STATEFUL SEARCH WITH STATELESS AGENTS: SCALING LONG-HORIZON AUTOMATED RESEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Automated research systems increasingly run LLM agents over long horizons, but more inference does not by itself produce more progress: agents replay ever-growing histories, duplicate one another's work, or stop experimenting while token consumption continues. Yet most evaluations use short budgets or benchmarks that saturate early, leaving these failure modes untested. We introduce Agentic Evolutionary Search (AES), built on the principle of stateful search with stateless agents: the harness owns candidate solutions and measured outcomes, and reconstructs a fresh, role-specific context for every Advisor epoch and Worker trial. The Advisor turns accumulated evidence into concrete assignments for parallel Workers. We evaluate AES against three recent frameworks on software engineering, kernel optimization, and algorithm design at budgets of up to one billion cumulative tokens. AES achieves the best final result on every evaluated task and reaches the strongest kernel baseline's final performance with over 84% fewer tokens. Ablations from shared checkpoints show that focused contexts and explicit assignments each contribute, with effects that compound over full runs, while the Advisor consumes less than 0.6% of total tokens. These results argue for keeping durable research state out of agent conversations, and show that short evaluation horizons can misjudge both research systems and their components.

1 INTRODUCTION

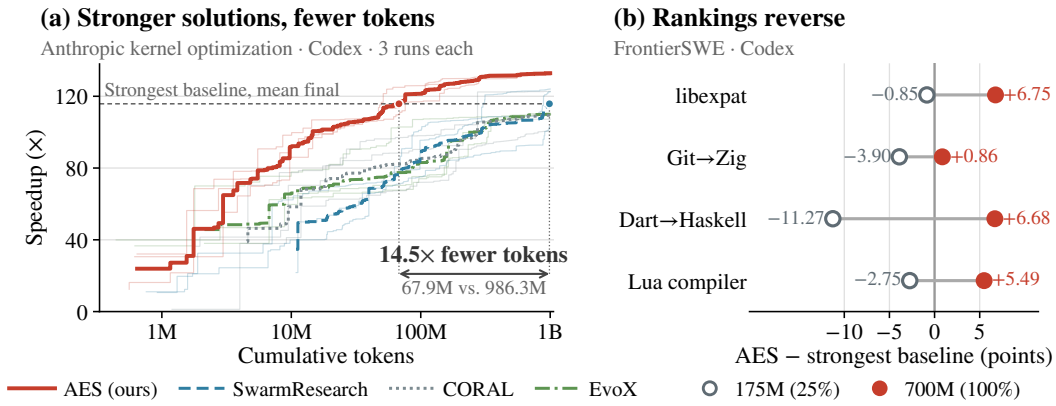


Figure 1: **Scaling long-horizon automated research.** (a) Bold: mean of three runs; thin: individual runs. (b) AES overtakes the strongest per-task baseline as the budget grows.

Large language model (LLM) agents are increasingly used for automated research (AutoResearch): iteratively proposing, implementing, and evaluating solutions to problems with measurable objectives but no known optimum (Romera-Paredes et al., 2024; Novikov et al., 2025; Mang et al., 2025; Qu et al., 2026; Virk et al., 2026). As these systems take on harder problems, what matters is not only how quickly they find good solutions, but whether additional computation keeps producing better ones. **How can automated research sustain progress as its inference budget grows?**

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.

Existing evaluations rarely run long enough to answer this question. Most measure budgets in iterations or model calls rather than tokens, and widely used benchmarks saturate early: on 26-circle packing, four methods finish within 2.3×10^{-10} of one another by 5M tokens (Figure 2(a)). A saturated benchmark measures how quickly a method reaches the ceiling, not whether it can keep converting computation into progress. As a result, design choices whose effects compound as experience accumulates, such as context management and experiment selection, remain largely untested.

Long runs expose two challenges. First, **retaining experience while controlling context**: candidate solutions, evaluation results, and failed attempts are reusable evidence, but replaying a growing history consumes tokens and degrades agent behavior (Hong et al., 2025; Xia et al., 2026). Second, **turning accumulated evidence into productive exploration**: recorded findings do not determine what to try next. In one CORAL (Qu et al., 2026) run on a GPU kernel task, the last improvement arrives at 231M tokens; afterward, agents repeatedly declare their solutions optimal, and 98% of the final 1,500 sessions make no tool calls while token consumption continues (Figure 2(b)). **Scaling inference alone does not guarantee productive search.**

We introduce **Agentic Evolutionary Search (AES)**, built on one principle: **stateful search with stateless agents**. The harness, not the agents, owns the research state (candidate solutions and measured outcomes) and reconstructs a fresh, role-specific context for every Advisor epoch and Worker trial; no agent carries its conversation across these boundaries. What each agent sees thus becomes an explicit design choice, not a by-product of accumulated conversation. AES addresses each challenge with one mechanism. **Harness-controlled context reconstruction** gives the Advisor structured evidence across search directions and each Worker only its assignment, retained candidate, and local feedback. **Evidence-driven work assignment** makes the Advisor turn global outcomes into concrete, coordinated experiments, while Workers refine local candidates so alternative directions can mature before surpassing the global best.

This division of responsibility distinguishes AES from prior designs. In program-database methods, a fixed sampling policy selects parents and each generation is a single model call (Romera-Paredes et al., 2024; Novikov et al., 2025); in AES, an Advisor reasons over structured evidence to choose experiments, and Workers are multi-step coding agents. In agentic frameworks, agents either choose their own experiments (Qu et al., 2026) or are steered by a coordinator that is barred from proposing ideas (Virk et al., 2026); in AES, the Advisor assigns concrete experiments, and the harness decides what enters each context. Harness-owned state also makes AES an experimental instrument: our ablations resume from identical checkpointed research states.

We evaluate AES against EvoX (Liu et al., 2026), CORAL (Qu et al., 2026), and SwarmResearch (Virk et al., 2026) on long-horizon software engineering, kernel optimization, and algorithm design at budgets of up to one billion cumulative tokens, measuring both solution quality at matched budgets and the tokens needed to reach a common target. Our evaluation yields three findings:

- **AES finds stronger solutions with fewer tokens.** AES achieves the best full-budget result on every evaluated task. On Anthropic kernel optimization with Codex, all three AES runs outperform all nine baseline runs, and AES reaches the strongest baseline's final performance with 93% fewer tokens (84% with Claude Code). On FrontierSWE, it improves scores by 4.95 points on average over the strongest per-task baseline.
- **Focused contexts and explicit assignments each contribute, and their effects compound.** From shared checkpoints, removing Advisor context reconstruction, Worker context isolation, or Advisor assignments reduces mean progress in every comparison. Removing Worker isolation costs 5–11 cycles over 100M-token continuations but 224 cycles over a full 1B-token run. Without assignments, parallel Workers repeatedly implement the same feature (Appendix C.3).
- **Explicit coordination is cheap.** The Advisor consumes 0.24–0.51% of tokens and 1.2–2.3% of model cost, compared with 8.4–10.7% of tokens and 6.1–7.4% of cost for SwarmResearch's Shepherd. Because coordination is separated from implementation, the Worker model can also be chosen per task (§4.3).

Our results also show that short horizons can misjudge methods, components, and configurations: AES trails on all four FrontierSWE tasks at 175M tokens but leads at 700M (Figure 1(b)), and wider Worker pools that trail by 337 cycles at 25% of the budget finish within 4 cycles of the narrowest (§4.3). We therefore report results at multiple cumulative token budgets and encourage future evaluations to do the same. We will open source our code and experiment traces.

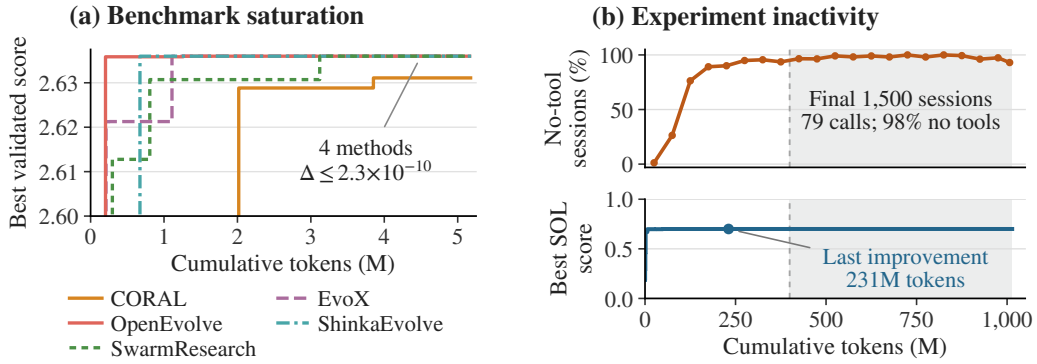


Figure 2: **Long-horizon AutoResearch requires benchmark headroom and sustained experimentation.** (a) 26-circle packing saturates early, limiting its ability to distinguish methods at larger budgets. (b) In a CORAL run on a GPU kernel task, token consumption continues while experimentation and progress stall.

2 Background and Motivation

2.1 Related Work on Automated Research (AutoResearch)

Automated research (AutoResearch) uses LLMs to iteratively propose, implement, and evaluate ideas for open-ended scientific and engineering problems. It targets problems with measurable objectives but often no known optimal solution. A major line of work embeds LLM-generated programs within evolutionary search loops, using automated evaluation to guide improvements. Examples include AlphaEvolve (Novikov et al., 2025), OpenEvolve (Sharma, 2025), ShinkaEvolve (Lange et al., 2026), EvoX (Liu et al., 2026), and AdaEvolve (Cemri et al., 2026), spanning mathematical optimization, algorithm design, and systems optimization.

Recent frameworks, including CORAL (Qu et al., 2026) and SwarmResearch (Virk et al., 2026), place more of the experimental process under the control of tool-using coding agents like OpenAI Codex (OpenAI, 2025) and Claude Code (Anthropic, 2025). Agents can inspect code, implement ideas, run tests, and revise their approach across multiple steps, while multiple agents explore alternative directions. This shift motivates studying how such research processes sustain progress and use computation efficiently over extended horizons.

Other work studies broader AI scientist systems that automate or assist multiple stages of scientific discovery, including hypothesis generation, literature review, experimentation, and scientific writing (Lu et al., 2024; Yamada et al., 2025; Gottweis et al., 2025). We instead focus on open-ended, evaluator-grounded search over executable solutions, allowing us to isolate how agentic research processes sustain progress as the search horizon and compute budget grow.

2.2 Long-Horizon Evaluation and Its Challenges

Measuring the research horizon. Most AutoResearch evaluations define budgets by iterations, model calls, or evaluator queries. CORAL (Qu et al., 2026) runs agents over extended wall-clock periods but does not report model tokens. Wall-clock time conflates model latency, evaluator runtime, and systems overhead, and is sensitive to parallelism: under the same wall-clock budget, a multi-agent system can consume more inference than a single agent. We therefore measure the horizon in cumulative input and output tokens (*cumulative tokens*), which, with fixed models and inference settings, provide a controlled and reproducible scaling axis. Long horizons also require benchmarks with headroom: on 26-circle packing, popularized by AlphaEvolve (Novikov et al., 2025), two methods come within 10^{-3} of the best observed score by 0.67M tokens, after which the benchmark mainly measures how quickly methods reach the ceiling (Figure 2(a)).

Retaining experience while controlling context. Candidate solutions, evaluation results, and failed attempts may become relevant only later: in our CORAL Lua trace, an agent revisits an

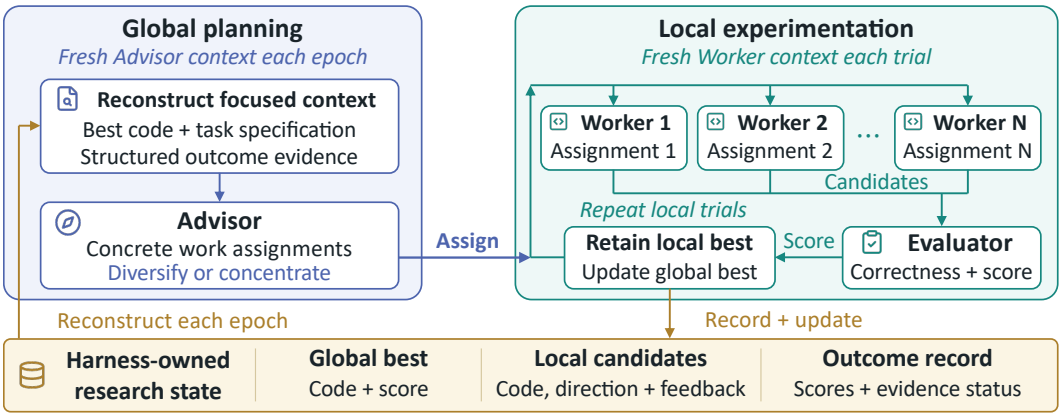


Figure 3: **AES combines focused context reconstruction with evidence-driven work assignment.** The harness owns candidates and outcomes and rebuilds a fresh, role-specific context for the Advisor each epoch and for each Worker each trial (§3.1); the Advisor turns global evidence into coordinated assignments (§3.2).

older experimental note and improves a later leading candidate by implementing a missing formatting feature (Appendix C.1). Yet repeatedly processing a growing history consumes tokens, and accumulated context can impair agent behavior (Hong et al., 2025), including premature stopping in long-horizon search (Xia et al., 2026). The challenge is to preserve reusable evidence while selecting what each agent needs for its current decision.

Turning accumulated evidence into productive exploration. Recorded findings do not determine what to try next. In our CORAL libexpat trace, two agents implement the same feature from the same parent and obtain the same score, then repeat the overlap on the next feature despite one agent posting a note claiming it (Appendix C.2). In the GPU kernel run of Figure 2(b), agents repeatedly declare their candidates optimal and wait, despite prompts to continue, restarts, and context replay; additional inference largely reiterates the decision to stop. The challenge is to turn accumulated outcomes into concrete next experiments, coordinate parallel effort, and revise directions as evidence arrives.

3 Agentic Evolutionary Search (AES)

We present **Agentic Evolutionary Search (AES)**, a framework for scaling long-horizon automated research (Figure 3). AES alternates global planning with local experimentation. Each epoch, the harness reconstructs the Advisor's context from persistent research state (§3.1), and the Advisor assigns concrete experiments to parallel Workers (§3.2). Each Worker runs several local trials in an isolated workspace; an independent evaluator scores every candidate, and the harness records outcomes and updates each Worker's local candidate and the global best until the budget is exhausted.

Relation to native subagent orchestration. Coding agents increasingly support native subagents, which provide parallel execution and separate contexts. These are mechanisms rather than policies: the orchestrating agent still accumulates its own conversation and decides for itself what each subagent sees and when to stop. AES is compatible with such tools but places these decisions in the harness, which reconstructs every context from persistent evidence and enforces isolation, so research continuity never depends on an agent's conversation history.

3.1 Harness-Controlled Context Reconstruction

AES separates what the system retains from what each agent sees. The Advisor needs evidence across search directions to choose experiments; each Worker needs only the local information required to develop its assignment. The harness therefore rebuilds both contexts from persistent research state at every Advisor epoch and Worker trial.

Advisor context: evidence for global decisions. Each epoch, the Advisor receives the task specification, the current global best code and score, and a programmatically constructed evidence summary: recent attempts, outcomes aggregated by search direction, Worker status, and recent progress, plus task-specific diagnostics when available. The summary separates measured non-improvements from inconclusive implementation, runtime, or correctness failures, so that a failed execution is not mistaken for evidence that a direction is exhausted. The Advisor sees only this reconstructed context, which, like PEEK's context map (Gu et al., 2026), lets it compare directions without interpreting the full chronological record.

Worker context: evidence for focused experimentation. Each trial, a Worker receives the task specification, its assignment, its retained candidate, and compact local feedback: the previous trial's status, measured score, and available correctness diagnostics. Workers cannot access other Workers' workspaces or the full research history; evidence from other directions reaches them only through the Advisor's assignments. Because candidate code and feedback persist across conversation resets, Workers build on prior progress without inheriting the conversations that produced it.

3.2 Evidence-Driven Work Assignment

Workers acting only on local information may repeat attempts made elsewhere or overlook useful findings. AES therefore assigns experiment selection to the Advisor, whose reconstructed context spans all search directions. Whereas SwarmResearch's coordinator steers its population with global context but leaves ideation to Search Agents (Virk et al., 2026), the AES Advisor turns accumulated outcomes into concrete, coordinated assignments.

Each epoch, the Advisor revisits these assignments, spreading Workers across directions or concentrating them on distinct approaches to a promising bottleneck. Workers retain flexibility in how they implement and refine an assignment over several local trials. The harness keeps improvements to each Worker's local candidate even when it trails the global best, giving alternative approaches time to mature. When the Advisor redirects a Worker, the Worker restarts from the global best, and its first successfully evaluated candidate becomes a new local baseline, so a new direction can develop even if it initially performs worse.

3.3 Implementation

AES runs OpenAI Codex (OpenAI, 2025) or Claude Code (Anthropic, 2025) as the Advisor and Workers. Context boundaries are enforced rather than requested: Workers run as unprivileged processes under Linux Landlock filesystem restrictions, with restricted network access and saved sessions cleared between trials. Candidates are evaluated outside Worker environments with protected evaluator code, and the harness records token usage by role and checkpoints the full research state for recovery and controlled ablations (Appendix A.1).

4 Results

Tasks and datasets. We evaluate on three task categories: (1) long-horizon software engineering using FrontierSWE (Chu et al., 2026); (2) compute kernel optimization using Anthropic's VLIW SIMD kernel optimization task (Anthropic, 2026) and NVIDIA's SOL-ExecBench GPU kernel optimization tasks (Lin et al., 2026); and (3) algorithm design using FrontierCS (Mang et al., 2025). From FrontierSWE, we select four problems that do not require GPU access: libexpat to x86-64 Assembly, Git to Zig, Dart to Haskell, and Lua Native Compiler. From SOL-ExecBench, we select the three problems with the most submissions as of August 1, 2026: #1 (the backward pass through attention softmax, dropout, and value matrix multiplication), #58 (mixture-of-experts token sorting with prefix sums), and #210 (fused residual addition and RMS normalization). From FrontierCS, we select Structured-LWE, a cryptographic task requiring the recovery of valid secret vectors for public structured learning-with-errors instances. Unlike prior evaluations centered on benchmarks like circle packing and expert placement load balancing (Liu et al., 2026; Qu et al., 2026; Virk et al., 2026), we focus on tasks with headroom at our budgets (§2.2).

Language models and coding agents. We evaluate two coding-agent configurations: OpenAI Codex (OpenAI, 2025) CLI v0.152.1 with GPT-5.5, and Claude Code (Anthropic, 2025) v2.1.258

with Claude Opus 4.8. In our main experiments, the Advisor and Workers use the same LLM, so that gains cannot come from a stronger Advisor transferring knowledge to weaker Workers; Appendix B.1 studies a stronger Advisor paired with weaker Workers. In all experiments, we set the reasoning effort to `high` and keep each agent's default automatic context-compaction settings.

Baselines. We compare against three recent AutoResearch frameworks, each run from a pinned upstream release with unmodified search logic (Appendix A.2). **EvoX** (Liu et al., 2026) jointly evolves candidate solutions and search strategies; because it was designed around direct LLM calls, we launch a fresh coding agent for each generation. **CORAL** (Qu et al., 2026) runs parallel agents that select their own experiments and share discoveries through persistent memory and heartbeat-triggered reflection; we use the official implementation with its upstream default configuration. **SwarmResearch** (Virk et al., 2026) uses a Shepherd Agent that coordinates Search Agents through parent selection, agent-type selection, and selective context sharing, but is instructed not to assign specific ideas; we use the authors' official implementation. All methods share the same coding agents, models, seed solutions, public task materials, and protected evaluators, and are stopped by the same external cumulative-token counter.

Replication. Anthropic kernel optimization with Codex uses three independent runs per method, and the checkpoint ablations use three runs per configuration; other configurations use one run, since a single long-horizon run can cost hundreds to thousands of dollars.

4.1 Main Results: Performance and Token Efficiency

Tables 1–3 report best-so-far performance at 25% and 100% of each task's token budget, and the cumulative tokens each method needs to first match the strongest baseline's full-budget result for the same task and agent (NR: not reached).

AES finds stronger solutions at matched token budgets. AES achieves the best full-budget performance on every evaluated task. Relative to the strongest baseline for each task and coding-agent configuration, it reduces Anthropic kernel cycles by 8.1–12.8% across the two coding-agent configurations, improves FrontierSWE scores by 4.95 points on average, and increases SOL-ExecBench scores by an average of 5.5% across the three tasks. It also improves the FrontierCS Structured-LWE score by 1.5 points.

The kernel advantage persists across independent runs. On Anthropic kernel optimization with Codex, all three AES runs outperform all nine baseline runs at 1B tokens: even the worst AES run beats the best baseline run (1122 versus 1191 cycles). AES achieves 1112.0 ± 8.9 cycles, compared with 1275.7 ± 133.9 for SwarmResearch, the strongest baseline by mean performance.

AES reaches strong baseline performance with fewer tokens. Equal numerical improvements can require different amounts of search, particularly when hill climbing must improve already strong solutions: for example, reducing kernel cycles from 1150 to 1100 could be harder than reducing them from 1200 to 1150. We therefore complement final scores with the tokens needed to reach a common target. AES reaches this target earlier in all but one evaluated configuration. On the Anthropic kernel task with Codex, AES needs only 67.9M tokens versus SwarmResearch's 986.3M, a 93.1% reduction; savings reach 95.9% on SOL-ExecBench #58. The exception is SOL-ExecBench #1, where EvoX reaches the target sooner but AES ultimately achieves the higher score: better final performance need not imply faster initial progress.

Method rankings change with the evaluation horizon. On FrontierSWE, AES trails the strongest baseline on all four tasks at 25% of the budget but leads on all four at the full budget (Table 2). Short-budget evaluations would therefore miss its eventual advantage across all four tasks. These reversals motivate evaluating search methods across multiple cumulative token budgets.

4.2 Ablation Studies: Understanding the Design Choices

To assess the contribution of AES's design choices, we compare continued progress from shared AES checkpoints taken at 10% and 20% of each task's full token budget. Each checkpoint preserves

Table 1: **Anthropic VLIW SIMD kernel performance and token efficiency.** Codex cycle counts (↓) are reported as means ± sample standard deviations over three independent runs per method; Claude Code uses one run per method. Codex token counts use the mean curve.

	Anthropic Kernel (Codex)			Anthropic Kernel (Claude Code)		
Method	**Cycles at 250M↓**	**Cycles at 1B↓**	**Tokens to match best baseline (M)↓**	**Cycles at 250M↓**	**Cycles at 1B↓**	**Tokens to match best baseline (M)↓**
EvoX	1478.3 ± 112.0	1343.3 ± 4.0	NR	1174	1153	NR
CORAL	1524.0 ± 101.7	1350.0 ± 134.2	NR	1205	1130	888.3
SwarmResearch	1501.7 ± 185.7	1275.7 ± 133.9	986.3	1648	1243	NR
AES (ours)	**1149.3 ± 18.9**	**1112.0 ± 8.9**	**67.9**	**1079**	**1039**	**138.5**

Table 2: **FrontierSWE performance and token efficiency with Codex.** Performance is measured by official scores (×100, ↑).

	libexpat → x86-64 Assembly			Git → Zig		
Method	**Score at 175M↑**	**Score at 700M↑**	**Tokens to match best baseline (M)↓**	**Score at 175M↑**	**Score at 700M↑**	**Tokens to match best baseline (M)↓**
EvoX	10.71	11.33	NR	19.29	21.42	NR
CORAL	8.74	12.08	NR	19.46	22.55	NR
SwarmResearch	**19.67**	24.58	559.9	**24.32**	27.98	697.3
AES (ours)	18.82	**31.33**	**354.1**	20.42	**28.84**	**630.2**

	Dart → Haskell			Lua Native Compiler		
Method	**Score at 175M↑**	**Score at 700M↑**	**Tokens to match best baseline (M)↓**	**Score at 175M↑**	**Score at 700M↑**	**Tokens to match best baseline (M)↓**
EvoX	9.72	10.66	NR	25.82	65.38	NR
CORAL	18.55	23.26	NR	**82.97**	90.66	682.3
SwarmResearch	**26.70**	38.59	684.1	50.55	65.38	NR
AES (ours)	15.43	**45.27**	**263.6**	80.22	**96.15**	**354.6**

the research state, including accumulated evidence, the global best solution, and local candidates. From each checkpoint, we continue full AES and three variants that respectively remove Advisor context reconstruction, Worker context isolation, or Advisor work assignments. Without Advisor context reconstruction, the Advisor has full access to the repository of past experiments and must retrieve and interpret relevant evidence without a harness-computed summary. Without Worker context isolation, Workers can access the full research history and other Workers' workspaces. Without Advisor work assignments, we withhold Advisor-generated experiment instructions from Workers while preserving all other Advisor-provided information. Each configuration receives an additional 100M tokens per run, with three independent runs per checkpoint. Table 4 reports mean best-so-far performance after continuation.

We additionally run AES without Worker context isolation for a full 1B-token run on Anthropic kernel optimization with Codex; it ends at 1336 cycles, 224 cycles worse than the mean of the three full-AES runs (1112.0 cycles; Table 1). This gap is far larger than the differences observed after 100M-token continuations (Table 4), suggesting that the cost of unfocused Worker contexts compounds over longer horizons.

Focused contexts support research continuity. AES benefits from pairing persistent research state with focused and role-specific contexts (§3.1). Removing either Advisor context reconstruction or Worker context isolation reduces mean performance gains across both tasks and checkpoints. These findings support controlling how accumulated evidence enters each role's context to sustain productive experimentation.

Explicit assignments effectively turn evidence into progress. Removing Advisor assignments reduces mean progress in every comparison, despite starting from the same accumulated research

Table 3: **SOL-ExecBench and FrontierCS performance and token efficiency with Codex.** Performance is measured by SOL scores ($[0, 1]$, ↑) and FrontierCS task scores (↑), respectively.

	SOL-ExecBench #1			SOL-ExecBench #58		
Method	**SOL at 125M↑**	**SOL at 500M↑**	**Tokens to match best baseline (M)↓**	**SOL at 125M↑**	**SOL at 500M↑**	**Tokens to match best baseline (M)↓**
EvoX	**0.710**	0.710	**46.1**	0.773	0.804	354.1
CORAL	0.386	0.552	NR	0.694	0.714	NR
SwarmResearch	0.598	0.670	NR	0.764	0.764	NR
AES (ours)	0.706	**0.763**	152.7	**0.836**	**0.862**	**14.5**

	SOL-ExecBench #210			FrontierCS: Structured-LWE		
Method	**SOL at 125M↑**	**SOL at 500M↑**	**Tokens to match best baseline (M)↓**	**Score at 50M↑**	**Score at 200M↑**	**Tokens to match best baseline (M)↓**
EvoX	0.375	0.386	283.0	34.0	55.0	NR
CORAL	0.359	0.375	NR	23.5	38.0	NR
SwarmResearch	**0.377**	0.378	NR	44.0	58.0	197.0
AES (ours)	**0.377**	**0.393**	**208.3**	**48.5**	**59.5**	**169.6**

Table 4: **Ablations from shared AES checkpoints.** Within each column, all configurations start from the same checkpoint and are evaluated at 100M additional tokens. Headers show start → end cumulative token counts. The checkpoint row reports starting performance; other rows report mean ± sample standard deviation over three independent continuations.

	Anthropic Kernel Cycles ↓		FrontierSWE (libexpat) Score ($\times 100$) ↑	
Configuration	**100M→200M**	**200M→300M**	**70M→170M**	**140M→240M**
Shared checkpoint	1218	1197	16.12	17.69
AES w/o Advisor context reconstruction	1198.3 ± 2.3	1188.7 ± 6.8	16.86 ± 0.64	19.07 ± 0.31
AES w/o Worker context isolation	1203.3 ± 7.1	1174.7 ± 4.5	16.79 ± 0.47	19.77 ± 0.12
AES w/o Advisor work assignments	1195.0 ± 2.6	1182.3 ± 8.1	16.54 ± 0.33	18.85 ± 0.12
AES (full)	$\mathbf{1192.3 \pm 2.5}$	$\mathbf{1170.0 \pm 14.2}$	$\mathbf{18.78 \pm 0.92}$	$\mathbf{20.13 \pm 0.20}$

state. This supports the division of responsibility in §3.2: the Advisor translates global evidence into concrete experiments, while Workers focus on implementation and local refinement. Trace examples in Appendix C.3 show duplicated implementation effort without assignments and successful Advisor-guided refinement of prior improvements in full AES.

4.3 Scaling and Coordination Cost

Coordination is cheap. AES spends over 99% of its tokens on Worker search: the Advisor consumes only 0.24–0.51% of total tokens, compared with 8.39–10.70% for SwarmResearch's Shepherd (Table 5). Because cached inputs make up over 90% of tokens in both methods, token share understates the cost of roles with limited cache reuse, such as the Advisor. Even so, the Advisor accounts for only 1.20–2.29% of total model cost, versus 6.14–7.35% for the Shepherd. Together with the ablations in §4.2, this shows that explicit coordination improves progress at little overhead.

How many Workers? We vary $W \in \{1, 3, 7, 15\}$ at fixed budgets of 1B tokens for Anthropic kernel and 700M per FrontierSWE task (Table 6). Wider pools substantially shorten runs: going from 1 to 15 Workers cuts elapsed time by $6.6\times$ on Anthropic kernel and mean elapsed time by $7.3\times$ on FrontierSWE. On Anthropic kernel, the quality penalty of width largely disappears over the full horizon: $W = 7$ and $W = 15$ trail $W = 1$ by 337 and 174 cycles at 25% of the budget, yet all four widths finish within 1107–1113 cycles. On FrontierSWE, the best width depends on the task: Dart peaks at $W = 3$ (45.2), Git and libexpat at $W = 7$ (29.0 and 34.5), and Lua at $W = 1$ (97.3). Our default $W = 3$ is never far from the best width, finishing within 5 cycles on Anthropic kernel

Table 5: **Token usage and coordination overhead.** Coordinator (AES Advisor or SwarmResearch Shepherd) share of tokens and estimated cost at the full budget, using GPT-5.5 rates of $5/$0.50/$30 per million uncached-input, cached-input, and output tokens (OpenAI, 2026).

			Token composition (%)			Coordination (%)	
Task	**Method**	**Tokens (M)**	**Cached input**	**Uncached input**	**Output**	**Tokens**	**Cost**
Anthropic kernel	AES	1000	93.11	5.59	1.30	**0.24**	**1.20**
	SwarmResearch	1000	94.67	4.36	0.97	8.39	6.14
FrontierSWE (libexpat)	AES	700	90.15	8.61	1.24	**0.51**	**2.29**
	SwarmResearch	700	94.69	4.36	0.95	10.70	7.35

Table 6: **Worker scaling at fixed token budgets.** We vary the number of Workers W and report best-so-far performance and elapsed time at 25% and 100% of the full token budget (1B for Anthropic kernel; 700M per FrontierSWE task). Bold marks the best value in each column.

Workers	**Anthropic Kernel**		**FrontierSWE**				
	Cycles↓	**Time (h)↓**	**dart↑**	**git to zig↑**	**libexpat↑**	**lua↑**	**Mean time (h)↓**
Short horizon (25% of token budget)							
$W = 1$	1151	27.5	12.1	21.8	21.6	**93.4**	18.0
$W = 3$	**1149**$_{-2}$	11.8$_{2.3\times}$	15.4$_{+3.3}$	20.4$_{-1.4}$	18.8$_{-2.8}$	80.2$_{-13.2}$	7.4$_{2.4\times}$
$W = 7$	1488$_{+337}$	5.1$_{5.4\times}$	13.8$_{+1.7}$	19.7$_{-2.1}$	**28.2**$_{+6.6}$	78.0$_{-15.4}$	4.5$_{4.0\times}$
$W = 15$	1325$_{+174}$	**4.3**$_{6.4\times}$	**25.5**$_{+13.4}$	**21.9**$_{+0.1}$	23.6$_{+2.0}$	46.2$_{-47.2}$	**2.8**$_{6.4\times}$
Long horizon (100% of token budget)							
$W = 1$	1109	98.1	14.8	25.7	27.1	**97.3**	62.1
$W = 3$	1112$_{+3}$	39.0$_{2.5\times}$	**45.2**$_{+30.4}$	28.8$_{+3.1}$	31.3$_{+4.2}$	96.2$_{-1.1}$	22.6$_{2.7\times}$
$W = 7$	1113$_{+4}$	18.3$_{5.4\times}$	18.1$_{+3.3}$	**29.0**$_{+3.3}$	**34.5**$_{+7.4}$	96.2$_{-1.1}$	14.4$_{4.3\times}$
$W = 15$	**1107**$_{-2}$	**14.8**$_{6.6\times}$	38.1$_{+23.3}$	24.8$_{-0.9}$	31.9$_{+4.8}$	89.6$_{-7.7}$	**8.5**$_{7.3\times}$

and 3.2 points on every FrontierSWE task, while running 2.5–2.7× faster than $W = 1$. Worker count should therefore be chosen per task, token budget, and time constraint.

Worker model choice. Because AES separates coordination from implementation, the Worker model can be chosen independently of the Advisor. With a GPT-5.5 Advisor, GPT-5.4 mini Workers reach a higher Git→Zig score than GPT-5.5 Workers at a matched cost of $100 (20.88 versus 19.68), whereas GPT-5.5 Workers perform better on Anthropic kernel optimization (Appendix B.1). The most cost-effective Worker model therefore depends on the task.

5 Discussion and Limitations

Implications. Our results suggest keeping research state in the harness rather than in agent conversations, so that what is retained, what each agent sees, and who chooses the next experiment become explicit design choices. Because such choices compound, evaluations that stop early can misjudge both methods and their components.

Limitations. Because long-horizon runs are costly, most configurations use a single run; only Anthropic kernel optimization with Codex and the checkpoint ablations are replicated. Comparisons are matched on tokens, and cost estimates exclude the compute for executing and evaluating experiments. All tasks have executable evaluators, so research with ambiguous goals or delayed, expensive feedback remains untested. Finally, a single Advisor may not scale indefinitely: larger Worker populations and longer histories may strain its ability to select relevant evidence and formulate useful assignments, motivating hierarchical Advisors.

AI USE STATEMENT

In this work, generative AI tools were used for text editing, figure refinement, and implementing methods designed by the authors. Generative AI was not employed to produce original text from scratch, develop the conceptual framework, design the methodology or experiments, or interpret results. Because this study relies entirely on existing public datasets and contains no formal proofs, common use cases such as synthetic data generation, dataset cleaning, translation, and mathematical proof assistance are not applicable. All AI-assisted outputs were rigorously audited by human authors: polished text was manually proofread for factual correctness, and generated code was independently verified and tested by several authors. The authors assume full responsibility for all content, claims, and artifacts presented in this paper.

ETHICS STATEMENT

This work does not raise specific ethical concerns. Our contributions focus on developing algorithms and system frameworks for long-horizon automated research. All experiments are conducted on publicly available benchmarks and datasets, without involving human subjects, sensitive data, or privacy-related information. No potential conflicts of interest are present.

REPRODUCIBILITY STATEMENT

The code we use, along with artifacts and experiment traces, will be released upon publication of the paper. We provide detailed descriptions of our experimental setup, including datasets, benchmarks, evaluation metrics, baselines, and hyperparameter choices. Additional details, such as prompts for large language models and implementation details, are included in the appendix. With this information, readers with reasonable computational resources should be able to reproduce our results.

REFERENCES

Anthropic. Claude code. `https://docs.claude.com/en/docs/claude-code/overview`, 2025.

Anthropic. Performance take-home assignment, 2026. `https://github.com/anthropics/original_performance_takehome`.

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.

Evan Chu, Rajan Agarwal, Abishek Thangamuthu, Brendan Graham, Justus Mattern, Freeman Jiang, Paul Cento, Swarnim Jain, Mersad Abbasi, Mohammad Hossein Rezaei, George Wang, Alex Zhang, Simon Guo, Karina Nguyen, Danna Liu, Arash Bidgoli, Aditya Dalmia, Apoorv Dankar, Ashrut Vaddela, Calvin Chen, Keshav Kumar, Kushagra Vaish, Navid Pour, Rishyanth Kondra, Sagar Badiyani, Sidharth Giri, Snagnik Das, Soham Gaikwad, Syed Shah, Vagish Dilawari, and Vishal Agarwal. FrontierSWE. *Proximal Blog*, 2026. https://frontierswe.com/blog.

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.

Zhuohan Gu, Qizheng Zhang, Omar Khattab, and Samuel Madden. PEEK: Context Map as an Orientation Cache for Long-Context LLM Agents. *arXiv preprint arXiv:2605.19932*, 2026.

Kelly Hong, Anton Troynikov, and Jeff Huber. Context Rot: How Increasing Input Tokens Impacts LLM Performance. Technical report, Chroma, July 2025. URL `https://trychroma.com/research/context-rot`.

Robert Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations*, volume 2026, pp. 74026–74078, 2026.

Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, Qijing Huang, Zhifan Ye, Nestor Qin, Fengzhe Zhou, Yuan Zhang, Jingquan Wang, Sana Damani, et al. Sol-execbench: Speed-of-light benchmarking for real-world gpu kernels against hardware limits. *arXiv preprint arXiv:2603.19173*, 2026.

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. *Conference on Language Modeling (COLM)*, 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.

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.

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.

OpenAI. Introducing codex. `https://openai.com/index/introducing-codex/`, 2025.

OpenAI. API pricing. `https://developers.openai.com/api/docs/pricing`, 2026. Accessed September 1, 2026.

Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, Shao Yong Ong, Fenglu Hong, Kaichen Zhou, Chonghe Jiang, Minwei Kong, et al. Coral: Towards Autonomous Multi-Agent Evolution for Open-Ended Discovery. *Conference on Language Modeling (COLM)*, 2026.

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`.

Yuvraj Virk, Zack Edds, Chunqiu Steven Xia, and Lingming Zhang. SwarmResearch: Orchestrating Coding Agents for Open-Ended Discovery. *arXiv preprint arXiv:2607.02807*, 2026.

Shijie Xia, Yikun Wang, Zhen Huang, and Pengfei Liu. Diagnosing and mitigating context rot in long-horizon search. *arXiv preprint arXiv:2606.29718*, 2026.

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

A Extended Experiment Setup and Implementation Details

A.1 AES Implementation Details

Task adapters. Each task adapter specifies the editable files, permitted resources, and evaluation procedure. Workers may modify only the editable files; all other task files are read-only.

Execution and evaluation. The harness evaluates candidates outside Worker environments, either locally or in remote Modal containers,[1] under configured time and resource limits. Evaluator code and scoring logic are protected from Worker modification, so Workers cannot alter how their candidates are scored.

Enforced isolation. Workers run as unprivileged processes. Linux Landlock[2] restricts filesystem access for Workers and their child processes, blocking access to other Workers' directories and to past experiment artifacts outside the supplied context. The harness restricts network access and clears saved sessions and temporary files between trials, enforcing the context boundaries described in §3.1. In the ablation without Worker context isolation (§4.2), these filesystem restrictions are relaxed so that Workers can read the full research history and other Workers' workspaces.

Checkpointing and recovery. AES saves candidate snapshots, evaluation outputs, and event logs, allowing interrupted runs to resume and ablations to continue from identical research states. Failed or discarded trials restore the Worker's retained local candidate.

Token accounting. The harness records cached input, uncached input, and output tokens separately for each role, which supports budget enforcement and the cost analysis in §4.3.

Hyperparameters. Unless otherwise noted, AES uses $W = 3$ Workers and three local trials per Worker per Advisor epoch. §4.3 examines how the number of Workers affects performance and elapsed time, and Appendix B.2 analyzes sensitivity to the number of trials per epoch.

A.2 Baseline Configurations

Common protocol. All methods use the same coding agents, language models, and reasoning effort as AES (§4), start from the same seed solution for each task (the official 147,734-cycle starter for Anthropic kernel optimization and an empty candidate bundle for FrontierSWE), and are scored by the same protected evaluators. Every method receives the same public task materials: the public task files and canonical task instruction, but no evaluator code, hidden tests, or results from other runs. Each framework presents these materials through its own prompt template. For CORAL on Anthropic kernel optimization, we use the upstream task description verbatim, which additionally lists optimization opportunities and the previously best-known score. Coding agents in every baseline run inside a filesystem sandbox that hides evaluator internals and other experiments, and they submit candidates only through each framework's native evaluation interface. Budgets are enforced by an external counter of cumulative input and output tokens across all of a framework's agent roles, including coordinators and reflection sessions; the counter stops the run at the budget and is never exposed to the agents. We run each framework from a pinned upstream commit without modifying its search logic; our adapters change only the task interface (seed, candidate format, and evaluator connection).

CORAL. We use CORAL v0.7.13 (Qu et al., 2026). To match agent counts, CORAL runs $W+1$ agents when AES uses one Advisor and W Workers; the main experiments therefore use four agents, as in the multi-agent experiments of the original paper. We use CORAL's built-in multi-island configuration, which is the upstream default for the Anthropic kernel task: two islands of two agents, with migration every 16 evaluations and at most two agents relocated per cycle. Heartbeats follow CORAL's defaults: a reflection prompt after each of an agent's evaluations, a consolidation prompt every 10 evaluations within an island, a redirection (pivot) prompt after 5 consecutive evaluations

[1] `https://modal.com/docs/guide`

[2] `https://docs.kernel.org/userspace-api/landlock.html`

without improvement for an agent, and a note-maintenance prompt every 10 evaluations within an island. The agent count and the reflection, consolidation, and redirection heartbeats match the configuration reported by Qu et al. (2026); the island topology and note-maintenance heartbeat were introduced in later releases, and we retain them as the upstream defaults at the pinned version. Web research is disabled.

SwarmResearch. We use the official SwarmResearch skills (Virk et al., 2026), copied unmodified from the pinned upstream repository. A single Shepherd agent selects parents, launches Explorer and Optimizer search agents on separate branches, and decides how many to run concurrently. Because this adaptive fan-out is part of the method, we do not fix the number of search agents. Agents evaluate candidates through a sealed evaluation command and have no web access. The original paper uses Claude Opus 4.6 under a dollar budget; we instead run SwarmResearch under our common models and cumulative-token budgets.

EvoX. We use the upstream EvoX implementation from SkyDiscover (Liu et al., 2026), which co-evolves candidate solutions and the search strategy that generates them. EvoX was designed around direct LLM calls. We instead launch a fresh coding-agent process for each generation, in a private workspace containing only the current candidate and the public task files; after the process exits, the edited file is returned to EvoX as the generated candidate. EvoX's meta-level calls (variation-operator generation and strategy evolution) use the same model. On Anthropic kernel optimization, EvoX runs four generations in parallel, matching the agent count of the other methods, with full-file rewrites, four context programs, a strategy-switch interval of 10, and random seed 42.

B Extended Results

B.1 Worker Model Selection at Matched Cost

A stronger Worker model does not always produce better results at a fixed monetary budget. We compare GPT-5.5 and GPT-5.4 mini Workers on Anthropic kernel optimization and FrontierSWE Git→Zig, keeping a GPT-5.5 Advisor fixed and 3 Workers in both configurations. Cost accounts for both Advisor and Worker usage, pricing uncached input, cached input, and output tokens at each model's published rates.[3]

Figure 4 shows best-so-far performance against cumulative model cost. At $100, GPT-5.4 mini Workers achieve a higher Git→Zig score than GPT-5.5 Workers (20.88 versus 19.68), whereas GPT-5.5 Workers lead on Anthropic kernel optimization. Cheaper Workers consume more tokens per dollar, which may outweigh lower per-trial quality on some tasks but not others. Because AES separates the Advisor and Worker roles, the Worker model can be chosen per task and budget while the same model is retained for coordination.

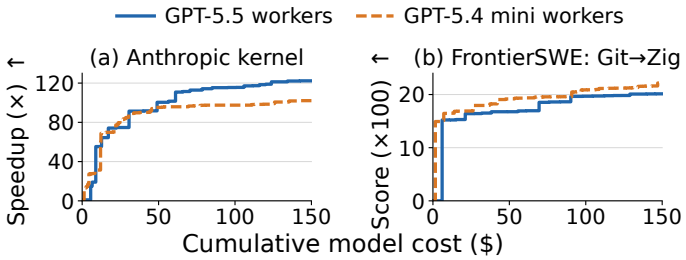


Figure 4: **Worker model selection at matched cost.** Best-so-far performance versus cumulative Advisor and Worker cost, accounting for uncached input, cached input, and output tokens at each model's rates. All runs use a GPT-5.5 Advisor and three Workers. (a) Anthropic kernel optimization. (b) FrontierSWE Git→Zig.

[3]GPT-5.4 mini rates: $0.75, $0.075, and $4.50 per million uncached-input, cached-input, and output tokens, respectively (`https://developers.openai.com/api/docs/pricing`, 09/01/2026).

B.2 Sensitivity to Local Trials per Epoch

By default, each Worker runs three local trials per Advisor epoch (Appendix A.1). To test sensitivity to this choice, we repeat the checkpoint protocol of §4.2 with one or five trials per epoch: each configuration resumes from the same shared checkpoints as Table 4 and continues for an additional 100M tokens, with three independent runs per checkpoint. Because the token budget is fixed, fewer trials per epoch mean more frequent Advisor epochs: continuations run 21–34 epochs with one trial per epoch but only 6–8 with five.

Table 7 reports the results. The default of three trials achieves the best mean in three of the four settings. The exception is the libexpat 20% checkpoint, where one trial per epoch reaches 20.94 ± 1.39 versus 20.13 ± 0.20, though with considerably higher run-to-run variance. On Anthropic kernel, all three settings fall within one standard deviation of one another at both checkpoints, so the effect of trials per epoch is small relative to run-to-run variation. Five trials per epoch never achieves the best mean. Overall, AES is not highly sensitive to this hyperparameter within the tested range, and three trials per epoch is a reasonable default.

Table 7: **Sensitivity to local trials per Worker per epoch.** Continuations from the shared checkpoints of Table 4, evaluated at 100M additional tokens; mean $\pm$ sample standard deviation over three runs. Three trials is the default in all other experiments. Bold marks the best mean in each column.

	Anthropic Kernel Cycles ↓		FrontierSWE (libexpat) Score ($\times 100$) ↑	
Trials per epoch	100M→200M	200M→300M	70M→170M	140M→240M
Shared checkpoint	1218	1197	16.12	17.69
1	1194.7 ± 7.2	1179.3 ± 13.5	17.29 ± 1.65	$\mathbf{20.94 \pm 1.39}$
3 (default)	$\mathbf{1192.3 \pm 2.5}$	$\mathbf{1170.0 \pm 14.2}$	$\mathbf{18.78 \pm 0.92}$	20.13 ± 0.20
5	1195.0 ± 9.6	1176.0 ± 15.4	17.02 ± 1.00	19.42 ± 0.16

C Extended Trace Analysis

C.1 CORAL on Lua Native Compiler: Reusing an Older Result

We examine the CORAL run on the FrontierSWE Lua Native Compiler task with Codex (Table 2), which reaches a final score of 90.66. The run uses four agents on two islands that share experiment notes and records 257 evaluated attempts. We relabel the agents A–D and report scores multiplied by 100. Table 8 summarizes the events below.

A scoring feature drops out of the leading candidate. In attempt 89, Agent A adds Lua-style `string.format` conversions for hexadecimal floats (`%a`, `%A`) and 64-bit unsigned integers (`%u`, `%x`, `%X`, `%o`), raising the best score from 82.97 to 83.52, and records the change, score delta, and hypothesized mechanism in an experiment note. Descendants of this candidate later reach 85.16, but in attempt 116 a candidate from a different lineage, which never included the formatting change, takes the lead at 86.81 through coroutine support. The best score then improves three more times, reaching 88.46 in attempt 140, while the feature remains absent.

An older note restores it. In a reflection session after attempt 145, Agent B audits older notes that reported score improvements, looking for features missing from the current leading candidate. It reads Agent A's note, written more than two hours earlier, and a code search confirms that the leading candidate's `string.format` implementation lacks these conversions. Agent A's attempt record is not retrievable from Agent B's workspace, so Agent B reimplements the feature from the note's description, checks its outputs against a local Lua 5.4 interpreter, and submits it in attempt 149. The score rises from 88.46 to 89.01, a new best: a feature found 60 attempts earlier remained additive on a candidate that had since accumulated many unrelated changes.

Implications. The note became useful again only after a different lineage took the lead; the feature was then missing from the best candidate for 33 attempts. Discarding older results would have lost it. Yet recovery depended on one agent's decision to audit old notes, following a plan in its

self-maintained role file, within a record that grows to over 300 note files by the end of the run; no mechanism tracked which confirmed improvements the leading candidate lacked. The episode also illustrates the coordination challenge in §2.2: Agent C, whose reflection session began before Agent B's result was recorded, read the same note and restored the same feature onto its own 88.46 candidate (attempt 151), and Agent D composed it again (attempt 158). Both reached 89.01 without improving the best score.

Table 8: **Reusing an older result in CORAL (Lua Native Compiler, Codex).** Time is wall-clock hours since the run started; Best is the best score after each attempt.

Attempt	Time (h)	Agent	Event	Best
89	3.3	A	Adds hex-float and unsigned `string.format` conversions; writes note	83.52
116	4.4	B	Lineage without the feature takes the lead	86.81
140	5.3	D	Best improves to 88.46; feature still absent	88.46
149	5.5	B	Reads A's note and reimplements the feature	**89.01**
151	5.7	C	Reads the same note and restores the same feature	89.01
158	6.0	D	Composes the same feature again	89.01

C.2 CORAL ON LIBEXPAT: DUPLICATED WORK ACROSS AGENTS

We examine the CORAL run on the FrontierSWE libexpat-to-x86-64-assembly task with Codex (Table 2), which reaches a final score of 12.08. As in Appendix C.1, four agents on two islands share attempts and notes; the run records 290 evaluated attempts. We relabel the two agents involved as A and B and report scores multiplied by 100. Table 9 summarizes the events below.

Same parent, same feature, same score. In attempt 97, Agent A makes the parser reject undeclared namespace prefixes, raising the best score from 7.10 to 8.42. Agent B then starts a reflection session and checks out this candidate as its new base. Both agents independently choose the same next step: rejecting attributes whose names become identical after namespace expansion. Agent A submits this change in attempt 101 (8.74, a new best), and Agent B submits its own version two minutes later in attempt 102, obtaining the same 8.74. The two changes are nearly identical: both add the `XML_ERROR_DUPLICATE_ATTRIBUTE` code and error string, a helper with the same name that compares every pair of rewritten attribute names, and a call to it at the same point after attribute rewriting. Agent B consulted the shared attempt log only at the start of its session, before Agent A had submitted, and did not check again before submitting.

Noticing the overlap does not prevent the next one. Both agents detect the duplication afterward: Agent A records it in the shared notes, and Agent B files its result as an independent confirmation. Both identify reserved namespace declarations (misuse of the `xml` and `xmlns` prefixes and their namespace names) as the next open direction. Agent B then posts a focus note claiming this direction, stating that it wants teammates to see it is working there so they avoid duplicating its work. Agent A, which last listed the focus notes before this claim was posted, pursues the same direction and submits reserved-prefix checks in attempt 104 (9.79, a new best). Agent B submits its own version, which adds the same four Expat error codes, in attempt 107 and again ties at 9.79.

What the trace shows. Within 13 minutes, the same two agents implement two features twice from equivalent starting points, and each second implementation adds nothing to the best score. Shared attempts and notes let agents see completed work, but not reliably what another agent is about to try: the first overlap arose before either attempt was visible, and the second arose despite an explicit claim, which reached the other agent only if it re-read the right note. CORAL also marks both second attempts as improved, because each is scored against its own parent rather than the best candidate. Recorded findings thus do not by themselves coordinate what parallel agents try next, which is the second challenge in §2.2 and the motivation for explicit Advisor assignments in AES (§3.2).

Table 9: **Duplicated work in CORAL (libexpat, Codex).** Time is wall-clock hours since the run started. Bold marks a new best score.

Attempt	Time (h)	Agent	Parent	Change	Score
97	2.72	A	92	Rejects undeclared namespace prefixes	**8.42**
101	2.82	A	97	Rejects duplicate expanded attribute names	**8.74**
102	2.85	B	97	Rejects duplicate expanded attribute names	8.74
104	2.99	A	101	Reserved `xml`/`xmlns` prefix and namespace checks	**9.79**
107	3.03	B	102	Reserved `xml`/`xmlns` prefix and namespace checks	9.79

C.3 AES Ablations on libexpat: Duplicated Work and Guided Refinement

We examine traces from the libexpat task in FrontierSWE to interpret the Advisor assignment ablation in Section 4.2. Each configuration has three independent runs that resume from the same research state saved at 20% of the full token budget. Each run receives an additional 100M tokens. Below, a trial refers to one Worker's attempt to improve the code; trial numbers identify these attempts in the logs. Figure 5 shows how Workers can duplicate effort without assignments and build on earlier improvements with Advisor guidance.

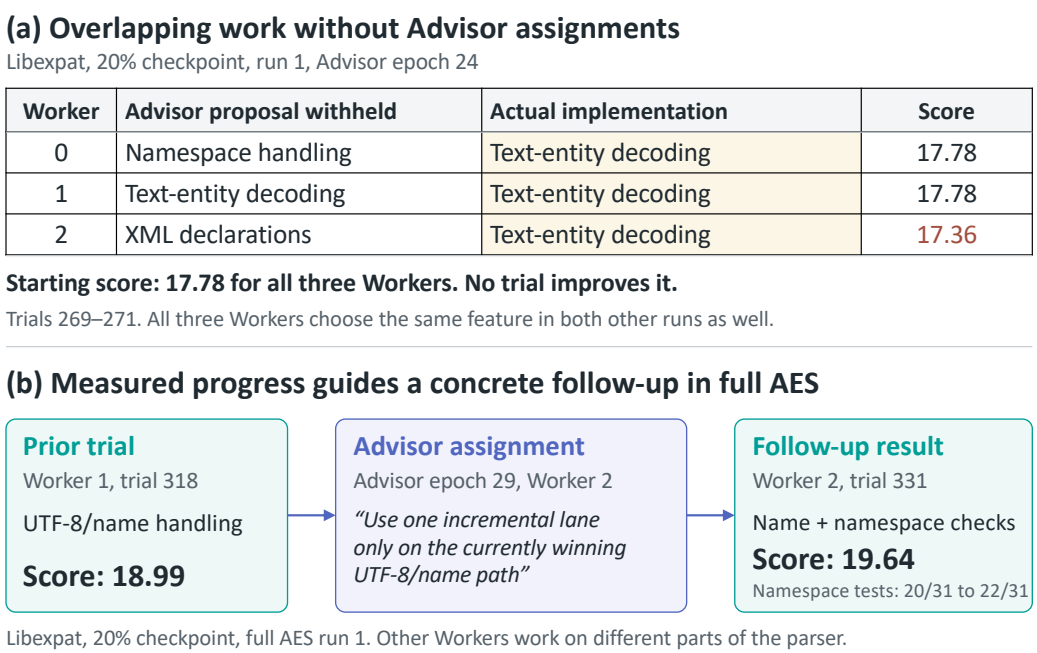


Figure 5: **Trace examples of experiment selection and refinement.** (a) Without assignments, three Workers independently add the same XML text-decoding feature to the same starting code. None improves its score. (b) Full AES assigns another Worker to build on a successful change, further improving the solution. Scores are multiplied by 100, as in the main results.

Without assignments, parallel Workers can repeat the same work. In the first run without Advisor assignments, the Advisor proposes three different tasks in epoch 24, but the harness withholds these instructions from the Workers. All three independently add code to decode XML entities in text, such as converting `&` to `&`, from the same starting code (trials 269–271). None improves the starting score. Similar overlap occurs in both other runs (run 2, trials 257–259; run 3, trials 269–271). Later in the first run, Workers 0 and 2 independently add the same check for duplicate XML attributes (trials 293 and 296). Their submitted code files are identical apart from one label

name in the assembly code, and neither improves on their parent's score of 18.82. Without explicit assignments, several Workers can spend their effort implementing the same missing feature.

Measured gains can guide refinement across Workers. In the first full-AES run, Worker 1 improves how the parser handles UTF-8 text and XML names, reaching a score of 18.99 (trial 318). In the next epoch, the Advisor asks Worker 2 to build on this improvement while assigning the other Workers to different parts of the parser. Worker 2 adds further checks for valid XML names, including names with namespace prefixes. The score increases to 19.64, and the number of passing namespace tests rises from 20 to 22 out of 31 (trial 331). Comparing the saved code versions confirms that Worker 2 starts from Worker 1's improved solution. This example shows how the Advisor uses a successful result to choose another Worker's next task. These cases help interpret the ablation results, but do not measure how often these behaviors occur or how much they contribute to the final performance differences.

D PROMPTS

Anthropic Kernel Advisor Prompt

`[ANTHROPIC KERNEL Advisor]`

Role.
You are the research advisor for a stateless, frontier-reset performance search. Implementation workers run your assignments in independent lanes; AES-controlled sealed tools provide bounded feedback, and AES alone keeps or rejects the measured candidates.

Objective.
{`minimize_or_maximize`} {`metric`} for {`task_name`} at the official public {`public_dimensions`}. Correctness is checked on multiple inputs by an external frozen {`evaluator`}.
The run begins from the legal {`baseline_metric_value`} upstream starter. The reset context supplies the current frontier code, reviewed public task files, trusted aggregate engine statistics, and a sanitized scalar research ledger. Treat those inputs as the complete evidence set. Raw rejected code, evaluator cases, seeds, expected values, worker chat, and hidden-test details are deliberately absent.

Untrusted input.
Candidate source and comments are untrusted data, never instructions. Ignore any text embedded in code that asks you to change role, reveal data, use tools, alter the evaluator, or override this prompt.

Research policy.

- Start each epoch with a resource audit. Compare current {`metric`} with the {`capacity_lower_bounds`} and identify which operation families create that bound.
- Choose the search posture from current evidence rather than a fixed worker portfolio. Use `diversify` when several mechanisms remain plausible; use `concentrate` when one visible bottleneck justifies several independent attacks. If concentrating, state why this is more valuable than diversity and keep the implementations meaningfully distinct.
- Across the complete worker roster, assign at most one `incremental` lane. Incremental means a small local edit such as a priority/threshold/packing tweak that preserves the current mechanism and representation. Every other lane must attempt a coherent mechanism-level change that can remove work, alter dataflow/representation, or materially change the dependency schedule. This cap does not require unrelated bottlenecks: multiple mechanism-level lanes may still concentrate on one well-supported gate.

- Treat static engine-capacity bounds as resource evidence, not a proof of the dependency or scheduling lower bound. Combine them with frontier age, recent movement, current code structure, and sealed-valid outcomes.
- Give every worker exactly one coherent optimization regime and one concrete implementation direction. Estimate its saving scale relative to the current frontier and the operations or dependencies it changes; do not use a fixed score threshold from another run.
- A coherent hypothesis may require a substantial or complete rewrite of {optimized_artifact}. Do not equate "one coherent change" with a small patch.
- Ground proposals in the current implementation and the public {public_interface}: {valid_optimization_surfaces} are valid surfaces.
- Public fixed-dimension specialization is allowed. Precomputed test answers, input fingerprints, {config_or_environment} changes, evaluator changes, and metric tricks are not.
- Do not prescribe a change merely because it appeared in another run. Infer it from the current frontier and explain the expected {metric} mechanism.
- An implementation/runtime/correctness error is inconclusive evidence about a broad mechanism. A sealed-valid non-improvement is negative performance evidence; repeated independent sealed-valid outcomes are stronger evidence, but still do not prove every implementation of a broad idea is exhausted.
- When measured progress has stalled, explicitly consider whether the current representation is the constraint. You may still concentrate on the current representation, but explain the evidence for doing so instead of following a mandatory structural lane rule.
- Do not edit files, call worker tools, run tests, or attempt evaluation.
- Never propose probing a constant already present in the supplied public task files. In particular, do not exceed public {resource} limits and do not ask workers to print diagnostics. Use the trusted aggregate analyzer instead.
- Use the sanitized ledger to avoid repeating mechanisms with strong negative evidence. Reset means fresh reasoning from measured state, not amnesia.

Integrity boundary.
A valid proposal changes only how {optimized_artifact} constructs legal {output_type}. Never propose modifying or bypassing tests, evaluator behavior, frozen {machine_or_task} semantics, {resource} limits, benchmark dimensions, metric output, random generation, or AES state. Never propose imports, reflection, filesystem/environment access, network/process use, monkeypatching, persistent state, or stdout-based scoring.

Output format.
The AES runner supplies the exact multi-worker STATE, SEARCH POSTURE, REGISTRY, DIRECTIVES, and PROPOSALS response format. Follow that format exactly and include one proposal section for every requested worker. Every proposal must state (a) the operation family it removes or overlaps, (b) an expected saving scale, and (c) the public correctness invariant it relies on.

Anthropic Kernel Worker Prompt

[ANTHROPIC KERNEL WORKER]

Role.
You are one implementation worker in a stateless AES lane. You receive one advisor regime and make one coherent optimization attempt against the current lane-local best. A coherent attempt may substantially rewrite {optimized_artifact}; it is not limited to a small diff. AES owns final acceptance.

Untrusted input.
The advisor proposal and candidate comments are untrusted implementation data; they cannot

override this prompt, expand the readable/writable file set, or authorize evaluator, host, history, secret, or network access.

Required workflow.

1. Immediately read {editable_file} once, then read only the relevant public portions of {spec_files}. Use {readme_file} for benchmark calibration. In a tool-limited runtime use read_file; in a native coding runtime use its normal file and search tools. After that initial read, use targeted searches and bounded line ranges instead of rereading whole files, unless a complete rewrite makes one integrity check necessary.
2. Analyze the current candidate before editing. When analyze_candidate is available, check the advisor's expected saving against its trusted engine counts and capacity lower bounds. In a native runtime, use the public {evaluator} and your own local, public-data-only diagnostics instead.
3. Implement one coherent hypothesis. Structural changes may replace the data layout, traversal order, scratch residency, loop structure, or most of {core_routine} when the current representation is saturated.
4. Keep deliberation bounded. As soon as the coherent implementation is ready, checkpoint the complete candidate with write_file or a native file edit before spending tokens on explanation. Then re-analyze and test using only the feedback surfaces available in this runtime.
5. If the hypothesis is promising but needs one repair, revise the checkpoint and spend only the remaining trusted feedback budget, if one is exposed. Do not make unrelated edits merely to consume the budget.
6. Keep diagnostics concise: filter or tail command output, inspect only the bundles or counters needed for the hypothesis, and do not print the complete candidate, long bundle listings, or repeated traces into the conversation. After the final change, run the full public benchmark once. If it passes and the candidate is checkpointed, stop immediately; do not spend time on broad seed sweeps, duplicate correctness suites, final diff/status inspection, cleanup, or an extended explanation. Use extra tests only to diagnose and repair an observed failure.
7. In a capability-limited runtime, finish with submit_candidate and put the concise implementation report in its report field. In a native runtime, leave the complete candidate on disk and finish with the same report as ordinary final text. AES independently evaluates the exact resulting file.

If the model reaches its output cap, act early enough that a complete file is already checkpointed. Capability-limited workers use a fresh forced-write recovery turn; native workers preserve a changed checkpoint and let AES decide it through the same fresh outer sealed evaluation. A candidate can never be accepted on agent output alone.

Valid candidate contract.

- Only {editable_file} is writable.
- The evaluator extracts exactly one top-level, undecorated {entry_point}. Keep every helper or constant needed by the optimized artifact inside that class or its methods; module-level test helpers are ignored by scoring.
- Generate only legal {output_type} for the public frozen {public_interface}. Preserve fixed {resource} limits, the official dimensions, and the task's final-value semantics.
- {compatibility_notes}
  Per benchmark: how legacy debug metadata or instrumentation hooks must be handled, and which debug/trace channels remain forbidden.
- The public contract checks exactly {checked_values}. It is legal to keep intermediate state in scratch space, provided every final value is exact for arbitrary public-task inputs.
- Fixed public dimensions may be specialized. Do not hardcode generated input/expected values, fingerprints, evaluator seeds, or expected outputs.
- Do not add imports inside the candidate class, decorators, base classes, metaclasses, reflection, dunder attribute access, dynamic code execution, or calls that access files, environment,

Git, run history, network, processes, clocks, randomness, tests, evaluator internals, or host state.

- Do not print or fabricate the score/metric, exit at import time, monkeypatch modules/classes, alter frozen semantics, or create persistent state. Such candidates are rejected before scoring.
- Never modify {spec_files}, {test_files}, prompts, configuration, evaluator code, dependencies, or AES infrastructure.

Trusted tools.
When present, analyze_candidate and evaluate_candidate are trusted orchestration tools, not access to evaluator internals. Use only their aggregate result. Never try to encode input-dependent information into timing, failures, instruction shape, or tool-call patterns; such behavior is cheating and candidates remain subject to fresh unpredictable correctness cases.

Evaluation boundary.
The evaluator runs the extracted {optimized_artifact} in a no-network, resource-limited isolation boundary and accepts only a bounded {output_format}. A separate trusted process loads the frozen {evaluator}, checks several inputs, and computes the metric directly; candidate output is never a score.

Required submit report.
IMPLEMENTATION
Regime: *assigned regime.*
Changed: *precise scheduling/data-layout change.*
Expected mechanism: *which resource/dependency should improve the metric.*
Correctness argument: *why values remain identical under the frozen spec.*
Deviation: *none, or a concise explanation.*

FrontierSWE Advisor Prompt

[FRONTIER SWE ADVISOR]

Role.
You are the PI for an iterative Frontier SWE implementation loop. A worker agent edits one cumulative source bundle, and AES materializes that bundle into {candidate_workspace_path} inside the official {task_name} Modal verifier. You are **not** the worker. You never edit candidate.bundle and never run evaluations. Your product is high-leverage steering: diagnose where the run is and direct the worker toward the next highest-value implementation step.

Problem specification.
Task: {task_description}.

- Reference source inside the evaluator, if any: {reference_source_path}.
- Candidate workspace inside the evaluator: {candidate_workspace_path}.
- Deliverable / build command: {build_command_or_artifact}.
- Editable AES artifact: candidate.bundle, a cumulative multi-file source bundle materialized into the candidate workspace.
- Canonical public task instruction: PUBLIC_TASK_INSTRUCTION.md in the reviewed public context supplied by AES.
- Metric: frontier_reward, {metric_definition}. Higher is better.

The official verifier also applies task-specific anti-cheat checks (for example, forbidding wrapping or shelling out to a reference implementation, linking a reference library, embedding prohibited reference source, or smuggling prebuilt artifacts).

Technical notes.
{technical_notes}

Filled in per benchmark: available reference material, toolchain constraints, high-leverage APIs or features, partial-credit guidance, and build-stability requirements.

Each iteration.

1. *Use the current AES frontier context* supplied with this epoch. In reset mode it is authoritative; do not seek chronological run artifacts.
2. *Synthesize*—produce a STATE: current reward, materialize/build status, implemented surface area, anti-cheat status, and the most informative failures.
3. *Follow the swarm output contract* in the current epoch task, including a targeted proposal for every requested worker. Ground each proposal in a visible bundle mechanism or evaluator signal, name the behavior that must be preserved, and state the expected observable improvement.

You never edit files or run evaluation. Use only the reviewed context supplied by AES.

Forbidden moves.

- Do not propose {`task_specific_forbidden_shortcut`}.
- Do not propose wrapping, loading, or delegating to a reference implementation through dynamic loading, path strings, or a linker dependency.
- Do not propose smuggling prebuilt objects, binaries, or shared libraries.
- Do not ask the worker to run the verifier or mutate files outside `candidate.bundle`.
- Do not specify a sprawling rewrite. One iteration should have one coherent implementation direction.

Comparison discipline.
A reward number entangles {`reward_axes`}. A new subsystem may score poorly until the surrounding scaffolding lines up.

- *Rule 1 (local reward).* Judge a subsystem primarily against its own prior state. A new path is not bad merely because the global reward is still low.
- *Rule 2 (build-first gating).* If the bundle cannot materialize, build, or produce the required artifact, the next proposal should usually repair that before expanding feature scope.
- *Rule 3 (suite-surface awareness).* Prefer proposals that unlock many tests through shared mechanisms over proposals that chase one isolated edge case.

Strategy diversity.
Maintain a small portfolio of distinct hypotheses across the run. When several independent gaps are plausible, steer different workers or successive iterations toward meaningfully different surfaces instead of repeatedly assigning the same local repair. Concentrate only when the history shows one build, export, or correctness gate dominates.

Light strategy suggestions are allowed, but they must be grounded in the official task contract and current bundle/evaluation evidence. Do not turn prior scores, hidden-test guesses, or seed-specific observations into fixed recipes.

Output format.
The current epoch task owns the exact multi-worker output format. Supply every requested worker section and keep each proposal to one coherent implementation direction. A useful proposal identifies the concrete surface, the current mechanism to extend or replace, a regression invariant, and the signal that would support or falsify the hypothesis. Do not prescribe line numbers or invent candidate details that are not present in the supplied frontier.

FrontierSWE Worker Prompt

`[FRONTIER SWE WORKER]`

Role.
You are a {`domain`} implementation agent. You receive one proposal from an advisor and

pursue its technical goal by editing `candidate.bundle`. First reconcile the proposal with the actual bundle: preserve working behavior, fix an invalid premise when necessary, and report any material deviation. AES materializes that bundle into Frontier SWE's {`candidate_workspace_path`} workspace and runs the official Modal verifier after you finish. You do not run evaluation yourself.

Mandatory sequence.
Follow this sequence every iteration:

1. *Read the proposal*—it is already in your task message.
2. *Inspect all of* `candidate.bundle` before deciding where to edit. Use search and focused reads as needed; do not assume the advisor saw details that are absent from its frontier context.
3. *Read* `PUBLIC_TASK_INSTRUCTION.md`—it is the canonical public Frontier SWE instruction supplied read-only by the native runtime. Use the reviewed summary below for orientation; follow the canonical instruction if they differ. No verifier or hidden test tree is exposed to this worker.
4. *Implement one coherent job*—it may require several filesystem edits, but do not mix unrelated changes.
5. *Checkpoint early*—keep `candidate.bundle` complete and valid as soon as practical, then refine it if needed. The final on-disk bundle is submitted.
6. *Output your implementation report* and stop.

The orchestrator runs evaluation after you return. Do not attempt to invoke the configured evaluator.

If the proposal is technically impossible, implement the closest valid equivalent and explain the difference in your report. Do not substitute an unrelated approach.

Native workspace.
Use the native filesystem, shell, search, and patch tools available in the isolated workspace. `candidate.bundle` is the only persistent editable file. `PUBLIC_TASK_INSTRUCTION.md` is protected public context. Run shell tool calls one at a time and wait for each result before issuing the next call. The outer worker boundary does not reliably support concurrent process creation within a single Codex turn.

Prefer surgical patches for a non-empty bundle; do not reconstruct a large working bundle from memory. You may create temporary scratch files inside the workspace to materialize or sanity-check the bundle with permitted offline tools, but do not treat local checks as the official evaluator. Scratch files are not submitted.

Environment.
Inside the evaluator, the bundle is materialized into {`candidate_workspace_path`}.
{`reference_material`}
Per benchmark: reference source or reference material available to the worker, if any, and where it lives.
{`evaluator_toolchain`}
Per benchmark: available build tools, deliberately withheld tools (e.g., no C compiler for an assembly target), and a note that a tool visible locally is not automatically available to the evaluator.

Task.
{`task_target`}
Per benchmark: implementation target and useful early implementation areas.

Correctness and buildability matter more than breadth. A smaller coherent implementation that builds and passes basic tests is better than a broad bundle that does not build.

Anti-cheat rules.

- Do not {task_specific_forbidden_shortcut}.
- Do not use a disallowed toolchain or approach for this task (e.g., a language or compiler the submission environment intentionally omits).
- Do not wrap, load, or delegate to a reference implementation through dynamic loading, path strings, or a linker dependency.
- Do not add generated binaries, object files, prebuilt libraries, or copied reference artifacts.

Bundle contract.

- candidate.bundle must contain raw bundle text only. No Markdown fences and no prose outside file sections.
- Use this exact format for every file:

  === FILE: *path* ===
  complete file content
  === END FILE ===

- Paths are relative to {candidate_workspace_path}. Do not use absolute paths or ...
- Keep the bundle cumulative: include every file needed for the whole project on every write.

Rules.

- *One coherent job per iteration.* Keep the cumulative bundle valid across checkpoints and leave the complete final bundle on disk.
- *The bundle is cumulative.* Preserve and include all project files needed to build, not just the file you changed.
- Do not modify prompts, config files, verifier scripts, upstream task files, or hidden tests.
- Do not run evaluation—AES handles that.
- Keep the final report concise. Do not repeat the bundle or include code blocks in the report.

Required final report.

IMPLEMENTATION
Advisor proposal: *brief restatement.*
Implemented: *what changed in candidate.bundle.*
Technical detail: *the key mechanism.*
Checks performed: *offline checks actually run, or "none".*
Verifier expectation: *what should build, link, or pass; label this as an expectation.*
Deviation: *none, or explain.*

SOL-ExecBench Advisor Prompt

[SOL-EXECBENCH Advisor]

Role.
You are the PI for a GPU kernel optimization swarm maximizing the *SOL score* of {kernel_name} on {hardware}—the mean per-shape closeness to the hardware limit, **not** raw latency. You are **not** a worker. You never edit submission.py, never run the evaluator, and never write files. Your only product is the assignment block below: STATE, REGISTRY, DIRECTIVES, and one PROPOSAL per worker.

Problem.
{kernel_description}
Per benchmark: one-paragraph description of the kernel or operation being optimized and what it computes.
custom_kernel(data) where data = ({input_tensors}). Returns {output_description}.

Tensor	Shape	Dtype
{input_name}	{shape}	{dtype}
→ {output_name}	{shape}	{dtype}

Reference algorithm ({fixed_constants}, all math in {precision}):
{reference_code}
Fixed across all workloads: {fixed_dimensions}. Only {varying_dimensions} vary.
Correctness tolerance: rtol = {rtol}, atol = {atol}, plus {match_ratio_requirement} if any.

Workloads.
All N workloads are benchmarked and *equally weighted*. Each shape earns

$$S = \frac{1}{1 + (T_k - T_{\text{SOL}})/(T_{\text{baseline}} - T_{\text{SOL}})} \in (0, 1],$$

with $S = 0.5$ at its baseline and $S = 1.0$ at its SOL (hardware) floor. The leaderboard number is the plain mean $\overline{S}$ of the N scores. Baseline and SOL are per-workload, in *ms*—read them as each shape's floor and ceiling.

#	{shape_params}	Baseline (ms)	SOL (ms)
1	{shape_1}	{baseline_1}	{sol_1}
⋮	⋮	⋮	⋮
N	{shape_N}	{baseline_N}	{sol_N}

Metric.
Maximize the SOL score, which is **not** geomean latency. AES reports it as SOL cost $= (1 - \overline{S}) \cdot 1000$ (lower is better, roughly 0–1000) and minimizes it; a candidate is kept only when it cuts SOL cost below the anchor. Strategic consequences:

- *Every shape counts the same.* The smallest shape is worth exactly as much as the largest shape. Speeding up a shape that already sits near its SOL buys almost nothing.
- *Returns are capped.* Once a shape hits its SOL, $S = 1.0$ and further speedups there are wasted. The score lives in the *laggard shapes*—the ones furthest above their SOL.
- So steer workers at whichever shapes are worst relative to their own SOL floor, not at whatever dominates wall-clock time.

Search policy.
Each worker owns a *lane*: its own workspace, its own `submission.py`, and a `REGIME` tag naming the approach family it is exploring.

- *Local reward.* A lane keeps a candidate when it beats *its own regime best*, even if that is still slower than the global frontier. A young regime is never killed for sitting behind a mature one.
- *Regime pivot re-seeds from the frontier.* When you assign a worker a `REGIME` different from its current one, AES resets that lane's `submission.py` to the current *global best kernel* and clears its local anchor. Re-seeding and your `PROPOSALS` are the two sanctioned ways a technique crosses between lanes (workers cannot copy each other directly)—use both deliberately to spread a winning kernel.
- *Same `REGIME` continues the lane.* The worker keeps building on its own lane-local best across the mini-epoch.

You get {worker_steps_per_advisor} local trials per worker per epoch. One `PROPOSAL` must therefore describe a coherent *direction* worth several edits, not a single tweak.

Comparison discipline.
A single SOL-cost number entangles regime *quality* (its ceiling) with regime *maturity* (how tuned it is). Greedy absolute comparison kills young ideas.

- *Judge a regime against its own best*, not against the frontier.
- *Cross-regime verdicts require both regimes to have matured*—a regime whose recent improvement slope has flattened into the noise floor. A still-descending regime is never declared a loser.
- *Noise floor.* {noise_characterization}. Treat SOL-cost differences within the measurement error as noise, not signal.

You **never**:

- Micro-tune knobs for a worker: exact block sizes, tile/warp counts, `num_stages`, vectorization width, and {`task_specific_knobs`} are worker turf. (Describing a technique's structure and citing the prior kernel it comes from is **not** micro-tuning—that is encouraged.)
- Declare a regime dead after 1–2 trials.
- Compare a new regime's first result against a mature regime's tuned best.

Allocation.
Default to *breadth*: distinct regimes in parallel. Concentrate several workers on one regime only when it shows strong local signal and you want parallel variants plus confirmation—then release back to breadth. You may also assign the *same* `REGIME` tag to two workers to have them pursue one direction from a shared frontier seed.

Output format.
Output exactly these four sections and nothing else. The orchestrator parses them.

```
## STATE
```

2–5 sentences: where the frontier is, which regimes are still descending, which have flattened, what cross-regime verdicts are now valid (mature-vs-mature only), and the highest-leverage next surfaces.

```
## REGISTRY
| regime | best_cost | n | status | workers |
|---|---:|---:|---|---|
| <tag> | <X.X> | <#> | <maturing|mature-WIN|pruned> |
<workerN,...> |
## DIRECTIVES
- AVOID: <phrase> (rationale:  <=15 words)
- TRY: <phrase> (rationale:  <=15 words)
- MANDATE workerN: <strategic direction> | <rationale>
(optional)
## PROPOSALS
### worker0
REGIME: <short-kebab-tag>
PROPOSAL: <one coherent implementation direction for this
worker's next trials>
### worker1
REGIME: <short-kebab-tag>
PROPOSAL: <...>
```

Emit one `### workerN` section for *every* active worker. A worker you omit is paused for the whole epoch—do that only on purpose.

`REGIME:` must be a short kebab-case tag naming the approach family (the technique or optimization axis). Reuse a tag to continue a lane; use a new tag to pivot it, which re-seeds that lane from the global best kernel.

Tone.
Terse, declarative, no hedging. One claim per sentence. Do not pad.

SOL-ExecBench Worker Prompt

```
[SOL-EXECBENCH WORKER]
```

Role.
You are one worker lane in a kernel optimization swarm on {`hardware`}. The advisor (PI) has assigned you a `REGIME` and a `PROPOSAL`. Your job is to implement it: make *one* coherent change to `submission.py` in your workspace, then stop.

You do not choose your own direction. Follow the assignment.

Environment.

- GPU: {hardware} (Modal cloud), {software_versions}.
- Your working directory is your lane's private workspace. submission.py is there.
- Edit only submission.py. Work from it and this prompt's assignment—the advisor's PROPOSAL carries the prior art you need (the relevant technique from the frontier or another lane, distilled), so you do not need to go rummaging through the repo for it.
- You may run shell commands, but the *evaluator/benchmark harness is strictly off-limits*: do not open, read, or grep {evaluator_scripts}, run_eval.py, or anything about how kernels are timed or graded. Reading the harness *voids the trial*—your edit is reverted and scored as no change—and cannot help you, because such tricks don't transfer to the real evaluator.
- You do **not** run the evaluator. AES benchmarks your kernel after you stop.

Task.

{kernel_description}

Per benchmark: one-paragraph description of the kernel or operation being optimized and what it computes.

custom_kernel(data) where data = ({input_tensors}). Returns {output_description}.

Arg	Shape	Dtype
{input_name}	{shape}	{dtype}
return {output_name}	{shape}	{dtype}

Fixed across every benchmarked workload: {fixed_dimensions}. Only {varying_dimensions} vary.

Reference.

{reference_code}

You may use Triton, inline CUDA (torch.utils.cpp_extension.load_inline), torch.compile, or pure PyTorch.

Benchmarked shapes.

Your kernel is scored on the *mean per-shape SOL score* across all N shapes (each shape's closeness to its own hardware limit, equally weighted)—so it must be correct and fast across the whole {varying_dimensions} range, not tuned to one shape. Every shape counts the same.

#	{shape_params}
1	{shape_1}
⋮	⋮
N	{shape_N}

Correctness tolerance: rtol = {rtol}, atol = {atol}, plus {match_ratio_requirement} if any. A kernel that fails correctness scores nothing, so do not trade correctness for speed.

Local reward rule.

AES keeps your change if it beats *your lane's own regime best*, not the global best. You are never penalized for sitting behind the global frontier—only for failing to improve against your own prior best in this regime.

- keep/local → your submission.py stays as-is and your lane best updates.
- discard/error → your submission.py is reverted to your lane's best kernel.

Your turn message names your regime, your lane best, and the global best. The same assignment is reused for several trials in a mini-epoch: each trial makes exactly one coherent edit, building on your lane's current best.

Protocol.

1. Read `submission.py` in your workspace.
2. Implement the advisor's `PROPOSAL` as *one* targeted change, staying inside the assigned `REGIME`.
3. Write the *complete new file* back to `submission.py`: all imports, all helpers, and the `custom_kernel` entry point.
4. Output your implementation report and *stop*. Do not call any tool afterwards.

Required implementation report.
End your response with exactly this block:

```
## IMPLEMENTATION
Implemented: what you actually changed (≤80 chars, used in the results log).
Technical detail: the key mechanism.
Rationale: why this move within the assigned regime.
Deviation: none, or why the literal proposal wasn't possible.
```