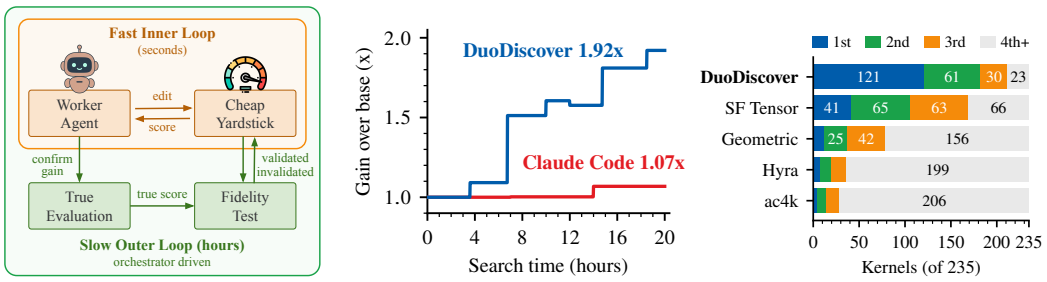
DUODISCOVER: CODE OPTIMIZATION BY DISCOVERING WHAT TO OPTIMIZE AND HOW TO EVALUATE

Anonymous authors
Paper under double-blind review

ABSTRACT

Most discovery and optimization frameworks assume that candidate solutions can be evaluated cheaply over many iterations. In many real settings, however, a single true evaluation, such as a full benchmark or serving trace, can take hours, so a search can afford only a handful of true evaluations. When evaluation is the bottleneck, search must discover both promising targets to optimize and cheap ways to measure progress on them to find a solution. We present DuoDiscover, a search system that builds on this idea in three ways. First, each worker agent in a fleet of coding agents co-discovers a target and builds a cheap proxy evaluator for it, which we call a yardstick. The agent then hill-climbs the yardstick rather than the true evaluation. Second, because search can exploit any gap between a yardstick and the true objective, fidelity tests are introduced to continually check that each yardstick still ranks edits like the true objective and invalidate any that drift. Every gain is confirmed by the true evaluation before it is accepted. Third, the worker fleet is bottom-up, with no central decomposition of the problem. A lightweight orchestrator spawns and terminates agents, briefs running ones, and seeds new agents with others' confirmed work. We evaluate DuoDiscover on 2 repository-level benchmarks requiring multi-file edits and on 4 GPU and TPU kernel suites covering hundreds of kernels. On SWE-fficiency, it doubles the average speedup of human experts. On SOL-Exec, it ranks first on 121 of 235 kernels and in the top three on 212. In addition to those benchmarks, on the heavily optimized vLLM and SQLite engines, DuoDiscover achieves $1.92\times$ and $1.38\times$ gains over the base code, respectively. On a CPU microarchitecture design task with evaluations of up to 28 hours, DuoDiscover surpasses both AlphaEvolve and the best human design in 9 days, $4\times$ shorter search time, a big step toward autonomous chip design.



(a) The DuoDiscover loop (b) vLLM serving engine (c) SOL-Exec leaderboard

Figure 1: DuoDiscover at a glance. (a) A worker agent hill-climbs a cheap *yardstick* in an inner loop (seconds per edit), while an outer loop of true evaluations (hours each) confirms gains and runs the fidelity test. (b) On vLLM, DuoDiscover reaches $1.92\times$ on a four-objective geometric mean, against Claude Code's $1.07\times$. (c) On NVIDIA's SOL-Exec leaderboard, DuoDiscover takes first place.

1 INTRODUCTION

Search-based discovery systems that couple large language models (LLMs) with iterative search and evaluator feedback have achieved strong results across mathematics (Romera-Paredes et al., 2024), algorithms (Novikov et al., 2025), and systems optimization (Cheng et al., 2025). However, most

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.

prior work focuses on refining the search harness itself (Lange et al., 2026; Cemri et al., 2026), treating the evaluator as a given that can be called cheaply and repeatedly. This assumption breaks down when evaluation requires running a full benchmark or serving trace (Gupta et al., 2026). For instance, when optimizing vLLM for multi-objective serving under service-level objectives (SLOs, Section 3.2), 16.8 hours (84%) of a 20-hour budget went to true evaluation, comprising correctness checks and workload execution. In such scenarios, the central question becomes how to search effectively when true evaluation is the bottleneck, which is the focus of our work.

A natural solution is to guide search with cheap proxy evaluators instead of expensive true evaluations. This works because large speedups often come from focused edits to a few small regions of code, which we call *targets*, whose progress can be measured much more cheaply than the entire true evaluation. However, such proxies are target-specific, making it impractical to design one by hand for every target. They are also vulnerable to reward hacking, since search may exploit discrepancies between the proxy and the true objective, improving the proxy score without improving the true objective. Targets and proxies therefore cannot be chosen independently. A target is useful only if it admits a cheap proxy that tracks the true objective, while a proxy is useful only if its target can meaningfully improve that objective. This motivates co-discovering targets and their cheap proxy evaluators, which we call *yardsticks*, and hill-climbing the yardsticks rather than the true evaluation to find solutions.

We propose **DuoDiscover**, a search system for settings where evaluation is the bottleneck. Beyond co-discovering targets and yardsticks, DuoDiscover addresses two further challenges. The first is *drift*, a form of reward hacking. A yardstick that initially ranks edits like the true objective degrades as the agent adapts to it, because hill-climbing seeks out the edits where the yardstick and the true objective diverge. DuoDiscover continually runs fidelity tests to invalidate drifting yardsticks and requires a true evaluation to confirm every gain before it is accepted. The second is *target selection*, choosing which regions to optimize among many candidate targets with unpredictable payoffs. Rather than decomposing the problem centrally, DuoDiscover deploys a bottom-up fleet of worker agents that independently co-discover targets and yardsticks. Because worker agents are cheap but true evaluations are expensive, a lightweight orchestrator manages only the true evaluations, steering them toward agents making progress, terminating stalled ones, and combining compatible gains.

We evaluate DuoDiscover on two repository optimization benchmarks that require codebase editing, and 4 kernel optimization benchmarks (GPUs and TPUs), achieving consistent gains (Table 1). On repositories, DuoDiscover doubles the human-expert speedup on SWE-fficiency and beats all baselines by 21% for FormulaCode. For GPU kernels, DuoDiscover achieves a 14% higher speed-of-light (SOL) score than the best baseline, ranking 1st on SOL-Exec for 121 of 235 kernels and in the top-3 for 212 at the time of submission. On TPU-Inference kernels, DuoDiscover attains a $2.84\times$ speedup over optimized Pallas kernels, ahead of the best baseline at $2.51\times$. On JAXBench, DuoDiscover achieves a $4.19\times$ speedup over XLA. On highly optimized systems (vLLM and SQLite), DuoDiscover finds up to a $1.92\times$ speedup. Finally, on an expensive CPU microarchitecture design task (data prefetcher), whose evaluation takes up to 28 hours, it surpasses both ArchAgent v2, an AlphaEvolve-based system, and the best human design in 9 days, against ArchAgent v2's 40 days.

2 DuoDiscover: Co-discovering targets and yardsticks

DuoDiscover treats code optimization under an expensive true objective as a search that must also decide *what to optimize* and *how to measure progress cheaply*. It runs a fleet of parallel coding agents, each constructing and hill-climbing its own (target, yardstick) pair, coordinated by a lightweight *orchestrator* that also runs a *fidelity test* against the true evaluation (Figure 2).

2.1 Problem: Optimization under expensive true evaluation

Let $\mathcal{X}$ be the space of program states, from a single file (e.g., kernel code) to an entire codebase, and let $x_0 \in \mathcal{X}$ be the initial program state. An *edit* by an agent turns one state into another (e.g., updating files in a codebase). The expensive *true objective* $f : \mathcal{X} \to \mathbb{R} \cup \{\perp\}$ maps a state to a scalar score, where higher is better, or to $\perp$ (a failure value) if any correctness check fails. We denote the cost of one call to f by c_f (e.g., the time it takes vLLM to serve a benchmark workload end to end). Within a run, N agents work under one shared wall-clock budget and evaluation resources.

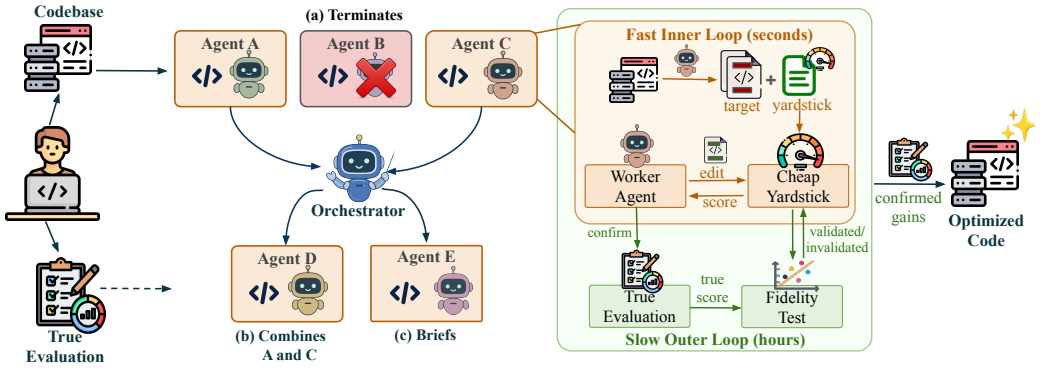


Figure 2: Overview of DuoDiscover. Left: a fleet of worker agents, each on its own copy of the codebase, coordinated by an orchestrator that terminates agents (a), spawns new ones, here one combining A and C (b), and briefs running ones (c). Right: inside one agent, a fast loop co-discovers a target and yardstick and climbs it on cheap scores, while a slow loop confirms gains with true evaluations and runs the fidelity test that validates or invalidates the yardstick.

For our experiments, c_f ranges from minutes to hours (Table 7), so each agent gets only tens of true evaluations in total. The goal of a run is to find a state that scores as high as possible under f.

2.2 Co-discovery: Targets and their yardsticks

The building block of DuoDiscover is the worker agent, an off-the-shelf LLM coding agent, following recent agentic optimizers (Luo et al., 2026; Liao et al., 2026; Deng et al., 2026; Agarwal et al., 2026). Each agent co-discovers a target and a yardstick for it and then climbs the yardstick to find solutions. The optimization steps (profiling, choosing a target, building a yardstick, and making edits) are all made by the agent, while the orchestrator adds measurement and coordination.

We denote a *target* by T, which is a region of code (e.g., a single function in a kernel, or a module in a large codebase). The set $\mathcal{X}_T \subseteq \mathcal{X}$ denotes the program states that can be reached by changing code only inside T. Each agent comes up with its own target given the task and the initial program state x_0. This is in contrast to AlphaEvolve (Novikov et al., 2025), where the user marks the code to be evolved for the system with explicit begin and end markers in the source. In DuoDiscover, the agent also writes down a rough guess, $w_T \in [0, 1]$, of what share of the true score the target region might be responsible for. This is useful to know because, by Amdahl's law (Amdahl, 1967), even if the target code region became infinitely fast, the overall score could improve by at most a factor of $1/(1 - w_T)$ (i.e., a region that is responsible for 20% of the score cannot buy more than $1.25\times$).

We denote a *yardstick* for T by $g_T : \mathcal{X}_T \to \mathbb{R} \cup \{\perp\}$, a scorer that is much cheaper to run than the true evaluation, $c_g \ll c_f$. The agent writes this as a script that returns a score and a correctness flag. Each agent builds its own yardstick for its chosen target, and what it computes is up to the agent (e.g., a representative subset of cases, or a cost model). The orchestrator measures how cheap a yardstick is as the ratio $r = c_f/c_g$ of the two costs, and passes it back to the agent, which decides whether to keep it. The absolute value of a yardstick does not need to track f, only the order it puts edits in, since the yardstick is used only to decide which edits to pursue. For two edits $x, x' \in \mathcal{X}_T$, we want $\mathrm{sign}(g_T(x) - g_T(x')) = \mathrm{sign}(f(x) - f(x'))$.

By co-discovery we mean that the target and the yardstick are chosen together. A concrete example of this co-discovery process is shared in Appendix D.1. When optimizing vLLM for performance, one agent targeted the scheduler's prefill batching and wrote a yardstick that was a simple model of decode-step time, which runs in seconds while the true evaluation takes 85 minutes. Co-discovery is needed since a target is only valuable if a cheap yardstick that correlates with f can be built for it, while a yardstick is only valuable for a target that can actually change f.

2.3 Bottom-up search and coordination

Within a target, an agent runs two loops (Figure 2, right): a fast one, in which it edits and scores with its yardstick, and a slow one, in which a true evaluation confirms a gain in score that the yardstick

reports. In the fast loop, where it spends most of its time, it edits its target region, scores each edit with its yardstick g_T, keeps the edit if the score improves, and reverts it otherwise.

The search is bottom-up, in that each agent picks its own target without an a priori decomposition of the problem, so the orchestrator's role is coordination rather than planning. Periodically, the orchestrator reads a short summary of every agent's progress and coordinates them through three actions. It *terminates* agents that are not making progress, *spawns* new ones on promising or uncovered targets, and *briefs* running ones with a short message about what the rest of the fleet has found. A spawned agent (i) starts fresh, (ii) is seeded with the confirmed state of the best agent (a *transfer*), or (iii) re-implements the confirmed changes of several agents in one place, keeping each only if a true evaluation confirms it (a *combine*). For example, over the course of the vLLM run, the orchestrator terminated one agent, briefed the other three to leave throughput and explore, and later spawned a fourth on a new target with a transferred solution. A run ends when the budget is exhausted, a target score is reached, or the fleet empties.

2.4 Fidelity testing: Validating yardsticks

The fidelity test measures whether a yardstick ranks states in the same order as f, using Kendall's τ rank correlation (Kendall, 1938), a standard tool in nonparametric statistics (Hollander et al., 2013). We compute the correlation only on pairs of states whose true scores differ by more than the evaluation noise, which avoids giving credit for distinctions that the true evaluation itself cannot resolve. A yardstick counts as correlated with f when $\tau \geq 0.5$ over these pairs.

Since the search optimizes against its yardstick, it can be rewarded for edits that exploit the gap between the yardstick and f, a form of reward hacking (Gao et al., 2023; Skalse et al., 2022; Karwowski et al., 2024). Consequently, a yardstick that initially tracks f may fail to do so for the specific edits targeted during search. We refer to this degradation as *drift*. For this reason, the orchestrator re-tests every yardstick after every true evaluation. A new yardstick has not been tested yet, which does not stop the agent from climbing it. Once enough states have been scored by both g_T and f and the correlation clears the bar, the yardstick is *validated*. If the correlation later drops, it is *invalidated*, and the agent either rebuilds it or moves on to a new target. In the vLLM case, a yardstick built from a short load sweep was validated, later invalidated when higher load reversed its ranking, and validated again once rebuilt. Appendix A.1 gives the noise estimation and thresholds.

3 Results

We detail the setup, then report benchmark results (§3.1) and five real-world case studies (§3.2).

Benchmarks. We evaluate DuoDiscover on tasks across two repository optimization benchmarks (FormulaCode and SWE-fficiency; Sehgal et al., 2026; Ma et al., 2025) and four GPU and TPU kernel suites (SOL-Exec, JAXBench, TPU-Inference, and FA4; Lin et al., 2026a; Tschand et al., 2026; vLLM Project, 2026; Chen et al., 2026a). Each task requires speeding up the source code while maintaining functional correctness. Unlike algorithmic benchmarks (Novikov et al., 2025), where evaluation finishes in seconds, a single true evaluation here can take minutes to hours (Table 7).

Scoring. All methods are evaluated under the same wall-clock time and number of agents. Repositories are scored by end-to-end execution speedup while checking for correctness. Kernels are scored by speedup over reference code or as a percentage of theoretical peak hardware throughput (speed-of-light or SOL score; Lin et al., 2026a). The reported score of a run is the median of several re-evaluations of its final candidate, chosen by lower confidence bound, with a Student-t interval. The best score seen during the run is never reported, and a candidate that fails the correctness check is reported as no result. Search statistics and the analyses of §4 use a stratified sample of 20 SWE-fficiency tasks and 50 SOL-Exec kernels that covers every task type and evaluation time.

Baselines. We compare DuoDiscover against agentic (Claude Code; Anthropic, 2026), evolutionary (OpenEvolve; Sharma, 2025; its agentic variant on the repository benchmarks), and domain-specific baselines, namely AVO (Chen et al., 2026a), KDA (Zou et al., 2026), and a profiler-guided approach modeled on PerfAgent (Deng et al., 2026) (Appendix B.2). Unless noted, all runs execute for 8 hours on identical hardware with $N = 4$ parallel agents driven by Claude Code v2.1 on claude-opus-5, with claude-sonnet-5 as the harness's small fast helper model, for DuoDiscover and for

Table 1: Benchmark results under an 8-hour, four-agent budget (24 hours on the long FormulaCode tasks, 10 hours on TPU-Inference), higher is better. Numbers are speedups over the base code, except FA4 (% over Flash Attention 4), JAXBench (over XLA), TPU-Inference (over tuned Pallas kernels), and SOL-Exec, the mean % of speed-of-light over 235 leaderboard kernels. Dashes mark benchmarks a method does not target.

Method	Repositories		GPU Kernels		TPU Kernels	
	FormulaCode (×)	SWE-fficiency (×)	SOL-Exec (% SOL)	FA4 (%)	JAXBench (×)	TPU-Inference (×)
Claude Code	1.778	12.81	67.31	7.05	3.82	2.51
OpenEvolve	1.161	3.76	52.36	1.48	2.66	1.13
PerfAgent	1.774	12.27	64.10	6.86	4.08	2.27
KDA	–	–	70.22	8.93	–	–
DuoDiscover (ours)	**2.156**	**15.02**	**79.74**	**11.97**	**4.19**	**2.84**

every Claude Code-based baseline. In §4, we evaluate another backbone, Antigravity with Gemini 3.8 Flash (Google, 2026). Main results are single-run, and §4 reports variance across three runs on 50 SOL-Exec kernels and 20 SWE-fficiency tasks.

3.1 Benchmark results

Under the same wall-clock budget, DuoDiscover scores above every baseline on every repository and kernel benchmark (Table 1). The rest of this section dives deeper into each of the benchmarks.

FormulaCode. FormulaCode (Sehgal et al., 2026) is a repository-level benchmark of real-world performance optimization tasks mined from merged, performance-improving pull requests across scientific Python projects. We evaluate on a 25-task subset of FormulaCode-V, the human-checked 108-task suite. Human expert PRs give a 1.048× geometric-mean speedup over the base code, and the goal is to exceed the human patch while maintaining functional correctness. Selected tasks have evaluation times from 7 minutes to 6.8 hours, and all baselines share the same budget, 8 hours for tasks under 2 hours and 24 hours for tasks over 2 hours.

On these tasks (Table 1), OpenEvolve barely clears the human bar at 1.161×, while the agentic baselines average 1.78×. DuoDiscover achieves 2.16×, ahead of all baselines and the human PRs, a 21% gain over the strongest agentic baseline. It also makes broader repository-level changes, modifying on average 22 files, 51 functions, and 1,800 lines per task, against more localized edits from PerfAgent (14 files, 800 lines) and OpenEvolve (2 files, 250 lines).

SWE-fficiency. We evaluate DuoDiscover on the 100 SWE-fficiency-Lite (Ma et al., 2025) tasks, which require agents to speed up pull-request workloads in major Python libraries like NumPy and pandas without breaking tests. Unlike FormulaCode, SWE-fficiency targets a single workload per task. DuoDiscover gives a 15.02× geometric-mean speedup, above the expert patches (6.90×) and all baselines: Claude Code (12.81×), PerfAgent (12.27×), and OpenEvolve (3.76×). The tasks are difficult because locating bottlenecks in large codebases is hard, and slow true evaluations (15 minutes on median, up to 93 minutes) strictly cap candidate testing within an 8-hour budget. DuoDiscover succeeds through search efficiency (Table 16). On the 20-task sample, Claude Code crowds onto 2 to 4 targets per task and runs true evaluations on roughly 72 candidates to land 6 confirmed gains, while DuoDiscover isolates about 9 targets per task and builds a fast yardstick for each (5 of which pass the fidelity test). Because a yardstick runs in just 40 seconds on median, 22× faster than a true evaluation, agents test 358 candidates on it and send only 59 to true evaluation. This doubles confirmed gains to roughly 12 per task, for a 16% higher geometric mean (14.93× against 12.85×), and passes Claude Code's 8-hour score after 2.5 hours (median over 20 tasks).

GPU and TPU kernels. Finally, we extend our evaluation to kernel optimization tasks across GPUs and TPUs. Unlike multi-file repository tasks, a kernel is a single file, but it still has several stages, such as expert routing, gating, projections, and memory dispatch in a mixture-of-experts model, and a true evaluation runs the whole kernel on every run (14 minutes on median, up to an hour on SOL-Exec). DuoDiscover takes each stage as a target, builds a fast yardstick timing only that component, and hill-climbs on it before confirming gains on the complete kernel. Similar to repository opti-

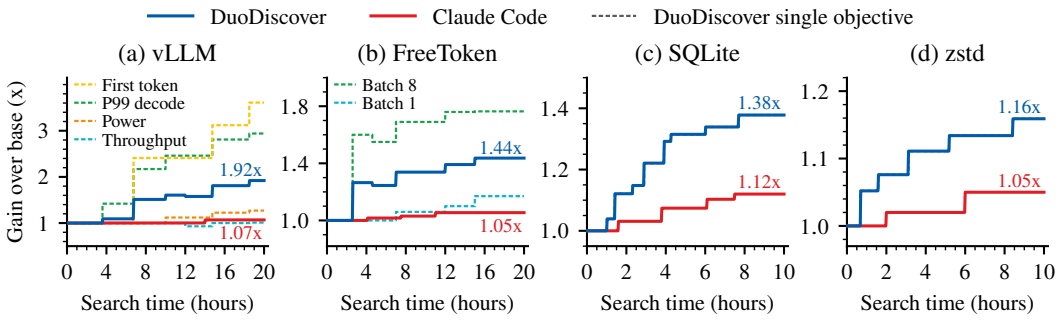


Figure 3: Gain over the base code across search time. Thick lines: geometric mean over the objectives (DuoDiscover blue, Claude Code red). Dashed lines: individual DuoDiscover objectives.

mization on SWE-fficiency, DuoDiscover searches more broadly with cheap yardsticks while using fewer true evaluations, leading to more confirmed gains and faster progress on SOL-Exec (Table 16).

For GPU kernels, we evaluate on SOL-ExecBench (SOL-Exec) (Lin et al., 2026a), a recent GPU kernel benchmark of 235 kernels spanning emerging models and suites (e.g., FlashInfer-Bench; Xing et al., 2026). It scores implementations by the fraction of hardware speed of light (SOL) attained, and its public leaderboard enables direct comparison with external and proprietary submissions. At the time of submission[1], DuoDiscover ranks first for 121 of the 235 kernels, second for 61, and third for 30 (Figure 1c, Appendix C.3), with an average SOL score of 79.74%, ahead of several proprietary entries. We also tested strong baselines, including KDA (70.22% SOL, top-ranked in MLSys 2026 Competition (2026)). We further evaluate on Flash Attention 4 (FA4), designed for Blackwell GPUs (Zadouri et al., 2026) and not yet included in SOL-Exec. Across 24 B200 configurations, DuoDiscover achieves an 11.97% geometric-mean speedup over the FA4 reference, ahead of KDA (8.93%), NVIDIA's AVO (Chen et al., 2026a) (4.11%), and cuDNN attention (1.30%).

Moving beyond GPUs, we first evaluate on JAXBench, covering 17 priority JAX/TPU kernels. DuoDiscover achieves a 4.19× geometric-mean speedup over the XLA baseline, ahead of PerfAgent (4.08×), Claude Code (3.82×), and OpenEvolve (2.66×). We further evaluate on the TPU-Inference suite (vLLM Project, 2026), which includes kernels from recent frontier models such as GLM-5.2 and Kimi K3, with reference implementations optimized by expert performance engineers. Speedups here are measured against those optimized production Pallas kernels. DuoDiscover attains a 2.84× speedup, against 2.51× for Claude Code, 2.27× for PerfAgent, and 1.13× for OpenEvolve.

3.2 Real-world case studies

In this section, we investigate whether DuoDiscover can find performance headroom in heavily optimized real systems like vLLM (Kwon et al., 2023, LLM serving), FreeToken (Yang et al., 2026, on-device inference), SQLite[2] (embedded database), zstd[3] (compression), and data-prefetcher design (Bera et al., 2026), which has its own budget and baselines. Every confirmed edit passes the system's full test suite and correctness checks. Both methods use four agents under the same budget, 20 hours for vLLM and FreeToken and 10 hours for SQLite and zstd. Core maintainers of vLLM and FreeToken reviewed our results to confirm correctness and rule out reward hacking, and upstream pull requests are in preparation. Search and solution details are in Appendix D.

vLLM: Multi-Objective Serving under SLOs. The task is to serve GLM-5.2 in NVFP4, NVIDIA's 4-bit floating-point format, on an 8×B200 node under four competing service-level objectives (SLOs), namely 99th-percentile (p99) decode latency, first-token latency, GPU power, and throughput, scored by an 85-minute true evaluation. On this heavily tuned stack, four Claude Code agents improve neither p99 decode latency nor power and end at a 1.07× geometric mean after 12 true evaluations. DuoDiscover advances all four (Figure 3(a)) and after 7 true evaluations lands a roughly 500-line, purely system-level solution at 1.92× geometric mean, with 2.94× lower p99 decode latency, 3.61× lower first-token latency, 21% lower power, and unchanged throughput.

[1] Our rankings are based on the SOL-ExecBench leaderboard state on September 10, 2026.
[2] https://www.sqlite.org/ [3] https://github.com/facebook/zstd

Table 2: Yardstick speed and fidelity across SWE-fficiency tasks and SOL-Exec kernels. Here *Validated*: targets whose yardstick passes the fidelity test. *Invalidated*: percentage invalidated later for drift. τ: median rank correlation with the true evaluation.

	True evaluation	Yardstick	Faster by	Validated	Invalidated	τ
SWE-fficiency tasks	15 min	40 s	22×	5.3 of 9.2	41%	0.66
SOL-Exec kernels	14 min	16 s	53×	11 of 15	45%	0.76

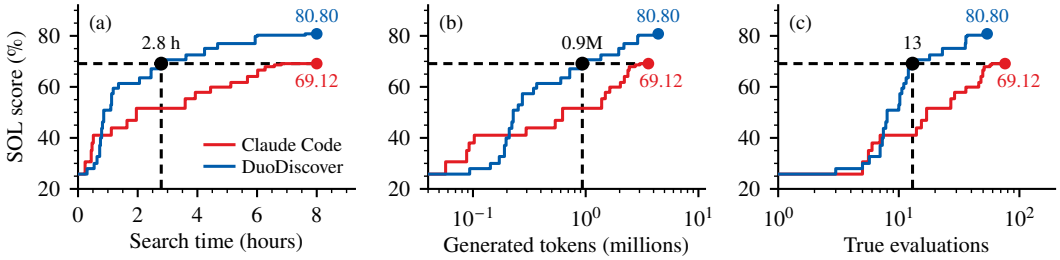


Figure 4: Best-so-far SOL (mean) over 50 kernels vs. (a) time, (b) tokens, and (c) true evaluations. DuoDiscover attains Claude Code's score (69.12%) in 2.8 hours, 0.93M tokens and 13 evaluations.

FreeToken: On-Device LLM Inference. FreeToken is an open-source engine that serves mixture-of-experts (MoE) models on a single GPU by keeping most weights in host memory (Yang et al., 2026). The task is to serve GLM-5.2-NVFP4 (a 753B-parameter MoE model with 433 GB of weights) on a single 48 GB L40S GPU, scored by a 70-minute true evaluation of decode throughput at batch sizes 1 and 8. Bound by CPU expert execution and PCIe transfers, Claude Code ends at $1.05\times$ after 14 true evaluations. DuoDiscover finds complementary optimizations across the CPU expert path, GPU kernels, expert scheduling, and caching, and after 6 true evaluations ends at $1.17\times$ at batch 1 and $1.76\times$ at batch 8, a $1.44\times$ geometric mean (Figure 3(b)).

SQLite and zstd. Both are mature, heavily tuned C codebases, and zstd is the Zstandard lossless compression library. For SQLite, the task uses its own `speedtest1` workload, and each 10-minute true evaluation rebuilds the engine, runs the 13,249-test suite twice, and checks hidden result hashes on unseen seeds and query shapes. DuoDiscover speeds up the bytecode engine, sorter, recursive-query path, and decimal conversion, ending at $1.38\times$ against $1.12\times$ for Claude Code, with 17 true evaluations against 45 (Figure 3(c)). For zstd, each 15- to 25-minute true evaluation counts instructions across 16 workload configurations and requires byte-identical compressed output. DuoDiscover improves the optimal parser, match finder, and match compare, ending at a $1.16\times$ instruction ratio against $1.05\times$ for Claude Code, with 7 true evaluations against 21 (Figure 3(d)).

Computer Architecture Design. Following recent advances in autonomous chip design[4], we evaluate DuoDiscover on data-prefetcher optimization (Bera et al., 2026), a CPU microarchitecture design problem. A single true evaluation takes up to **28 hours**. With about 1,500 candidates over 9 days, against over 12,000 candidates and more than 40 days in the first of five search stages of ArchAgent v2, an AlphaEvolve-based system (Gonzalez et al., 2026), DuoDiscover reaches a $1.0504\times$ geometric-mean speedup over the reference core across all three configurations, outperforming both ArchAgent v2 ($1.0382\times$) and the best human design ($1.0352\times$). On multi-core workloads, DuoDiscover reaches $1.0287\times$, matching the best human design ($1.0285\times$) and exceeding ArchAgent v2 ($1.0133\times$). To our knowledge, DuoDiscover is the first autonomous system to reach the human best on these multi-core cases, a step toward autonomous chip design (Appendix D.5).

4 Analysis

We evaluate DuoDiscover on 20 SWE-fficiency tasks and 50 SOL-Exec kernels, examining yardstick quality, search efficiency, variance, component ablations, and backbone generalization.

Yardstick Quality. Yardsticks score candidate edits $22\times$ faster on repositories (40 s vs. 15 min) and $53\times$ faster on kernels (16 s vs. 14 min) than true evaluations (Table 2). Each measures only what its target affects, such as a shrunken input or a profiled hot loop, rather than the full workload. Yardsticks are validated on 5.3 of 9.2 repository targets and 11 of 15 kernel targets per task, and

[4] `https://www.kimi.ai/blog/kimi-k3`

Table 3: Adding one component at a time, up to the full DuoDiscover (shaded).

	SWE-fficiency (×)	SOL-Exec (% SOL)
Claude Code ($N = 4$)	12.85	69.12
+ target	12.71	69.68
+ yardstick (no fidelity test)	10.33	66.91
+ fidelity test	13.25	72.57
+ co-discovery	13.76	74.43
+ orchestrator (DuoDiscover)	**14.93**	**80.80**

Table 4: Removing one orchestrator mechanism at a time from the full DuoDiscover (shaded).

	SWE-fficiency (×)	SOL-Exec (% SOL)
DuoDiscover (full)	**14.93**	**80.80**
− spawn and terminate	14.10	77.93
− allocate by progress	14.30	78.23
− brief	14.28	78.49
− transfer	13.92	77.46
− combine	14.03	76.38

their rankings correlate closely with the true evaluation (median Kendall τ of 0.66 and 0.76). As code evolves, rank drift invalidates 41% of repository yardsticks and 45% of kernel yardsticks. This wastes budget but never counts a false gain, since every reported gain is confirmed by a true evaluation. Appendix F reports yardstick quality by kind (Table 15) and per-candidate fidelity (Figure 12).

Search Efficiency. DuoDiscover builds yardsticks in its first hour, then pulls ahead of Claude Code (Figure 4). On kernels, DuoDiscover equals Claude Code's final 69.12% score in 2.8 hours using 74% fewer tokens (0.93M vs. 3.6M) and 83% fewer true evaluations (13 vs. 76), cutting run cost from $309 to $79.60 (Appendix F.3). While faster loops let agents generate more tokens per hour, DuoDiscover turns them into real gains, ending at 80.80% (4.4M tokens, 54 true evaluations) and leading on 41 of 50 kernels. On repositories, it passes Claude Code's score in a median of 2.5 hours.

Variance across Discovery Runs. Across three seeds per task, DuoDiscover shows low run-to-run variance with final scores staying within $\pm 1.12\%$ of its per-kernel mean and $\pm 1.8\%$ of its per-task mean, against $\pm 3.27\%$ and $\pm 5.21\%$ for Claude Code. Most gains reproduce across seeds, with 78% on kernels and 81% on repositories.

Component Ablations. We add components incrementally (Table 3) and remove orchestrator actions one by one (Table 4). Target selection alone changes little, as true evaluations still bottleneck candidate testing. Untested yardsticks hurt because rank drift misguides search. The fidelity test recovers this loss (13.25×, 72.57%), co-discovery adds more (13.76×, 74.43%), and the orchestrator adds +1.17× and +6.37 SOL points, the largest single step on kernels, to the full DuoDiscover (14.93×, 80.80%). Removing any single action costs 0.63 to 1.01× in repository speedup and 2.3 to 4.4 kernel points, hitting transfer hardest on repositories and candidate combination on kernels. The removal losses sum to more than the orchestrator's net gain, so its actions depend on one another.

Backbone Coding-Agent Generalization. We also run DuoDiscover on Antigravity/Gemini 3.8 Flash, a different coding-agent backbone, under the same 8-hour, 4-agent budget (Table 5). With Antigravity, DuoDiscover raises repository speedup from $9.72\times$ to $12.66\times$ and kernel SOL score from 58.79% to 73.26%, corresponding to $1.30\times$ and $1.25\times$ gains, larger than the $1.16\times$ and $1.17\times$ gains with Claude Code/Opus 5. DuoDiscover largely closes the gap between the two agents. With DuoDiscover, Antigravity/Gemini 3.8 Flash nearly matches Claude Code/Opus 5 on repositories ($12.66\times$ vs. $12.85\times$) and surpasses it on kernels (73.26% vs. 69.12%). As expected for the faster model, Gemini 3.8 Flash generates 32% more tokens, but its $6.7\times$ lower per-token API cost makes the overall run $5.1\times$ cheaper than Claude Code/Opus 5 (Appendix F.3).

Table 5: Generalization across coding-agent backbones under the 8-hour, 4-agent budget.

	SWE-fficiency (×)	SOL-Exec (% SOL)
Claude Code/Opus 5	12.85	69.12
+DuoDiscover	**14.93**	**80.80**
Antigravity/Gemini 3.8 Flash	9.72	58.79
+DuoDiscover	**12.66**	**73.26**

The appendix extends this analysis. Appendix E scales the fleet from 1 to 16 agents and compares per-target yardsticks with a single task-wide evaluator, and Appendix F reports yardstick quality by kind, search behavior, the cost of a run, examples of discovered targets and yardsticks, and robustness to missing, unreliable, or noisy yardsticks.

5 Related work

In this section, we describe three lines of related research: LLM-driven coding and discovery agents, LLM-generated tests and evaluators, and multi-fidelity optimization methods.

Coding and discovery agents. LLMs are increasingly used as operators for search and optimization, transforming automated code generation into an engine for scientific and algorithmic discovery. FunSearch (Romera-Paredes et al., 2024) was among the first methods of this kind and found new best-known constructions for combinatorics problems. AlphaEvolve (Novikov et al., 2025) scaled this into an autonomous pipeline that evolves a population of programs from programmatic feedback. Subsequent work has extended this evolutionary approach along several axes: improving efficiency (Lange et al., 2026; Jiang et al., 2026), dynamically adapting mutation strategies (Cemri et al., 2026; Liu et al., 2026), optimizing the model weights with test-time reinforcement learning (Wang et al., 2026b; Yuksekgonul et al., 2026), and overcoming local minima (Yan et al., 2026). Discovery agents have been proposed for a variety of tasks, including repository-level runtime optimization (Deng et al., 2026; Lin et al., 2026b), low-level kernel acceleration (Zou et al., 2026; Chen et al., 2026a; Wang et al., 2026a; Luo et al., 2026; Dai et al., 2026; Liao et al., 2026; Wei et al., 2025), and computer architecture discovery (Gupta et al., 2026; Gonzalez et al., 2026; Blasberg et al., 2026). For some tasks, the cost of evaluations can be a major bottleneck for optimization. To mitigate this bottleneck, SIFT (Fu et al., 2026) uses the expensive evaluations only for the most promising solutions. While prior methods like AlphaEvolve allow users to specify cheaper checks ahead of the full evaluation, DuoDiscover is the first discovery agent that finds both the solution and cheap evaluators which are validated or invalidated during the discovery process.

LLM-generated tests and evaluators. Prior work has also used LLMs to synthesize test cases and evaluators to guide code search. Generated unit tests are used to rank candidate implementations (Chen et al., 2023) and to drive iterative refinement (Shinn et al., 2023; Chen et al., 2024; Huang et al., 2023; Ruan et al., 2025; Ehrlich et al., 2025). Alternatively, ExecCritic (Tao et al., 2026) freezes generated tests before changing the implementation so unverified checks cannot misguide revisions (Chen et al., 2026b). While these works use LLMs and coding agents to construct software correctness tests, DuoDiscover extends this idea beyond correctness testing to discover cheap, target-specific yardsticks for fast optimization.

Multi-fidelity optimization (MFO). MFO methods balance cheap, approximate evaluations and expensive, accurate ones by spending most of the budget on a low-cost proxy and querying the expensive objective sparingly (Li & Li, 2026). There are two common approaches. The first is to fit a cheap *surrogate model* that predicts the objective and is corrected as real measurements arrive (Kennedy & O'Hagan, 2000; Forrester et al., 2007; Kandasamy et al., 2016; Chen et al., 2018; Fernández-Godino et al., 2019). The second is *adaptive resource allocation*, which evaluates many candidates cheaply, then progressively concentrates budget on the promising ones and discards the rest (Li et al., 2018; Falkner et al., 2018). Uniquely, our method uses agents to synthesize cheap proxy evaluators and validate or invalidate them during search.

6 Conclusion

Code optimization is often limited by the cost of evaluating candidate solutions. In this work, we have argued that search in this regime must discover both *what to optimize (target)* and *a cheaper way to measure progress on it (yardstick)*. We present DuoDiscover, a bottom-up search system that builds on this idea for evaluation-bottlenecked settings. Across two repository benchmarks, four kernel suites, and five real-world case studies, DuoDiscover improves over competitive baselines, and its margin widens as the number of agents increases. We expect *co-discovery*, letting agents find their own targets and yardsticks, to help other agentic searches where the true evaluation is expensive.

Limitations. While DuoDiscover accelerates optimization across diverse domains, it has two primary limitations. First, when no faithful yardstick can be constructed for a target, DuoDiscover falls back to standard search over the expensive true objective. Second, constructing yardsticks and running initial fidelity tests incurs an upfront warmup overhead, making DuoDiscover less advantageous under extremely short evaluation budgets.

AI USE STATEMENT

In this work, we used generative AI tools for assisting code implementations. We have not used generative AI tools to generate synthetic data sets, help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, or interpret results. The remaining required disclosure tasks, assisting with translation, cleaning and reformatting datasets, and supporting qualitative and thematic data analysis, are not applicable to this work. We also used generative AI tools as writing aids, but only for initial drafting of certain paragraphs and copy-editing. AI-generated text was substantially rewritten and edited by the authors, who are fully responsible for the final wording and content. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The defaults of the fidelity test are given in Appendix A.1. For each benchmark, the budget, the hardware, and the true evaluation time can be found in Appendix B.1 (Table 7), together with how each baseline was run (Table 8). Per-task results, the case studies, the ablations, as well as the robustness tests are in Appendices C to F, while the token cost of a run is shared in Appendix F.3. As explained in the Scoring paragraph of Section 3, the score we report for a run is obtained by re-evaluating its final candidate several times. Since LLM agent runs are not deterministic, we reran a subset of our tasks (50 kernels and 20 code base tasks) with three seeds each and report the spread in Appendix F.5. All main experiments were run in August and September 2026; the harness and model versions we used are given in the Baselines paragraph of Section 3. We plan to release the code, the agent prompts, and the per-run records behind the numbers and figures in the paper.

REFERENCES

Mohamed S. Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas Donald Lane. Zero-cost proxies for lightweight NAS. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=0cmMMy8J5q.

Shubham Agarwal, Alexander Krentsel, Shu Liu, Mert Cemri, Audrey Cheng, Rui Meng, Tomas Pfister, Chun-Liang Li, Sylvia Ratnasamy, Aditya Parameswaran, Matei Zaharia, Ion Stoica, and Mohsen Lesani. Inductive deductive synthesis: Enabling ai to generate formally verified systems. *arXiv preprint arXiv:2605.23109*, 2026.

Yash Akhauri, Juan Pablo Munoz, Nilesh Jain, and Ravishankar Iyer. EZNAS: Evolving zero-cost proxies for neural architecture scoring. In *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=lSqaDG4dvdt.

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference (AFIPS)*, pp. 483–485, 1967. doi: 10.1145/1465482.1465560.

Anthropic. Claude code. Computer software, 2026. URL https://claude.com/product/claude-code. Terminal-based agentic coding tool. Accessed 2026-09.

R. Bera, K. Kanellopoulos, and O. Mutlu. Dpc4 2026. https://sites.google.com/view/dpc4-2026/home, 2026.

Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1121–1137, 2021.

Alexander Blasberg, Vasilis Kypriotis, and Dimitrios Skarlatos. Agentic architect: An agentic ai framework for architecture design exploration and optimization. *arXiv preprint arXiv:2604.25083*, 2026.

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

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations*, 2023.

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. *arXiv preprint arXiv:2603.24517*, 2026a.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pp. 578–594, 2018.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024.

Zhi Chen, Zhensu Sun, Yuling Shi, Chao Peng, Xiaodong Gu, David Lo, and Lingxiao Jiang. Rethinking the value of agent-generated tests for LLM-based software engineering agents. *arXiv preprint arXiv:2602.07900*, 2026b.

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

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. CUDA agent: Large-scale agentic RL for high-performance CUDA kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

Ryan Deng, Yuanzhe Liu, Bastian Lipka, Yao Ma, Xuhao Chen, Tim Kaler, and Jatin Ganhotra. PerfAgent: Profiler-guided iterative refinement for repository-level code optimization. *arXiv preprint arXiv:2607.19653*, 2026.

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

Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1437–1446. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/falkner18a.html.

M. Giselle Fernández-Godino, Chanyoung Park, Nam H. Kim, and Raphael T. Haftka. Issues in deciding whether to use multifidelity surrogates. *AIAA Journal*, 57(5):2039–2054, 2019. doi: 10.2514/1.J057750. URL https://doi.org/10.2514/1.J057750.

Alexander I. J. Forrester, András Sóbester, and Andy J. Keane. Multi-fidelity optimization via surrogate modelling. *Proceedings: Mathematical, Physical and Engineering Sciences*, 463(2088): 3251–3269, 2007. ISSN 13645021. doi: 10.1098/rspa.2007.1900. URL http://www.jstor.org/stable/20209374.

Xinghong Fu, Aravinth Kulanthaivelu, and Yutaro Yamada. Self improvement via fast tree-search. *arXiv preprint arXiv:2609.19526*, 2026.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning (ICML)*, pp. 10835–10866, 2023.

Nathan Gober, Gino Chacon, Lei Wang, Paul V Gratz, Daniel A Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. The championship simulator: Architectural simulation for education and competition. *arXiv preprint arXiv:2210.14324*, 2022.

Abraham Gonzalez, Raghav Gupta, Akanksha Jain, Hanna Alam, Alexander Novikov, Po-Sen Huang, Matej Balog, Marvin Eisenberger, Sergey Shirobokov, Ngân Vũ, Hank Levy, Borivoje Nikolić, Sagar Karandikar, Martin Dixon, and Parthasarathy Ranganathan. Archagent v2: A case study with the data prefetching championship. *arXiv preprint arXiv:2608.09874*, 2026.

Google. Gemini 3.8 flash: Model card. Google DeepMind Model Card, 2026. URL https://deepmind.google/models/model-cards/gemini-3-8-flash/. Accessed: 2026-09-19.

Raghav Gupta, Akanksha Jain, Abraham Gonzalez, Alexander Novikov, Po-Sen Huang, Matej Balog, Marvin Eisenberger, Sergey Shirobokov, Ngân Vũ, Martin Dixon, Borivoje Nikolić, Parthasarathy Ranganathan, and Sagar Karandikar. Archagent: Agentic AI-driven computer architecture discovery. *arXiv preprint arXiv:2602.22425*, 2026.

Myles Hollander, Douglas A Wolfe, and Eric Chicken. *Nonparametric statistical methods*. John Wiley & Sons, 2013.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

Jiachen Jiang, Tianyu Ding, and Zhihui Zhu. Deltaevolve: Accelerating scientific discovery through momentum-driven evolution. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=ownPRHrXMx.

Kirthevasan Kandasamy, Gautam Dasarathy, Junier B Oliva, Jeff Schneider, and Barnabas Poczos. Gaussian process bandit optimisation with multi-fidelity evaluations. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

Jacek Karwowski, Oliver Hayman, Xingjian Bai, Klaus Kiendlhofer, Charlie Griffin, and Joar Skalse. Goodhart's law in reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2024.

Maurice G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1–2):81–93, 1938. doi: 10.1093/biomet/30.1-2.81.

Marc C. Kennedy and Anthony O'Hagan. Predicting the output from a complex computer code when fast approximations are available. *Biometrika*, 87(1):1–13, 2000. doi: 10.1093/biomet/87.1.1.

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

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. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/7886b9bafe76c52fd568db10ff9772df-Paper-Conference.pdf.

Ke Li and Fan Li. Multi-fidelity methods for optimization: A survey. *ACM Comput. Surv.*, 58(12), May 2026. ISSN 0360-0300. doi: 10.1145/3801959. URL https://doi.org/10.1145/3801959.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018. URL http://jmlr.org/papers/v18/16-558.html.

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Ruichao Xiao, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, Zewei Jiang, Roman Levenstein, Dianshi Li, Liyuan Li, Ajit Mathews, Varna Puvvada, Feng Shi, Nathan Yan, Xiayu Yu, Uladzimir Pashkevich, Matt Steiner, Carole-Jean Wu, and Gaoxiang Liu. KernelEvolve: Scaling agentic kernel coding for heterogeneous AI accelerators at Meta. In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 733–747. IEEE, 2026. doi: 10.1109/ISCA66397.2026.00063.

Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, Qijing Huang, Zhifan Ye, Nestor Qin, Fengzhe Zhou, Yuan Zhang, Jingquan Wang, Sana Damani, Dheeraj Peri, Ouye Xie, Aditya Kane, Moshe Maor, Michael Behar, Triston Cao, Rishabh Mehta, Vartika Singh, Vikram Sharma Mailthody, Terry Chen, Zihao Ye, Hanfeng Chen, Tianqi Chen, Vinod Grover, Wei Chen, Wei Liu, Eric Chung, Luis Ceze, Roger Bringmann, Cyril Zeller, Michael Lightstone, Christos Kozyrakis, and Humphrey Shi. SOL-ExecBench: Speed-of-light benchmarking for real-world GPU kernels against hardware limits. *arXiv preprint arXiv:2603.19173*, 2026a.

Hannah Lin, Martin Maas, Maximilian Roquemore, Arman Hasanzadeh, Fred Lewis, Yusuf Simonson, Ameya Shringi, Hongwen Dai, Patrick Musau, Tzu-Wei Yang, Amir Yazdanbakhsh, Deniz Altınbüken, Florin Papa, Maggie Nolan Edmonds, Aditya Patil, Don Schwarz, Satish Chandra, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. ECO: An AI-Driven code efficiency optimizer for warehouse scale computers (operational systems). In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 589–625, Seattle, WA, July 2026b. USENIX Association. ISBN 978-1-939133-55-7. URL https://www.usenix.org/conference/osdi26/presentation/lin-hannah.

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*, 2026.

Mao Luo, Hongbin Li, Feng Lin, Hanling Yi, and Zhe Huang. Agentic kernel optimization: Generating state-of-the-art gpu kernels without hand-written cuda. *arXiv preprint arXiv:2608.14560*, 2026.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. SWE-fficiency: Can language models optimize real-world repositories on real workloads? *arXiv preprint arXiv:2511.06090*, 2025.

MLSys 2026 Competition. MLSys 2026 FlashInfer AI Kernel Generation Contest, 2026. URL https://mlsys26.flashinfer.ai/.

Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñals-Yúfera, and Alberto Ros. Berti: an accurate local-delta data prefetcher. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 975–991. IEEE, 2022.

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

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.

Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. SpecRover: Code intent extraction via LLMs. In *IEEE/ACM International Conference on Software Engineering*, 2025.

Atharva Sehgal, James Hou, Akanksha Sarkar, Ishaan Mantripragada, Swarat Chaudhuri, Jennifer J. Sun, and Yisong Yue. Formulacode: Evaluating agentic optimization on large codebases. *arXiv preprint arXiv:2603.16011*, 2026.

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

Yu Shen, Yang Li, Jian Zheng, Wentao Zhang, Peng Yao, Jixiang Li, Sen Yang, Ji Liu, and Bin Cui. ProxyBO: Accelerating neural architecture search via Bayesian optimization with zero-cost proxies. In *Proceedings of the AAAI conference on artificial intelligence*, volume 37, pp. 9792–9801, 2023. doi: 10.1609/aaai.v37i8.26169.

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

Simranjit Singh, AN Torres, and Alberto Ros. Pushing the limits of the berti prefetcher. *4th Data Prefetching Championship (DPC4)*, 2026.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pp. 9460–9471, 2022.

Leitian Tao, Baolin Peng, Haorui Wang, Hang Wang, Hao Cheng, Wenlin Yao, Qianhui Wu, Tao Ge, Sharon Li, and Jianfeng Gao. ExecCritic: Learn to test, test to improve for coding agents. *arXiv preprint arXiv:2609.09133*, 2026.

Arya Tschand, Charles Hong, Julian Walker, Nina Cai, Shangkun Wang, Suvinay Subramanian, Sundar Dev, Vijay Janapa Reddi, Amir Yazdanbakhsh, and Sethu Sankaran. JAXBench: Benchmarking autonomous TPU kernel optimization. *arXiv preprint arXiv:2607.20466*, 2026.

vLLM Project. tpu-inference: Tpu inference for vllm, with unified jax and pytorch support, 2026. URL https://github.com/vllm-project/tpu-inference. GitHub repository. Accessed 2026-09-21.

Shangkun Wang, Nina Cai, Charles Hoong, Julian Walker, Gerson Kroiz, George Vanica, Deepak Patil, Andi Gavrilescu, Hassan Sipra, and Sethu Sankaran. MaxKernel: Agentic kernel generation for TPUs. *arXiv preprint arXiv:2609.04523*, 2026a.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. Thetaevolve: Test-time learning on open problems. In *Forty-third International Conference on Machine Learning*, 2026b. URL https://openreview.net/forum?id=XHP568OuIl.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for GPU kernel performance optimization. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025. URL https://openreview.net/forum?id=IZKZIcPaHz.

Colin White, Arber Zela, Robin Ru, Yang Liu, and Frank Hutter. How powerful are performance predictors in neural architecture search? In *Advances in Neural Information Processing Systems*, volume 34, pp. 28454–28469. Curran Associates, Inc., 2021.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, Aksara Bayyapu, Luis Ceze, and Tianqi Chen. FlashInfer-Bench: Building the virtuous cycle for AI-driven LLM systems. In *Proceedings of Machine Learning and Systems*, volume 8, pp. 2016–2064. MLSys, 2026.

Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Shuo Chen, Zhankui He, Noveen Sachdeva, Isabella Ye, Weili Wang, Chi Wang, Ed H. Chi, Fernando Pereira, Wang-Cheng Kang, Derek Zhiyuan Cheng, and Beidou Wang. PACEvolve: Enabling long-horizon progress-aware consistent evolution. In *ICLR 2026 Workshop on Lifelong Agents: Learning, Aligning, Evolving*, 2026.

Shuo Yang, Xiaoze Fan, Melissa Pan, Haocheng Xi, Zhe Wang, Shanlin Sun, Kurt Keutzer, Song Han, Matei Zaharia, Chenfeng Xu, and Ion Stoica. Freetoken: Efficient edge-native moe serving with bandwidth-adaptive execution. *arXiv preprint arXiv:2608.16157*, 2026.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=96zNuQrH9Y.

Ted Zadouri, Markus Hoehnerbach, Jay Shah, Vijay Thakkar, and Tri Dao. FlashAttention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. In *Proceedings of Machine Learning and Systems*, volume 8, pp. 912–926. MLSys, 2026.

Dongyun Zou, Ligeng Zhu, and Yixin Dong. KDA: Kernel design agents for high-performance gpu kernel generation, 2026. URL https://github.com/NVlabs/kda. GitHub repository.

A METHOD DETAILS

This appendix expands §2. Table 6 collects the notation of the Method. The terms the Method introduces are the following. A *target* is what an agent optimizes, a region of the code, and a *yardstick* is the cheap scorer the agent builds for it (§2.2); choosing the two together is *co-discovery*. The *fidelity test* is the orchestrator's rank-correlation test of a yardstick against f, repeated after every true evaluation. A yardstick that has passed it is *validated*, and one that fails it after having passed is *invalidated* (§2.4). The *orchestrator* is an LLM, with supporting code, that coordinates the agents and runs the fidelity test (§2).

Table 6: Notation of the Method, in the order it is introduced.

Symbol	Meaning	Defined in
x, $\mathcal{X}$	a program state (a version of the code); the set of all states	§2.1
x_0	the initial program	§2.1
f	the true evaluation, $f : \mathcal{X} \to \mathbb{R} \cup \{\perp\}$, higher is better; $\perp$ means the state fails the task's correctness check	§2.1
c_f	the cost of one call to f, measured as its running time	§2.1
N	the number of agents in a run	§2.1
T, $\mathcal{X}_T$	a target (a region of the code); the states reachable by editing only inside T	§2.2
w_T	the agent's rough guess of the share of the true score that T accounts for	§2.2
g_T	the yardstick for T, $g_T : \mathcal{X}_T \to \mathbb{R} \cup \{\perp\}$	§2.2
c_g, $r = c_f/c_g$	the cost of one yardstick run; how many yardstick runs fit in one true evaluation	§2.2
τ	Kendall's rank correlation between g_T and f over the resolved pairs (Eq. 1)	§2.4

A.1 FIDELITY TEST DETAILS

This section gives the details of the fidelity test of §2.4: how the noise floor is measured, which statistic and significance test we use, and when a yardstick is validated or invalidated and what follows.

Let $S = \{x_1, \ldots, x_n\}$ be the states scored by both f and the current yardstick, pooled across all agents that share the yardstick, and let $\hat{\sigma}$ be the relative evaluation noise, estimated from repeated true evaluations of the same state. Only pairs that the true evaluation itself can resolve are counted:

$$P_{\hat{\sigma}}(S) = \left\{(i,j) : |f(x_i) - f(x_j)| > 2\hat{\sigma}\bar{f}_{ij}\right\}, \qquad \tau(g_T, f) = \frac{n_c - n_d}{n_c + n_d}, \tag{1}$$

where $\bar{f}_{ij}$ is the mean score of the pair, and n_c and n_d count the pairs in $P_{\hat{\sigma}}(S)$ that g_T orders the same way as f and the opposite way, respectively (pairs tied under the yardstick are excluded). Two states within noise of each other can therefore neither add to nor subtract from the correlation. Throughout this appendix, τ means this restricted statistic.

Noise floor. We never assume a value for the relative evaluation noise $\hat{\sigma}$; it is measured from the run itself. Whenever the same snapshot (identified by a content hash of its source files) has been evaluated more than once, the residuals of $\log f$ across those evaluations are pooled over all such snapshots, and $\hat{\sigma}$ is the 95% upper confidence bound on their standard deviation, so that a lucky low sample cannot understate the noise. Until at least one snapshot has been evaluated twice, the noise is unmeasured and no yardstick is validated. A minimum separation of 1% applies in any case, so that sub-percent jitter never counts as a resolved difference. A pair of states is resolved when its true scores differ by more than $2\hat{\sigma}$ times their mean (Eq. 1).

Statistic and significance test. Rank correlation between a proxy and the objective is a commonly used test (Abdelfattah et al., 2021; White et al., 2021; Akhauri et al., 2022; Shen et al., 2023). All runs in this paper use Kendall's τ (Kendall, 1938) over the resolved pairs, with yardstick ties

excluded, and a one-sided permutation test in each direction: the yardstick scores are permuted across states, the same restricted τ is recomputed under each permutation, and the p-value is the share of permutations at or beyond the observed value.

When a yardstick is validated or invalidated. A yardstick is *validated* when the noise has been measured, at least $m = 6$ distinct states resolve beyond it, $\tau \geq \tau_{\min} = 0.5$, and the permutation test rejects independence at level $\alpha = 0.05$. Until then the yardstick is simply untested, and the agent is told what evidence would settle the question, typically more confirmed states that are far apart. A yardstick is *invalidated* when it has that much evidence yet $\tau < \tau_{\min}$. A significantly negative τ also forces a switch to a different kind of yardstick at once. After an invalidation, the orchestrator stops spending true evaluations on confirming states climbed under that yardstick, and the agent either rebuilds the yardstick or moves on to a new target. The agent sees the measured τ and the ratio r whenever it is launched, and it may rebuild on its own before the test has reached a conclusion. If no yardstick for the target validates, the agent scores candidates with f alone.

B Benchmark and baseline details

This appendix expands the setup of §3. It lists the benchmarks (Table 7) and how each baseline was run (Table 8).

B.1 Benchmarks

We report every benchmark in Table 7: its score, reference, task count, budget, true-evaluation time, and the floor and ceiling of its score. Floor is the reference implementation's score, for SOL-Exec the mean over all 235 kernels. Ceiling is the hardware peak for SOL-Exec and the best known result elsewhere. For FormulaCode and SWE-fficiency it is the expert patch, one merged pull request per task. Every method exceeds it on FormulaCode, and DuoDiscover, Claude Code and PerfAgent exceed it on SWE-fficiency.

Hardware. SOL-Exec and FA4 kernels run and are evaluated on NVIDIA B200 GPUs, one GPU per kernel. TPU-Inference and JAXBench kernels run on TPU v6e. Repository tasks are timed one per machine, on 16-vCPU, 32 GB nodes for SWE-fficiency and 2-vCPU, 4 GB nodes for FormulaCode. The case studies use an 8×B200 node for vLLM, one 48 GB L40S, with the model's 433 GB of weights in host memory, for FreeToken, and a 44-vCPU node for SQLite and zstd (Appendix D). The data-prefetcher study runs on the ChampSim simulator (Gober et al., 2022) on a 3000-vCPU cluster in three configurations, one core at full bandwidth, one bandwidth-limited, and four cores sharing memory. Every method we run on a benchmark uses the same hardware, and the true-evaluation times in Table 7 are measured on it.

Budget. On the benchmarks every method gets 8 hours (24 hours on the FormulaCode tasks whose true evaluation exceeds 2 hours, and 10 hours on TPU-Inference), four agents, and at most four concurrent true evaluations. The other four case studies run true evaluations one at a time, since each uses a single machine (the 8-GPU node for vLLM, the L40S for FreeToken, and one node for SQLite and zstd). The data-prefetcher study instead runs true evaluations in parallel on a cluster, each fanning its 610 traces and 484 mixes across cores, so we report both its wall-clock time and its total CPU-hours (Appendix D.5).

B.2 Baselines

Table 8 details each evaluated baseline alongside its execution setup. Every baseline executed within our benchmark suite shares DuoDiscover's hardware setup, budget (8 hours on the benchmarks and 20 or 10 hours on the case studies, with per-benchmark exceptions in Table 7), agent count, base code, task prompts, and true-evaluation procedures. Every method we run is evaluated on the full set of each benchmark it targets, all 100 SWE-fficiency tasks or all 235 SOL-Exec kernels, except Antigravity, which runs on the 20-task and 50-kernel sample. The FA4 numbers for AVO and cuDNN are adopted directly from the AVO paper.

Table 7: The benchmarks, with their task type, score, comparison baseline, task count n, budget, true-evaluation time, and score floor and ceiling. On each benchmark, every method we run shares the same hardware, budget, agent count (four), and concurrency limit (four concurrent true evaluations on the benchmarks, one on the case studies). A true evaluation is one sequential run of the full workload, so adding machines cannot shorten it. The data-prefetcher study is the exception. It uses three agents, runs its traces in parallel, and takes its baselines as reported rather than rerun.

Benchmark	Task	Score	Compares against	n	Budget	Eval time	Floor	Ceiling
FormulaCode	repo runtime	speedup (×)	base code	25	8 h/ 24 h	7 min - 6.8 h	1.00	1.048
SWE-fficiency	repo runtime	speedup (×)	base code	100	8 h	15 min (3 to 93)	1.00	6.90
SOL-Exec	GPU kernel	% of SOL	hardware peak	235	8 h	14 min (2 to 60)	26	100
JAXBench	kernel	speedup (×)	XLA baseline	17	8 h	17 min (7 to 32)	1.00	–
TPU-Inference	kernel	speedup (×)	optimized Pallas kernel	7	10 h	3 min (1 to 8)	1.00	–
FA4	kernel	% over FA4	Flash Attention 4	1	8 h	4 min	0	–
vLLM	open-source	geomean gain (×)	tuned, unmodified vLLM	1	20 h	about 85 min	1.00	–
FreeToken	open-source	geomean gain (×)	unmodified FreeToken	1	20 h	about 70 min	1.00	–
SQLite	open-source	speedup (×)	base code	1	10 h	about 10 min	1.00	–
zstd	open-source	instruction ratio (×)	base code	1	10 h	15 to 25 min	1.00	–
Data prefetcher	hardware design	speedup (×)	reference core	1	9 d	up to 28 h	1.00	–

Table 8: Summary of baseline methods, benchmark coverage, and execution configurations.

Baseline	Benchmarks	Setup
Claude Code	all	Four independent agents on Opus 5, same prompt as DuoDiscover, each on its own copy of the code. Every candidate scored by a true evaluation. Best of four reported.
OpenEvolve	all	On FormulaCode and SWE-fficiency we run OpenEvolve-Agentic, with the mutation operator replaced by a coding agent so it can edit whole codebases. On the kernel suites, unmodified OpenEvolve. Both run on Opus 5 under the same budget and true-evaluation setup as the other baselines. True evaluation as fitness, best of four independent runs reported.
PerfAgent	all	Our reimplementation on Opus 5, four agents. Each profiles with the task's profiler, edits the hottest region, re-profiles. Every candidate scored by a true evaluation.
KDA	SOL-Exec, FA4	Run by us, 8-hour budget, same hardware. Same model and agents as DuoDiscover, run as KDA kernel agents.
AVO	FA4	As reported in the AVO paper.
cuDNN	FA4	NVIDIA's attention kernels, no search. As reported in the AVO paper.
Antigravity	SWE-fficiency, SOL-Exec	Antigravity harness on Gemini 3.8 Flash, run as Claude Code above, on the 20-task and 50-kernel sample (§4).
ArchAgent v2	data prefetcher	AlphaEvolve-based system; its first of five search stages alone ran over 40 days and graded 12,000+ candidates. Numbers as reported (Gonzalez et al., 2026).
BertiGO	data prefetcher	Best human-designed prefetcher, no search (Singh et al., 2026). Numbers as reported in the ArchAgent v2 paper (Gonzalez et al., 2026).

C Extended main results

This appendix expands §3.1, one subsection per benchmark in the order of Table 1.

C.1 FormulaCode

Per-task scores for every method on the 25 FormulaCode tasks can be seen in Table 9. Each task provides a snapshot of the repository just before the human expert PR, the original issue and discussion

context, correctness tests, and a suite of community-maintained Airspeed Velocity (ASV) performance benchmarks. FormulaCode exposes an average of 264 performance workloads per task and scores a solution by its geometric-mean speedup across these workloads. We report the geometric-mean metric for each task across all benchmark loads. DuoDiscover beats the other baselines in 16 out of 25 tasks, with 21% higher mean speedup across the selected tasks.

Task	OpenEvolve	Claude Code	PerfAgent	DuoDiscover
TileDB-Py_2206	1.175	1.627	1.461	**1.730**
uxarray_1249	1.116	3.207	4.093	**6.076**
networkx_6879	1.430	3.349	2.310	**3.554**
networkx_7181	1.056	1.652	1.566	**2.627**
networkx_7249	1.420	2.137	1.984	**3.666**
networkx_7595	1.015	1.607	2.237	**2.616**
networkx_7971	1.008	**2.557**	1.647	2.182
networkx_8077	1.098	1.815	1.990	**2.218**
networkx_8206	1.001	1.341	**1.404**	1.305
PyBOP_301	1.707	**2.748**	2.557	2.108
bottleneck_329	1.038	**1.478**	1.385	1.193
pymc_6711	1.742	2.682	2.913	**3.947**
pymc_7578	1.585	**2.527**	2.334	2.399
shapely_2283	1.068	2.034	1.078	**2.097**
flox_300	1.049	1.145	**3.292**	2.064
flox_360	1.075	**2.416**	1.004	2.344
flox_415	1.052	1.005	2.067	**2.281**
xdsl_4909	1.357	1.826	1.776	**1.971**
pandas_51344	1.015	1.023	1.031	**1.039**
pandas_53397	1.006	1.045	1.036	**1.089**
pandas_55104	1.016	0.993	**1.068**	1.041
pandas_57651	1.002	1.020	1.027	**1.061**
pandas_59647	1.001	1.024	1.029	**1.131**
pandas_60461	1.004	**1.132**	1.028	1.052
pandas_62623	0.999	1.071	1.028	**1.100**
Mean Speedup	1.161	1.778	1.774	**2.156**
Wins (out of 25)	0	6	3	**16**

Table 9: Speedup comparison across 25 tasks for FormulaCode. Entries report geometric mean speedups (higher is better). The best result in each row is bold. Mean Speedup is the arithmetic mean of the task scores, and Wins counts first-place finishes.

C.2 SWE-FFICIENCY

A true evaluation runs the repository's own tests that cover the changed code and times the task's workload. A candidate that fails a test scores base speed, and a score of 1.00 means no gain over the base code. DuoDiscover's final patch passes the tests on all 100 tasks.

C.3 SOL-EXEC

On September 10, 2026, DuoDiscover held first place on the public SOL-Exec leaderboard (Figure 5), with a mean SOL score of 79.74% over all 235 kernels against 78.95% for the second entry. The entry is our single 8-hour, four-agent run per kernel from Table 1, submitted once. Every method's SOL scores are graded by the same public leaderboard. The lead comes from breadth rather than a few kernels (Figure 5). DuoDiscover ranks in the top three on 212 of the 235 kernels, the second entry on 169, and it holds first place on 121 kernels against the runner-up's 41. Across the leaderboard's four collections it is first on L1, L2 and FlashInfer and second on Quant.

C.4 FA4

We compare every method on the 24 FA4 configurations, each scored as a speedup over Flash Attention 4, and present the geometric mean in Table 10 and the per-configuration speedups in Figure 6. DuoDiscover is 11.97% faster than Flash Attention 4, ahead of KDA at 8.93%, Claude

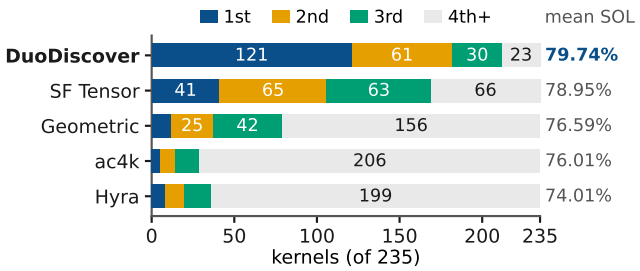


Figure 5: Per-kernel finishing position of the top five SOL-Exec entries, with mean SOL at the bar.

Code at 7.05%, PerfAgent at 6.86%, OpenEvolve at 1.48%, and the two entries reported in the AVO paper, AVO at 4.11% and cuDNN at 1.30%.

Table 10: Geometric-mean speedup over Flash Attention 4 across the 24 FA4 configurations. cuDNN and AVO are as reported in the AVO paper (Chen et al., 2026a). The rest are our runs.

Method	Speedup over FA4 (%)
FA4	0.00
cuDNN	1.30
AVO	4.11
KDA	8.93
Claude Code	7.05
PerfAgent	6.86
OpenEvolve	1.48
DuoDiscover (ours)	**11.97**

C.5 JAXBENCH

We evaluate every method on all 17 priority kernels of JAXBench (Tschand et al., 2026), a suite of production JAX/TPU operators drawn from models such as Llama-3.1, DeepSeek-V3, Mixtral-8x7B, Mamba-2, and RetNet (attention variants, mixture-of-experts, general matrix multiplication (GEMM), normalization, and more). Each kernel is evaluated on 13 input shapes taken from production use, with random seeds that the agent never sees. A candidate passes the correctness gate only if at least 98% of its output elements lie within a tolerance, set per data type, of the XLA baseline's output, and its score is the geometric-mean speedup over that baseline across the 13 shapes (XLA $= 1.00\times$). The per-method geometric means are in Table 1; a value of $1.00\times$ means the method found no speedup over XLA.

C.6 TPU-INFERENCE

The TPU-Inference results cover 7 production TPU kernels provided by the vLLM TPU-inference project (vLLM Project, 2026), including DeepSeek-v4 multi-head latent attention (MLA) variants, DeepSeek-v4 and GLM-5.2 stream-index top-k, and Kimi Delta Attention (KDA). The speedup obtained on each configuration of every kernel is plotted in Figure 7. The configurations include representative prefill and decode inference settings. These kernels were chosen after correspondence with the vLLM TPU-inference team because they cover a variety of important kernels from recent open frontier models and have robust testing and benchmarking scripts available. Note that, at the time of writing, the KDA kernel is not part of the public TPU-inference GitHub repository yet. The kernel was provided to us through correspondence with the vLLM TPU-inference team. The reference implementation for each kernel is the optimized production Pallas kernel, so a speedup of $1.00\times$ means no improvement over already-tuned code; unlike JAXBench, the denominator is not XLA, and the two suites should not be compared. Every method ran with a 10-hour budget and four agents (for DuoDiscover, three explorers and one control). The geometric mean over all 40 configurations is $2.84\times$ for DuoDiscover, $2.51\times$ for Claude Code, $2.27\times$ for PerfAgent, and

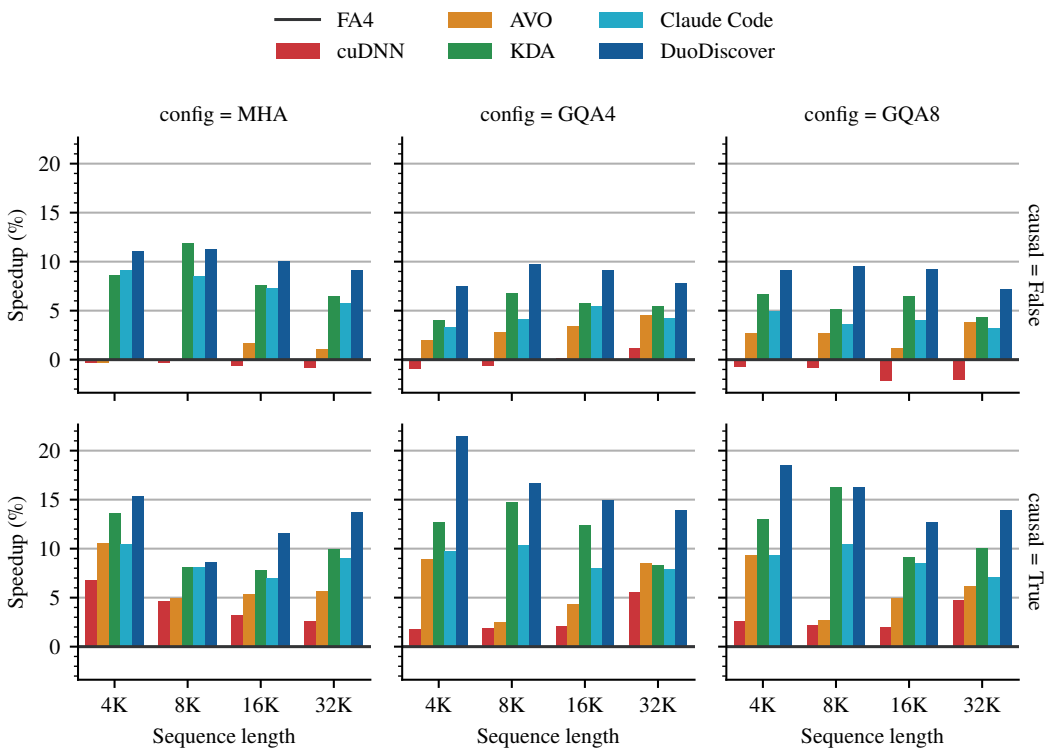


Figure 6: Speedup over Flash Attention 4 on each of the 24 FA4 configurations.

1.13× for OpenEvolve (Table 1). DuoDiscover is first on 5 of the 7 kernels. PerfAgent is ahead on DeepSeek-v4 MLA and Claude Code on DeepSeek-v4 MLA SWA.

D CASE STUDY DETAILS

This appendix expands §3.2. The runs are those of §3.2, the budgets are in Table 7, and the baseline is four Claude Code agents on Opus 5 as in Table 8. The data-prefetcher study is the exception, comparing against ArchAgent v2 and the best human design (Table 13). The fidelity test's floor is two evaluated states, the original code and the first candidate, instead of the six used on the benchmarks, since a target sees only a few true evaluations of 10 to 85 minutes each, and up to 28 hours for the prefetcher.

D.1 vLLM

- **Setting.** vLLM serves GLM-5.2-NVFP4 on one 8×B200 node, tensor parallel over the eight GPUs with expert parallelism and an fp8 key-value (KV) cache. The true evaluation scores p99 decode latency, first-token latency, power and throughput and takes their geometric mean. The baseline is unmodified vLLM tuned for the workload, with priority scheduling on, a long-prefill token threshold set, and at most 64 sequences in flight. A true evaluation sweeps a workload of 8192 input and 1024 output tokens at rising arrival rates, samples power once a second, requires the test suite to pass, and checks that greedy output equals the unmodified build's. It takes about 85 minutes.
- **Search.** Figure 8a shows the 20 hours. All four agents started on throughput until one profiled the server and found decode bound by memory latency. Streaming multiprocessor (SM) issue was at 7.8% and DRAM reads at 21% of peak. The orchestrator dropped the throughput targets, terminated one agent, and briefed the other three to explore. They found targets in p99 decode latency, GPU power, and first-token latency for a light client while a heavy client runs, all at fixed throughput. A decode cost model scored the prefill

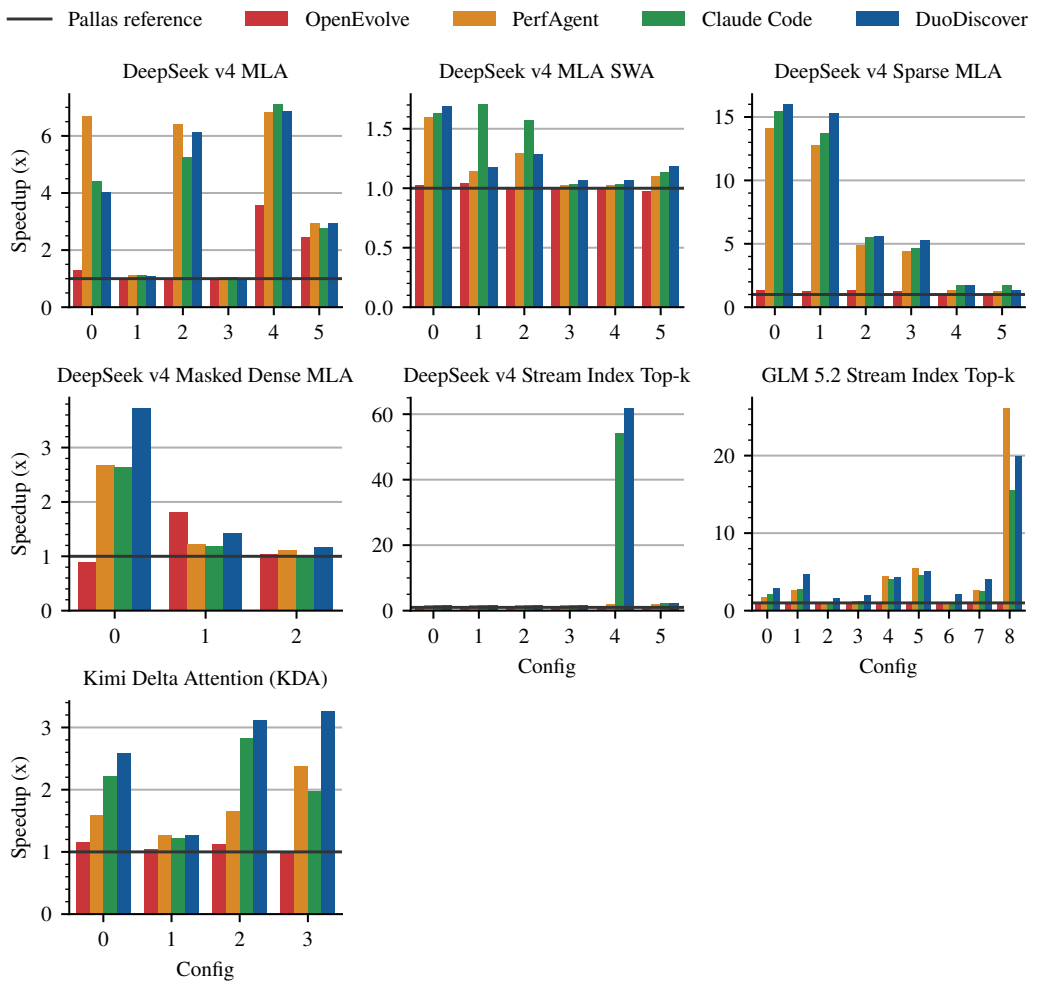
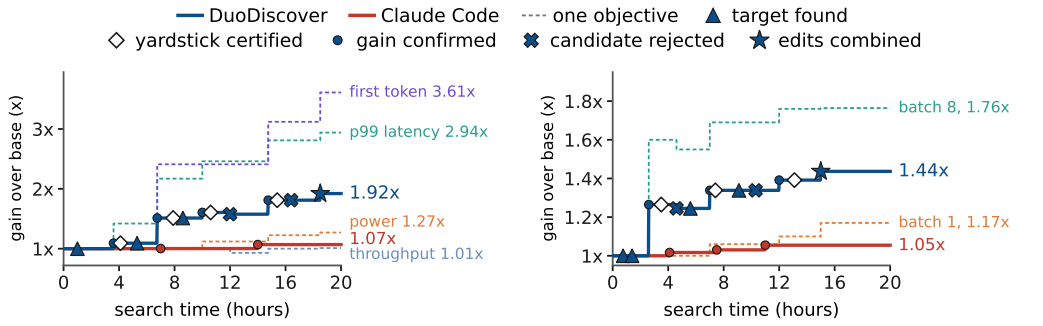


Figure 7: Speedup over the optimized production Pallas reference (horizontal line at $1.00\times$) on every evaluated configuration of each TPU-Inference kernel, for DuoDiscover, Claude Code, PerfAgent, and OpenEvolve.

cap in seconds, and a 13-minute load sweep scored the per-client fair scheduler and the clock control. Each yardstick passed the fidelity test after its target's first true evaluation (the diamonds). The test invalidated the load sweep when higher load reversed its ranking, and at hour 15.4 validated the sweep rebuilt on two arrival rates. At hour 7 the orchestrator spawned a fourth agent onto the first-token target and handed it the p99 agent's validated prefill cap, re-tested on arrival. The fourth agent's queue change, confirmed at hour 14.75, lifted the first-token gain from $2.4\times$ to $3.1\times$. Of about 90 candidates, 7 went to true evaluation, 9.9 hours of evaluation time in the 20-hour run. Two candidates were rejected (the crosses). At hour 12 a prefill cap set below the decode floor (the step time of decode alone) cost 7% of throughput. At hour 16.4 a shortest-job-first schedule turned out to be one the unmodified server already provides. Prefix co-scheduling and preemptive fairness failed on their yardsticks and never cost a true evaluation. The seventh true evaluation, at hour 18.5, confirmed the combined edits (the star). Claude Code kept every agent on throughput for 10 hours, then moved to first-token latency. It confirmed two gains, throughput $1.01\times$ at hour 7 and first-token latency $1.29\times$ at hour 14, with p99 decode latency and power unchanged, and ended at $1.07\times$ after 12 true evaluations, 16.8 hours of evaluation time, against DuoDiscover's $1.92\times$.

- **Solution.** The solution is about 500 lines in four pieces. None of it touches a kernel, and greedy output equaled the unmodified build's byte for byte on all six check prompts.



(a) vLLM on an 8×B200 node, 20 h budget each. DuoDiscover 7 true evaluations (9.9 h of evaluation time), Claude Code 12 (16.8 h).

(b) FreeToken on a 48 GB L40S, 20 h budget each. DuoDiscover 6 true evaluations (6.9 h of evaluation time), Claude Code 14 (16.3 h).

Figure 8: Gain over the base code as the search runs. Thick solid lines are the geometric mean over the objectives at each true evaluation, DuoDiscover in blue and Claude Code in red. The main text shows the same curves without event markers (Figure 3). Thin dashed lines are the individual objectives behind that mean. Markers on the thick line are the search events in the legend. For Claude Code only confirmed gains are marked.

`scheduler.py` caps the prefill tokens per step from the cost model, so a decode step cannot stall behind a long prefill. The cap is recomputed each step from the current decode batch. The unmodified server's per-step limit, max-num-batched-tokens, is static and cannot do that. `dvfs_governor.py` lowers the GPU clock while decode latency has slack, which cuts power. A per-client fair scheduler, a virtual token counter, gives each client a weighted share of served tokens, so the light client's first token no longer waits behind the heavy client's backlog. The unmodified scheduler's priority is static and per request and gives no such share. One flag, the decode tail latency to hold, sets the prefill cap and the clock together. At hour 20 p99 decode latency is 2.94× lower, first-token latency 3.61× lower, and power 21% lower, a geometric mean of 1.92× over the four objectives with throughput at 1.01×. Table 11 uses fixed arrival rates rather than the true evaluation's rising-rate sweep, so its numbers exceed the hour-20 ones. The throughput row is the full solution at hour 20. The p99 and power rows use 1.0 and 1.5 requests per second. The first-token row compares the fair scheduler with first come first served. The tokens-per-joule row holds the clock at 1200 MHz against 1965 MHz on a short-prompt load, where the clock alone cuts power 24.9% and throughput 1.3%. The concurrent-prefill row compares the per-step cap with the unmodified server's long-prefill threshold at 256 or 2048 tokens. The clock control lowers the tail as well as power. A lower clock keeps more decodes in flight per step, so prefill is spread thinner and the collision spikes that set the tail shrink. The p99 then sits just above the decode floor set by the clock, so the cap and the clock control do not stack on tail latency. Together they give 23.1 ms p99 on the served workload, the same as the clock control alone, and the cap alone at full clock gives 16.5 ms. The pair gives up 6.6 ms of tail latency for the table's 34% power cut at equal throughput. Against the unmodified server's default chunked prefill at 16,384 batched tokens per step, the cap is a worst-case tail guard. It cuts median decode latency 10 to 15% at some decode counts and leaves it unchanged at others. Its gain is the p99 row of Table 11.

Table 11: Each mechanism of the vLLM solution measured after the run on 8×B200 serving GLM-5.2-NVFP4, under the fixed-rate conditions given in the text. The first-token row is against first-come-first-served scheduling, the throughput row is the full solution.

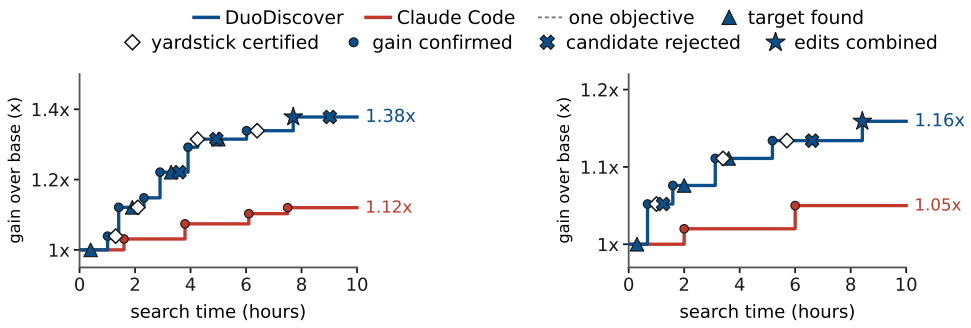
Measure	Tuned unmodified vLLM	DuoDiscover	Change
p99 decode latency (inter-token latency), real workload	141 to 337 ms	22 to 26 ms	6 to 13× lower
GPU power at equal throughput	4594 to 4968 W	3038 to 3237 W	34% lower
Tokens per joule, fixed low clock	baseline	1.314× baseline	31.4% higher
Light-client first-token latency under a heavy client (vs. first come first served)	9654 ms	1114 ms	8.7× lower
p99 decode latency under 8 concurrent long prefills	221.6 to 507.7 ms (baseline cap)	24.3 ms	9 to 21× lower
Peak throughput	baseline	unchanged	within 1%

D.2 FREETOKEN

- **Setting.** FreeToken runs GLM-5.2-NVFP4, a 753B mixture-of-experts model with 40B active parameters, on one 48 GB L40S. The model's 433 GB of weights live in host memory, with a cache of hot experts on the GPU. Each decode step fetches a fixed fraction of the experts it needs over PCIe and computes the rest on the CPU. Decode is bound by those fetches and by the CPU expert path. A true evaluation builds the engine and decodes 256 tokens at batch 1 and batch 8, three times each. The test suite must pass and greedy output must equal the unmodified build's on the evaluation's prompts. It scores decode throughput at batch 1 and batch 8 and takes their geometric mean. A true evaluation takes about 70 minutes.
- **Search.** Figure 8b shows the 20 hours. Each agent explored alone, chose a target, and built its own yardstick, a two-minute microbenchmark or a byte-count model. Two converged on the CPU expert loop, so the orchestrator terminated one and spawned a new agent, which found the shared-expert schedule. A third agent's target was the fp8 linear layer, scored against the floor set by the bytes it must move. A fourth's was the expert cache policy, scored by the bytes its routing sends over PCIe. The CPU-loop, shared-expert, and fp8 yardsticks passed the fidelity test at hours 3.5, 7.4, and 13.1 (the diamonds), each after a true evaluation on its target. At hour 4.6 a true evaluation rejected the 1.25× that the cache-policy yardstick had credited to a cut of a few percent in expert fetches. That yardstick was dropped (the first cross). Of about 80 candidates, 6 went to true evaluation, 6.9 hours of evaluation time in the 20-hour run. Gains were confirmed at hours 2.6, 7, and 12 (the circles). A tuned fetch fraction at hour 10.3 gave no gain over the confirmed best (the second cross), and the combined edits were confirmed at hour 15 (the star). Claude Code rewrote the batch-8 kernels and ended at 1.05× after 14 true evaluations, 16.3 hours of evaluation time, against DuoDiscover's 1.44×.
- **Solution.** `cpu_moe_ext.cpp` visits each expert's weights once while they are still in cache. The CPU expert loop is the largest gain. The shared expert runs inside the window where the GPU would otherwise wait for the CPU. `fp8_pertensor_linear.py` had a dispatch rule, written when decode meant one token, that sent every batch above one to the general tiled GEMM. At batch 8 that grid is 48 blocks of four warps on a 142-SM GPU, so most SMs sit idle. One added branch routes those batches to a new split-K kernel that fills the GPU as the batch-1 path does. Per batch-8 decode step the layer's calls fall from 38.2 to 24.1 ms. That 14.1 ms is most of the 19 ms cut from the 469 ms step. The 24.1 ms is close to the 22.9 ms the batch-1 path takes to move the same bytes. With this kernel on top of the other edits, the true evaluation's batch-8 ratio went from 1.69× to 1.76×, three true evaluations within 0.7% of each other. The rule returned correct values on every branch, so no test caught the misroute, and the byte-floor yardstick showed how far above the floor the layer ran. An early skip for the expert routes that the mask already zeroes removes about 37% of the GPU expert-GEMV memory reads. A worker loop that spins briefly before it parks removes a wakeup stall per layer. The masked-route skip and the shared-expert overlap are bit-identical by construction. The others passed the greedy-output gate. The final engine also carries edits confirmed as part of the combined edits at hour 15. Hot experts stay GPU-resident under a least-frequently-used policy with recency decay. Fetches are ordered by how many tokens need each expert, host-to-device copies overlap CPU compute, and the PCIe fetch fraction is set per model and GPU. At batch 1 each expert serves one token, so visiting its weights once saves nothing, and the batch-1 gain comes from the other edits. Table 12 gives the confirmed gain at each batch size, a geometric mean of 1.44×.

Table 12: FreeToken on one L40S, decode tokens per second against the unmodified build, measured by the true evaluation, three runs each.

Model	Batch	Unmodified	DuoDiscover	Speedup
GLM-5.2-NVFP4 (753B, 40B active)	1	6.75	7.9	1.17×
GLM-5.2-NVFP4 (753B, 40B active)	8	10.10	17.8	1.76×



(a) SQLite, wall time, 10 h budget each. DuoDiscover 17 true evaluations (2.8 h of evaluation time), Claude Code 45 (7.5 h).

(b) zstd, instruction count, 10 h budget each. DuoDiscover 7 true evaluations (2.4 h of evaluation time), Claude Code 21 (7.3 h).

Figure 9: Gain over the base code as the search runs, for SQLite and zstd, each a single objective, the zstd axis an instruction-count ratio. Lines and markers as in Figure 8.

D.3 SQLite

- **Setting.** The task is to make SQLite's own `speedtest1` workload faster, editing only `src/`. A true evaluation rebuilds the engine, runs the 13,249-test suite twice, times `speedtest1` at size 750 three times, and checks a result hash on seeds and query shapes the agent never sees. It takes about 10 minutes.
- **Search.** The agents found four targets in turn, the floating-point paths of the bytecode engine, the sorter, the recursive query path, and the decimal conversion (the triangles in Figure 9a). Each target used the same yardstick, the `cte` set of `speedtest1` at sizes 100 and 200, a recursive query over about 43 million rows that runs in seconds, tested separately per target. The four fidelity tests passed at hours 1.3, 2.1, 4.3, and 6.4 (the diamonds), each after a true evaluation on its target. Their scores stayed within 2 to 4% of the true evaluations. Of about 200 candidates, 17 went to true evaluation, 2.8 hours of evaluation time in the 10-hour run. Gains were confirmed at hours 1.0, 1.4, 2.3, 2.9, 3.9, 4.3, and 6.0 (the circles). The combined edits were confirmed at hour 7.7 (the star). Two candidates the yardstick ranked above the best confirmed candidate scored below it at hours 3.6 and 4.9, and one broke a test at hour 9.0 (the crosses). The other six scored below the best confirmed candidate and are not marked. Two agents converged on one target, and the kept edit was transferred from an earlier confirmed one. Claude Code ran 45 true evaluations, 7.5 hours of evaluation time, and ended at 1.12×, against DuoDiscover's 1.38×.
- **Solution.** The bytecode engine had fast paths for integer arithmetic, comparison and sorting but not for floats. `vdbe.c`, `vdbeaux.c`, and `vdbesort.c` add the same paths for floats, and the sorter merges 512 runs per pass instead of 16. `expr.c` and `select.c` compute a repeated subexpression once and, for a recursive query with no ORDER BY, pop the newest row from the queue first, a depth-first traversal that SQL leaves unspecified. `util.c` converts a decimal with a single correctly rounded multiply or divide when the mantissa is at most 2^{53} and the power of ten is within ± 22. `pcache1.c` computes its page-cache hash index with a bit mask instead of the original modulo. Two further edits are build settings inside the editable `src/`: `sqliteInt.h` adds an O3 optimize pragma, and `global.c` sets the default mutexes off, which the single-threaded workload never contends. Both passed the 13,249-test suite twice and the true evaluation. The confirmed gain is 1.38×, with the result hash equal to the unmodified build's.

D.4 zstd

- **Setting.** The task is to make zstd 1.6.0 execute fewer instructions on a fixed workload while producing the same bytes, editing only `lib/`. The task scores instruction count, so we report an instruction ratio, not a speedup. A true evaluation runs the 2,038-test suite over

14 branches and counts instructions under callgrind, Valgrind's instruction-count profiler, across 16 cells, four files from the Silesia compression corpus at levels 3, 9, 15, and 19. The score is the geometric mean over cells of the unmodified build's count over the candidate's. Any difference from the unmodified build's bytes scores 0. A true evaluation takes 15 to 25 minutes.

- **Search.** The agents found two targets, the optimal parser in `compress/zstd_opt.c` that level 19 uses and the row hash match finder in `compress/zstd_lazy.c` that level 9 uses. A third came later, the 32-byte match compare (the triangles in Figure 9b). The yardstick for each target took seconds, counting instructions under callgrind on one small file at one level. The parser's yardstick passed the fidelity test at hour 1.0, the match finder's at hour 3.4, and the compare's at hour 5.7 (the diamonds), each after a true evaluation on its target. The parser yardstick's score was within 0.04% of the first true evaluation. Of about 90 candidates, 7 went to true evaluation, 2.4 hours of evaluation time in the 10-hour run. Gains were confirmed at hours 0.7, 1.6, 3.1, and 5.2 (the circles). The combined edits were confirmed at hour 8.4 (the star). Two candidates the yardstick ranked above the best confirmed candidate scored below it at hours 1.2 and 6.6 (the crosses). The orchestrator transferred two of the final edits from a second agent. Claude Code ran 21 true evaluations, 7.3 hours of evaluation time, and ended at 1.05×, against DuoDiscover's 1.16×.
- **Solution.** `compress/zstd_opt.c` builds the length and offset price tables once per forward pass instead of repricing every match. `compress/zstd_lazy.c` compares each match candidate as it is found instead of collecting and revisiting them. `compress/zstd_compress_internal.h` compares 32 bytes at a time with AVX2, `common/compiler.h` removes the software prefetches, and 13 compressor files gain an AVX2 `#pragma GCC target` line. The prefetch removal and single-pass finder cut the instruction count, and the AVX2 pragma compiles the hot files for AVX2. The confirmed gain is 1.16× with identical bytes.

D.5 Data-prefetcher design

- **Setting.** The task is the 4th Data Prefetching Championship (DPC4) (Bera et al., 2026) on the ChampSim simulator (Gober et al., 2022). The starting point is one reference core already running Berti (Navarro-Torres et al., 2022) at the L1D and Pythia (Bera et al., 2021) at the L2, under storage limits of 32, 128, and 256 KB per cache level. A design is scored by its geometric-mean speedup in instructions per cycle (IPC) over the reference core in three configurations: FullBW runs one core at full bandwidth, LimitBW throttles the memory bus, and MC runs four cores sharing memory. The overall score is the geometric mean of the three. FullBW and LimitBW run 610 traces and MC runs 484 four-trace mixes. A true evaluation consists of three cluster jobs totaling about 2,900 CPU-hours (5 to 9 hours of wall-clock for FullBW and LimitBW, and 25 to 28 hours for MC, simulating up to 120 traces or mixes in parallel), so the agents hill-climb yardsticks built from short subsets of the traces, and only the best designs reach a true evaluation.
- **Search.** The search ran for 9 days as a sequence of 12-hour runs with three agents each, every run seeded from the previous best. The seed was picked at the end of each run from a faster 610-trace re-grade; the official evaluations ran asynchronously in parallel alongside subsequent runs and did not gate them. Early designs improved FullBW but lost most of their gain once the memory bus was throttled, so the orchestrator briefed the agents to target the geometric mean of FullBW and LimitBW, which penalizes wasted prefetches. On day 6 the orchestrator briefed the agents to extend their edits to the L2 prefetcher, where a temporal correlator and an accuracy gate gave the largest gains. MC was not optimized directly. A short yardstick ranked single-core candidates for the MC true evaluation. Of about 1,500 candidates graded on the 131 released traces, 50 reached a true evaluation on FullBW and LimitBW and 12 reached MC.
- **Solution.** The final design is about 4,700 lines of C++. The L1D is a rewritten Berti with deeper cross-page lookahead and latency-triggered throttling. The L2 extends Pythia with an accuracy gate that disables it where its prefetches are inaccurate, a spatial-footprint engine, and a temporal correlator. The last-level cache has no prefetcher. It reaches a $1.0504\times$ overall speedup against ArchAgent v2's $1.0382\times$ and the best human design's $1.0352\times$ (Table 13).

Table 13: Official DPC4 scores, geometric-mean IPC speedup over the reference core (610 traces for FullBW and LimitBW, 484 mixes for MC). BertiGO (the best human design) and ArchAgent v2 are as reported in the ArchAgent v2 paper (Gonzalez et al., 2026).

Configuration	BertiGO	ArchAgent v2	DuoDiscover
FullBW	1.0516	1.0561	1.0750
LimitBW	1.0257	1.0456	1.0481
MC (484 mixes)	1.0285	1.0133	1.0287
Overall	1.0352	1.0382	1.0504

E Ablation and scaling details

This appendix expands §4 with a single task-wide evaluator in place of per-target yardsticks, the fleet-size sweep, and the fleet-scaling curves over time.

E.1 Per-target yardsticks against a single task evaluator

		Score	Validated	Cost
SWE-fficiency	DuoDiscover	14.93×	58%	4%
	one evaluator	12.98×	21%	29%
SOL-Exec	DuoDiscover	80.80%	73%	2%
	one evaluator	71.36%	33%	18%

Table 14: A yardstick per target against one evaluator per task. Validated is the share of targets (DuoDiscover rows) or tasks (one-evaluator rows) whose yardstick passes the fidelity test, Cost the run time over one true evaluation.

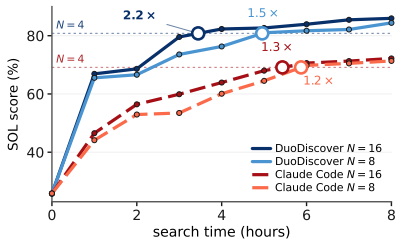


Figure 10: 8- and 16-agent fleets over 8 hours on the 50 SOL-Exec kernels. Dotted lines mark each method's four-agent result, rings where each fleet crosses it, labeled with the wall-clock speedup.

Instead of a yardstick per target, an agent could build one fast evaluator for the whole task and hill-climb on it. We test this design on the 20 repository tasks and 50 kernels of Section 4 with everything else unchanged (Table 14). A task-wide evaluator cannot be both fast and correlated with the true evaluation.

On the repositories it passes the fidelity test on 21% of tasks and, where validated, costs 29% of a true evaluation. The score drops from 14.93× to 12.98× on repositories and by 9.44 in SOL score, from 80.80% to 71.36%, on kernels. Amdahl's law (Amdahl, 1967) explains the breakdown. A yardstick only has to rank candidates within one region of the code. A task-wide evaluator must rank edits from every region, and to run fast it must trim its workload. An edit's gain grows with the share of the measured workload's time its region takes, so a trimmed workload divides time across regions differently from the real one, and the evaluator misranks edits from different regions relative to the true evaluation.

E.2 Fleet scaling

To test fleet scaling, we evaluate DuoDiscover and Claude Code across 50 SOL-Exec kernels for fleet sizes $N \in \{1, 2, 4, 8, 16\}$ under equal wall-clock time (Figure 11). At $N = 1$, DuoDiscover attains 70.26% SOL versus 66.47% for Claude Code due to higher candidate throughput on validated yardsticks. While the baseline rises slowly to 72.20% at $N = 16$ as isolated agents duplicate each other's search, DuoDiscover climbs to 84.41% at $N = 8$ and 85.96% at $N = 16$ as orchestration transfers discoveries across agents, so the gap widens at every fleet size, from 3.8 to 13.8 SOL. At $N = 16$, DuoDiscover attains its 4-agent benchmark score 2.2× faster (3.4 h vs. 7.6 h), compared to 1.3× for Claude Code (Appendix E.3).

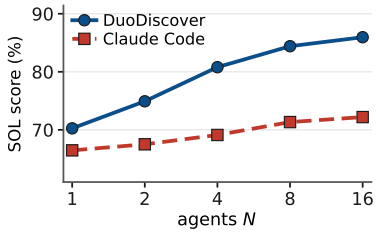


Figure 11: SOL score against agent count N, 8-hour runs on 50 SOL-Exec kernels.

E.3 Fleet scaling over time

Figure 10 tracks the 8- and 16-agent fleets over the 8-hour budget. Each curve starts at the reference implementation's score of about 26% and climbs to that method's final 8-hour score. On the 50 SOL-Exec kernels, a larger fleet buys DuoDiscover far more time than it buys Claude Code. With 16 and 8 agents, DuoDiscover attains its four-agent result of 80.80% at 3.4 and 5.0 hours. The four-agent run itself gets there at 7.6 hours, so the larger fleets are 2.2× and 1.5× faster. Claude Code needs 5.4 hours with 16 agents and 5.9 hours with 8 for its 69.12%, against 6.8 hours with four, so its larger fleets are 1.3× and 1.2× faster. The four-agent times are the hours at which the four-agent curves of Figure 4a settle at their final scores, and the speedups are computed before rounding. The difference comes from the orchestrator transferring discoveries across agents, where isolated agents duplicate each other's search.

F Yardstick and search analysis

This appendix expands Section 4. It gives yardstick speed and fidelity per target, how a run finds targets and hill-climbs, the token cost of a run, four examples of discovered targets and yardsticks, and the stress tests and run-to-run spread.

F.1 Yardstick quality

Table 2 (Section 4) gives the speed and rank fidelity of the yardsticks agents build on the 20 SWE-fficiency tasks and 50 SOL-Exec kernels. Fidelity is Kendall's rank correlation τ between a yardstick's scores and the true evaluation's over the candidates that yardstick sent to a true evaluation, reported as the median over validated yardsticks. The fidelity test validates a yardstick at $\tau \geq 0.5$ over at least six resolved states (§2.4). The test reuses states that any agent sharing the yardstick has already truly evaluated, so one true evaluation serves several tests, which is how a kernel run's 54 true evaluations cover its 11 validated yardsticks.

Two selection effects pull τ in opposite directions. True evaluations go only to candidates the yardstick favors, which narrows the score range and lowers τ, and only validated yardsticks are counted, which raises it. Over the kernel yardsticks that were never validated, the median is 0.03. A second view needs neither selection, the share of truly evaluated candidates that turn out to be gains. On kernels, 37.0% (20 of 54) are gains found on a yardstick, plus 1 combined edit, against 21.1% (16 of 76) for Claude Code. On repositories, 17.5% (10.3 of 59) are, plus about 2 combined edits, against 8.6% (6.2 of 72). Per confirmed gain, DuoDiscover spends 2.6 true evaluations on kernels and 4.8 on repositories, against 4.8 and 11.6 for Claude Code. Figure 12 plots per-target speed and, for every truly evaluated candidate, its yardstick rank against its true-evaluation rank on 15 of the 50 kernels. Its τ is pooled over all candidates of a kind, so it differs from the per-yardstick median in the table and from the fidelity test, which computes τ per target over noise-resolved pairs (§2.4).

F.2 Search behavior

The run logs of the 50 SOL-Exec kernels and 20 SWE-fficiency tasks show how agents discover targets, build yardsticks and hill-climb. All counts are means over the 50 kernels or the 20 tasks unless a median is named.

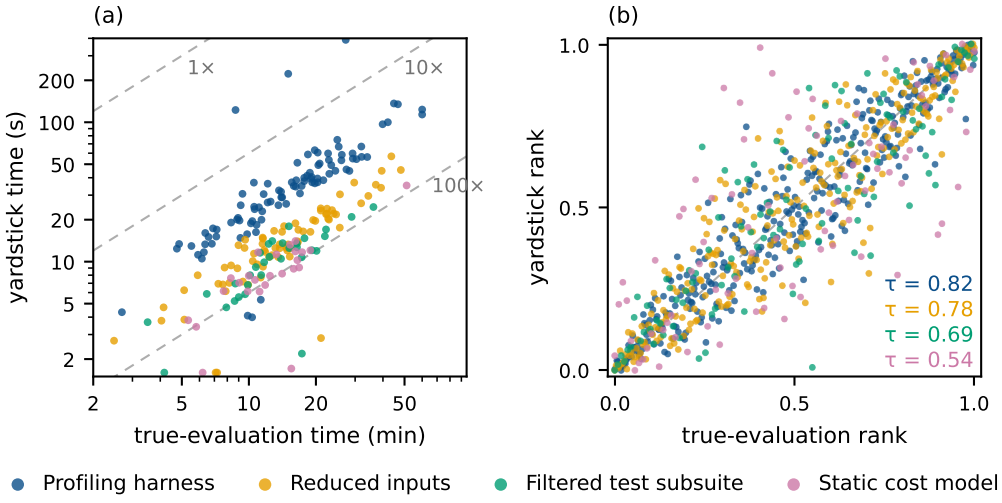


Figure 12: Yardstick quality by kind on 15 of the 50 SOL-Exec kernels. (a) Yardstick time against true-evaluation time, one point per target, dashed lines at 1, 10 and 100× faster. (b) Rank correlation between yardstick and true evaluation, one point per truly evaluated candidate, its rank by the yardstick against its rank by the true evaluation, pooled over the candidates of each kind, with Kendall τ per kind.

Target and yardstick co-discovery. Agents pick their own targets and build a yardstick for each, with no manual rule. A run pursues 15 targets per kernel, 11 of which get a validated yardstick, against 3 to 7 targets for Claude Code (Table 16). On repository tasks it pursues 9.2 targets per task, 5.3 validated, against Claude Code's 2 to 4. Two of the 20 tasks end with no validated yardstick. The yardsticks fall into four kinds (Table 15), the same four on both benchmarks with different shares: reduced inputs, a profiling harness, a filtered test subsuite, and a static cost model. The filtered test subsuite times the tests on the changed code, so it scores speed like the others, and it also has to keep those tests passing. Static cost models are the fastest kind and the least often validated, 32% on kernels and 18% on repositories, and one found the fp8 dispatch misroute in FreeToken that no test could catch (Appendix D.2).

Table 15: The four kinds of yardstick agents build on the 50 SOL-Exec kernels and the 20 SWE-fficiency tasks. Weighted by share, the kernel columns give the 53× and 11 of 15 of Table 2, the repository columns the 22× and 5.3 of 9.2.

		SOL-Exec kernels			SWE-fficiency tasks		
Yardstick kind	Scores a candidate on	Share	Faster by	Validated	Share	Faster by	Validated
Reduced inputs	smaller or low-precision inputs	33%	63×	75%	42%	20×	67%
Profiling harness	an isolated hot loop	42%	28×	81%	29%	11×	58%
Filtered test subsuite	tests on the changed code	14%	78×	69%	24%	31×	51%
Static cost model	instruction or memory counts	11%	89×	32%	5%	52×	18%

Per kernel, the 50-kernel sample behind Figure 4 ends with DuoDiscover ahead of Claude Code on 41 kernels, level on 5 and behind on 4.

Hill-climbing and drift. Agents try 285 candidates per kernel and 358 per repository task on their yardsticks, which run far faster than a true evaluation (Table 16). The yardstick screens out 81% of kernel candidates and 84% of repository candidates before any true evaluation. The rest yield 20 gains plus 1 combined edit on kernels, one gain per 14 candidates and 1.4 per pursued target. On repositories they yield 10.3 gains plus about 2 combined edits, one per 29 candidates. As later edits change the code a yardstick was built on, its ranking can drift. The fidelity test rechecks each yardstick after every true evaluation and invalidates it when it drifts, each recheck a significance test at level α (Appendix A.1). In a run, 45% of validated kernel yardsticks are invalidated, each after

Table 16: Search per task, means over 20 SWE-fficiency tasks and 50 SOL-Exec kernels, times are medians. Ratio is DuoDiscover over Claude Code, arrow up for more or faster, down for fewer true evaluations. Confirmed gains include the combined edits. Gains per true evaluation counts only edits found on a yardstick.

	SWE-fficiency (20 tasks)			SOL-Exec (50 kernels)		
	Claude Code	DuoDiscover	ratio	Claude Code	DuoDiscover	ratio
Targets pursued	2 to 4	**9.2**	–	3 to 7	**15**	–
Yardsticks validated	–	**5.3**	–	–	**11**	–
Time per candidate check	15 min	**40 s**	22× ↑	14 min	**16 s**	53× ↑
Candidates scored	72	**358**	5.0× ↑	76	**285**	3.8× ↑
True evaluations	72	**59**	0.8× ↓	76	**54**	0.7× ↓
Confirmed gains	6.2	**12.3**	2.0× ↑	16	**21**	1.3× ↑
Gains per true evaluation	8.6%	**17.5%**	2.0× ↑	21.1%	**37.0%**	1.8× ↑
Score	12.85×	**14.93**×	1.16× ↑	69.12%	**80.80%**	1.17× ↑

roughly 15 candidates, and 41% of repository yardsticks, after 19. Because each recheck reuses the true evaluation that confirmed a gain, rechecking adds no true evaluations of its own.

Patch footprint. Confirmed gains are rewrites, not parameter tweaks. On kernels, 98% of confirmed gains change more than 100 lines of the reference kernel and 83% more than 200. The median change is about 430 lines. On repository tasks, the final patch changes a median of four to five files and about 800 lines.

Explore, then exploit. The orchestrator spends the fleet's budget in two phases. Early on it keeps agents on different regions of the code, and when two converge on one target it usually terminates one and spawns a fresh agent on another region. This holds the share of agents on the same or a related target to 5 to 15% on kernels and 18% on repositories. Claude Code's uncoordinated agents sit at 40 to 60% and 55%, about 5× more at the kernel midpoints. Later it concentrates the budget on targets with confirmed gains, transfers discoveries between agents, combines complementary gains, and terminates the rest. Wide exploration is cheap. Of the 15 targets a kernel run pursues, 8 (53%) never confirm a gain, four because no yardstick is validated and four because a validated one finds nothing. On repository tasks, 5 of the 9.2 never confirm a gain. The orchestrator drops them early, so they take only 19% of the true evaluations on kernels and 20% on repositories.

F.3 Cost of a run

Table 17 breaks down the token consumption and API cost of an 8-hour, four-agent run on a single SOL-Exec kernel, averaged across 50 kernels at public Opus 5 list prices. Although a full 8-hour DuoDiscover search accumulates $375 in API charges against $309 for Claude Code, DuoDiscover equals Claude Code's final 8-hour score in 2.8 hours. At that point DuoDiscover has cost $79.60, or 25.8% of Claude Code's full run. All prices are public API rates as of September 2026. The Gemini cost comparison in Section 4 rests on a 6.7× price ratio between Opus 5 and Gemini 3.8 Flash per million tokens.

Table 17: Tokens and API cost of one 8-hour, four-agent run on one SOL-Exec kernel, mean over 50 kernels, at Opus 5 list prices with cache writes billed at the one-hour rate. GPU time for true evaluations is not included.

		DuoDiscover		Claude Code	
Token class	$ per M tokens	tokens	cost	tokens	cost
generated	25	4.4M	$110	3.6M	$90
cache writes	10	13.6M	$136	11.2M	$112
cache reads	0.50	258.7M	$129	213.2M	$107
total		**276.7M**	**$375**	**228.0M**	**$309**

Token spend follows the loop of four parallel agents re-reading their context and generating a reply on every turn. A DuoDiscover run executes 1,990 turns across its fleet, whereas Claude Code executes 1,640 because each of its agents pauses to wait on 19 true evaluations. On average a turn generates 2.2k tokens, re-reads 130k tokens of context billed as cache reads, and appends 4.8k tokens, the reply and tool returns, billed as cache writes. When the context hits the 200k-token limit it is compacted to 60k tokens and written back, which happens 68 times in a DuoDiscover run and 56 times in a Claude Code run. Cache writes are billed at the one-hour rate because true evaluations often idle an agent past the five-minute cache expiry.

F.4 Examples of discovered targets and yardsticks

Three of the 50 kernels and one of the 20 repository tasks show what a discovered target and its yardstick look like, and how gains add up.

Mixture-of-experts forward pass.

- **Setting.** A 128-expert mixture-of-experts layer at width 2880 with three stages, a gate and up projection, a clamped GLU activation, and a down projection. The reference scores 9.8% SOL. A true evaluation runs every shape on fresh inputs and takes about 16 minutes.
- **Search.** Each stage, and the routing between experts, became a target with its own yardstick. The first agent took the gate and up projection, the dominant matrix multiply. Its yardstick was a profiling harness, two warm-ups and five timed iterations over the real shapes on fixed inputs, 18 seconds. On it the agent found a tiling that uses TF32 for short token counts and a batched FP16 multiply above that count, confirmed at 58% SOL. The orchestrator transferred that kernel to the other agents. The second agent took the memory-bound activation with a byte-count model of high-bandwidth memory (HBM) traffic. The model ran in under a second and launched no kernel. The agent's fused activation halved the round trips and lifted the kernel to 67% on top of the tiling. The third agent took the down projection with a reduced-input microbenchmark, one batch at a shrunk token count in about two seconds. Its GEMM with in-place buffer reuse brought it to 74% on top of the tiling. The fourth agent took the routing between experts with a profiling harness sweeping occupancy as its yardstick. The sweep favored a two-stream pipeline, but a true evaluation showed no gain because the projection stages still dominated, and the target was dropped.
- **Solution.** Merging the activation and down-projection edits failed at first. Both reused the same scratch buffers, and the merged kernel returned wrong results. The combine step reconciled the three edits into one kernel, the tiled projection feeding the fused activation and the in-place GEMM, at 77.4% SOL.

Grouped-query attention with rotary embeddings.

- **Setting.** The reference scores 22.4% SOL and a true evaluation takes about two minutes.
- **Search.** One target was the kernel's tile size and occupancy. Its yardstick, a reduced-input microbenchmark, timed one long-context shape at a reduced token count, 13 seconds. It assumed that a tile or warp change shifts every shape the same way. The fidelity test checked that assumption against true evaluations and validated the yardstick at hour 1.1. The agent then swept tile and warp configurations.
- **Solution.** The winning configuration split the causal-pass loops and unified the tiling at 128 by 32. This edit alone took the kernel to 39% SOL, and with the edits on its other targets the kernel scored 71.5%.

GEMM with $N = 5120$ and $K = 2048$.

- **Setting.** The reference scores 48% SOL, already close to the cuBLAS ceiling. A true evaluation takes about 24 minutes.
- **Search.** The target was the tile size and pipeline depth for each range of M. The yardstick timed the matmul over the shapes in each range in about 20 seconds, about 70× faster, so the agent swept many tilings between true evaluations. The yardstick passed the fidelity test after six true evaluations.

- **Solution.** The kernel finished at 51.4% SOL. As in the mixture-of-experts kernel, the yardstick is built around the shape ranges the target depends on.

SciPy fast Fourier transform.

- **Setting.** A SWE-fficiency task on SciPy's fast Fourier transform code. The expert patch gives $1.71\times$ over the base code. A true evaluation runs the tests and times the workload, about 3.4 minutes, the shortest in the sample.
- **Search.** One of the targets was the code for prime transform lengths. Agents built six yardsticks on this task, a shortened run and small paired benchmarks of the transform, and four passed the fidelity test. One agent found a faster method for prime lengths (Rader's algorithm), another a build that picks newer CPU vector instructions at run time, and a third one merged step for lengths divisible by 13.
- **Solution.** The combine step merged the three edits into one patch, confirmed at $4.45\times$ over the base code, against $3.89\times$ for Claude Code.

F.5 Robustness

We stress DuoDiscover where a fast yardstick that correlates with the true evaluation is hard to find and measure its run-to-run spread across seeds (Table 18).

Table 18: DuoDiscover under missing, unreliable or noisy yardsticks, and its score sensitivity to α and seed, on the 50 SOL-Exec kernels and 20 SWE-fficiency tasks.

Condition	What happens	Confirmed score
No reliable yardstick	search runs on true evaluations alone until the orchestrator moves the agent on	equals Claude Code's search, the method's floor
Unreliable yardstick	rejected before validation, 5 to 9% of true evaluations spent	never below the baseline, search resumes on a rebuilt yardstick
Noisy true evaluation	the noise is measured and folded into the test	no false gain from timing jitter
α from 0.01 to 0.10	the fidelity test is stricter or looser	within a 1.4% band, lowest at 0.01
Three seeds, 50 kernels	each kernel rerun three times	within $\pm$1.12% of the mean, 78% of gains shared, Claude Code $\pm$3.27%
Three seeds, 20 repository tasks	each task rerun three times	within $\pm$1.8% of the mean, 81% of gains shared, Claude Code $\pm$5.21%

No reliable yardstick. Over the 50 SOL-Exec kernels and 20 SWE-fficiency tasks whose run logs we analyzed (Appendix F.2), on average 4 of the 15 targets per kernel run and 2 of the 20 tasks never produce a validated yardstick. On those, the search runs on true evaluations alone until the orchestrator moves the agent on. That is the method's floor.

Unreliable yardstick or noisy evaluation. We plant a misleading yardstick. The fidelity test catches the disagreement within 3 to 5 true evaluations, rejects it, and the search recovers on a rebuilt one. For a noisy true evaluation, the fidelity test measures the run's own noise and folds it into its test, so timing jitter does not raise a false gain.

α and seeds. α is the significance level of each fidelity-test recheck (Appendix A.1). It is 0.05 throughout. Sweeping it from 0.01 to 0.10 keeps the confirmed score within a 1.4% band, lowest at 0.01. A smaller α validates a yardstick only after more of the run's true evaluations, so fewer go to hill-climbing, and a larger α lets more drifted yardsticks through, which the true evaluations catch. Three seeds per task give the spreads in Table 18, summarized in Section 4, and 78% and 81% of confirmed gains recur in at least two of the three runs.