arXiv:2610.36848v1 [cs.AI] 27 Oct 2026

Evolving Agent Teams

Shiyi Cao **Ziming Mao** **Dacheng Li** **Joseph E. Gonzalez** **Ion Stoica**
University of California, Berkeley
{shicao, ziming.mao, dacheng177, jegonzal, istoica}@berkeley.edu

Abstract

The rapid progress of software engineering agents drives interest in applying them to open-ended problems such as GPU kernel optimization. However, we find that existing coding agents converge to local optima on real-world, expert-written kernels (e.g., FLASHINFER). We introduce EVOLVING AGENT TEAMS, a multi-agent framework that escapes such local optima. It features (1) a central supervisor that diagnoses each local optimum and restructures a team of agents; and (2) a persistent file system that comprises a learning document, an evolution history, and per-team artifacts, grounding restructuring with accumulated experience. Under a matched token budget, EVOLVING AGENT TEAMS consistently outperforms strong coding-agent baselines. With an extended token budget, EVOLVING AGENT TEAMS generates a CUDA kernel that reaches $1.2\times$ mean speedup over the production FLASHINFER MLA paged decode kernel from scratch and wins on 36 of 47 production workloads from FLASHINFER-BENCH (0.86–$0.99\times$ on the remaining 11). It also demonstrates superior performance on tasks such as KDA prefill on AMD MI350X ($3.01\times$ over vLLM) and NVIDIA B200 ($3.44\times$ over FlashKDA), MoE expert-offload serving on Apple silicon (30.3 tokens/s decode for Qwen3.8-Flash-Next in MLX 4-bit on a 48 GB M4 Pro), and RoboDojo robot manipulation (62.1 on held-out seeds vs. 55.3 for the best leaderboard entry). Code is available at `https://github.com/caoshiyi/EAT`.

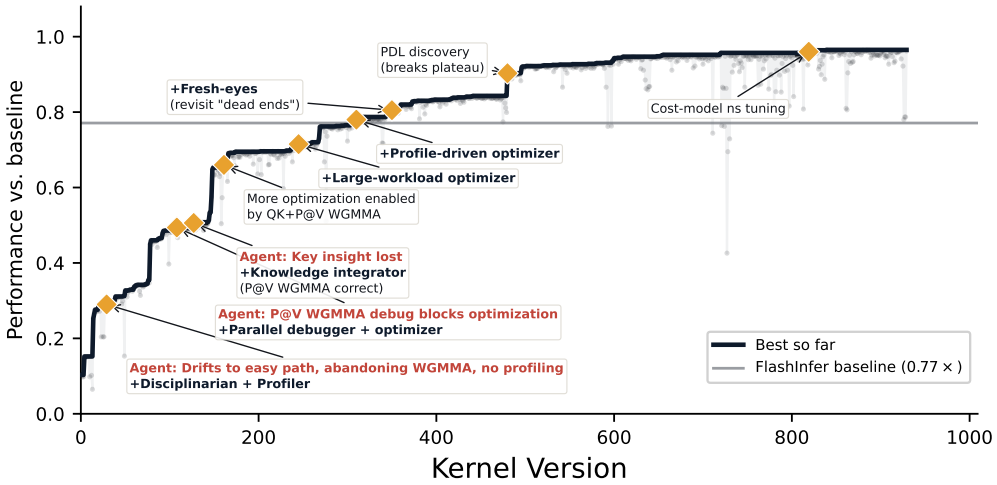


Figure 1: MLA decode kernel long-running optimization trajectory with Claude Opus 4.6. Diamond markers indicate supervisor interventions that restructure the team. Scores are measured against FLASHINFER's `run()` stage directly, excluding `plan()` overhead ($1.0\times$); production FLASHINFER solutions cache `plan()` and only re-execute it when workload metadata changes [11] (solid baseline, $0.77\times$). The kernel ultimately reaches $0.96\times$ the plan-excluding baseline, corresponding to $1.2\times$ the production baseline.

Preprint.

1 Introduction

The rapid progress of coding agents on software engineering tasks [15, 42, 38] has driven interest in applying them to open-ended discovery tasks such as GPU kernel optimization. Here, the goal is not just to produce a correct kernel but to search a large, hardware-dependent design space for high performance. Expert-written kernels such as FLASHINFER [46] and FLASHATTENTION [31] navigate this space across two levels: low-level primitive use (e.g., WGMMA tensor cores, swizzled shared-memory layouts) and high-level architectural design (e.g., dispatch strategies, warp specialization, dataflow). While low-level primitive use can substantially improve performance within a chosen design, high-level architectural choices often determine which hardware bottlenecks remain and thus largely shape the attainable performance ceiling.

Unfortunately, even strong coding agents struggle to write challenging GPU kernels. They often make rapid initial progress, but then converge to local optima far below expert-written baselines. A common failure mode is that agents select design directions that are initially easier to implement but architecturally limited, spend most of their budget tuning within those directions, and fail to revisit harder alternatives that require substantial debugging or architectural rethinking.

For example, when writing the MLA decode kernel used in DeepSeek-V3 inference [10], single-agent runs often abandoned difficult WGMMA-based implementations after repeated correctness failures, and instead optimized simpler but less efficient WMMA-based designs. These designs plateaued far below expert-written baselines: Codex 5.5 xHigh reaches only 28% of expert performance despite a $1K budget and multiple days of effort. Static agent teams show the same limitation (Section 4): a fixed 4-role team built with Claude Code [2] Opus 4.7 1M reaches only 55%. These results suggest that neither stronger single agents nor fixed teams are sufficient for challenging GPU kernel optimization. The process moves through distinct phases, from establishing correctness to debugging hardware primitives, profiling bottlenecks, and redesigning the kernel architecture. Each phase requires different expertise, tools, and coordination patterns, causing fixed agent organizations to become mismatched to the current stage.

We introduce EVOLVING AGENT TEAMS, a multi-agent framework that escapes such local optima by evolving the agent team's organization alongside the optimization process. This evolution is driven by a central supervisor agent, grounded in a persistent file system with accumulated experience. Concretely, the supervisor monitors team progress, reflects on the agent behavior, diagnoses the cause of plateau (e.g., a knowledge gap or an exhausted search), and adapts the team accordingly: spawning specialists for high-effort tasks, introducing profiling roles to get more insights, or running parallel roles for different directions (Section 3.2). The file system contains a learning document of empirical observations, per-team artifacts, and an evolution history.

On the MLA decode kernel, EVOLVING AGENT TEAMS explores over 100 team configurations. The supervisor repeatedly reconfigures the team around the current failure mode, which improves token efficiency and enables successive architectural advances, including performant WGMMA-based matrix multiplication, a PV accumulation strategy, and L2 prefetching. Together, these improvements allow EVOLVING AGENT TEAMS to substantially outperform prior single-agent and static-team approaches (Figure 1).

We evaluate on four challenging CUDA kernel optimization tasks spanning two GPU architectures (Section4): Flash Attention 4 [49] and Gated Delta Net [43] prefill on Blackwell B200, and MLA paged decode [10, 46] and fused GEMM+AllReduce (G+AR) [35] on Hopper H100. EVOLVING AGENT TEAMS outperforms strong coding agent baselines on every task. Notably, on the MLA paged decode task, evaluated on 47 production DeepSeek-V3 workloads from FLASHINFER-BENCH [11], CLAUDE CODE agent plateaus at $0.51\times$ the expert-written FLASHINFER kernel, while our system reaches $0.74\times$ under a matched token budget. With additional budget, the system surpasses the production FLASHINFER baseline, achieving $1.21\times$ mean speedup , faster on 36 of 47 workloads, while the previous best result is $0.61\times$ from K-SEARCH [5].

2 Related Work

Multi-agent LLM systems. CHATDEV [29], METAGPT [13], and MAGIS [36] assign fixed roles (programmer, reviewer, tester) to distinct agents for software engineering. AUTOGEN [40] provides a general multi-agent conversation framework; DYLAN [22] and AGENTVERSE [9] dynamically

adapt the team's role composition during execution, but either select from a hand-crafted pool of role prompts (DyLAN) or regenerate the group from scratch each round without preserving global context across re-compositions (AgentVerse). Our framework differs in two respects: (1) it operates over multi-day horizons where the *nature* of productive work changes qualitatively (from knowledge acquisition to search to verification), requiring changes to agent *roles and mandates*, not merely routing; and (2) the supervisor maintains a persistent global file system across team restructurings.

Self-improving agents and evolutionary search. Reflexion [33] accumulates verbal feedback across trials; Voyager [37] builds a growing skill library. These approaches improve a single agent's behavior within a fixed role, whereas our framework adapts the broader optimization process around a single agent. FunSearch [30], OpenEvolve[32] and AlphaEvolve [24] use population-based evolutionary search with LLM-driven mutation over programs. Our approach differs in that it employs no population and no crossover; instead, a supervisor performs sequential, diagnosis-driven adaptation of the agent team architecture. ADAS[14] uses an LLM with an archive to iteratively propose new agent architectures. Our work also uses an LLM to iteratively adapt the team architecture. However, the purpose of this adaptation is different: rather than searching for a standalone best agent architecture, we evolve the team to better facilitate the optimization process at different stages.

Meta-optimization and text optimization. A growing body of work uses LLMs to optimize text and code artifacts iteratively. OPRO [41] optimizes prompts conditioned on past (solution, score) pairs; TextGrad [48] provides textual feedback on current artifacts; GEPA [1] uses reflective feedback from rollout traces. Meta-Harness [19] optimizes the code surrounding a fixed LLM by giving a coding-agent proposer full filesystem access to prior candidates' source code, scores, and execution traces. Our work shares the insight that richer access to prior experience enables better meta-optimization. However, the object of optimization is different: Meta-Harness optimizes the surrounding harness, whereas our method adapts the search process itself to better discover high-performance kernel implementations.

Code generation and optimization, and inference-time scaling. SWE-agent [42], SkyRL-Agent [4] and SWE-Gym [27] advance agents in software engineering tasks. In GPU kernel optimization, most of the existing LLM-based solutions [39, 18, 21, 50] adopt a fixed human-designed system, where LLMs refine kernels based on compilation or profiling feedback. Some recent works [12] add evolutionary strategies into the system for maintaining and selecting candidates. K-Search [5] decouples planning from implementation via a co-evolving world model but employs a fixed two-role configuration. Inference-time compute scaling [34, 20] improves reasoning within a fixed inference-time search procedure; our framework addresses the complementary problem of adapting the search process itself.

3 Method

3.1 Problem Setup

We consider open-ended program optimization tasks: given a goal description, a codebase, and correctness constraints $\mathcal{C}$, find a solution $s \in \mathcal{P}$ (an executable program) that maximizes a scoring function $\mathcal{F} : \mathcal{P} \to \mathbb{R}$ while satisfying $\mathcal{C}$. Performance is measured against a finite token budget.

Existing LLM-based systems tackling this problem [42, 3, 29, 13, 30, 24, 7] can be characterized by an *agent team configuration* $T = (R, \Pi, M)$ governing three design choices:

- *Roles* R — what each agent is responsible for, e.g., a single generalist [42, 26, 2], fixed programmer/reviewer/tester [29, 13], or an LLM mutation operator [30, 24, 7].
- *Coordination* Π — how agents interact, e.g., no coordination (single agent), a fixed role-to-role pipeline [29, 13], or a select-mutate-evaluate cycle [30, 24, 7].
- *Knowledge management* M — how experience is accumulated and shared, e.g., the filesystem with no explicit management [42, 2], structured message passing enforced by fixed protocol [29, 13], or a candidate archive [30, 24, 32, 7].

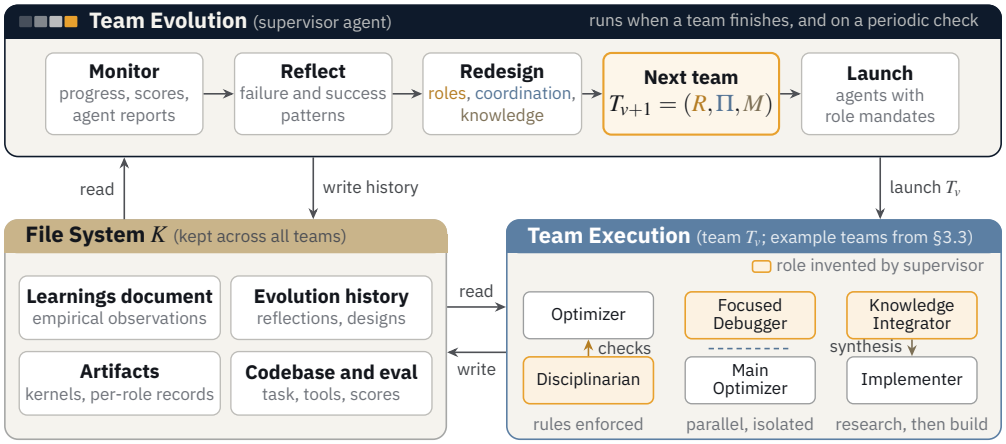


Figure 2: The EVOLVING AGENT TEAMS framework. Team execution runs coding agents on the task through tool use and execution feedback. Team evolution acts as a meta-optimizer: it analyzes the past success and failure patterns, and adapts the team composition, role specialization, coordination protocol, and knowledge management for the current stage of discovery.

In most existing frameworks, all three are human-designed and fixed before the run begins.[1] These systems optimize the solution s but do not adapt T itself, making them prone to stagnation at local optima during long-running open-ended optimization tasks.

3.2 Evolving Agent Teams

EVOLVING AGENT TEAMS co-evolves T alongside s, adapting the optimization system to better facilitate the discovery. It operates at two levels, following the loop shown in Figure 2: at the *team evolution* level, a supervisor agent monitors each team's execution, analyzes success and failure patterns, and designs the next team T_{v+1}. At the *team execution* level, a team of LLM-based coding agents works on the task under the current configuration T_v. A persistent file system K connects the two levels, accumulating all records across team executions and restructurings: a *learnings document* of accumulated empirical observations, per-team *artifacts* (kernels and per-role records), and the *evolution history* (the supervisor's reflections and design decisions from each version), so that each team can build on and improve upon prior work.

Team Evolution. The supervisor is triggered by two signals: the current team finishing its run, or a fixed-interval cron job. Upon either, it checks the team's progress so far, reflects on any observed failure or success patterns (e.g., drifting to easy paths, key insights from previous teams lost, stuck in debugging, and heavily micro-optimizing), and designs the next team along the three axes of T:

- *Roles.* The supervisor invents roles as needed, rather than selecting from a predefined set. A single agent handling implementation, profiling, and research simultaneously divides its limited context across all three, sacrificing depth in each. The supervisor addresses this by creating focused roles such as PROFILING SCIENTIST (designs targeted microbenchmarks to isolate bottlenecks), CHALLENGER (stress-tests claimed performance ceilings), PRIMITIVE STUDY AGENT (studies potentially useful primitives and their usage, running small tests), and KNOWLEDGE INTEGRATOR (synthesizes findings scattered across prior teams into actionable plans). Table 3 in the appendix lists other observed roles.
- *Coordination.* The supervisor decides how agents interact: it can pipeline them (research $\rightarrow$ implement $\rightarrow$ profile), run them in parallel on separate branches, or combine both. For example, the supervisor may spawn one agent solely to debug a blocking issue (forbidden from touching the main codebase) while a second agent continues optimizations on the main branch, so that one agent's failure does not block the other's progress.

[1] Some evolutionary systems adapt search *hyperparameters* during a run (e.g., AdaEvolve [7] modulates exploration intensity and budget allocation); here "fixed" refers specifically to the team configuration $T = (R, \Pi, M)$, the roles, coordination protocol, and knowledge management strategy, which remain static in all listed systems.

- *Knowledge management.* The supervisor evolves how teams manage their experience and learning within the file system K, specifying where each agent should output: e.g., write findings into a dedicated document, append learnings during execution to a team-shared log, or commit code to a specific branch. For example, when early insights were getting lost across teams, the supervisor introduced a cross-agent synthesis document that connected all prior findings into an actionable plan, and mandated future teams to read it.

Team Execution. Each team execution deploys the agents specified by T_v on the task. Each agent is an LLM-based coding agent [42, 3] with shell access, able to read and write files, execute commands, run compilers and profilers, and iterate autonomously on the codebase. Agents receive their role mandates and access to the codebase, evaluation tools, and the file system K. One agent's outputs (committed code, profiling reports, written analyses) become inputs for others according to the coordination plan.

File System. The file system K is a shared filesystem that persists and grows across all team versions. K contains the codebase, evaluation tools, and all records produced by prior teams: the *learnings document*, per-team *artifacts* (kernels and per-role records), and the *evolution history*. This continuity is critical for long-running optimization: each team inherits all prior learnings, failed attempts, and successful strategies, enabling continuous experience accumulation that may open up new optimization directions.

3.3 MLA Decode: A Case Study in Team Evolution

In this section, we present a detailed case study of the MLA decode kernel optimization with EVOLVING AGENT TEAMS.

Evolution trajectory. Figure 1 shows the performance trajectory of the evolving team. The boxes below show representative evolution steps, each summarizing the observed agent behavior, the supervisor's reflection, the team design change, and the outcome.

V1 → V2 Enforcing discipline — **0.15 → 0.065** (first correct WGMMA kernel)

Behavior Solo agent drifted from WGMMA to scalar code under pressure; no profiling was done; multi-variable changes made attribution impossible.
Reflection The agent needs external discipline. Without enforcement, it abandons hard constraints when they become inconvenient.
Design Roles +Disciplinarian (blocks non-WGMMA commits, enforces one-change-per-commit). Coordination Disciplinarian checks after every commit. Knowledge Introduced `LEARNINGS.md` for persistent cross-run discoveries.
Outcome First validated correct WGMMA kernel. Discipline held (one rule-gaming attempt blocked by supervisor).

V5 → V6 Supervisor-orchestrated parallelism — **0.34 → 0.49** (+44%)

Behavior V3–V5 each independently attempted P@V WGMMA integration and failed on the same root cause. Structural changes require isolation debugging then integration — one agent session is too short for both.
Reflection Offering pair-spawning to the agent doesn't work (V5 never used it). The supervisor should orchestrate parallelism directly.
Design Roles Focused Debugger (P@V WGMMA only, forbidden from main kernel) + Main Optimizer (micro-ops only, forbidden from P@V). Coordination Fully parallel, isolated branches.
Outcome Debugger failed P@V conclusively (dead end confirmed). Optimizer made 12 commits for +44%, unblocked by the parallel pattern.

V7 → V8 Cross-team knowledge synthesis — **0.50 → 0.51** (knowledge breakthrough)

Behavior V4 had recommended "use WGMMA RS with P in registers" but V5–V7 all ignored this because the recommendation was in V4's report, not in the shared learnings.
Reflection Lost knowledge between teams is the biggest structural weakness. Per-experiment facts are captured but cross-experiment strategic recommendations are not.
Design Roles +Knowledge Integrator. Knowledge Introduced `SYNTHESIS.md` connecting all agents' findings into an actionable plan. Mandated all future teams to read it.
Outcome RS standalone test passed (first P@V WGMMA success ever). Directly enabled V9's integration milestone.

V9 → V10 Re-profiling after a structural change **0.50 → 0.66** (+32%)

Behavior V9 integrated P@V WGMMA RS, changing the kernel's bottleneck structure completely.
Reflection With both GEMMs on WGMMA, the kernel needs fresh profiling — all prior bottleneck analysis is stale.
Design Roles Standard Profile-Driven Optimizer on the new architecture. Coordination Solo, profile-first discipline.
Knowledge Inherited full K including V9's integration learnings.
Outcome 9 commits, +32%. Split-K retune alone gave +4.1. Structural change unlocked entirely new optimization surface.

This trajectory illustrates the strength of EVOLVING AGENT TEAMS in responding to failures that no fixed design can foresee: the supervisor observes a concrete dysfunction, diagnoses its structural cause, and restructures roles, coordination, or knowledge management to address it.

Section 4 evaluates how these adaptations translate into quantitative gains across multiple tasks.

4 Experiments

Table 1: Settings of Main Evaluation Tasks.

Task	GPU	Language	Workloads	Baseline	Seed
FA4	B200	CuTe DSL	24	FlashAttention-4 [49]	0.17× CuTe DSL kernel
GDN [43]	B200	CUDA	100	FlashInfer [46]	PyTorch reference
MLA [10]	H100	CUDA	47	FlashInfer [46]	PyTorch reference
G+AR	H100	CUDA	5	CUTLASS [25]+NCCL	PyTorch reference

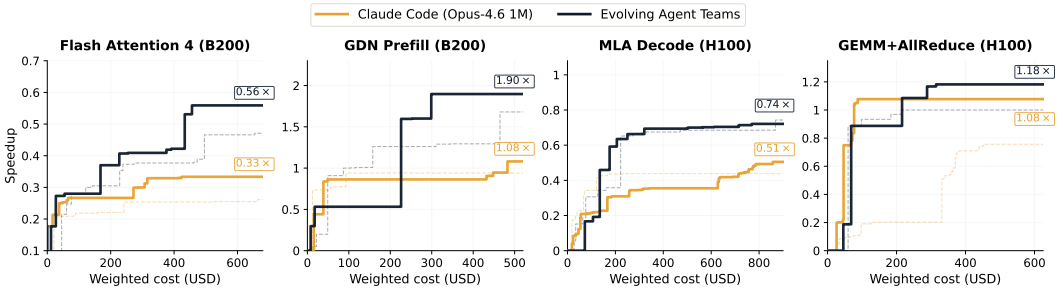
4.1 Experimental Setup

Tasks. We evaluate on four GPU kernel optimization tasks spanning two architectures (Table 1). *FA4* targets the attention forward pass [49], written in CuTe DSL [6]; the challenge is co-designing algorithm and kernel pipelining for Blackwell's asymmetric hardware scaling, where tensor core throughput doubles but shared memory bandwidth and non-matmul units do not. *GDN* computes a prefill pass for Gated DeltaNet [43], a linear RNN with a matrix-valued hidden state; the challenge is parallelizing the inherently sequential recurrence over long sequences via a chunkwise algorithm. *MLA* implements paged decode for DeepSeek-V3's Multi-head Latent Attention [10]; the challenge is overlapping CUDA core and tensor core work efficiently under extreme register pressure due to MLA's large head dimension. *G+AR* fuses GEMM with inter-GPU AllReduce into a single kernel, overlapping computation and communication; efficient fusing requires writing a custom WGMMA GEMM kernel with tile-level scheduling to overlap reduction with compute.

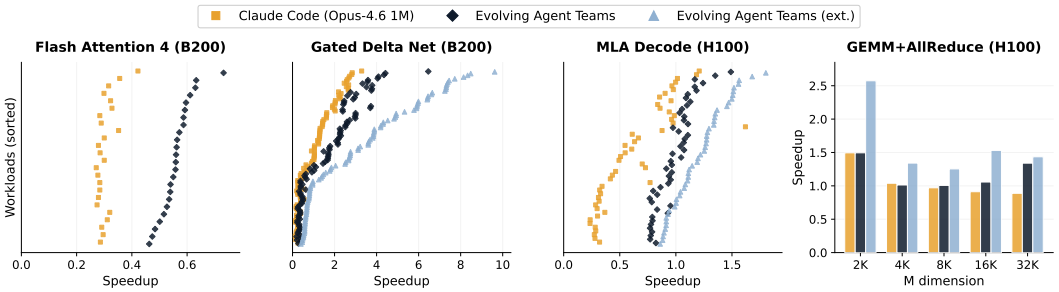
Metrics. For each task, we report the speedup relative to the production baseline: the geometric mean of $t_{\text{base}}/t_{\text{kernel}}$ across a set of workloads, where t_{base} and t_{kernel} are the baseline and optimized kernel latencies respectively. Values above $1.0\times$ indicate the optimized kernel is faster than the baseline; values below $1.0\times$ indicate it is slower. All settings are run twice; we report the best result from each pair. We acknowledge this limits statistical conclusions and discuss implications in Section 5.

Workloads. GDN prefill and MLA workloads are drawn from FLASHINFER-BENCH [11], which provides representative production serving shapes. FA4 workloads use the same shapes as used in the performance report in the FlashAttention-4 paper [49]. GEMM+AR (G+AR) workloads are taken from the ThunderKittens benchmark suite [35].

Baselines. We compare against three baseline systems. (1) *Claude Code* [2]: CLAUDE CODE powered by Claude Opus 4.6 with 1M context window, a state-of-the-art coding agent with file editing, shell access, and compiler invocation that autonomously writes, compiles, profiles, and iterates on kernel code. It receives the same task description, reference kernel, and test harness as the evolving team. Evaluated on all four tasks in the budget-controlled experiment (Section 4.2). (2) *Fixed four-role team*: a static team of four common high-value roles for GPU kernel optimization (Profiler, Hardware Researcher, Planner, Implementer), each powered by Claude Code with the same Opus 4.7 1M model, but with no supervisor and no team restructuring. Evaluated on MLA in the ablation study (Section 4.5). (3) *Codex* [26]: a strong coding agent powered by GPT-5.5 xhigh with 258K context. Evaluated on MLA in the ablation study (Section 4.5). (4) *Kernel-specific systems*: K-SEARCH [5] and CUDAFORGE [50], run from their publicly released implementations with



(a) Budget-controlled progress curves.



(b) Per-workload speedups.

Figure 3: Performance comparison across four kernel tasks. (a) Solid lines show the best of two runs; dashed lines show the second run. Cost computed under published per-token pricing (Claude Opus 4.6 1M). (b) Each dot is one workload. "Evolving Teams (ext.)" denotes longer runs beyond the budget-controlled setting; EVOLVING AGENT TEAMS shifts the distribution rightward on all tasks.

Claude Opus 4.6. Evaluated on MLA (Section 4.6). For Claude Code, K-SEARCH, CUDAFORGE, and EVOLVING AGENT TEAMS, we run each setting twice and report the best result; the fixed-team and Codex ablations are single runs.

Implementation. EVOLVING AGENT TEAMS is built on top of CLAUDE CODE [2] agents powered by Claude Opus 4.6 with 1M context. The supervisor is itself a Claude Code agent that orchestrates team evolution by spawning and monitoring worker agents. There are no predefined initial roles or team configurations; the supervisor designs all roles, coordination plans, and knowledge management strategies from scratch based on the task description.

4.2 Budget-Controlled Progress Curves

Figure 3a plots performance vs. cumulative weighted cost, while Figure 3b shows per-workload speedup distributions. Cost is computed from per-token API pricing: for Claude Opus 4.6 (1M context), we use \$0.50 / \$6.25 / \$5.00 / \$25.00 per million tokens for cache-read / cache-create / input / output respectively; for GPT-5.5 xhigh (Codex ablation), \$0.50 / \$5.00 / \$30.00 per million tokens for cached-input / input / output.

EVOLVING AGENT TEAMS outperforms Claude Code: Claude Code curves flatten early on FA4 and GDN Prefill, while EVOLVING AGENT TEAMS continues improving with more budget spent. On MLA decode, Claude Code plateaus around 0.44–$0.51\times$: it abandons the difficult WGMMA tensor core implementation after repeated failures at correctness and instead uses the WMMA API, which limits the tensor core utilization compared to WGMMA. The evolving team reaches $0.75\times$ under the budget-controlled setting ($1.21\times$ with additional budget, as shown in Figure 1).

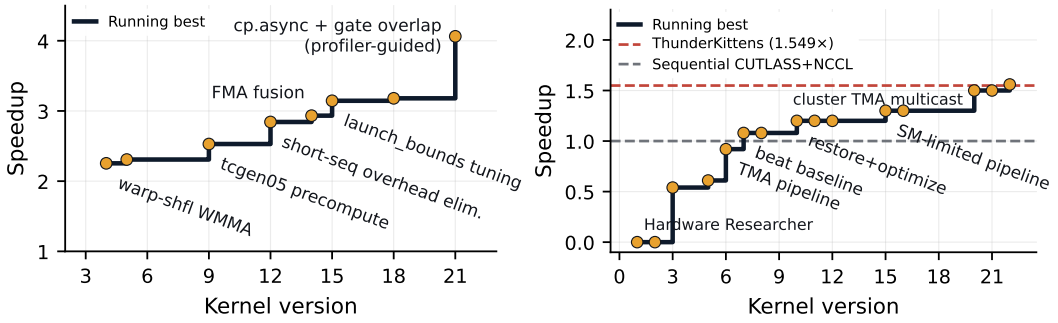
The GDN prefill panel shows particularly high speedups ($> 2\times$) because the 100-workload benchmark includes many small input sizes where FlashInfer's planning and preprocessing overhead dominates end-to-end latency, even with `plan()` cached across calls. Our generated kernel requires no planning stage, so on these small workloads it achieves disproportionately large speedups. Under the budget-controlled setting, the best EVOLVING AGENT TEAMS run has a worst workload of $0.19\times$ with 43 of 100 below the baseline, compared to $0.13\times$ and 49 of 100 for the best Claude Code run. With

additional budget (Section 4.3), the worst workload improves to $0.40\times$ with 37 of 100 below the baseline.

On GEMM+AllReduce, the single-agent run produces a cuBLAS-based kernel, yielding a mean speedup of $1.08\times$ compared to the cuBLAS + NCCL baseline. The budget-controlled evolving team similarly converges on a cuBLAS+NCCL pipeline, achieving $1.18\times$ with stronger gains at larger matrix sizes (32K: $1.34\times$). With additional budget (Section 4.3), the team discovers a fused WGMMA+NVLS kernel that bypasses cuBLAS entirely and leverages NVLink multicast for the all-reduce, reaching $1.49\times$ overall and $2.57\times$ on the smallest workload where communication latency dominates.

Across all four tasks, the best EVOLVING AGENT TEAMS kernel matches or exceeds the best Claude Code kernel on nearly every workload (Figure 3b), with a single exception on MLA decode. This broad coverage suggests the system does not overfit to a narrow regime; we attribute it to the supervisor's tendency to evolve roles such as WORKLOAD ANALYST that explicitly diagnose per-workload regressions and steer optimization toward balanced performance (Appendix A).

4.3 Scaling with Additional Budget



(a) GDN Prefill (B200).

(b) GEMM+AllReduce (H100).

Figure 4: Evolution trajectories for GDN and G+AR with additional budget beyond the controlled comparison. Each point is one team version. The step line tracks the running best.

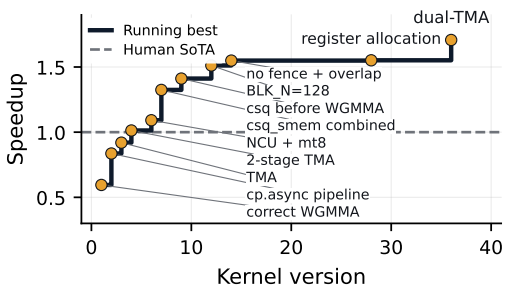
Beyond the budget-controlled comparison, we ran EVOLVING AGENT TEAMS with additional token budget on GDN and G+AR (Figure 4).

GDN Prefill (Figure 4a) continues to improve from a $2.2\times$ kernel to $4.06\times$ mean speedup across 21 team versions. With the additional budget, the worst-case workload improves $0.40\times$, and the number of workloads below the baseline drops from 43 to 37 of 100, indicating that extended optimization disproportionately benefits the hardest workloads where the initial kernel architecture was less suited.

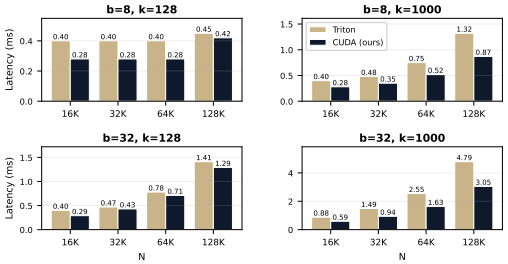
G+AR (Figure 4b) reaches $1.56\times$ over the cuBLAS+NCCL baseline, comparable to ThunderKittens [35], which is at $1.54\times$ over cuBLAS+NCCL. The trajectory illustrates three distinct phases that require qualitatively different capabilities. Versions 1–5 focus on correctness: the team debugs WGMMA register layouts and IPC cache coherence for multi-GPU communication, producing a working but slow two-phase kernel ($0.54\times$). Versions 6–10 introduce TMA asynchronous loads and a warp-specialized pipeline that first exceeds the baseline ($1.08\times$). Versions 15–22 shift to a fundamentally different architecture: the team replaces cuBLAS with a custom WGMMA kernel fused with NVLink multicast for the all-reduce, culminating in cluster-level TMA multicast ($1.56\times$).

4.4 Optimizing an Already-Optimized Implementation

The four tasks above start from naïve reference kernels. To test whether EVOLVING AGENT TEAMS can also improve code that has already been carefully optimized by domain experts, we apply it to *Flash-KMeans* [45], a recent GPU-optimized exact K-Means implementation. Flash-KMeans fuses distance computation with online argmin (FlashAssign) to bypass HBM materialization of the $N\times K$ distance matrix and introduces a sort-inverse update that transforms high-contention atomic scatters into segment-level localized reductions, achieving up to $17.9\times$ end-to-end speedup over



(a) Evolution trajectory.



(b) Benchmark grid: latency comparison (ours vs. Triton).

Figure 5: Flash-KMeans (H100). (a) Evolution trajectory; each point is one kernel version. (b) Per-iteration latency comparison across the paper's benchmark grid (D=128, FP16). Our CUDA WGMMA kernel achieves up to $1.5\times$ lower latency on compute-bound configurations (high K), while matching Triton on memory-bound regimes.

Table 2: Comparison with kernel-specific optimization systems on MLA decode (H100) under a matched $800 model-inference budget. All kernels are re-evaluated on the same H100 across the 47 production workloads and pass all 47 correctness tests. Speedup is relative to production FLASHINFER.

System	Run 1	Run 2	Best-kernel MMA path
CUDAFORGE [50]	$0.448\times$	$0.283\times$	scalar/SIMT (0/2 WGMMA)
K-SEARCH [5]	$0.602\times$	$0.598\times$	WMMA (0/2 WGMMA)
EVOLVING AGENT TEAMS	$\mathbf{0.721}\times$	$\mathbf{0.685}\times$	WGMMA (2/2)

prior GPU baselines on H200. The released implementation uses hand-tuned Triton kernels. We task the EVOLVING AGENT TEAMS with rewriting the assignment kernel in CUDA, starting from the Triton source as a reference rather than a naïve PyTorch baseline. As shown in Figure 5, the system produces a CUDA WGMMA kernel that reaches $1.71\times$ speedup over the Triton kernel on the default workload, and achieves up to $1.5\times$ lower latency on compute-bound configurations (high K) across the full benchmark grid (Figure 5b).

4.5 Ablation Studies

Figure 6 compares EVOLVING AGENT TEAMS against two ablation baselines on MLA decode. The *fixed four-role team* (Profiler, Hardware Researcher, Planner, Implementer) reaches $0.31\times$: role specialization helps early on, but the static configuration cannot adapt when the optimization landscape shifts, e.g., it lacks a KNOWLEDGE INTEGRATOR to prevent cross-agents information asymmetry or a CHALLENGER to break premature ceiling declarations (examples provided in Appendix B). *Codex* (GPT-5.5 xhigh, 258K context) reaches $0.28\times$, confirming that the gains of EVOLVING AGENT TEAMS ($0.74\times$) are not solely due to model choice. The $0.31\times \rightarrow 0.74\times$ gap between the fixed team and the evolving team isolates the value of team evolution: both use the same model and multi-agent structure, but only the evolving system can restructure roles and coordination in response to diagnosed failure modes.

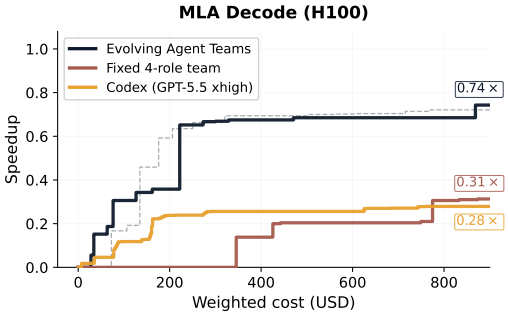


Figure 6: Ablation on MLA decode.

4.6 Comparison with Kernel-Specific Systems

We further compare against two systems designed specifically for GPU kernel optimization, K-SEARCH [5] and CUDAFORGE [50], on MLA decode. All systems use Claude Opus 4.6 with medium thinking effort and the same $800 model-inference budget, and each is run twice. As shown

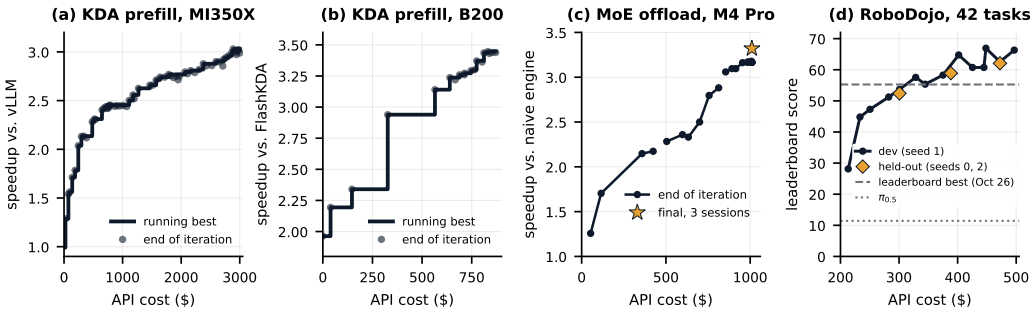


Figure 7: Full runs on four further settings: harness evaluation at the end of each iteration vs. API cost. (a) MI350X, from a port of vLLM's kernels. (b) B200, from a port of FLASHINFER's KDA kernels ($1.96\times$). (c) MoE offload from the naive engine; star: final engine on the recorded sessions. (d) RoboDojo, 42 tasks: dev (seed 1) and held-out (seeds 0, 2) scores, with leaderboard references as of Oct. 26, 2026.

in Table 2, EVOLVING AGENT TEAMS outperforms both systems in each run. The gap is explained by the implementation family each system converges to: EVOLVING AGENT TEAMS produces WGMMA-based kernels in both runs, whereas K-SEARCH converges to WMMA and CUDAFORGE to scalar/SIMT code in both runs. As with Claude Code (Section 4.2), without restructuring the search they settle on an easier-to-get-correct implementation.

4.7 Surpassing State-of-the-Art Systems

Finally, we ask whether full runs can surpass the best existing systems on new hardware and in new domains, with the same supervisor prompt, no predefined roles, and newer models (Opus/Sonnet 5.5; single runs; details in Appendix C, solo ablations in Appendix D). On Kimi Delta Attention (KDA) prefill [16], the team's kernel is $3.01\times$ faster than vLLM's hand-written production HIP kernels [17] on AMD MI350X, and $3.44\times$ faster than FlashKDA [23], a fused CUTLASS implementation, on B200 (Figure 7a,b). On B300, the final kernel scores $3.54\times$ in the same harness, against $3.02\times$ for the agent-written KDA^2 kernels [51] (Appendix C). It makes a Metal port of FreeToken [44], which runs Qwen3.8-Flash-Next in MLX 4-bit on a 48 GB Apple M4 Pro by streaming experts from SSD, serve recorded coding-agent sessions $3.32\times$ faster (Figure 7c). On RoboDojo [8] it writes a code policy for 42 simulated dual-arm manipulation tasks that scores 62.1 on held-out seeds, above the best leaderboard entry (55.3 as of Oct. 26, 2026) and the $\pi_{0.5}$ VLA [28] alone (11.4) (Figure 7d). The B200 and RoboDojo runs had no web access, and the MoE run had web tools disabled.

5 Conclusion

Strong coding agents make rapid initial progress on GPU kernel optimization but converge to local optima well below expert-written baselines. EVOLVING AGENT TEAMS escapes these local optima by evolving the team's organization alongside the search: a central supervisor diagnoses each plateau and restructures the team's roles, grounded in a persistent file system. Across four CUDA kernel optimization tasks on Hopper and Blackwell GPUs, it consistently outperforms strong coding-agent baselines under a matched token budget, and with additional budget it surpasses the production FLASHINFER MLA decode kernel at $1.21\times$ mean speedup across 47 workloads. It also demonstrates superior performance on tasks such as KDA prefill on AMD and NVIDIA GPUs, MoE expert-offload serving on Apple silicon, and robot manipulation.

Limitations. We run each main setting twice and each setting in Section 4.7 once, which limits statistical comparisons; a cost-vs-performance Pareto analysis and a controlled study of noisier domains remain future work.

Acknowledgments and Disclosure of Funding

This work was done at the Sky Computing Lab. We gratefully acknowledge compute support from SemiAnalysis, Amazon, AMD, and NVIDIA. Shiyi Cao and Dacheng Li are supported by the Amazon Fellowship.

References

[1] L. A. Agrawal, S. Tan, D. Soylu, N. Ziems, R. Khare, K. Opsahl-Ong, A. Singhvi, H. Shandilya, M. J. Ryan, M. Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

[2] Anthropic. Claude code. `https://docs.anthropic.com/en/docs/claude-code`, 2025.

[3] N. D. Bui. Building effective ai coding agents for the terminal: Scaffolding, harness, context engineering, and lessons learned. *arXiv preprint arXiv:2603.05344*, 2026.

[4] S. Cao, D. Li, F. Zhao, S. Yuan, S. R. Hegde, C. Chen, C. Ruan, T. Griggs, S. Liu, E. Tang, et al. Skyrl-agent: Efficient rl training for multi-turn llm agent. *arXiv preprint arXiv:2511.16108*, 2025.

[5] S. Cao, Z. Mao, J. E. Gonzalez, and I. Stoica. K-search: Llm kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026.

[6] C. Cecka. Cute layout representation and algebra. *arXiv preprint arXiv:2603.02298*, 2026.

[7] M. Cemri, S. Agrawal, A. Gupta, S. Liu, A. Cheng, Q. Mang, A. Naren, L. E. Erdogan, K. Sen, M. Zaharia, et al. Adaevolve: Adaptive llm driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

[8] T. Chen, Y. Chen, Z. Li, J. Tang, K. Su, H. Lu, W. Wan, B. Chen, S. Liu, H. Yan, et al. RoboDojo: A unified sim-and-real benchmark for comprehensive evaluation of generalist robot manipulation policies. *arXiv preprint arXiv:2607.04434*, 2026.

[9] W. Chen, Y. Su, J. Zuo, C. Yang, C. Yuan, C.-M. Chan, H. Yu, Y. Lu, Y.-H. Hung, C. Qian, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*, 2023.

[10] DeepSeek-AI. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[11] FlashInfer Team. FlashInfer-Bench: MLSys 2026 AI kernel generation contest. `https://bench.flashinfer.ai/`, 2026. Accessed: 2026-05-05.

[12] P. Guo, C. Zhu, S. Chen, F. Liu, X. Lin, Z. Lu, and Q. Zhang. Evoengineer: Mastering automated cuda kernel code evolution with large language models. *arXiv preprint arXiv:2510.03760*, 2025.

[13] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024.

[14] S. Hu, C. Lu, and J. Clune. Automated design of agentic systems. In *arXiv preprint arXiv:2408.08435*, 2024.

[15] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[16] Kimi Team. Kimi Linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692*, 2025.

[17] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pages 611–626, 2023.

[18] R. T. Lange, Q. Sun, A. Prasad, M. Faldor, Y. Tang, and D. Ha. Towards robust agentic cuda kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025.

[19] Y. Lee, R. Nair, Q. Zhang, K. Lee, O. Khattab, and C. Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

[20] D. Li, S. Cao, C. Cao, X. Li, S. Tan, K. Keutzer, J. Xing, J. E. Gonzalez, and I. Stoica. S*: Test time scaling for code generation. *arXiv preprint arXiv:2502.14382*, 1(2), 2025.

[21] G. Liao, H. Qin, Y. Wang, A. Golden, M. Kuchnik, Y. Yetim, J. J. Ang, C. Fu, Y. He, S. Hsia, et al. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta. *arXiv preprint arXiv:2512.23236*, 2025.

[22] Z. Liu, Y. Zhang, P. Li, Y. Liu, and D. Yang. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization. *arXiv preprint arXiv:2310.02170*, 2023.

[23] MoonshotAI. FlashKDA: High-performance Kimi Delta Attention kernels. `https://github.com/MoonshotAI/FlashKDA`, 2026.

[24] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. Ruiz, A. Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[25] NVIDIA. Cuda templates and python dsls for high-performance linear algebra. `https://github.com/NVIDIA/cutlass`, 2023.

[26] OpenAI. Codex. `https://openai.com/index/introducing-codex/`, 2025.

[27] J. Pan, X. Wang, G. Neubig, N. Jaitly, H. Ji, A. Suhr, and Y. Zhang. Training software engineering agents and verifiers with swe-gym. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

[28] Physical Intelligence, K. Black, N. Brown, J. Darpinian, K. Dhabalia, D. Driess, A. Esmail, M. Equi, C. Finn, N. Fusai, et al. $\pi_{0.5}$: a vision-language-action model with open-world generalization. *arXiv preprint arXiv:2504.16054*, 2025.

[29] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

[30] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. R. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, P. Kohli, and A. Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

[31] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. In *Advances in Neural Information Processing Systems*, 2024.

[32] A. Sharma. Openevolve: An open-source evolutionary coding agent. `https://github.com/algorithmicsuperintelligence/openevolve`, 2025. GitHub repository. Accessed: 2026-05-07.

[33] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36: 8634–8652, 2023.

[34] C. Snell, J. Lee, K. Xu, and A. Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

[35] B. F. Spector, S. Arora, A. Singhal, D. Y. Fu, and C. Ré. Thunderkittens: Simple, fast, and adorable ai kernels. *arXiv preprint arXiv:2410.20399*, 2024.

[36] W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems*, 37: 51963–51993, 2024.

[37] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[38] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

[39] A. Wei, T. Sun, Y. Seenichamy, H. Song, A. Ouyang, A. Mirhoseini, K. Wang, and A. Aiken. Astra: A multi-agent system for gpu kernel performance optimization. *arXiv preprint arXiv:2509.07506*, 2025.

[40] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2024.

[41] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2023.

[42] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[43] S. Yang, J. Kautz, and A. Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In *The Thirteenth International Conference on Learning Representations*, 2025.

[44] S. Yang, X. Fan, M. Pan, H. Xi, Z. Wang, S. Sun, K. Keutzer, S. Han, M. Zaharia, C. Xu, and I. Stoica. FreeToken: Efficient edge-native MoE serving with bandwidth-adaptive execution. *arXiv preprint arXiv:2608.16157*, 2026.

[45] S. Yang, H. Xi, Y. Zhao, M. Li, X. Fan, J. Zhang, H. Cai, Y. Lin, X. Li, K. Keutzer, et al. Flash-kmeans: Fast and memory-efficient exact k-means. *arXiv preprint arXiv:2603.09229*, 2026.

[46] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *Proceedings of Machine Learning and Systems*, 7, 2025.

[47] Z. Ye, Y. Huang, H. Jin, B. Hou, J. Shao, Z. Yu, J. Chen, M. Cowan, S. Cao, S. Xing, H. Chen, V. Grover, T. Chen, and L. Ceze. CAKE: Compiler–agent co-design for frontier kernel evolution. *arXiv preprint arXiv:2608.12629*, 2026.

[48] M. Yuksekgonul, F. Bianchi, J. Boen, S. Liu, Z. Huang, C. Guestrin, and J. Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

[49] T. Zadouri, M. Hoehnerbach, J. Shah, T. Liu, V. Thakkar, and T. Dao. Flashattention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. *arXiv preprint arXiv:2603.05451*, 2026.

[50] Z. Zhang, R. Wang, S. Li, Y. Luo, M. Hong, and C. Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025.

[51] D. Zou, Y. Dong, H. Jin, J. Guo, Y. Cui, A. Huang, Z. Ye, J. Shao, C. Li, Z. Zhang, S. Liu, S. Bian, and L. Zhu. KDA^2: Kernel design agents (KDA) optimize kimi delta attention (KDA). `https://nvlabs.github.io/kda/blog/2026-09-27-kda-for-kda/`, 2026. NVIDIA, blog post, Sep. 27, 2026.

A Observed Roles Invented by the Supervisor

Table 3 lists the roles invented by the supervisor across our experiments. None of these roles were predefined; the supervisor created each in response to a specific diagnosed need.

Table 3: Roles invented by the supervisor, grouped by category. All roles emerged from the supervisor's reflection on team performance.

Role	Description
Implementation & Optimization	
Solo Optimizer	Full-stack agent with complete autonomy over design, implementation, and profiling
Focused Optimizer	Targets a specific subsystem or bottleneck (e.g., Large-Workload, Memory, L1 Cache, Barrier-Free Reduce, Stage Pipelining)
Focused Debugger	Scoped to a single standalone test file; forbidden from touching the main kernel
Focused Integrator	Integrates a specific validated component with corrected context
Fresh-Eyes Agent	Retries previously dismissed approaches without preconceptions
Analysis & Profiling	
Profile-Driven Optimizer	Mandated to run `ncu` before every change and cite profiling data
Workload Analyst	Per-workload performance breakdown to identify the biggest gap
Compiler Specialist	Investigates compiler behavior (`ptxas`, SASS, register allocation)
Phase Profiler	Instruments kernel with cycle counters to measure per-phase costs
Stall Localizer	`ncu -set source` per-PC profiling to attribute stalls to code lines
Roofline Analyst	Computes mathematical upper bounds on achievable performance
Knowledge & Research	
Knowledge Integrator	Synthesizes findings from prior sessions into a single actionable document
Dead-End Revisitor	Re-evaluates all revisit log entries against current kernel state
Research Optimizer	Web search to study external implementations for structural insights
Hardware Cataloguer	Builds and maintains a catalog of available hardware primitives
Primitive Study Agent	Studies potentially useful primitives and runs small validation tests
Quality Assurance	
Disciplinarian	Enforces constraints (WGMMA, one-change-per-commit, profiling-first)
Contrarian / Challenger	Questions assumptions and stress-tests claimed performance ceilings
Auditor	Cross-checks proposed designs against known dead ends and hardware constraints
Architecture & Design	
Architect	Proposes new kernel designs from a blank slate based on hardware constraints
Schedule Explorer	Cost model to systematically enumerate and rank parallelization strategies
DAG Topology Explorer	Analyzes the kernel's dataflow graph to identify overlapping opportunities

B Case Study: Why the Fixed Four-Role Baseline Starts Slowly

Figure 6 shows the fixed four-role baseline (Planner, Hardware Researcher, Implementer, and Profiler; all Claude Opus 4.7, coordinating through a shared filesystem and a SendMessage primitive) reaching its first correctness-passing commit substantially later than the single agent on MLA decode. Most of this delay is traceable to a single class of bug: a low-level SMEM-layout mismatch in which

WGMMA on SM90 requires 8×64 atom-tiled shared memory, but the Implementer wrote row-major tiles, silently corrupting the QK output.

The information needed to identify the bug is present in the team but partitioned across agents: the Hardware Researcher holds the layout requirement, the Implementer holds the actual SMEM declaration, and both agree on the descriptor numerics, leading them to localize the bug elsewhere. The team iterates through incorrect hypotheses without any agent reading the Implementer's source to inspect the layout, despite all having Read access. A single agent holding both specification and code in its context would identify the discrepancy on direct inspection.

The failure is structural rather than capability-driven. Each role's charter implicitly scopes its outputs (the Hardware Researcher operates as a recipe expert, not an auditor); spawn prompts specify reactive behavior, locking roles into question-and-answer loops that preclude proactive code inspection; and the SendMessage primitive carries prose, encouraging abstract agreement ("`LBO=16`, confirm?") over byte-level cross-checking. Fixed-role multi-agent systems can therefore stall on debugging tasks that cross role boundaries. EVOLVING AGENT TEAMS addresses this class of failure: the supervisor diagnoses the stall as a *knowledge gap* and introduces a Knowledge Integrator (Section 3.3) that mandates cross-agent artifact reads.

C Details of the Additional Settings

All four runs use the same supervisor prompt and harness as the main experiments, with no predefined roles. Each is a single run, so we report them as case studies rather than controlled comparisons; the matched-budget solo ablations are in Appendix D. Costs are booked agent and supervisor API spend at published per-token prices.

KDA prefill on AMD MI350X. Kimi Delta Attention [16] is a gated delta rule with a per-key-channel forget gate and a matrix-valued fp32 state; the kernel computes the chunked prefill from the raw projections (normalization, gates and recurrence). The task uses one MI350X (CDNA4) and the tensor-parallel shape of a production deployment ($H{=}12$ heads per GPU, $K{=}V{=}128$). The baseline is vLLM's hand-written fused HIP prefill kernels used in production [17]; the starting kernel is a port of them (score 0.99). The score is the arithmetic mean of baseline-over-candidate latency across 10 variable-length workloads (single sequences of 1K–64K tokens, batches from $4{\times}4$K to $64{\times}128$, and one mixed batch), with the cache flushed before each timed run. Correctness requires relative L_2 error $\leq 2{\times}10^{-2}$ on outputs and final states; the team itself found that the evaluation's fast-decaying gates let state-handoff bugs pass, and added a slow-decay check that every subsequent kernel had to pass, along with a held-out shape set that is checked but not scored. The run used Claude Opus 5.5 (medium effort) for all agents, lasted 73 iterations (186 hours), and booked $3.0K. The final kernel scores $3.01\times$ (mean of five re-measurements; per-workload 2.09–$4.63\times$) and is $7.3\times$ faster than the Triton kernel from Flash Linear Attention. Its main structural changes are a "relay" that fuses the prologue with the recurrence and passes the state between workgroups through the shared L2 cache, bf16/fp16 state handoffs and non-temporal loads, fp16 matrix operands ($+5.5\%$ and more accurate than bf16), and a "twin route" that feeds one 64-token recurrence step from two 32-token prologues.

KDA prefill on B200. The task is the same operator on B200 (`sm_100a`), scored as the geomean speedup over FlashKDA [23], a fused CUTLASS implementation, in a clean room with no web access. The team could read FlashKDA, the Triton KDA kernels of Flash Linear Attention, the CUTLASS/CuTe examples, FlashAttention-4 [49], and a read-only snapshot of FLASHINFER [46] at commit `a9e03bf2` (Aug. 13, 2026), which includes the SM100 KDA prefill kernels that CAKE [47] contributed to FLASHINFER; the team ported these in the first iteration ($1.96\times$). The score covers six workloads (8,192 tokens, $H \in \{64, 96\}$, fixed and variable-length layouts). Iterations 1–5 used Claude Opus 5.5 and later iterations Claude Sonnet 5.5. *Comparison with KDA*^{2}. KDA^2 [51] reports $2.96\times$ over FlashKDA on B300 on the same six workloads. We recompiled the team's final kernel for B300 (`sm_103a`, no code change) and scored it and KDA^2's released kernels with the same harness on the same B300: $3.54\times$ for ours, against $3.02\times$ (TIRx), $2.98\times$ (PTX), and $2.85\times$ (CuTe-DSL) for KDA^2. On B200, the same harness gives $3.44\times$, against $2.91\times$ (TIRx) and $2.74\times$ (CuTe-DSL); the PTX kernels run only on B300. Because the run starts from a port of an existing kernel, it measures improvement over a strong start rather than from scratch.

MoE expert-offload serving on Apple silicon. The task is to make a Metal port of the FreeToken MoE serving system [44] serve Qwen3.8-Flash-Next at 4 bits (about 79 GB of weights plus a 32 GB

per-layer embedding table; 512 routed experts per layer, top-10 plus a shared expert) as fast as possible on a 48 GB M4 Pro Mac, streaming experts that do not fit in memory from SSD through an offload cache. The model and math must stay the same: no expert skipping or pruning, no lower top-k, no fewer bits. Gates check top-1 agreement with the naive engine's greedy predictions, KL divergence, peak memory ($\leq$ 37 GiB), swap growth, and load time, and a candidate that fails any gate scores 0. The score is the mean over workload terms of baseline time over candidate time; the workload replays sessions recorded from a real coding-agent loop, teacher-forced from a cold page cache, plus a batch-1 decode benchmark. All agents used Claude Opus 5.5, with web search and fetch tools disabled. The supervisor resized the team from one agent to three and finally to two, and re-cut its roles six times: by file type (I/O vs. compute), by bottleneck (speculation, cache, offload), by time slice (decode vs. prefill), and by measured gap (GPU time per token). Main changes include a victim cache for evicted experts, compact GPU expert slots with shared palette indices, (scale, bias) palette compression that cuts expert reads by 7%, and admission rather than eviction as the residency lever, after a cache role found that most decode misses had already been read during the preceding prefill. The final engine decodes at 27.3 tokens/s on the batch-1 decode benchmark, against 13.4 for the naive engine. In Figure 7c, iterations 3–6 have no valid evaluation, and each iteration's cost is its team cost scaled so that the total matches the booked $1.0K, which includes the supervisor.

RoboDojo. RoboDojo [8] has 42 dual-arm manipulation tasks in Isaac Sim, grouped into five dimensions (generalization, memory, precision, long-horizon, open-ended); the leaderboard score is the mean of the five dimension means of per-task partial-credit scores (0–100). The team edits a Python controller package with per-task skill modules. At run time the policy may call the frozen $\pi_{0.5}$ checkpoint [28] and a runtime LLM (Claude, at most 20 calls per episode); inputs are three RGB cameras, joint states, and the instruction, and no model training is allowed. The agents' outbound network is limited to the Anthropic API and the evaluation fleet, and other leaderboard entries' policy code is removed from the hosts. Agents develop on seed 1; seeds 0 and 2 are held out and evaluated only by the operator, and agents never see those scores. Iterations 1–36 optimize a 10-task subset; iterations 37–51 use all 42 tasks. The supervisor and all agents used Claude Sonnet 5.5, and the run booked $472 through iteration 50 (excluding runtime-LLM calls). The iteration-50 policy scores 62.1 held-out (50 episodes per task on each held-out seed) and 62.6 on dev.

D Matched-Budget Solo Ablations

On two of the additional settings we compare the team against a single Claude Code agent with the same model at matched spend. On MI350X, from the same starting commit and task statement, at $300 the solo agent (Opus 5.5, medium effort) reaches 1.65×, while the team reaches 2.13×. On the offload task, starting from the team's 3.2× engine, the two are close at $200 (1.047× for solo vs. 1.056×, averaged over the dev and two held-out workloads), but within the same 20-hour window the solo agent progresses more slowly and spends only $200, while the team reaches 1.083× at $509.

KDA prefill on AMD MI350X. The solo run starts from the team's starting commit and task statement. Figure 8 compares the team and the solo run at matched spend.

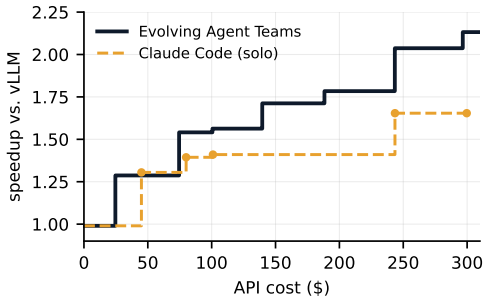


Figure 8: KDA prefill on MI350X over the first $300.

MoE expert-offload serving, from an optimized start. A second run starts both the team and a solo agent from the final engine of the run in Appendix C (about 3.2× naive), each on two identical Macs, both shown a $1,000 budget, and both stopped at the same 20-hour deadline. Table 4 reports all

versions re-measured afterwards in the same batch on the same machines, on the dev workload and on two held-out workloads: (A) a long prefix-reuse session and (B) four real Claude Code sessions. At $200 the team and the solo agent are close; the solo agent leads on the workload it optimized and the team on both held-out workloads. The solo agent's later changes are mostly small parameter tweaks (e.g., the half-life of the expert-cache eviction policy), which help the dev workload more than the held-out ones. The solo agent progressed more slowly and spent $200 in the 20 hours, while the team kept improving through $509 (decode lookahead hints, then exact multi-row verification enabling speculative decoding, then a larger victim cache), and its final engine is the fastest on every workload. Measured together on the same Macs, it decodes at 30.3 tokens/s on the batch-1 decode benchmark, against 26.7 for the starting commit and 28.2 for the solo agent's final engine.

Table 4: MoE offload serving, second run: speedup over the shared starting engine, re-measured in the same batch on the same machines (macOS 26.7). "All" is the equal-weight mean of the three workloads. Rows are labeled by wall-clock time since the start; both runs stopped at the same 20-hour deadline, and the team reached the solo run's final cost ($200) after about 8 hours.

Version	**Cost**	**Dev**	**Held-out A**	**Held-out B**	**All**
Claude Code (solo), 20 h	$200	1.060	1.036	1.045	1.047
EVOLVING AGENT TEAMS, 8 h	$200	1.050	1.062	1.055	1.056
EVOLVING AGENT TEAMS, 20 h	$509	**1.098**	**1.080**	**1.072**	**1.083**