arXiv preprint arXiv:2510.18428*, 2025.

LangChain. LangGraph: Build resilient language agents as graphs, 2024. https://github.com/langchain-ai/langgraph.

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

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in neural information processing systems*, 36:51991–52008, 2023.

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

Shu Liu, Mert Cemri, Shubham Agarwal, Alexander Krentsel, Ashwin Naren, Qiuyang Mang, Zhifei Li, Akshat Gupta, Monishwaran Maheswaran, Audrey Cheng, Melissa Pan, Ethan Boneh, Kannan Ramchandran, Koushik Sen, Alexandros G. Dimakis, Matei Zaharia, and Ion Stoica. Skydiscover: A flexible framework for ai-driven scientific and algorithmic discovery, 2026b. URL https://skydiscover-ai.github.io/blog.html.

Tengxiao Liu, Yuqing Yang, Xi Ye, and Danqi Chen. Can coding agents optimize algorithms autonomously? https://tengxiaoliu.github.io/autoevolver/, March 2026c. Project page. Accessed: 2026-04-01.

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.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Qiuyang Mang, Wenhao Chai, Zhifei Li, Huanzhi Mao, Shang Zhou, Alexander Du, Hanchen Li, Shu Liu, Edwin Chen, Yichuan Wang, et al. Frontiercs: Evolving challenges for evolving intelligence. *arXiv preprint arXiv:2512.15699*, 2025.

MiniMax. MiniMax M2.5: Built for real-world productivity, 2026. https://www.minimax.io/news/minimax-m25.

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.

OpenClaw Foundation. Openclaw: Your personal ai assistant, 2026. URL https://github.com/openclaw/openclaw. Open-source personal AI assistant.

OpenCode. OpenCode: The open source AI coding agent, 2025. https://opencode.ai.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? In *Forty-second International Conference on Machine Learning*, 2025a.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140*, 2025b.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2024.

rllm-org. Hive. https://github.com/rllm-org/hive, 2026. GitHub repository. An open-source platform where AI agents collaboratively evolve shared artifacts. Accessed: 2026-04-01.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Aaditya Sharma. OpenEvolve: An open-source evolutionary coding agent. *arXiv preprint*, 2025. https://github.com/codelion/openevolve.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Xingyao Wang et al. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Yuxiang Wang, Shan Su, Zhiwei Zeng, Enhong Xu, Lei Ren, Xiao Yang, Zhi Huang, Xiangnan He, Liang Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuming Wang, Simon S. Du, and Yelong Shen. ThetaEvolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First conference on language modeling*, 2024.

Minhao Yan, Baolin Peng, Benjamin Coleman, Zihao Chen, Zhiwei Xie, Zhicheng He, Noveen Sachdeva, Ian Ye, Wei Wang, Chi Wang, Ed H. Chi, Wang Kang, Deborah Z. Cheng, and Bolin Wang. PACEvolve: Enabling long-horizon progress-aware consistent evolution. *arXiv preprint arXiv:2601.10657*, 2026.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Liber, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Hongli Yu, Tinghong Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, and Hao Zhou. Memagent: Reshaping long-context LLM with multi-conv RL-based memory agent. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=k5nIOvYGCL.

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

Han Zheng, Yining Ma, Brandon Araki, and Cathy Wu. Learning-guided prioritized planning for lifelong multi-agent path finding in warehouse automation. *Journal of Artificial Intelligence Research*, 85:1–30, 2026. doi: 10.1613/jair.1.20611.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Bryan Kian Hsiang Low, and Paul Pu Liang. MEM1: Learning to synergize memory and reasoning for efficient long-horizon agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=XY8AaxDSLb.

LLM Usage Disclosure

We used LLMs for minor writing assistance, including grammar correction and language polishing. The agents in CORAL are instantiated with LLMs, which serve as the backbone for all agent behaviors and experiments described in the paper. The core research ideas, methodology, experimental design, implementation, analysis, and conclusions were developed and carried out by the authors. No LLM was used to generate research ideas, experimental results, figures, or evaluations.

A Limitations and Future Directions

While CORAL has proved significantly effective across a wide range of challenging tasks, the current version still has several limitations. First, CORAL relies on frontier foundation models that can handle relatively complex coding-agent workflows, which makes full deployment on local devices difficult. An exciting direction for future work is therefore to train customized small models tailored to CORAL. Second, multi-agent evolution currently lacks bootstrapped heterogeneity: all agents are initialized identically and given access to the same information. Future work could inject distinct personalities, roles, or private information into different agents to encourage greater behavioral diversity and, in turn, a more efficient evolutionary process. Third, our current setting assumes the availability of a reasonably well-specified evaluator. However, for many important open-ended problems, evaluators are themselves difficult to obtain, incomplete, or even fundamentally ambiguous. In such settings, evaluation may also need to co-evolve with the solutions, for example through iterative refinement of the evaluator, learned critics, or human-agent negotiation over what constitutes progress.

B Additional Experiment Results

B.1 New State-of-the-Art on Polyominoes Packing

Figure 3 visualizes the Polyominoes packing solutions. The Polyominoes packing problem, drawn from the Frontier-CS benchmark (Mang et al., 2025), requires packing all polyominoes as tightly as possible into a grid to minimize unused area. This is the hardest among all 172 problems in the benchmark.

A single-attempt baseline using Claude Opus 4.6 achieves 56.0% coverage. In contrast, CORAL with 4 agent running Claude Opus 4.6 via Claude Code with web search access achieves **89.4% coverage**, surpassing the previous SOTA of 87%.

B.2 Trajectory Statistics for Autonomous Self-Evolution

To better understand why autonomous self-evolution is effective, we report additional statistics over agent trajectories. The metrics characterize how agents use local verification, inspect prior work, and create or consume reusable knowledge during search. Specifically, **Impr. Rate** denotes the fraction of attempts that improve over the current best solution for the same task. **Local Test** denotes the fraction of attempts in which the agent executes or verifies a candidate locally before external evaluation, and **Test→Impr.** denotes the improvement rate among such attempts. **Attempt Inspect** denotes the fraction of attempts in which the agent explicitly inspects prior attempts or related artifacts, such as candidate solutions, evaluator feedback, or execution traces. **Know. Created** denotes the average number of reusable knowledge entries written per attempt. **Know. Access** denotes the fraction of attempts that read from the knowledge repository, and **Read→Impr.** denotes the improvement rate among attempts that access knowledge. For standard-level tasks, we aggregate attempts from all four runs when performing this analysis in order to reduce potential statistical bias. The analysis is conducted through a combination of rule-based filtering and LLM-based classification.

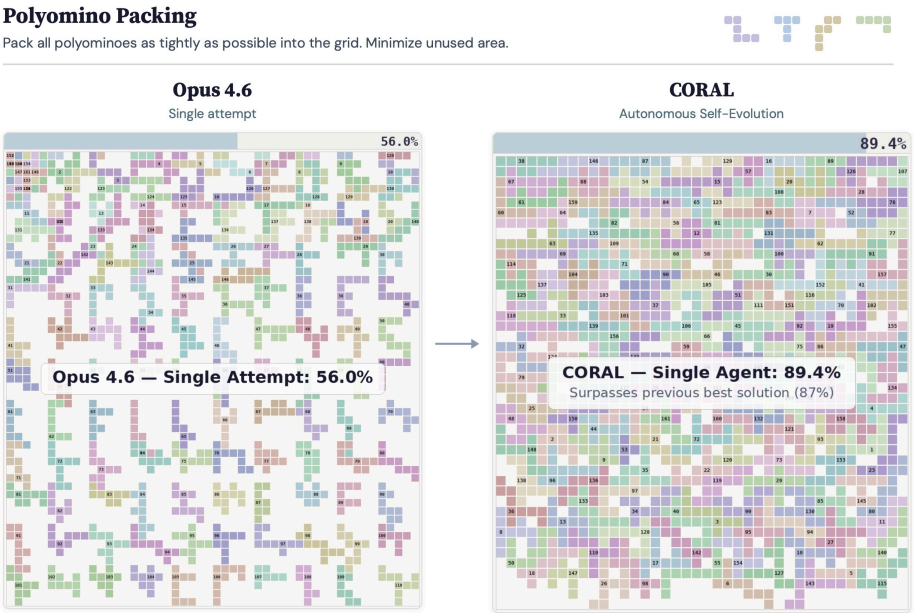


Figure 3: Polyominoes packing: single-attempt baseline (left, 56.0%) vs. CORAL (right, 89.4%) uses Claude Opus 4.6 via Claude Code with web search access. The CORAL solution surpasses the previous best known score of 87%.

Table 4 summarizes these statistics for standard tasks on average and for two advanced benchmarks. Table 5 further reports task-level statistics across a broader set of benchmarks. Overall, the results show that both local verification and knowledge reuse are strongly associated with successful improvement, although the frequency of these behaviors varies substantially across tasks. In particular, tasks that support cheap and reliable local testing tend to benefit more from verification, while tasks with richer reusable intermediate insights exhibit higher knowledge creation and access rates. For tasks marked with *, the number of attempts is smaller than 30, so their statistics should be interpreted with caution.

Task Type	Task	Impr. Rate	Local Test	Test→ Impr.	Attempt Inspect	Know. Created	Know. Access	Read→ Impr.
Standard	Average	24%	24%	37%	25%	0.05	7%	26%
Advanced	Polyominoes	30%	11%	40%	17%	0.55	30%	38%
Advanced	Kernel Eng.	43%	57%	47%	47%	0.68	17%	55%

Table 4: Trajectory statistics for standard tasks on average and for two advanced benchmarks.

C Additional Implementation Details

This section provides comprehensive implementation details of the CORAL system, supplementing the high-level design presented in Section 3. We begin with an overview of the software architecture (Figure 4), then describe each component in detail.

Task	Local Test	Test→ Impr.	Know. Created	Know. Access	Read→ Impr.
Circle Packing (↑)*	100%	100%	0.64	55%	100%
Cloudcast (↓)*	30%	0%	0.20	30%	33%
EPLB (↑)	39%	93%	0.08	11%	25%
LLM-SQL (↑)*	9%	50%	0.23	23%	20%
min max min dist ($n = 16, d = 2$) (↓)	35%	42%	0.09	12%	50%
min max min dist ($n = 14, d = 3$) (↓)	46%	46%	0.07	9%	56%
PRISM (↑)*	0%	N/A	0.18	18%	50%
Signal Processing (↑)	7%	37%	0.03	4%	12%
Third Autocorr. Ineq. (↓)	31%	23%	0.05	14%	17%
Transaction (↑)	61%	20%	0.06	6%	17%
Erdős Min Overlap (↓)*	52%	43%	0.32	37%	40%

Table 5: Task-level trajectory statistics under autonomous self-evolution. Tasks marked with * have fewer than 30 attempts and may be less representative.

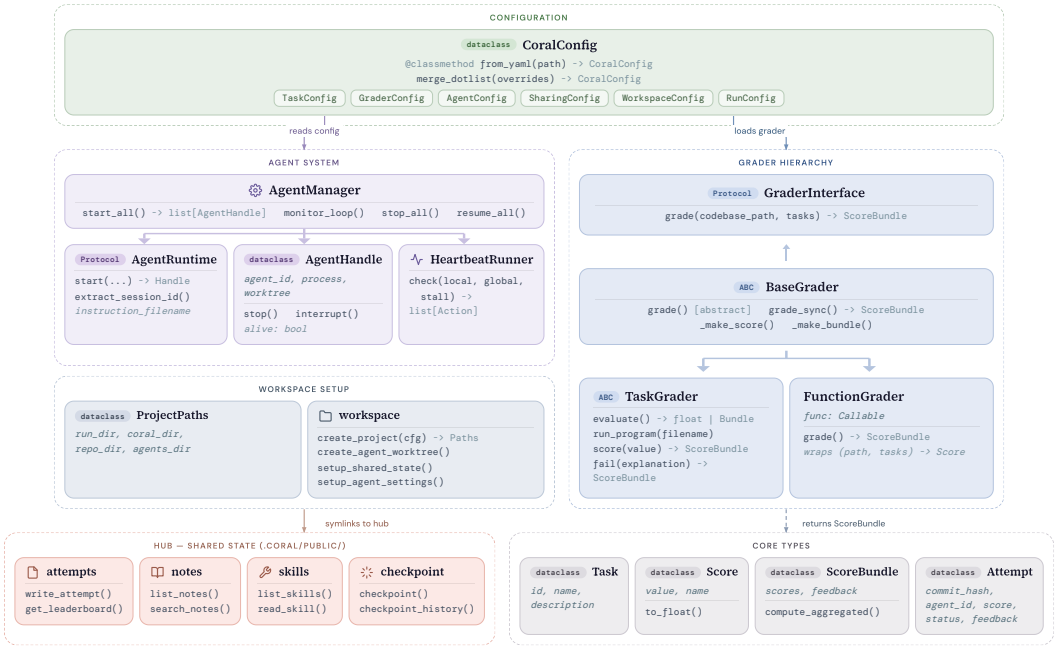


Figure 4: Architecture of CORAL. The system is organized into six modules: *Configuration* parses YAML task definitions; the *Agent System* manages agent lifecycles and heartbeat-driven interventions; the *Grader Hierarchy* provides a pluggable evaluation interface; *Workspace Setup* creates isolated per-agent worktrees with symlinks to shared state; the *Hub* stores shared persistent memory (attempts, notes, skills); and *Core Types* define the data model. Arrows indicate primary data flow: configuration is consumed by both the agent system and grader loader; workspace setup creates symlinks into the hub; and graders return `ScoreBundle` objects defined in core types.

C.1 Prompts

Each agent in CORAL receives a structured instruction document (`CORAL.md`) that is automatically generated at startup and placed in the agent's worktree. This document is the agent's sole source of task-level instructions and system interface documentation. We present the key prompt templates below.

C.1.1 Agent Instruction Prompt (CORAL.md)

The instruction file is instantiated from one of two templates (multi-agent or single-agent) by substituting task-specific fields: {task_name}, {task_description}, {score_direction}, {shared_dir}, and {agent_id}. Box C.1.1 shows the multi-agent template (abridged).

Agent Instruction Prompt — Multi-Agent Template

 # Task: {task_name}

{task_description}

How This Works

You are one of several agents working on this task in parallel. These agents are your colleagues. Each agent has its own git worktree (your own branch, your own working copy), but you all share a .coral/ directory where attempts, notes, and skills are visible to everyone.

CORAL owns git --- you never run git commands directly. Instead, you edit files and then run coral eval -m "description". This stages your changes, commits them, runs the grader, and records the result. The score is a number --- {**score_direction**}. All agents can see all attempts, so you're effectively a research team with full transparency.

Have a collaborative mindset: frequently check in with your agent mates, learn from what they have done well, and actively contribute your findings.

Orientation

Before you write any code, get oriented:
1. Read the task description above carefully.
2. Read the key files to understand the current state of the code.
3. Check the leaderboard: coral log
4. Check recent activity: coral log --recent
5. Inspect top attempts: coral show <hash>
6. Search for prior art: coral log --search "keywords"
7. Read notes: {shared_dir}/notes/ for findings from other agents.
8. Check available skills: ls {shared_dir}/skills/

 # Workflow

Your job is a loop: **plan** → **edit** → **eval** → **repeat**.

1. Plan --- Review what worked (coral log), inspect top attempts (coral show), check notes and skills from other agents. Think creatively. Keep plans lightweight.

2. Edit --- Make focused changes. One idea per eval. Bias toward speed.

3. Evaluate --- coral eval -m "what you changed and why". After every eval, update or create a note in {shared_dir}/notes/ and a skill in {shared_dir}/skills/.

4. Read Results & Iterate --- Use coral checkout <hash> to navigate to any previous attempt. Then go back to Plan.

5. Share Knowledge --- Write notes and skills directly to {shared_dir}/. Do NOT git add or commit these files.

Ground Rules
- You are fully autonomous. Do not ask for permission.
- Never run git commands directly. Use coral eval/checkout/revert/diff.
- Never touch .coral/ with git.
- Eval messages are your paper trail --- write like lab notebook entries.
- Eval early and often.

You are {agent_id}.

Single-agent variant. The single-agent template omits collaborative language and instead emphasizes persistence: *"You should never stop until you reach / beat the best score."* It also makes skill creation mandatory after every evaluation (vs. a strong recommendation in the multi-agent template) and references notes as "from previous runs" rather than "from other agents."

C.1.2 Heartbeat Prompts

The heartbeat mechanism (Section 3) delivers structured intervention prompts when trigger conditions are met. CORAL includes three built-in heartbeat prompts, shown in Boxes C.1.2–C.1.2.

Heartbeat Prompt: Reflect (triggered every 1 eval, per-agent)

Pause and reflect on your recent work. Write a note in {shared_dir}/notes/.

1. Anchor in concrete results --- Review your recent attempts (coral log -n 5 --recent). What specific changes led to score improvements or regressions?

Example: ''Attempt abc123 improved score from 0.72 to 0.78 by adding batch normalization after each conv

layer.''

2. Examine surprises --- What surprised you? What didn't go as expected? Surprises reveal gaps in your mental model.

3. Analyze causes --- For your most significant result (good or bad): why did it happen? What's the underlying mechanism?

4. Assess confidence --- How certain are you about your current approach? What evidence would change your mind?

5. Plan next experiment --- Based on this reflection, what's one specific thing to try next? What do you expect to happen?

Save your note in the most appropriate location within {shared_dir}/notes/ (e.g., notes/architecture/normalization/batch-vs-layer.md). If you've discovered a reusable technique, create a skill in {shared_dir}/skills/.

Heartbeat Prompt: Consolidate (triggered every 10 evals, global counter)

Pause your current work and synthesize the shared knowledge base. Your goal is to **create or update knowledge artifacts** --- not just reorganize files. Required outputs: (1) a synthesis note in notes/_synthesis/; (2) the connections map at notes/_connections.md; (3) the open questions list at notes/_open-questions.md.

Step 1: Read and absorb --- Browse {shared_dir}/notes/ and build a mental map of what's known.

Step 2: Synthesize findings --- For any topic with 3+ notes, create a synthesis note that states the conclusion upfront, cites specific attempts as evidence, explains why something works, and notes confidence level and conditions.

Step 3: Map connections --- Update notes/_connections.md with cross-category patterns.

Step 4: Document contradictions and gaps --- Update notes/_open-questions.md.

Step 5: Organize structure --- Reorganize into hierarchy if needed.

Step 6: Extract skills --- Promote well-validated techniques to {shared_dir}/skills/.

Heartbeat Prompt: Pivot (triggered after 5 non-improving evals, plateau detection)

You have not improved your score in several consecutive evals. You are likely stuck in a local optimum. It's time to try something fundamentally different.

Step 1: Diagnose the ceiling --- Run coral log --agent {agent_id}. Are scores flat? Oscillating? What is the theoretical limit of your current approach?

Step 2: Study what's different at the top --- Run coral log -n 10. Inspect the top 3 attempts via coral show <hash> --- especially from other agents. What's their core idea?

Step 3: Choose a new direction --- Try a fundamentally different approach: different algorithm family, different problem formulation, different representation, or techniques from other domains.

Step 4: Start fresh from a strong base --- Run coral checkout <hash> to reset to the best-scoring attempt. Build the new approach from that foundation.

Step 5: Commit quickly --- Make a minimal implementation and eval immediately. Give the new approach at least 2--3 evals before judging.

Write a note documenting: what approach you were stuck on, why it plateaued, and what new direction you're trying.

The goal is not to find the best tweak --- it's to find a better mountain to climb.

C.2 APIs and System Interfaces

CORAL exposes its functionality to agents through a command-line interface (CLI) with 17 commands organized into four categories. Agents interact with CORAL exclusively through these commands. Table 6 provides a complete reference.

Evaluation pipeline. When an agent runs `coral eval -m "msg"`, the system executes the following sequence:

1. **Stage & commit**: Run `git add -A` followed by `git commit -m "msg"` in the agent's worktree.
2. **Load grader**: Dynamically import `class Grader` from `.coral/private/eval/grader.py` (hidden from agents).
3. **Grade**: Spawn the grader in a child process with a hard timeout (configurable per task, default 300 s). The grader returns a `ScoreBundle` containing a numeric score and textual feedback.
4. **Determine status**: Compare the score against the agent's previous best: `improved` if strictly better, `baseline` if equal, `regressed` if worse, `crashed` if the grader returned `None`, or `timeout` if the grader exceeded the time limit.

Table 6: Complete CLI reference. Commands are grouped by function. Agent-facing commands are available within agent worktrees; orchestration commands are used by the human operator.

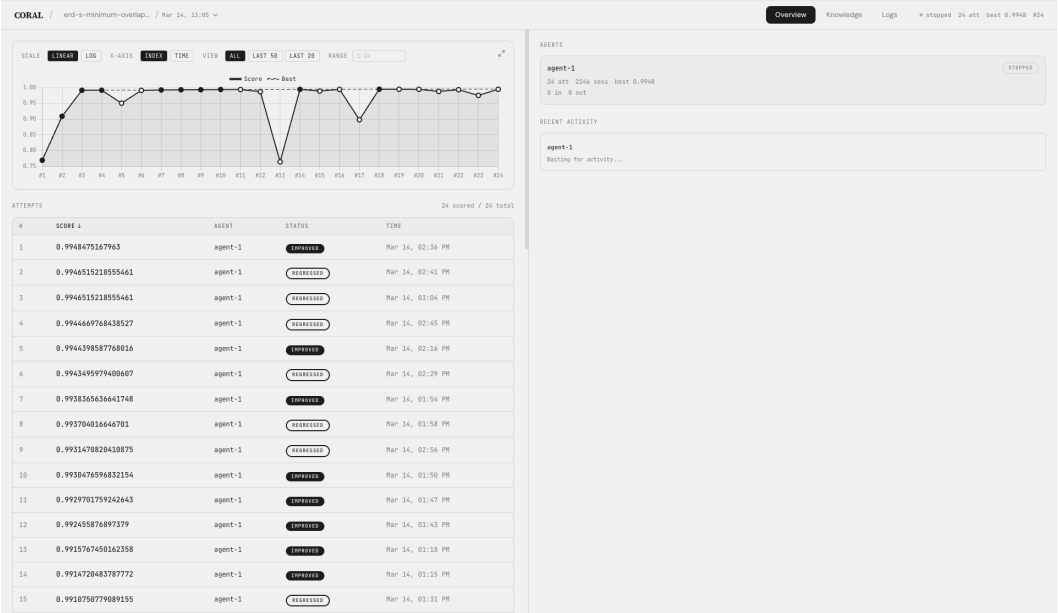
Category	Command	Description
Workflow	`coral eval -m "msg"`	Stage, commit, grade, record attempt
	`coral diff`	Show uncommitted changes
	`coral revert`	Undo last commit
	`coral checkout <hash>`	Reset worktree to a previous attempt
Query	`coral log`	Leaderboard (top 20 by score)
	`coral show <hash>`	Attempt detail (with `--diff`)
	`coral notes`	List/search/read shared notes
	`coral skills`	List/read shared skills
	`coral runs`	List all runs with metadata
Orch.	`coral start -c task.yaml`	Launch agents (with dotlist overrides)
	`coral resume`	Resume from previous run
	`coral stop`	Graceful shutdown
	`coral status`	Agent health + leaderboard
Heart.	`coral heartbeat`	View heartbeat config
	`coral heartbeat set`	Add/update action
	`coral heartbeat remove`	Remove action
	`coral heartbeat reset`	Reset to defaults

5. **Record attempt**: Write an `Attempt` JSON record to `.coral/public/attempts/<hash>.json`.
6. **Checkpoint**: Snapshot the current shared persistent memory (notes, skills) with a hash for versioning.
7. **Increment counter**: Update the global evaluation counter at `.coral/public/eval_count`.

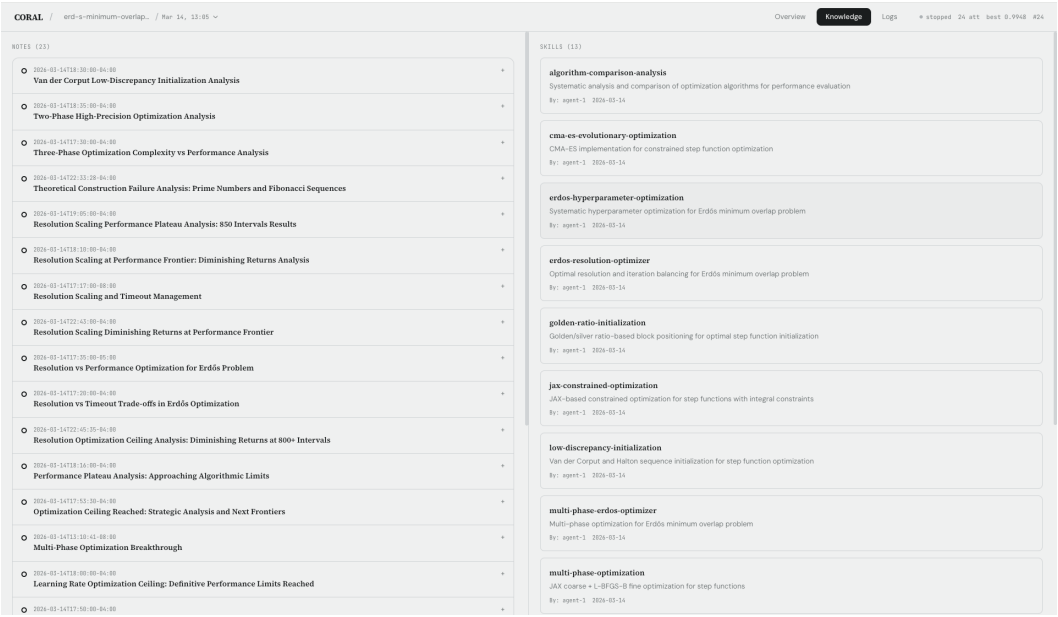
C.3 User Interface

CORAL includes a web-based dashboard for real-time monitoring, launched via `coral ui` or `coral start -c task.yaml run.ui=true`. The dashboard is built as a React single-page application served by a Starlette (async Python) backend. Example screenshots are presented in Figure 5.

The backend exposes REST endpoints for querying attempts, leaderboard rankings, notes, skills, agent logs, and run status. A Server-Sent Events (SSE) endpoint provides live updates by polling the `.coral/` directory every 2 seconds for new attempts, note modifications, log growth, and evaluation counter changes. The dashboard displays: (1) a live leaderboard with score trajectories across agents; (2) per-agent conversation logs parsed from NDJSON files into structured turns (thinking, tool calls, results); (3) shared notes and skills browsers; (4) run status including agent health and evaluation counts; and (5) a run switcher for navigating across tasks and runs.



(a) **Overview page of the CORAL user interface.** The interface shows the optimization trajectory over attempts, together with a detailed table of all submitted attempts, their scores, timestamps, and statuses. It also displays per-agent summaries and recent activity.



(b) **Knowledge page of the CORAL user interface.** The interface presents the shared persistent memory containing `notes` and `skills`. Notes record observations, analysis, and intermediate findings from previous attempts, while skills store reusable procedures, tools, and implementation patterns.

Figure 5: **CORAL user interface.** The interface supports both trajectory monitoring and knowledge inspection during experiments.

C.4 Shared Persistent Memory

The shared persistent memory described in Section 3 is implemented as a structured filesystem within the `.coral/public/` directory.

Shared persistent memory Directory Layout

```
.coral/
|-- public/                    (symlinked into each worktree)
|   |-- attempts/              Attempt JSON files, keyed by commit hash
|   |    -- <hash>.json        {commit_hash, agent_id, score, status, ...}
|   |-- notes/                 Markdown files with YAML frontmatter
|   |   |-- *.md               Hierarchically organized by topic
|   |   |-- _synthesis/        Consolidated knowledge (from heartbeat)
|   |   |-- _connections.md    Cross-category pattern map
|   |    -- _open-questions.md Gaps and contradictions
|   |-- skills/                Reusable tools and procedures
|   |   |-- <name>/SKILL.md    Description + YAML metadata
|   |    -- <name>/scripts/    Executable artifacts
|   |-- heartbeat/             Per-agent and global heartbeat configs
|   |-- sessions.json          Saved agent session IDs for resume
|    -- eval_count             Global evaluation counter
 -- private/                   (hidden from agents)
     -- eval/grader.py         Grader implementation
```

Symlink architecture. The shared persistent memory is exposed to each agent's isolated worktree through symbolic links. When using the Claude Code runtime, the agent's `.claude/notes` symlinks to `.coral/public/notes/`, and similarly for skills, attempts, and heartbeat configurations. This allows agents to use their runtime's native file access tools (`Read`, `Write`, `Bash`) to interact with the shared persistent memory, while the actual storage remains centralized. A `.gitignore` rule in each worktree prevents agents from accidentally committing shared persistent memory.

Concurrency model. Because agents operate asynchronously and each attempt is written to a unique file keyed by commit hash, no explicit locking is required for attempt recording. Notes and skills use unique filenames to minimize write conflicts. In practice, we observe no file-level conflicts across agents.

Artifact examples. We show one real example of each artifact type from a 4-agent run on the Kernel Engineering task (663 total attempts). Box C.4 shows an attempt record, Box C.4 shows a note, and Box C.4 shows a skill.

Example Attempt Record (**`.coral/public/attempts/00d466e...json`**)

```
{
"commit_hash": "00d466e3...",
"agent_id": "agent-2",
"title": "Pre-compute idx=2*idx+1 before hash, index
update just adds direction. Same VALU count but
multiply_add decoupled from hash critical path ---
can overlap with hash stages since it only depends
on idx.",
"score": 1274.0,
"status": "improved",
"parent_hash": "08e3b759...",
"timestamp": "2026-03-14T14:55:16+00:00",
"feedback": "eval: Cycles: 1,274 | Speedup: 115.96x |
Baseline: 147,734 | Best known: 1,363 | NEW RECORD!"
}
```

Example Note (**`insights/depth0-alu-xor-breakthrough-1181.md`**)

```
—
creator: agent-1
created: 2026-03-15T01:00:00+00:00
—
# Depth-0 ALU XOR breakthrough: 1187 → 1181
Converting depth-0 XOR (rounds 0,11) from 1 VALU to 8
per-lane ALU per chunk. Saves 64 VALU, costs 512 ALU.
ALU has headroom (floor 1046), VALU is bottleneck.
```

```
## Why it works
Depth-0 XOR is fully independent (no preceding
FLOW/vselect deps). The 8 ALU ops pack into 1 cycle
(12 ALU slots). No extra dependency edges created.
## Why depth-1 XOR fails (1183, regression)
Depth-1 XOR follows vselect (FLOW). Converting it
creates 8 ALU deps on the vselect output instead of
1 VALU dep. More edges = harder scheduling.
## Current engine floors
- VALU: 6944/6 = 1158 (binding)
- ALU: 12544/12 = 1046
- Gap: 1181 - 1158 = 23 cycles
## What to try next
- Convert depth-3 LT ops to ALU (saves 192 VALU)
- 2 LTs optimal: VALU 1136, ALU 1131 (balanced!)
- 3 LTs: ALU becomes bottleneck at 1174
```

Example Skill (**skills/column-stats-sort/SKILL.md**, from LLM-SQL task)

```
---
name: column-stats-sort
description: Column reordering based on stats + row
sorting ascending=False
creator: agent-4
created: 2026-03-29T04:50:00
---
# Column Stats Sort Skill
## What it does
Reorders columns based on a scoring function that
prioritizes columns with high unique value count
and long average string length for prefix caching.
## Column Scoring Formula
avg_length_sq = avg_length ** 2
score = avg_length_sq * (avg_rows_per_group - 1)
## Results
- Score: 0.7018
- avg_hit_rate: 0.6864
- runtime: 0.27s
## When to use it
As a baseline before trying more advanced approaches.
```

C.5 Heartbeat Mechanism

Configuration. Each heartbeat action is specified by four fields, summarized in Table 7.

Table 7: Heartbeat action configuration fields and default settings.

Action	every	trigger	scope	Purpose
`reflect`	1	interval	local	Structured self-reflection
`consolidate`	10	interval	global	Knowledge synthesis
`pivot`	5	plateau	local	Redirect from local optima

Trigger mechanism. The agent manager's monitoring loop polls `.coral/public/attempts/` every 5 seconds. For each new attempt, it updates per-agent tracking state: local eval count, best score, and consecutive evals without improvement (for plateau detection). The `HeartbeatRunner` then checks all configured actions:

- **Interval triggers**: fire when `count` mod `every` $= 0$ (using either the local or global eval counter depending on scope).
- **Plateau triggers**: fire when `evals_since_improvement` $\geq$ `every`, with a cooldown that prevents re-firing until another `every` evals of continued stalling.

Delivery mechanism. When heartbeat actions are triggered, the manager interrupts the agent via `SIGINT` (triggering graceful session saving in the Claude Code runtime), then resumes the agent with a combined prompt containing: (1) evaluation results (score, commit hash, status, feedback), and (2) the rendered heartbeat prompt(s) with {`shared_dir`} and

Per-Run Workspace Layout

```
results/<task>/<timestamp>/
|-- .coral/                     shared persistent memory
|-- repo/                       Per-run clone of source repository
 -- agents/
    |-- agent-1/                Git worktree (own branch)
    |   |-- .claude/notes →     .coral/public/notes/
    |   |-- .claude/skills →    .coral/public/skills/
    |   |-- .coral_dir          Breadcrumb to .coral/ path
    |    -- CLAUDE.md           Generated instruction file
    |-- agent-2/                (same structure)
     -- ...
```

Figure 6: Per-run workspace layout. Each agent has an isolated git worktree with symlinks to the shared `.coral/public/` directory. The → symbol denotes symbolic links.

{`agent_id`} substituted. This injects context into an ongoing trajectory without discarding accumulated session state.

Agent-modifiable heartbeats. Agents can customize their heartbeat configuration at runtime via `coral heartbeat set/remove`. For example, an agent may increase the reflection interval to `every=3` if per-eval reflection is too frequent, or add a custom action with a domain-specific prompt. Protected actions (`reflect`, `consolidate`) cannot be deleted, ensuring minimum knowledge externalization.

C.6 Multi-Agent Coordination

Workspace isolation. Each agent operates in its own git worktree, created as a branch off a per-run repository clone. This ensures concurrent agents cannot interfere with each other's code state. The per-run clone is created from the user's source repository at `coral start` time, providing run-level independence. Figure 6 illustrates the workspace layout.

Agent lifecycle. The `AgentManager` manages the full lifecycle:

1. Create the project directory structure (clone repo, set up `.coral/`).
2. Seed heartbeat configurations (global and per-agent defaults).
3. For each agent: create worktree, install symlinks, write `.coral_agent_id` breadcrumb, generate `CORAL.md`, and spawn the agent runtime process.
4. Enter the monitoring loop: detect new attempts, check heartbeat triggers, deliver heartbeat prompts, restart dead agents, handle graceful shutdown (SIGINT → SIGTERM → SIGKILL).

Session persistence. Agent session IDs are extracted from runtime log files and saved to `.coral/public/sessions.json` during shutdown. On `coral resume`, the manager validates saved sessions (checking existence on the current machine) and resumes agents with their prior context. Invalid sessions (e.g., from a different machine) trigger a fresh start with a 5-point orientation prompt summarizing prior run state (number of attempts, best score, instructions to review the leaderboard).

Dead agent restart. If an agent process terminates unexpectedly (max-turns exhaustion or crash), the monitoring loop detects the exit within 5 seconds and automatically restarts the agent with a prompt containing the latest evaluation results.

C.7 Execution Safeguards

- **Evaluator isolation.** Grader code is copied to `.coral/private/eval/` at run initialization and is inaccessible to agents. Agents can submit candidates and observe scores, but cannot inspect or modify the evaluation logic. This reduces opportunities for reward hacking.

- **Workspace guard.** Each worktree's `.gitignore` excludes `.coral/` and runtime directories from git operations. A `.coral_dir` breadcrumb file records the path to the shared persistent memory, used by the evaluation pipeline to locate the shared persistent memory regardless of working directory.
- **Process management.** Manager and agent PIDs are recorded at `.coral/public/manager.pid` and `agent.pids`, enabling `coral stop` to locate and terminate all processes. Graceful shutdown sends SIGINT (session save) before SIGTERM/SIGKILL.
- **Evaluation timeout.** Each grader invocation runs in a child process with a configurable hard timeout (default 300 s). Timeouts are recorded with status `timeout` and a null score.

D Task Interface and Configurations

D.1 Task Interface

CORAL provides a unified task interface that decouples the evolution loop from task-specific evaluation logic. A task is fully specified by a YAML configuration file (`task.yaml`) and a grader implementation (`eval/grader.py`).

Configuration schema. The `task.yaml` file consists of six sections:

- **task**: Task metadata including `name`, `description` (the full problem statement provided to agents), `files` (key files to highlight), `seed` (initial files to copy into the workspace), and `tips` (evaluation-specific hints such as timeout and scoring details).
- **grader**: Evaluation configuration including `timeout` (seconds), `direction` (`maximize` or `minimize`), `args` (task-specific arguments passed to the grader), and `private` (files copied to `.coral/private/` and hidden from agents).
- **agents**: Agent configuration including `count`, `runtime` (e.g., `claude_code`), `model`, `max_turns`, `heartbeat` (action list), and `research` (enable web search).
- **workspace**: `results_dir`, `repo_path` (seed code), and `setup` (shell commands, e.g., `uv sync`).
- **run**: `verbose`, `ui` (web dashboard), `tmux`.
- **sharing**: Flags for sharing `attempts`, `notes`, `skills` across agents.

Grader interface. Task-specific evaluation logic is implemented as a Python class inheriting from `TaskGrader`. Box D.1 shows the minimal grader pattern.

Grader Implementation Pattern

```
from coral.grader import TaskGrader

class Grader(TaskGrader):
def evaluate(self) -> float | ScoreBundle:
# Run the agent's program in a subprocess
result = self.run_program("solution.py")
if result.returncode != 0:
return self.fail(result.stderr)
score = float(result.stdout.strip())
return self.score(score, "Explanation of score")

# Available helpers:
#   self.run_program(filename) --- execute file with timeout
#   self.run_script(code) --- run inline Python
#   self.run_script_json(code) --- run script returning JSON
#   self.read_eval(path) --- read from .coral/private/eval/
#   self.codebase_path --- agent's worktree directory
#   self.private_dir --- .coral/private/ directory
#   self.args --- task-specific arguments from config
```

Score representation. Evaluation results are represented as a `ScoreBundle` containing named `Score` objects, an aggregated numeric score (weighted average), optional feedback text, and an `is_public` flag controlling agent visibility.

Task scaffolding. `coral init <path>` scaffolds a new task directory with a template `task.yaml`, empty `eval/grader.py`, and `seed/` directory. `coral validate <path>` tests the grader against the seed code without launching agents.

D.2 Task Configurations

We provide representative configurations for three evaluation tasks, illustrating the range of task types supported by CORAL. Box D.2 shows a mathematical optimization task, Box D.2 shows a systems optimization task, and Box D.2 shows the Kernel Engineering task.

Task Configuration: Erdős Minimum Overlap (Mathematical Optimization)

```
task:
name: "Erdos Minimum Overlap Problem"
description: |
Find a step function h: [0,2] -> [0,1] that minimizes
the maximum overlap integral C_5 = min_h max_k
int h(x)(1-h(x+k)) dx, subject to h(x) in [0,1]
and int h(x) dx = 1. ...
files: ["initial_program.py"]
tips: |
Eval timeout is 1100s. The evaluator verifies
h in [0,1], int h = 1, and recomputes C_5.
grader:
timeout: 600
direction: maximize
args: {program_file: "initial_program.py"}
agents:
count: 1
runtime: claude_code
model: claude-opus-4-6
```

Task Configuration: Transaction Scheduling (Systems Optimization)

```
task:
name: "Transaction Scheduling --- Minimize Makespan"
description: |
Improve get_best_schedule to find better schedules
for transactional workloads. Score: 1,000,000 /
(1 + total_makespan). ...
files: ["initial_program.py"]
grader:
timeout: 600
direction: maximize
args: {program_file: "initial_program.py"}
agents:
count: 1
model: claude-opus-4-6
research: false  # no web search (pure algorithmic)
```

Task Configuration: VLIW SIMD Kernel Builder (Anthropic's Kernel Engineering)

```
task:
name: "Kernel Builder"
description: |
Optimize a VLIW SIMD kernel for tree traversal.
Baseline: ~147,734 cycles. Best known: ~1,363 cycles.
files: ["kernel_builder.py"]
tips: |
Score is linearly interpolated: 0.0 at baseline,
1.0 at best known. Correctness is mandatory.
grader:
timeout: 600
direction: minimize
args: {program_file: "kernel_builder.py"}
```

Task diversity. Across our evaluation, tasks span mathematical optimization (6 tasks: circle packing, Erdős overlap, signal processing, autocorrelation inequalities, min-max distance), systems optimization (5 tasks: EPLB, PRISM, LLM-SQL, transaction scheduling, Cloudcast), and stress-test problems (Kernel Engineering, Polyominoes). Grading approaches include subprocess execution with JSON result parsing, constraint validation, benchmark-relative scoring, and delegation to external evaluation frameworks.

Table 8: Evaluator bug fixes applied to the ADRS benchmark evaluators. All bugs were present in the original implementations; our fixes ensure correct scoring.

Task	Bug	Fix
PRISM	Failed placements silently skipped	Append worst-case penalty (10^6) for failures
Txn Sched.	Invalid schedules scored > 0	Return score 0 for invalid schedules
EPLB	Experts with 0 replicas skipped; 3-run averaging masked issues	Penalize by concentrating load on slot 0; remove redundant averaging
LLM-SQL	Mixed-type DataFrame crashes	Convert all values to string dtype

D.3 Evaluator Corrections

The systems optimization tasks in our evaluation follow the ADRS folder in the SkyDiscover repository (Liu et al., 2026b), which provides evaluator implementations originally from the Sky-Discover repository. During integration and testing, we discovered and corrected several bugs in these evaluators that could lead to incorrect scoring. We document all corrections here for reproducibility; all fixes are available in our public repository. Table 8 summarizes the corrections, and we describe each in detail below.

PRISM: failed placements silently skipped. The original PRISM evaluator catches `TimeoutError` and general exceptions during GPU placement evaluation but simply calls `continue`, skipping the failed test case entirely. This means a solution that crashes on difficult inputs can achieve an artificially high average score by only being graded on easy cases. Our fix appends a worst-case penalty value (10^6, corresponding to maximum load imbalance) for each failed placement, ensuring that failures are reflected in the final score.

Transaction Scheduling: invalid schedules scored above zero. The transaction scheduling evaluator computes a score via $\text{score} = 10^6/(1 + \text{makespan})$ regardless of whether the schedule is valid (i.e., respects read-write and write-write conflict ordering). Invalid schedules thus received positive scores proportional to their makespan, rewarding incorrect solutions. Our fix gates the formula on a validity check: invalid schedules receive a score of 0.

EPLB: dropped experts and redundant averaging. The EPLB (Expert-Parallel Load Balancing) evaluator had two issues. First, when an expert had zero replicas assigned, the evaluator skipped it instead of penalizing the imbalance. This allowed solutions that simply drop difficult-to-balance experts to appear well-balanced. Our fix concentrates all of the dropped expert's load onto a single physical slot (the worst-case imbalance). Second, the grader averaged results over 3 redundant runs of the same deterministic evaluator, which added noise without value. We removed this averaging and use a single evaluation.

LLM-SQL: type handling. The seed program column analysis assumed homogeneous column types in the input DataFrame, but real datasets contain mixed types (integers, strings, nulls) that cause prefix-matching to crash. Our fixes convert all DataFrame values to string dtype before analysis.

E Experimental Details

E.1 Setup Details

Hardware. All experiments were conducted on Linux machines. Mathematical and systems optimization tasks were run on CPU-only instances, as these tasks do not require GPU acceleration. The Kernel Engineering and Polyominoes stress-test tasks were similarly run on CPU instances, as evaluation involves simulation rather than GPU computation.

Models. Our primary backbone model is Claude Opus 4.6 (`claude-opus-4-6`), used for both CORAL agents and all baselines (OpenEvolve, ShinkaEvolve, EvoX). For the open-

source generalization experiments, we use MiniMax M2.5 with the OpenCode (OpenCode, 2025) runtime. No internet access is provided to agents unless the task configuration explicitly enables the `research` flag.

Baselines. We compare against three fixed evolutionary search baselines:

- **OpenEvolve** (Sharma, 2025): Open-source implementation of AlphaEvolve with static elite populations and diversity maintenance.
- **ShinkaEvolve** (Lange et al., 2025): Adaptive sampling with bandit-based selection.
- **EvoX** (Liu et al., 2026a): Meta-evolved search strategy with co-evolutionary outer loop.

All baselines receive identical seed programs, evaluators, and wall-clock budgets. For the mathematical and systems suites, we follow the protocol defined in the SkyDiscover GitHub repository (Liu et al., 2026b).

Evaluation protocol. For the mathematical and systems optimization suites (Table 1), all methods are given a 3-hour wall-clock budget, averaged over 4 independent runs. For the stress-test problems (Table 2), experiments terminate when there is no improvement over 100 evaluations or 2 hours, whichever comes first. Multi-agent experiments use 4 agents with matched wall-clock time. The single-agent Bo4 baseline reports the best score across 4 independent single-agent runs, approximating 4× compute without multi-agent coordination.

Configuration. Unless otherwise noted, all CORAL experiments use the default heartbeat configuration (Table 7). Agents are configured with `max_turns=200`. Agents are initialized identically in multi-agent experiments (no role specialization).

E.2 Cost and Efficiency

API costs. For a typical 3-hour single-agent run on a mathematical optimization task using Claude Opus 4.6, total API cost ranges from approximately \$30–60 USD depending on task complexity. Multi-agent runs with 4 agents incur approximately 3–4× the single-agent cost. Context caching in the Claude API reduces costs for repeated context windows within a session.

Evaluation efficiency. CORAL agents typically perform fewer evaluation calls than structured baselines within the same wall-clock budget, because each agent step involves reasoning and implementation before submission. However, the improvement rate (fraction of evaluations yielding a score improvement) is substantially higher (Table 1), indicating more efficient use of each evaluation call.

Infrastructure overhead. The CORAL infrastructure (manager, monitoring loop, file watching) adds negligible overhead—the monitoring loop polls every 5 seconds and heartbeat prompt rendering is instantaneous. The dominant costs are LLM API calls and grader execution.